# Post-Training Mastery

LLM Fine-Tuning, RLHF and Reinforcement Learning,
from First Principles to Interview Ready

Supervised fine-tuning • RLHF • DPO and its variants • GRPO and reasoning RL • RL environments • Systems • Libraries • 150+ interview questions

**Mohammed Saqlain**

Draft, October 2026

# Contents

**Preface** i

**How to Use This Book** ii
Three reading paths . . . ii
Boxes used in this book . . . ii
Running the code . . . iii

**Notation** iv

**I Foundations** 1

**1 The Post-Training Landscape** 2
1.1 What a base model is . . . 2
1.2 The modern pipeline . . . 2
1.3 What "alignment", "helpfulness" and "reasoning" mean in practice . . . 4
1.4 The main job roles . . . 4
1.5 A short history . . . 4
1.6 How the rest of Part I fits in . . . 5
1.7 Two published recipes, stage by stage . . . 6
1.8 Where the compute goes: a back-of-the-envelope estimate . . . 6
1.9 How to read a post-training paper in ten minutes . . . 7
1.10 Common mistakes . . . 8

**2 Probability and Statistics You Actually Need** 10
2.1 Random variables and distributions . . . 10
2.2 Expectation, variance and covariance . . . 10
2.3 Joint, marginal and conditional probability . . . 11
2.4 Common distributions . . . 12
2.5 Autoregressive language models . . . 13
2.6 Maximum likelihood estimation . . . 14
2.7 Monte Carlo estimation . . . 15
2.8 Importance sampling . . . 16
2.9 The log-derivative trick . . . 16
2.10 The logistic function and the Bradley-Terry model . . . 18
2.11 Numerical stability: working in log space . . . 19

**3 Information Theory** 22
3.1 Surprise and entropy . . . 22

3.2 Cross-entropy . . . 23
3.3 KL divergence . . . 24
3.4 Forward vs reverse KL . . . 25
3.5 Estimating KL from samples . . . 26
3.6 Perplexity and bits per byte . . . 29
3.7 Mutual information and Jensen-Shannon divergence . . . 30
3.8 Entropy of a policy and entropy collapse . . . 30

**4 Optimisation and Training Mechanics 33**
4.1 Loss functions and gradients . . . 33
4.2 Gradient descent and SGD . . . 34
4.3 Momentum, RMSProp and Adam . . . 34
4.4 Learning-rate schedules . . . 37
4.5 Gradient clipping . . . 38
4.6 Batch size, gradient accumulation and the learning rate . . . 39
4.7 Why deep networks train at all . . . 40
4.8 Overfitting, validation and early stopping . . . 41
4.9 Reading a loss curve . . . 41

**5 The Transformer and LLM Pretraining Recap 43**
5.1 The big picture . . . 43
5.2 Embeddings and the residual stream . . . 43
5.3 Scaled dot-product attention . . . 44
5.4 MQA, GQA and MLA . . . 45
5.5 Positional information . . . 46
5.6 The feed-forward block . . . 47
5.7 Normalisation: LayerNorm vs RMSNorm . . . 49
5.8 The LM head and logits . . . 49
5.9 A tiny transformer block in PyTorch . . . 50
5.10 Counting parameters and FLOPs . . . 51
5.11 The KV cache and why generation is memory bound . . . 52
5.12 Pretraining in brief . . . 52
5.13 Base, instruct and reasoning checkpoints . . . 53

**6 Tokenisation 55**
6.1 Why tokenise? . . . 55
6.2 Byte-pair encoding (BPE) . . . 56
6.3 WordPiece, Unigram and SentencePiece . . . 58
6.4 Special tokens . . . 58
6.5 Where tokenisation goes wrong . . . 59
6.6 Tokenizers in Hugging Face . . . 59
6.7 Measuring tokenizer effects yourself . . . 61
6.8 Adding special tokens safely . . . 62
6.9 Common mistakes . . . 63

**7 Decoding and Sampling 65**

7.1 From logits to a distribution . . . 65
7.2 Greedy decoding and beam search . . . 65
7.3 Temperature . . . 66
7.4 Truncation: top-k, top-p and min-p . . . 66
7.5 Repetition, presence and frequency penalties . . . 68
7.6 Stop conditions, length limits and log-probabilities . . . 68
7.7 Defaults you did not choose . . . 69
7.8 Sampling for RL rollouts . . . 69
7.9 Speculative decoding . . . 70
7.10 What truncation does to the probabilities RL sees . . . 71
7.11 Many samples per question: self-consistency and best-of-n . . . 72
7.12 Reproducibility and seeds . . . 73
7.13 Common mistakes . . . 73

**II Supervised Fine-Tuning 75**

**8 Instruction Tuning and Chat Data 76**
8.1 What SFT changes in a model . . . 76
8.2 Data formats . . . 77
8.3 Chat templates . . . 78
8.4 Where SFT data comes from . . . 79
8.5 Multi-turn and system prompts . . . 81
8.6 What actually changes: token distribution shift . . . 81
8.7 The data mixture is a hyperparameter . . . 81
8.8 Choosing examples: from heuristics to influence . . . 82
8.9 Common mistakes . . . 83

**9 The SFT Loss in Detail 85**
9.1 Token-level cross-entropy with a mask . . . 85
9.2 What the SFT loss really minimises . . . 86
9.3 The gradient with respect to the logits . . . 86
9.4 Which tokens to train on . . . 87
9.5 Loss normalisation . . . 88
9.6 Packing and padding-free batching . . . 88
9.7 Truncation . . . 90
9.8 Catastrophic forgetting . . . 90
9.9 Variants of the SFT loss . . . 90
9.10 Hyperparameters that matter for SFT . . . 91
9.11 Reading loss numbers: perplexity . . . 91
9.12 Gradient accumulation and the token count . . . 92
9.13 Teacher forcing and exposure bias . . . 93
9.14 Noise in the embeddings: NEFTune . . . 93
9.15 TRL's defaults, checked . . . 93
9.16 Common mistakes . . . 94

**10 Parameter-Efficient Fine-Tuning** **96**
10.1 Where the memory goes . . . 96
10.2 LoRA: Low-Rank Adaptation . . . 96
10.3 The LoRA hyperparameters . . . 98
10.4 rsLoRA: fixing the scale . . . 99
10.5 LoRA+: different learning rates for A and B . . . 99
10.6 PiSSA: start from the important directions . . . 99
10.7 DoRA: magnitude and direction . . . 100
10.8 QLoRA: LoRA on a 4-bit base . . . 100
10.9 LoRA for RL . . . 101
10.10 The gradients of LoRA . . . 101
10.11 How much rank does a task need? . . . 102
10.12 The capacity argument for RL, with numbers . . . 103
10.13 Compute and memory, worked out . . . 103
10.14 Other parameter-efficient methods . . . 104
10.15 Using PEFT in code . . . 105
10.16 Common mistakes . . . 105

**11 Number Formats and Quantisation** **107**
11.1 Floating-point formats . . . 107
11.2 Quantisation basics . . . 108
11.3 Outliers and LLM.int8 . . . 109
11.4 Weight-only post-training quantisation: GPTQ and AWQ . . . 109
11.5 Quantisation-aware training . . . 110
11.6 FP8 training and inference . . . 110
11.7 Quantisation and RL . . . 111
11.8 Quantisation error: the numbers behind the methods . . . 111
11.9 How many bits does "4-bit" really cost? . . . 112
11.10 Quantising the KV cache . . . 113
11.11 Common mistakes . . . 113

**12 Knowledge Distillation** **115**
12.1 The idea . . . 115
12.2 Three kinds of LLM distillation . . . 115
12.3 Forward vs reverse KL . . . 115
12.4 GKD: on-policy distillation . . . 117
12.5 On-policy distillation as dense-reward RL . . . 117
12.6 Practical points . . . 118
12.7 Token-level distillation: the gradients . . . 118
12.8 Inside TRL's distillation loss . . . 119
12.9 What distillation costs: a worked comparison . . . 120
12.10 Common mistakes . . . 120

**III Reinforcement Learning from the Ground Up** **122**

**13 The Reinforcement Learning Problem** **123**
13.1 Agent, environment, reward . . . 123
13.2 Markov decision processes . . . 124
13.3 Policies . . . 124
13.4 Returns and discounting . . . 124
13.5 The LLM as a policy . . . 125
13.6 Exploration and exploitation . . . 126
13.7 Worked example: one LLM episode, written out as an MDP . . . 127
13.8 Partial observability: when the prefix is not the whole state . . . 128
13.9 Horizon, discounting and the "effective horizon" . . . 129
13.10 Common mistakes . . . 129

**14 Value Functions and Bellman Equations** **131**
14.1 Three value functions . . . 131
14.2 The Bellman expectation equations . . . 131
14.3 The Bellman optimality equations . . . 132
14.4 Dynamic programming . . . 132
14.5 Monte Carlo vs temporal-difference learning . . . 133
14.6 Q-learning and DQN . . . 134
14.7 A worked example: three ways to get the same values . . . 134
14.8 $n$-step returns and TD($\lambda$) . . . 136
14.9 What a value means for a language model . . . 136
14.10 Function approximation and the "deadly triad" . . . 137
14.11 Common mistakes . . . 137

**15 Policy Gradient Methods** **139**
15.1 The goal . . . 139
15.2 Deriving the policy gradient . . . 139
15.3 Reward-to-go: using causality . . . 140
15.4 Baselines . . . 140
15.5 Advantage normalisation . . . 141
15.6 REINFORCE . . . 141
15.7 Code: REINFORCE on CartPole . . . 142
15.8 Variance and its sources . . . 143
15.9 The score function of a softmax policy . . . 144
15.10 The optimal baseline . . . 144
15.11 Common mistakes . . . 145

**16 Actor-Critic Methods and GAE** **147**
16.1 From REINFORCE to actor-critic . . . 147
16.2 Generalised Advantage Estimation . . . 147
16.3 The critic in LLM RL . . . 149
16.4 Practical details of critics . . . 150
16.5 Seeing the bias-variance trade-off . . . 150

16.6 Length-adaptive $\lambda$ and decoupled $\lambda$ . . . 151
16.7 Bootstrapping at truncation . . . 152
16.8 Common mistakes . . . 152

**17 Trust Regions: TRPO and PPO** **154**
17.1 Why big policy steps break training . . . 154
17.2 The performance difference lemma . . . 154
17.3 TRPO . . . 155
17.4 PPO-Clip . . . 155
17.5 PPO-Penalty and adaptive KL . . . 157
17.6 Entropy bonus . . . 157
17.7 The full PPO algorithm . . . 157
17.8 Implementation details that matter . . . 158
17.9 Code: PPO from scratch on CartPole . . . 158
17.10 The gradient of the clipped objective, written out . . . 160
17.11 Why the clip does not bound the KL . . . 160
17.12 The value loss and value clipping . . . 162
17.13 PPO hyperparameters: control vs LLMs . . . 162
17.14 PPO as approximate natural-gradient or mirror descent . . . 163
17.15 Common mistakes . . . 163

**18 Off-Policy, Offline and Other RL Ideas** **166**
18.1 Off-policy correction with importance sampling . . . 166
18.2 V-trace . . . 166
18.3 SAC and DDPG, briefly . . . 167
18.4 Offline RL . . . 167
18.5 Bandits and best-of-$n$ . . . 168
18.6 How bad does importance sampling get? A measurement . . . 168
18.7 Offline RL for language models: weighting, filtering and DPO . . . 170
18.8 Where off-policy data comes from in LLM RL . . . 171
18.9 Common mistakes . . . 172

**IV Reinforcement Learning from Human Feedback** **174**

**19 Reward Models** **175**
19.1 Collecting preference data . . . 175
19.2 The Bradley-Terry model . . . 175
19.3 Architecture: a value head on a language model . . . 176
19.4 Training details . . . 177
19.5 Ensembles and uncertainty . . . 178
19.6 Generative reward models and LLM judges . . . 178
19.7 Evaluating reward models . . . 178
19.8 Where Bradley-Terry comes from: random utilities . . . 179
19.9 The ceiling set by label noise . . . 179
19.10 Shortcut features: the length example . . . 179

19.11 What a scalar reward cannot represent . . . 180
19.12 Using TRL's `RewardTrainer` . . . 181
19.13 Common mistakes . . . 182

**20 The PPO-RLHF Loop** **184**
20.1 The four models . . . 184
20.2 The KL-regularised objective . . . 184
20.3 The per-token reward . . . 185
20.4 One iteration, step by step . . . 185
20.5 The KL coefficient and KL controllers . . . 186
20.6 Whitening and length effects . . . 187
20.7 The InstructGPT recipe . . . 187
20.8 The N+ implementation details . . . 187
20.9 Memory and compute budget . . . 188
20.10 Why the per-token penalty gives the sequence KL . . . 188
20.11 A complete toy PPO-RLHF loop . . . 189
20.12 Training the critic well . . . 190
20.13 Monitoring a PPO-RLHF run . . . 191
20.14 PPO-RLHF in today's frameworks . . . 191
20.15 Common mistakes . . . 192

**21 Reward Hacking and Overoptimisation** **194**
21.1 Goodhart's law . . . 194
21.2 Scaling laws for overoptimisation . . . 194
21.3 Common forms of hacking . . . 195
21.4 Mitigations . . . 195
21.5 Goodhart in a simulation: light vs heavy-tailed errors . . . 196
21.6 Hacking in reasoning and agent training . . . 197

**V Preference Optimisation without a Separate Reward Model** **200**

**22 Direct Preference Optimisation** **201**
22.1 Step 1: the optimal policy of KL-regularised RL . . . 201
22.2 Step 2: solve for the reward . . . 201
22.3 Step 3: plug into Bradley-Terry . . . 202
22.4 Step 4: the DPO loss . . . 202
22.5 Step 5: the gradient . . . 202
22.6 Code: the DPO loss . . . 202
22.7 $\beta$ and the reference model . . . 203
22.8 What DPO does to likelihoods . . . 204
22.9 Length exploitation . . . 205
22.10 What to monitor . . . 205
22.11 From pairs to rankings: the Plackett-Luce form . . . 205
22.12 Why DPO overfits: the deterministic-preference argument . . . 205
22.13 The token-level view: DPO as a Q-function . . . 207

22.14 Is DPO really better than PPO? . . . 207
22.15 Using TRL's `DPOTrainer` . . . 208
22.16 Choosing the hyperparameters . . . 208
22.17 A worked example of the gradient weight . . . 209
22.18 Common mistakes . . . 210

**23 DPO Variants** **212**
23.1 IPO: fixing overfitting to certain preferences . . . 212
23.2 Noisy labels: cDPO and robust DPO . . . 212
23.3 KTO: no pairs needed . . . 213
23.4 ORPO: no reference model, one stage . . . 213
23.5 SimPO: length-normalised, no reference . . . 214
23.6 RPO, APO and other likelihood fixes . . . 214
23.7 One comparison table . . . 215
23.8 Code: several losses side by side . . . 215
23.9 One picture: every pairwise loss is a function of the margin . . . 216
23.10 KTO's view: prospect theory and HALOs . . . 217
23.11 ORPO's odds ratio, worked out . . . 217
23.12 Choosing a method in practice . . . 218

**24 Online and Iterative Preference Learning** **221**
24.1 Offline vs online data . . . 221
24.2 Iterative DPO . . . 221
24.3 Self-rewarding models . . . 222
24.4 Nash and game-theoretic methods . . . 223
24.5 Offline vs online in a simulation . . . 223
24.6 Online DPO in TRL . . . 224

**25 RLAIF, Constitutional AI and Rubrics** **227**
25.1 AI feedback instead of human feedback . . . 227
25.2 Constitutional AI . . . 227
25.3 Rubrics and LLM judges as rewards . . . 228
25.4 Judge biases . . . 229
25.5 Deliberative alignment and spec-based training . . . 229
25.6 From judge outputs to training labels: the maths of soft labels . . . 229
25.7 Measuring a judge before you trust it . . . 231
25.8 Rubric rewards inside an RL loop . . . 231
25.9 Self-rewarding loops . . . 232
25.10 Common mistakes . . . 233

**VI Reasoning through Reinforcement Learning** **235**

**26 RL with Verifiable Rewards** **236**
26.1 What a verifiable reward is . . . 236
26.2 Answer extraction and checking . . . 237

26.3 Sparse rewards and pass@$k$ . . . 238
26.4 Does RL add new abilities or sharpen existing ones? . . . 239
26.5 A map of verifiers . . . 240
26.6 What verifier errors do to learning . . . 241
26.7 Code rewards in practice . . . 243
26.8 Common mistakes . . . 243

**27 GRPO** **245**
27.1 From PPO to GRPO . . . 245
27.2 The GRPO objective . . . 245
27.3 The KL term in practice . . . 246
27.4 Loss aggregation . . . 247
27.5 Code: a minimal GRPO step . . . 247
27.6 Using TRL's `GRPOTrainer` . . . 249
27.7 Derivation: what gradient does GRPO follow? . . . 250
27.8 A worked example of group advantages . . . 251
27.9 The gradient coefficient view (DeepSeekMath's unified paradigm) . . . 251
27.10 Process supervision and iterative GRPO . . . 252
27.11 Reward functions in TRL . . . 253
27.12 Compute and memory: GRPO vs PPO . . . 253
27.13 Common mistakes . . . 254

**28 GRPO Variants and Critic-Free Methods** **256**
28.1 A common frame . . . 256
28.2 Dr. GRPO: removing two biases . . . 257
28.3 DAPO: four practical techniques . . . 258
28.4 RLOO: leave-one-out baselines . . . 260
28.5 REINFORCE++ and ReMax . . . 261
28.6 GSPO: sequence-level importance ratios . . . 261
28.7 CISPO: clip the weight, keep every token . . . 262
28.8 SAPO and other soft gates . . . 264
28.9 Masking: which tokens and sequences to train on . . . 264
28.10 Off-policy data, asynchronous RL and train/inference mismatch . . . 265
28.11 One comparison table . . . 266
28.12 Common mistakes . . . 267

**29 Process vs Outcome Rewards and Verifiers** **269**
29.1 Outcome vs process supervision . . . 269
29.2 The evidence . . . 269
29.3 Making step labels without humans: Monte Carlo rollouts . . . 270
29.4 Generative verifiers and "thinking" PRMs . . . 271
29.5 Using verifiers: training time vs test time . . . 271
29.6 Training a PRM . . . 272
29.7 Selecting answers with verifiers: a simulation . . . 273
29.8 Common mistakes . . . 274

**30 Test-Time Compute and the DeepSeek-R1 Recipe** **277**
30.1 Parallel test-time compute: sample many, pick one . . . 277
30.2 Search: tree search and step-level beam search . . . 278
30.3 Sequential test-time compute: thinking longer . . . 279
30.4 DeepSeek-R1-Zero: RL straight from the base model . . . 279
30.5 DeepSeek-R1: the four-stage pipeline . . . 280
30.6 Distilling R1 into small models . . . 281
30.7 Open reproductions and what they taught us . . . 281
30.8 Common mistakes . . . 282

**31 Practical RL Engineering for Reasoning Models** **284**
31.1 Data: difficulty is everything . . . 284
31.2 Entropy: the exploration budget . . . 285
31.3 Length: the other runaway variable . . . 286
31.4 The metrics to log . . . 287
31.5 Evaluating reasoning: pass@k and its trap . . . 287
31.6 Long runs: resets, staging and stability . . . 288
31.7 A debugging walk-through . . . 288
31.8 Common mistakes . . . 289

**VII RL Environments and Agents** **291**

**32 Environment Basics** **292**
32.1 What an environment is . . . 292
32.2 The Gymnasium API . . . 292
32.3 Terminated vs truncated . . . 293
32.4 Writing a custom environment . . . 295
32.5 Wrappers: changing one thing at a time . . . 296
32.6 Vector environments . . . 297
32.7 Seeding and reproducibility . . . 298
32.8 From Gymnasium to LLM environments . . . 298
32.9 Common mistakes . . . 299

**33 LLM Environments** **302**
33.1 Text in, text out . . . 302
33.2 Single-turn and multi-turn rollouts . . . 303
33.3 Tokens in, tokens out . . . 305
33.4 Parsers, rubrics and reward functions as code . . . 306
33.5 LLM environment libraries in 2026 . . . 308
33.6 Designing the observation . . . 310
33.7 Common mistakes . . . 310

**34 Agentic and Multi-Turn RL** **312**
34.1 The multi-turn trajectory . . . 312
34.2 Credit assignment across turns . . . 314

34.3 Long horizons and context management . . . 316
34.4 Asynchronous rollout systems . . . 317
34.5 Putting it together: an agent RL recipe . . . 319
34.6 Common mistakes . . . 319

**35 Sandboxes and Execution** **321**
35.1 The job of a sandbox . . . 321
35.2 Isolation levels . . . 321
35.3 Resource limits . . . 322
35.4 Grading: where code rewards break . . . 324
35.5 Throughput . . . 325
35.6 Web, browser and computer-use environments . . . 326
35.7 The security model . . . 326
35.8 Common mistakes . . . 327

**36 Reward and Curriculum Design** **329**
36.1 Sparse and dense rewards . . . 329
36.2 Potential-based shaping . . . 329
36.3 Format, length and partial-credit rewards . . . 331
36.4 Curriculum learning . . . 332
36.5 Detecting reward hacking inside an environment . . . 334
36.6 A checklist for designing an environment's reward . . . 335
36.7 Common mistakes . . . 335

**VIII Evaluation, Safety and Data** **337**

**37 Evaluation** **338**
37.1 What we evaluate, and with what . . . 338
37.2 pass@k and maj@k . . . 339
37.3 Statistics for small benchmarks . . . 341
37.4 LLM-as-a-judge . . . 342
37.5 Contamination and leakage . . . 343
37.6 Building your own evaluation harness . . . 344
37.7 Common mistakes . . . 345

**38 Safety and Alignment** **347**
38.1 Helpful, honest, harmless . . . 347
38.2 Refusals and over-refusal . . . 347
38.3 Safety in the post-training pipeline . . . 348
38.4 Red-teaming and jailbreaks . . . 350
38.5 Specification gaming, sycophancy and deception . . . 351
38.6 Common mistakes . . . 352

**39 Data Curation** **354**
39.1 Sources of post-training data . . . 354

39.2 Quality over quantity . . . 355
39.3 Filtering . . . 355
39.4 Deduplication . . . 356
39.5 Decontamination . . . 357
39.6 Difficulty estimation and data mixing . . . 357
39.7 Rejection sampling fine-tuning . . . 358
39.8 Common mistakes . . . 360

**IX Systems for Post-Training 362**

**40 Memory Maths 363**
40.1 Counting parameters . . . 363
40.2 Bytes per parameter in training . . . 364
40.3 Activations . . . 364
40.4 Logits: the hidden giant . . . 365
40.5 The KV cache . . . 365
40.6 Putting it together: a calculator . . . 365
40.7 Memory for RL . . . 366
40.8 Quick estimation rules . . . 367
40.9 Common mistakes . . . 367

**41 Mixed Precision and Memory Savers 369**
41.1 Floating-point formats . . . 369
41.2 FP16 training and loss scaling . . . 370
41.3 BF16 and why it became the default . . . 371
41.4 FP8 training . . . 372
41.5 Gradient checkpointing . . . 372
41.6 FlashAttention . . . 373
41.7 Fused kernels and chunked cross-entropy . . . 374
41.8 CPU and NVMe offload . . . 375
41.9 Common mistakes . . . 375

**42 Distributed Training 377**
42.1 Collective operations . . . 377
42.2 Data parallelism and DDP . . . 378
42.3 ZeRO: sharding the data-parallel states . . . 378
42.4 FSDP and FSDP2 . . . 380
42.5 Tensor parallelism . . . 380
42.6 Pipeline parallelism . . . 381
42.7 Sequence and context parallelism . . . 382
42.8 Expert parallelism . . . 382
42.9 Choosing a layout . . . 382
42.10Common mistakes . . . 383

**43 Rollout Engines 385**

43.1 Why generation dominates . . . 385
43.2 Continuous batching . . . 386
43.3 PagedAttention and vLLM . . . 387
43.4 SamplingParams for RL . . . 388
43.5 Weight synchronisation . . . 389
43.6 Log-probability mismatch . . . 390
43.7 Speculative decoding for rollouts . . . 391
43.8 Common mistakes . . . 391

**44 Designing an RL Training System** **393**
44.1 The RL dataflow . . . 393
44.2 Step time: a whiteboard model . . . 393
44.3 Placement . . . 394
44.4 Synchronous, one-step-off and fully asynchronous . . . 395
44.5 Single controller versus multi-controller . . . 396
44.6 Rewards as a service . . . 397
44.7 Fault tolerance and checkpointing . . . 397
44.8 Monitoring: the dashboard . . . 397
44.9 A reference design . . . 397
44.10Common mistakes . . . 398

**X Library Reference, Parameter by Parameter** **400**

**45 How to Read This Reference; TrainingArguments and Trainer** **401**
45.1 How this reference was built . . . 401
45.2 The Trainer in one page . . . 402
45.3 Batch size and step arithmetic . . . 402
45.4 The parameters that matter most . . . 403
45.5 Common mistakes . . . 404
45.6 Full parameter list: TrainingArguments . . . 405

**46 PEFT LoraConfig and BitsAndBytesConfig** **411**
46.1 LoraConfig: the parameters that matter . . . 411
46.2 BitsAndBytesConfig: 4-bit and 8-bit loading . . . 413
46.3 Common mistakes . . . 414
46.4 Full parameter list: LoraConfig . . . 414
46.5 Full parameter list: BitsAndBytesConfig constructor . . . 417

**47 TRL SFTConfig and SFTTrainer** **418**
47.1 Dataset formats decide the loss . . . 418
47.2 Packing and padding-free training . . . 419
47.3 The other parameters that matter . . . 420
47.4 How the loss is averaged . . . 421
47.5 A complete LoRA SFT script . . . 422
47.6 Common mistakes . . . 423

47.7 Parameters specific to SFTConfig . . . 424

**48 TRL RewardConfig and RewardTrainer 426**
48.1 What RewardTrainer does . . . 426
48.2 The parameters that matter . . . 427
48.3 Evaluating the reward model before using it . . . 428
48.4 From scores to RL rewards . . . 429
48.5 Common mistakes . . . 430
48.6 Parameters specific to RewardConfig . . . 431

**49 TRL DPOConfig and KTOConfig 432**
49.1 DPOConfig essentials (TRL 1.16.0) . . . 432
49.2 KTOConfig . . . 434
49.3 Common mistakes . . . 434
49.4 Parameters specific to DPOConfig . . . 435
49.5 Parameters specific to KTOConfig . . . 437

**50 TRL GRPOConfig and RLOOConfig 439**
50.1 Batch geometry . . . 439
50.2 Loss options . . . 440
50.3 Generation and vLLM settings . . . 441
50.4 Rewards, tools and environments . . . 441
50.5 RLOOConfig . . . 443
50.6 Common mistakes . . . 444
50.7 Parameters specific to GRPOConfig . . . 444
50.8 Parameters specific to RLOOConfig . . . 450

**51 OpenRLHF and veRL 454**
51.1 The same knobs under three names . . . 454
51.2 OpenRLHF . . . 455
51.3 veRL . . . 457
51.4 Common mistakes . . . 459
51.5 Generated reference: OpenRLHF `train_ppo_ray` flags . . . 459
51.6 Generated reference: veRL PPO trainer configuration . . . 464

**52 Unsloth 479**
52.1 The workflow . . . 479
52.2 The parameters that matter . . . 480
52.3 Where Unsloth's savings come from . . . 481
52.4 A GRPO run with Unsloth . . . 482
52.5 Common mistakes . . . 483
52.6 Generated reference: `FastLanguageModel.from_pretrained` . . . 484
52.7 Generated reference: `get_peft_model` . . . 485
52.8 Generated reference: `save_pretrained_merged` . . . 485
52.9 Generated reference: `save_pretrained_gguf` . . . 486

**53 DeepSpeed Configuration and vLLM Settings 487**
53.1 A DeepSpeed config for post-training . . . 487
53.2 vLLM settings for RL . . . 489
53.3 Common mistakes . . . 489
53.4 Generated reference: DeepSpeed `zero_optimization` . . . 490
53.5 Generated reference: DeepSpeed offload settings . . . 493
53.6 Generated reference: DeepSpeed `bf16` and `fp16` . . . 493
53.7 Generated reference: vLLM `SamplingParams` . . . 494
53.8 Generated reference: vLLM engine configuration (selected classes) . . . 496

**XI Projects 503**

**54 Projects I: SFT, Rewards and Preferences 504**
54.1 How to run a project well . . . 504
54.2 Project 1: Instruction-tune a small model with correct loss masking . . . 504
54.3 Project 2: Train and audit a reward model . . . 507
54.4 Project 3: A DPO-family ablation . . . 508
54.5 Project 4: Distil a reasoning teacher into a small student . . . 510
54.6 Project 5: QLoRA on a single GPU, with a memory budget you can defend . . . 512

**55 Projects II: RL, Agents, Systems and Evaluation 515**
55.1 Project 6: GRPO on grade-school maths with a verifier . . . 515
55.2 Project 7: A multi-turn tool-use environment . . . 518
55.3 Project 8: Reward over-optimisation, measured . . . 520
55.4 Project 9: A mini asynchronous RL system . . . 521
55.5 Project 10: An evaluation harness you can trust . . . 523

**XII Interview Preparation 527**

**56 Interview Questions I: Concepts and Mathematics 528**
56.1 Answering well . . . 528
56.2 Conceptual questions: the post-training pipeline . . . 529
56.3 Conceptual questions: reinforcement learning . . . 530
56.4 Conceptual questions: rewards, preferences and data . . . 531
56.5 Conceptual questions: environments, agents and systems . . . 532
56.6 Conceptual questions: evaluation and safety . . . 533
56.7 Mathematical questions . . . 534

**57 Interview Questions II: Coding, System Design and Behaviour 538**
57.1 Coding questions . . . 538
57.2 System design questions . . . 548
57.3 Behavioural questions . . . 550

**58 Cheat Sheet 552**
58.1 Core formulas . . . 552

58.2 Numbers to know . . . 553
58.3 Default hyperparameters (TRL 1.16 unless noted) . . . 553
58.4 Debugging RL: symptom → first checks . . . 554
58.5 Which method when . . . 554

**59 Glossary** **556**

**A Study Plans** **568**
A.1 Two-week fast track . . . 568
A.2 Eight-week full path . . . 568
A.3 Reference path . . . 569

**B Versions and Verification** **570**
B.1 Versions used . . . 570
B.2 Checking a parameter yourself . . . 571

**C References** **572**

# Preface

This book has one goal. When you finish it, you should be able to work on the post-training of large language models, and you should be able to answer the questions that come up in interviews for these jobs.

**Post-training** means everything that happens to a language model after its first big training run on raw text. That first run is called **pretraining**. It gives a model that can continue text, but it does not follow instructions well, it does not hold a conversation, and it does not reason reliably. Post-training fixes this. It includes supervised fine-tuning, preference optimisation, reinforcement learning, distillation and safety training. Today, much of what makes a model useful is decided in post-training.

The book starts from the basics. You do not need to know reinforcement learning before you begin. You do need to be comfortable with Python, and you should have seen a neural network before. Every other idea is built up step by step.

Each idea is taught in the same way:

1. A plain explanation in short sentences.
2. The math, written out in full, with every step shown.
3. The intuition: what the math means in words.
4. A worked example with small numbers you can check by hand.
5. A short piece of PyTorch code you can run on a laptop.
6. Common mistakes, and quick interview questions.

Some parts of the field change every few months. Library parameters change. New variants of algorithms appear. Where this matters, the book says which library version it was checked against. The core ideas (likelihood, KL divergence, policy gradients, trust regions, preference models) change very slowly. If you understand them well, you can read any new paper in this area and see what is really new in it.

Read with a notebook open. Derive the equations yourself. Run the code. Change the numbers and see what happens. That is how this material becomes yours.

# How to Use This Book

## Three reading paths

**Fast track (about 2 weeks).** Read Chapters 1, 3, 7, 9, 10, 15, 17, 19, 22, 27 and 28. Then do Projects 1, 3 and 6 (Chapters 54 and 55), work through the interview questions (Chapters 56 and 57) and learn the cheat sheet (Chapter 58). This gives you the core of modern post-training. Appendix A turns this into a day-by-day plan.

**Full path (about 6 to 8 weeks).** Read the book in order. Do one project at the end of each part. Answer every "Interview check" box before you read its answers.

**Reference use.** The library chapters (Part X), the cheat sheet and the glossary are built for looking things up. Use them while you work on real training runs.

## Boxes used in this book

**Note.**
Extra detail or context. You can skip these on a first read.

**Tip.**
Practical advice from real training runs.

**Common mistake.**
A mistake that people often make, in code or in interviews.

**Worked example.**
A calculation with small numbers that you can check by hand.

**Why this, not that? Why this, not that?**

A design choice explained in three parts: the **Reason** for it, the **Trade-off** it costs, and **When the alternative is better**. Interviewers ask exactly these questions.

**Key takeaways**

The key points of a chapter, for revision.

**Interview check**

Short questions at the end of each chapter. Try to answer them out loud, as if in an interview, before you read the answer.

## Running the code

All code in Part I runs on a CPU. You need Python 3.10 or newer and PyTorch:

```
python -m venv .venv
source .venv/bin/activate
pip install torch numpy
```

Later parts use `transformers`, `trl`, `peft`, `datasets`, `accelerate` and `deepspeed`. Every runnable snippet in the book was executed on a CPU with the versions listed in Appendix B (for example TRL 1.16.0 and Transformers 5.20.2). Code that needs a GPU or a cluster (vLLM, veRL, OpenRLHF, Unsloth) is marked as not executed, and its parameter names were checked against the library source instead.

# Notation

The same symbols are used across the whole book. Come back to this table when a symbol is unclear.

| Symbol | Meaning |
|---|---|
| $x$ | A prompt (the input text given to the model) |
| $y$ | A response (the text the model writes) |
| $y_t$ | The $t$-th token of the response |
| $y_{<t}$ | All response tokens before position $t$ |
| $\theta$ | The trainable parameters (weights) of a model |
| $\pi_\theta(y \mid x)$ | The policy: the probability that the model with weights $\theta$ writes $y$ given $x$ |
| $\pi_{\text{ref}}$ | The reference policy, usually the frozen model before RL or preference training |
| $\pi_{\theta_{\text{old}}}$ | The policy that generated the current batch of samples |
| $r(x, y)$ | The reward for response $y$ to prompt $x$ |
| $r_\phi$ | A learned reward model with parameters $\phi$ |
| $s_t, a_t, r_t$ | State, action and reward at time step $t$ |
| $\gamma$ | Discount factor, between 0 and 1 |
| $\lambda$ | GAE parameter, between 0 and 1 |
| $V(s)$, $Q(s,a)$, $A(s,a)$ | State value, action value and advantage |
| $\beta$ | Strength of the KL penalty (or the DPO temperature) |
| $\epsilon$ | Clip range in PPO and GRPO |
| $\rho_t$ | Importance ratio $\pi_\theta(a_t \mid s_t) / \pi_{\theta_{\text{old}}}(a_t \mid s_t)$ |
| $\mathbb{E}_{x\sim p}[f(x)]$ | Expected value of $f(x)$ when $x$ is drawn from $p$ |
| $H(p)$ | Entropy of distribution $p$ |
| $H(p, q)$ | Cross-entropy of $q$ relative to $p$ |
| $D_{\text{KL}}(p \Vert q)$ | KL divergence from $q$ to $p$ |
| $\sigma(z)$ | The logistic (sigmoid) function $1/(1+e^{-z})$ |
| $\eta$ | Learning rate |
| $B$, $G$ | Batch size; group size (number of samples per prompt in GRPO) |
| $V$ (vocabulary) | The vocabulary size, when the context makes clear it is not a value function |

| Symbol | Meaning |
|---|---|
| $d$, $d_{\text{model}}$ | Hidden size of the transformer |
| $L$ | Number of layers |
| $N$, $D$ | Number of model parameters; number of training tokens |

Logarithms are natural logarithms (base $e$) unless the text says otherwise. Vectors are lowercase bold only when there could be confusion; most of the time the context is clear.

PART I

# Foundations

CHAPTER 1

# The Post-Training Landscape

> **Why this chapter matters.**
> Before learning the tools, you need the map. This chapter explains what post-training is, which stages it has, and what each stage is for. Interviewers often open with "walk me through how a chat model is made". By the end of this chapter you can answer that clearly.

## 1.1 What a base model is

A **language model** is a program that gives a probability to every possible next piece of text. The pieces are called **tokens**. A token is usually a word, part of a word, or a punctuation mark (Chapter 6 explains this in detail).

A **base model** is a language model straight out of pretraining. **Pretraining** means training on a huge amount of text, often trillions of tokens, with one simple task: predict the next token. The model sees "The capital of France is" and learns to put high probability on the token `" Paris"` (with its leading space).

A base model is very knowledgeable. But it is not an assistant. If you give it the prompt

```
How do I sort a list in Python?
```

it might continue with another question, because on the internet, questions often appear in lists of questions. It does not know that it is supposed to *answer*. It also has no sense of what a good answer is, beyond what is likely to come next in web text.

So a base model has knowledge and skills, but they are not organised into useful, safe, reliable behaviour. **Post-training** is the set of methods that turn a base model into a model that follows instructions, holds a conversation, uses tools, reasons step by step, and refuses harmful requests.

## 1.2 The modern pipeline

Most strong models today pass through the stages below. Not every model uses every stage, and the order can vary. But this is the standard picture.

**1. Pretraining.** Next-token prediction on a very large and mixed set of text and code. This is by far the most compute-hungry stage. Result: the base model.

**2. Mid-training (also called continued pretraining or annealing).** More next-token training, but on a smaller and higher-quality mix: maths, code, long documents, reasoning traces, and sometimes instruction-like data. The learning rate is usually lowered during this stage. Long-context training often happens here too.

**3. Supervised fine-tuning (SFT).** The model is trained on examples of the behaviour we want: a prompt and a good response. The loss is the same next-token loss as pretraining, but it is usually computed only on the response tokens. SFT teaches format and style: how to answer, how to use a chat template, how to call tools. Chapters 8 to 10 cover this.

**4. Preference optimisation.** People (or another model) compare two responses and say which one is better. The model is trained to make the better response more likely and the worse one less likely. The main methods are RLHF with a reward model (Part IV) and direct methods such as DPO (Part V).

**5. Reinforcement learning (RL) with rewards.** The model generates responses, a reward signal scores them, and the model is updated to get more reward. The reward can come from a learned reward model, from a rule (for example, "is the final maths answer correct?"), from running unit tests, or from an environment such as a code sandbox. RL on checkable tasks is called **RL with verifiable rewards (RLVR)**. It is the main tool for teaching reasoning (Part VI) and agentic skills (Part VII).

**6. Distillation.** A large, strong model (the *teacher*) produces outputs, and a smaller model (the *student*) is trained to copy them. This is a cheap way to give small models skills that were first learned by a big model with RL.

**7. Safety training and evaluation.** Safety data is mixed into SFT and preference stages. Red-teaming finds failures. Evaluations decide which checkpoint ships.

Pretraining gives knowledge. SFT gives format. Preference optimisation gives taste. RL gives skill on tasks with clear goals. Distillation spreads those gains to smaller models.

This summary is a simplification, but it is a useful one. In real pipelines the stages are often repeated. A typical recipe for a reasoning model looks like: SFT on a small set of long reasoning examples ("cold start"), then RL with verifiable rewards, then rejection sampling to build a bigger SFT set, then another SFT and RL round that also covers general chat and safety. You will study this exact recipe in Chapter 30.

**Why this, not that? Why several stages, not one big training objective?**

**Why.** Each stage needs a different kind of data and fixes a different problem. Pretraining needs trillions of cheap tokens. SFT needs thousands to millions of good demonstrations. Preference learning needs comparisons. RL needs only a way to score answers. Splitting the pipeline lets each stage use the cheapest data that does its job, and lets teams test and fix each step on its own.
**Trade-off.** More stages mean more hand-offs, more hyperparameters and more chances for one stage to undo another (for example, RL making a model forget skills learned in SFT). Each stage also needs its own evaluation.
**When the alternative wins.** For a narrow product task, one stage is often enough: SFT alone, or SFT plus DPO, can beat a full pipeline when you have good data and little compute. Some recent recipes also merge stages, for example mixing SFT data into the RL loss.

## 1.3 What "alignment", "helpfulness" and "reasoning" mean in practice

These words are used loosely. In practice they mean measurable things.

- **Helpfulness**: the model does what the user asked, in a useful format, with correct content. It is measured by human or model judges comparing responses, and by task benchmarks.
- **Harmlessness**: the model avoids giving seriously harmful help, and it does *not* refuse harmless requests. Over-refusal is a failure too.
- **Honesty**: the model does not make up facts, says when it is unsure, and does not just agree with the user to please them (this failure is called **sycophancy**).
- **Reasoning**: the model can solve multi-step problems, such as maths, code and planning, usually by writing out intermediate steps before the final answer.
- **Alignment**: the general goal that the model's behaviour matches what its developers and users intend. In job descriptions, "alignment" often just means the post-training team.

## 1.4 The main job roles

Post-training jobs come in a few flavours. They overlap, but they test different things in interviews.

| Role | Main work | What interviews focus on |
|---|---|---|
| Research scientist | New methods, experiments, papers | Derivations, research taste, reading papers, experiment design |
| Research engineer | Training runs, infrastructure, making research work at scale | PyTorch, distributed training, debugging runs, system design |
| Applied / product post-training engineer | Fine-tuning models for products, data pipelines, evaluation | SFT and DPO in practice, data quality, evals, libraries |
| RL environments engineer | Building tasks, sandboxes, verifiers and reward functions | Environment design, reward hacking, security, scale |

All of them expect you to know the material in this book: the losses, the algorithms, the failure modes and the systems. The research roles expect deeper math. The engineering roles expect deeper systems and library knowledge.

## 1.5 A short history

You do not need to memorise dates, but knowing the order of ideas helps you explain *why* each method exists.

- **2017.** *Deep RL from human preferences* (Christiano and others) trains agents from human comparisons of behaviour. The same year, **PPO** (Proximal Policy Optimisation) is pub-

lished. It becomes the default RL algorithm. The **Transformer** architecture is also published.

- **2019 to 2020.** RL from human feedback is applied to language: fine-tuning GPT-2 on human preferences, then learning to summarise from human feedback.
- **2022. InstructGPT** sets out the classic three-step recipe: SFT, then a reward model trained on human rankings, then PPO against that reward model with a KL penalty. ChatGPT follows in late 2022. **Constitutional AI** shows that AI feedback, guided by a written list of principles, can replace much of the human feedback for harmlessness.
- **2023. Llama 2-Chat** describes an open RLHF pipeline in detail. **DPO** (Direct Preference Optimisation) shows that the RLHF objective can be optimised with a simple classification-style loss, with no separate reward model and no RL loop. Many variants follow (IPO, KTO, ORPO, SimPO). Work on **process reward models** shows that rewarding each reasoning step can beat rewarding only the final answer.
- **2024. GRPO** (Group Relative Policy Optimisation) appears in DeepSeekMath. It removes PPO's value network by comparing several answers to the same prompt. OpenAI's **o1** shows that RL can teach models to "think" for longer before answering, and that more thinking at test time gives better answers.
- **2025. DeepSeek-R1** publishes an open recipe for a strong reasoning model, including **R1-Zero**, trained with RL directly from a base model with only rule-based rewards. A wave of open work follows: **DAPO**, **Dr. GRPO** and other fixes to GRPO; open RL libraries such as **veRL** and **OpenRLHF** mature; and attention moves towards **agentic RL**, where models learn multi-step tool use inside environments.
- **2026.** The focus continues to shift towards RL at larger scale, longer horizons, richer environments, and better systems for fast rollouts. The core building blocks you will learn in this book are the same ones these systems are made of.

> **Tip.**
> In interviews, explain each method as an answer to a problem with the one before it. PPO-RLHF was expensive and unstable, so DPO removed the RL loop. DPO uses fixed offline data, so online and iterative methods were added. PPO needs a value network as large as the policy, so GRPO replaced it with a group baseline. GRPO had length and difficulty biases, so Dr. GRPO and DAPO fixed them. This "problem, then fix" story shows real understanding.

## 1.6 How the rest of Part I fits in

Everything later in the book rests on six foundations:

- **Probability** (Chapter 2): sequences of tokens are probability distributions, and RL needs expectations, sampling and the log-derivative trick.
- **Information theory** (Chapter 3): the training loss is cross-entropy, and almost every post-training objective has a KL term.
- **Optimisation** (Chapter 4): Adam, learning-rate schedules, clipping and batch sizes decide whether a run works.
- **The transformer** (Chapter 5): you need to know what is inside the model to reason about memory, speed and what LoRA changes.

- **Tokenisation** (Chapter 6): many real post-training bugs are tokenisation and template bugs.
- **Sampling** (Chapter 7): RL learns from the model's own samples, so the sampling settings shape what the model learns.

## 1.7 Two published recipes, stage by stage

Interviewers like concrete pipelines. Two open reports are worth knowing in detail.

**DeepSeek-R1** (*DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*, arXiv 2501.12948). Starting from the DeepSeek-V3 base model:

1. **R1-Zero**: RL with GRPO directly on the base model, with rule-based rewards only (answer accuracy and format). Reasoning ability and response length grow during training, but the outputs mix languages and are hard to read.
2. **Cold start**: collect *thousands* of long chain-of-thought examples written in a readable, conversational style and fine-tune the base model on them.
3. **Reasoning RL**: GRPO on reasoning tasks with rule-based rewards plus a **language-consistency reward** (the proportion of target-language words in the chain of thought).
4. **Rejection sampling and SFT**: sample from the RL checkpoint, keep correct and readable responses (about 600k reasoning samples) and add about 200k non-reasoning samples (writing, factual QA, translation) reusing DeepSeek-V3's pipeline, about 800k in total; fine-tune on them.
5. **RL for all scenarios**: a second RL stage mixing rule-based rewards for reasoning with reward models for general helpfulness and harmlessness.
6. **Distillation**: the same 800k samples are used to fine-tune smaller Qwen and Llama models. The report notes that distilling from the large model works better for small models than running the large-scale RL on them directly.

**Tulu 3** (*Tulu 3: Pushing Frontiers in Open Language Model Post-Training*, arXiv 2411.15124) is a fully open recipe built on Llama 3.1 base models, with three stages: **SFT** on a curated mixture (general chat, maths, code, safety, instruction following), **DPO** on preference data that includes on-policy pairs generated from the SFT model, and **RLVR** (the paper's name for RL with verifiable rewards) on tasks with checkable answers, such as maths and constraint-following instructions. It also publishes its decontamination procedure and evaluation suite, which makes it a good model for how to report a pipeline.

The two share the modern shape: SFT for format and a starting point, preference optimisation or reward models for general quality, RL with verifiable rewards for skills that can be checked, and repeated rounds in which the best model generates the data for the next.

## 1.8 Where the compute goes: a back-of-the-envelope estimate

Training costs about $6N$ FLOPs per token for a model with $N$ parameters ($2N$ forward, $4N$ backward; Chapter 40); generating costs about $2N$ per token. That is enough to compare stages.

**Notation on this page:** $N$: number of examples

```
N = 8e9                                                    # an 8B model
pretrain = 6 * N * 15e12                                   # 15 trillion tokens
sft = 6 * N * 2e9 * 3                                      # 2B tokens, 3 epochs
dpo = (6 + 2) * N * 2 * 100e3 * 1e3                        # 100k pairs, 1k tokens each;
 ↪  policy train + reference forward
steps, prompts, G, L = 1000, 512, 8, 2000                  # RL: steps, prompts/step,
 ↪  samples/prompt, tokens/sample
tokens = steps * prompts * G * L
rl = tokens * N * (2 + 6 + 2 + 2)                          # generate + train +
 ↪  old-logprob pass + reference pass
for name, f in [("pretraining", pretrain), ("SFT", sft), ("DPO", dpo), ("RL", rl)]:
    print(f"{name:12s} {f:9.2e} FLOPs = {100 * f / pretrain:8.4f}% of pretraining")
print(f"RL generates {tokens:.1e} tokens; at ~2N FLOPs/token generation alone is {100 *
 ↪  2 * N * tokens / rl:.0f}% "
      "of RL FLOPs, but a larger share of wall-clock time (decoding is memory-bound)")
```

Post-training is a small fraction of pre-training compute for this setting (well under 1% here), which is why it is where most teams can afford many experiments. Two caveats make the real picture different: RL generation runs far below peak FLOPs because decoding is memory-bound (Chapter 43), so its wall-clock share is much larger than its FLOP share; and frontier reasoning models now spend far more on RL than this example, with reports of RL compute becoming a significant fraction of the total.

**Why this, not that? Why distil a large RL-trained model into small ones, not run RL on each small model?**

**Reason.** The large model discovers good reasoning behaviour once; its outputs carry it to many small models through cheap SFT. The DeepSeek-R1 report found distillation gave small models better results than large-scale RL on them, at much lower cost. **Trade-off.** Distilled students inherit the teacher's errors and style, can only be as good as the teacher on the distilled tasks, and need the teacher's tokenizer for token-level methods; sequence-level distillation also needs large-scale teacher generation. **When the alternative is better.** When no stronger teacher exists for the task, when the target behaviour depends on the student's own environment (agents in a specific product), or after distillation as a final RL stage on top, which often adds more.

## 1.9 How to read a post-training paper in ten minutes

A checklist for the "tell me about a recent paper" question, and for deciding whether a result is worth reproducing:

1. **Base model and starting checkpoint.** Results on one model family often do not transfer (Chapter 26's spurious-rewards result held for Qwen but not Llama).
2. **Data.** Source, size, how prompts were filtered, and whether evaluation sets were decontaminated.
3. **Compute and steps.** Same budget for baselines? Gains at 100 steps may vanish at 1,000.
4. **Evaluation.** Benchmarks, number of samples per problem, sampling settings (temperature,

maximum length), confidence intervals or several seeds.
5. **Baselines.** Is the comparison against a well-tuned baseline, or against defaults?
6. **Ablations.** Which parts of the method matter? Methods with five changes often owe most of the gain to one.
7. **What it would take to believe it.** The one experiment you would run to test the main claim.

## 1.10 Common mistakes

- **Saying "RLHF means PPO".** RLHF is learning from human feedback; PPO is one optimiser for it. DPO, RLOO and GRPO are also used with preference rewards.
- **Saying "DPO has no KL constraint".** DPO optimises the same KL-regularised objective; $\beta$ is its KL strength.
- **Treating the stages as fixed.** Real pipelines repeat SFT and RL rounds and mix data between stages.
- **Quoting a benchmark gain without the base model, sampling settings and number of seeds.**

**Interview check**

**Q1. Walk me through how a modern chat model is made.** Pretraining on trillions of tokens gives a base model. Mid-training improves quality on maths, code and long context. SFT on prompt-response pairs teaches format and instruction following. Preference optimisation (RLHF with a reward model and PPO, or DPO-style methods) teaches what people prefer. RL with verifiable rewards trains reasoning and agentic skills. Distillation moves skills into smaller models. Safety data and evaluations run through every stage.

**Q2. Why can't we just do more SFT instead of RL?** SFT can only copy the responses in the dataset. It never sees the model's own mistakes, and it gives no signal about which of two good-looking answers is better. RL trains on the model's own outputs and rewards outcomes, so the model can find solutions that are not in any dataset and learn to recover from its own errors. RL also only needs a way to *check* answers, which is often much easier than writing perfect answers.

**Q3. What is the difference between a base model and an instruct model?** A base model has only been trained to predict the next token on raw text. An instruct model has been post-trained (SFT and usually preference optimisation or RL) to follow instructions in a chat format.

**Q4. What problem did DPO solve, and what problem did GRPO solve?** DPO removed the need for a separate reward model and an RL loop when learning from preference pairs. GRPO removed the need for a value network in PPO by using the average reward of a group of samples for the same prompt as the baseline.

**Q5. Describe the DeepSeek-R1 pipeline.** R1-Zero showed RL from a base model with rule rewards works but reads poorly; R1 adds a cold-start SFT on thousands of readable long CoT examples, reasoning RL with a language-consistency reward, rejection sampling into about 800k SFT samples (600k reasoning, 200k other), SFT, a second RL stage for all scenarios, and distillation into small models.

**Q6. Roughly what share of compute does post-training use for a typical open 8B model?**

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay)

Well under 1% of pre-training FLOPs for SFT, DPO and a modest RL run; RL's wall-clock share is larger because generation is memory-bound, and frontier reasoning models spend much more on RL.
**Q7. What do you check first when reading a post-training paper?** The base model, the data and decontamination, the compute budget compared with baselines, the evaluation settings and seeds, and the ablations.

**Why this, not that? Why learn from comparisons, not from scores or more demonstrations?**

**Why.** People are much more consistent when they say "A is better than B" than when they give a score out of 10. Comparing is also easier than writing a perfect answer. Comparisons tell the model what to avoid, which demonstrations never do.
**Trade-off.** A comparison gives only about one bit of information. You need many pairs, and the labels are noisy (annotators often disagree 20 to 30% of the time). The pair also says nothing about how much better A is.
**When the alternative wins.** Demonstrations (SFT) win when you need a new format or skill the model cannot yet produce at all. Absolute scores or checkable rewards (unit tests, exact answers) win when a correct answer can be verified automatically; then RL with verifiable rewards is cheaper and more precise than collecting preferences.

**Key takeaways**

- A base model predicts text. Post-training turns it into a useful, safe assistant.
- The usual stages are: pretraining, mid-training, SFT, preference optimisation, RL, distillation, with safety and evaluation throughout.
- RL with verifiable rewards is the main driver of recent gains in reasoning and agentic skills.
- Learn each method as a fix for a problem in the method before it.

CHAPTER 2

# Probability and Statistics You Actually Need

> **Why this chapter matters.**
> A language model *is* a probability distribution over text. Every training objective in this book is an expectation, a log-probability or a ratio of probabilities. Reinforcement learning adds sampling, estimators and the log-derivative trick. If this chapter feels easy, skim it. If any part feels shaky, slow down here: everything later depends on it.

## 2.1 Random variables and distributions

A **random variable** is a quantity whose value depends on chance. We write it with a capital letter, such as $X$, and a value it can take with a small letter, such as $x$.

- A **discrete** random variable takes values from a countable set. The next token of a language model is discrete: it is one of the $V$ tokens in the vocabulary.
- A **continuous** random variable takes values on a range of real numbers. A reward from a reward model is continuous.

A **probability distribution** tells us how likely each value is.

For a discrete variable we use a **probability mass function (PMF)**, $p(x) = P(X = x)$. It has two rules:

$$p(x) \geq 0 \quad \text{for every } x, \qquad \sum_x p(x) = 1.$$

For a continuous variable we use a **probability density function (PDF)**, also written $p(x)$. Now the probability of a range is an area under the curve:

$$P(a \leq X \leq b) = \int_a^b p(x)\,dx, \qquad \int_{-\infty}^{\infty} p(x)\,dx = 1.$$

A density can be larger than 1 at a point. Only areas are probabilities.

We write $X \sim p$ to say "$X$ is drawn from $p$". Drawing a value is called **sampling**.

## 2.2 Expectation, variance and covariance

### 2.2.1 Expectation

The **expectation** (or expected value, or mean) of a function $f(X)$ is its average value, weighted by probability:

$$\mathbb{E}_{X\sim p}[f(X)] = \sum_x p(x) f(x) \quad \text{(discrete)}, \qquad \mathbb{E}_{X\sim p}[f(X)] = \int p(x) f(x)\,dx \quad \text{(continuous)}.$$

**Notation on this page:** $x$: a value taken by a random variable • $V$: vocabulary size • $p$: a probability distribution (PMF or PDF) • $\mathbb{E}$: expected value (average over the distribution in the subscript)

In RL, the central quantity is the **expected reward**:

$$J(\theta) = \mathbb{E}_{y\sim\pi_\theta(\cdot|x)}[r(x,y)].$$

In words: if the model with weights $\theta$ writes many responses $y$ to prompt $x$, what is their average reward? Training tries to make $J(\theta)$ large.

**Linearity of expectation.** For any constants $a, b$ and any random variables $X, Y$ (even if they depend on each other):

$$\mathbb{E}[aX + bY] = a\,\mathbb{E}[X] + b\,\mathbb{E}[Y].$$

This is the most used property in the whole book. It lets us split a loss over tokens into a sum of per-token terms.

### 2.2.2 Variance

The **variance** measures how spread out a variable is around its mean $\mu = \mathbb{E}[X]$:

$$\mathrm{Var}(X) = \mathbb{E}\big[(X-\mu)^2\big] = \mathbb{E}[X^2] - \big(\mathbb{E}[X]\big)^2.$$

The second form comes from expanding the square: $\mathbb{E}[X^2 - 2\mu X + \mu^2] = \mathbb{E}[X^2] - 2\mu^2 + \mu^2$.

The **standard deviation** is $\sqrt{\mathrm{Var}(X)}$. It has the same units as $X$.

Two rules you will use:

$$\mathrm{Var}(aX+b) = a^2\,\mathrm{Var}(X), \qquad \mathrm{Var}(X+Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\,\mathrm{Cov}(X,Y).$$

### 2.2.3 Covariance

The **covariance** measures whether two variables move together:

$$\mathrm{Cov}(X,Y) = \mathbb{E}\big[(X-\mathbb{E}X)(Y-\mathbb{E}Y)\big] = \mathbb{E}[XY] - \mathbb{E}[X]\,\mathbb{E}[Y].$$

If $X$ and $Y$ are **independent** (knowing one tells you nothing about the other), then $\mathrm{Cov}(X,Y) = 0$. The reverse is not always true.

> **Why variance matters so much in RL.**
> In RL we estimate gradients from a few samples. The estimate is random. If its variance is high, training is noisy and slow, or it diverges. Much of Part III (baselines, advantages, GAE) and Part VI (group baselines in GRPO) is about *reducing the variance* of gradient estimates without changing their mean.

## 2.3 Joint, marginal and conditional probability

The **joint** distribution $p(x,y)$ gives the probability that $X = x$ *and* $Y = y$.

The **marginal** distribution of $X$ comes from summing out $Y$:

$$p(x) = \sum_y p(x,y).$$

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $J(\theta)$: objective to maximise (expected reward) • $x$: a value taken by a random variable • $y$: an outcome or sequence (a response in LLM context) • $r_A$: hidden score (strength) of item $A$ in Bradley-Terry • $\mu$: the mean $\mathbb{E}[X]$ • $p$: a probability distribution (PMF or PDF)

The **conditional** distribution of $Y$ given $X = x$ is

$$p(y \mid x) = \frac{p(x, y)}{p(x)}.$$

Read $p(y \mid x)$ as "the probability of $y$ given that we know $x$". A chat model is a conditional distribution: $\pi_\theta(y \mid x)$ is the probability of response $y$ given prompt $x$.

**The product rule.** Rearranging the definition: $p(x, y) = p(x)\, p(y \mid x)$.

**The chain rule of probability.** Applying the product rule again and again to $n$ variables:

$$p(x_1, x_2, \ldots, x_n) = p(x_1)\, p(x_2 \mid x_1)\, p(x_3 \mid x_1, x_2) \cdots p(x_n \mid x_1, \ldots, x_{n-1}) = \prod_{t=1}^{n} p(x_t \mid x_{<t}).$$

This one line is the reason language models work the way they do. See the section on autoregressive models below.

**Bayes' rule.** Writing $p(x, y)$ in two ways and dividing:

$$p(y \mid x) = \frac{p(x \mid y)\, p(y)}{p(x)} = \frac{p(x \mid y)\, p(y)}{\sum_{y'} p(x \mid y')\, p(y')}.$$

Here $p(y)$ is the **prior** (belief before seeing $x$), $p(x \mid y)$ is the **likelihood**, and $p(y \mid x)$ is the **posterior** (belief after seeing $x$).

**Worked example.**
A test flags buggy code. 1% of programs have the bug. The test catches 90% of buggy programs. It also wrongly flags 5% of clean programs. A program is flagged. What is the chance it really has the bug?

$$P(\text{bug} \mid \text{flag}) = \frac{0.9 \times 0.01}{0.9 \times 0.01 + 0.05 \times 0.99} = \frac{0.009}{0.0585} \approx 0.154.$$

Only about 15%. The bug is rare, so most flags are false alarms. The same effect appears with reward models and verifiers: a "verifier says correct" signal on a hard task with a weak verifier is often wrong. The base rate matters.

## 2.4 Common distributions

**Bernoulli.** One trial with two outcomes. $X \in \{0, 1\}$ with $P(X = 1) = p$. Mean $p$, variance $p(1 - p)$. A pass/fail reward from a unit test is a Bernoulli variable. Note that the variance is largest at $p = 0.5$ and zero at $p = 0$ or $p = 1$. This fact comes back in GRPO: prompts that the model always solves, or never solves, give no learning signal.

**Categorical.** One trial with $K$ outcomes, with probabilities $p_1, \ldots, p_K$ that sum to 1. The next token of a language model is categorical over the vocabulary.

**Gaussian (normal).** The bell curve, with mean $\mu$ and variance $\sigma^2$:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right).$$

Gaussians appear in weight initialisation, in noise, and in continuous-action RL (for example, a robot arm).

**Notation on this page:** $x$: a value taken by a random variable • $y$: an outcome or sequence (a response in LLM context) • $p$: a probability distribution (PMF or PDF) • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $n$: number of Monte Carlo samples • $t$: token position • $\mu$: the mean $\mathbb{E}[X]$ • $\sigma(z)$: logistic function; $\sigma^2$: a variance

### 2.4.1 The softmax gives a categorical distribution

A neural network outputs a vector of real numbers $z = (z_1, \dots, z_K)$, called **logits**. They can be any size and any sign. The **softmax** turns them into a categorical distribution:

$$p_i = \text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}.$$

Every $p_i$ is positive and they sum to 1.

**Shift invariance.** Adding the same constant $c$ to every logit does not change the softmax:

$$\frac{e^{z_i+c}}{\sum_j e^{z_j+c}} = \frac{e^c e^{z_i}}{e^c \sum_j e^{z_j}} = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

Only *differences* between logits matter. This is used for numerical stability (subtract the largest logit before exponentiating), and it has the same shape as the Bradley-Terry model later in this chapter.

**Worked example.**
Logits $z = (2.0,\ 1.0,\ 0.1)$. Then $e^z \approx (7.389,\ 2.718,\ 1.105)$, which sum to 11.212. So

$$p \approx (0.659,\ 0.242,\ 0.099).$$

### 2.4.2 The gradient of log-softmax

This small result is used everywhere: in the cross-entropy loss, in policy gradients and in DPO. Take $\log p_i = z_i - \log \sum_j e^{z_j}$. Differentiate with respect to logit $z_k$:

$$\frac{\partial \log p_i}{\partial z_k} = \mathbb{1}[i = k] - \frac{e^{z_k}}{\sum_j e^{z_j}} = \mathbb{1}[i = k] - p_k.$$

Here $\mathbb{1}[i = k]$ is 1 if $i = k$ and 0 otherwise. In vector form, with $e_i$ the **one-hot** vector for token $i$:

$$\nabla_z \log p_i = e_i - p.$$

**Intuition.** To make token $i$ more likely, push its logit up by $(1 - p_i)$ and push every other logit $k$ down by $p_k$. Tokens that already have high probability get pushed down the most. The cross-entropy gradient for a target token $i$ is just the negative of this: $p - e_i$.

## 2.5 Autoregressive language models

Let a response be a sequence of tokens $y = (y_1, y_2, \dots, y_T)$, written after a prompt $x$. By the chain rule of probability,

$$\pi_\theta(y \mid x) = \prod_{t=1}^{T} \pi_\theta(y_t \mid x, y_{<t}).$$

A model that generates one token at a time, each conditioned on everything before it, is called **autoregressive**. A transformer language model outputs, at each position $t$, a softmax over the vocabulary: the distribution of the next token.

**Notation on this page:** $z$: logits (raw scores before softmax) • $p$: a probability distribution (PMF or PDF) • $k$: an index • $\mathbb{1}[\cdot]$: 1 if the condition is true, else 0 • $\nabla_\theta$: gradient with respect to $\theta$ • $y$: an outcome or sequence (a response in LLM context) • $T$: number of tokens in the response • $x$: a value taken by a random variable • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $y_{<t}$: the tokens before position $t$ • *and 1 more: see the Notation table*

Products of many numbers below 1 get tiny very fast, so we work with **log-probabilities**. The log turns the product into a sum:

$$\log \pi_\theta(y \mid x) = \sum_{t=1}^{T} \log \pi_\theta(y_t \mid x, y_{<t}).$$

**Worked example.**
A three-token response has per-token probabilities 0.6, 0.5 and 0.9. The sequence probability is $0.6 \times 0.5 \times 0.9 = 0.27$. The sequence log-probability is $\log 0.6 + \log 0.5 + \log 0.9 \approx -0.511 - 0.693 - 0.105 = -1.309$, and $e^{-1.309} \approx 0.27$.

**Common mistake.**
A sequence log-probability is a *sum* over tokens, so longer responses get more negative log-probabilities. Comparing raw sequence log-probabilities of responses of different lengths is biased towards short responses. Some methods divide by length (SimPO, Chapter 23); others do not (DPO, Chapter 22). Know which one you are using.

In code, you get per-token log-probabilities from the logits with `log_softmax` and `gather`. This exact pattern appears in every RL and DPO implementation:

```
import torch
import torch.nn.functional as F

torch.manual_seed(0)
batch, seq_len, vocab = 2, 5, 10
logits = torch.randn(batch, seq_len, vocab)        # model output
tokens = torch.randint(0, vocab, (batch, seq_len)) # the tokens that were generated

# Logits at position t predict token t+1, so shift by one.
logp_all = F.log_softmax(logits[:, :-1, :], dim=-1)          # (B, T-1, V)
target = tokens[:, 1:]                                       # (B, T-1)
token_logp = logp_all.gather(-1, target.unsqueeze(-1)).squeeze(-1)  # (B, T-1)

seq_logp = token_logp.sum(dim=-1)   # log pi(y|x) for each sequence
print(token_logp.shape, seq_logp)
```

The "shift by one" is the most common bug in hand-written training code. The logit at position $t$ is the model's prediction for the token at position $t + 1$.

## 2.6 Maximum likelihood estimation

Suppose we have a dataset of examples $\{y^{(1)}, \ldots, y^{(N)}\}$ and a model $p_\theta$. The **likelihood** is the probability the model gives to the whole dataset, assuming the examples are independent:

$$\mathcal{L}(\theta) = \prod_{n=1}^{N} p_\theta(y^{(n)}).$$

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $y_{<t}$: the tokens before position $t$ • $x$: a value taken by a random variable • $y$: an outcome or sequence (a response in LLM context) • $t$: token position • $T$: number of tokens in the response • $N$: number of training examples • $p$: a probability distribution (PMF or PDF) • $\mathcal{L}$: a loss to minimise • $n$: number of Monte Carlo samples

**Maximum likelihood estimation (MLE)** picks the $\theta$ that makes the data most likely. Taking logs (which does not move the maximum, because log is increasing) and flipping the sign gives the **negative log-likelihood (NLL)** loss, which we *minimise*:

$$\mathcal{L}_{\text{NLL}}(\theta) = -\frac{1}{N}\sum_{n=1}^{N} \log p_\theta(y^{(n)}).$$

For a language model, $\log p_\theta(y)$ is a sum of per-token log-probabilities, so the NLL is the average over tokens of $-\log p_\theta(\text{correct next token})$. This is exactly the pretraining loss and the SFT loss. Chapter 3 shows that it is also the **cross-entropy** between the data and the model.

> **MLE is mode covering.**
> The NLL punishes the model heavily (with $-\log p \to \infty$) whenever it gives near-zero probability to something that appears in the data. So an MLE-trained model spreads probability over *everything* in the data, including rare and low-quality text. This is one reason pretraining alone gives a model that can produce both good and bad answers, and why later stages that *sharpen* the distribution (preference optimisation and RL) help.

## 2.7 Monte Carlo estimation

Most expectations in this book cannot be computed exactly. The expected reward $\mathbb{E}_{y\sim\pi_\theta}[r(x,y)]$ is a sum over every possible response, and there are more possible responses than atoms in the universe. So we **estimate** it from samples. This is called **Monte Carlo estimation**:

$$\mathbb{E}_{y\sim p}[f(y)] \approx \hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} f(y_i), \qquad y_i \sim p.$$

An **estimator** is a rule that turns samples into a guess. Two properties matter:

- **Bias**: $\mathbb{E}[\hat{\mu}] - \mu$. An estimator is **unbiased** if its expected value equals the true value. The Monte Carlo average is unbiased, by linearity of expectation.
- **Variance**: how much the estimate jumps around between different sets of samples. For independent samples,
$$\text{Var}(\hat{\mu}_n) = \frac{\text{Var}(f(y))}{n}.$$
The **standard error** is $\sqrt{\text{Var}(f)/n}$. To halve the error you need four times as many samples.

The **bias-variance trade-off** appears all over RL. A slightly biased estimator with much lower variance often trains better than an unbiased but very noisy one. GAE (Chapter 16) is a dial between the two.

> **Worked example.**
> A model solves a maths problem 30% of the time, so each attempt is Bernoulli with $p = 0.3$ and variance $0.3\times0.7 = 0.21$. With $n = 8$ samples (a typical GRPO group size), the standard error of the estimated success rate is $\sqrt{0.21/8} \approx 0.16$. So a measured rate of 0.3 could easily be 0.14 or 0.46. Small groups give noisy baselines.

**Notation on this page:** $\theta$: the model's trainable weights • $\mathcal{L}$: a loss to minimise • $y$: an outcome or sequence (a response in LLM context) • $N$: number of training examples • $p$: a probability distribution (PMF or PDF) • $n$: number of Monte Carlo samples • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $x$: a value taken by a random variable • $r_A$: hidden score (strength) of item $A$ in Bradley-Terry • $\mu$: the mean $\mathbb{E}[X]$

## 2.8 Importance sampling

Sometimes we have samples from one distribution $q$ but want an expectation under another distribution $p$. **Importance sampling** fixes this by reweighting each sample:

$$\mathbb{E}_{y\sim p}[f(y)] = \sum_y p(y)f(y) = \sum_y q(y)\,\frac{p(y)}{q(y)}f(y) = \mathbb{E}_{y\sim q}\left[\frac{p(y)}{q(y)}f(y)\right].$$

The factor $w(y) = p(y)/q(y)$ is the **importance weight** (or **importance ratio**). This works whenever $q(y) > 0$ wherever $p(y)f(y) \neq 0$.

**Where you will see it.** In PPO and GRPO, responses are sampled from the *old* policy $\pi_{\theta_{\text{old}}}$, but we want to improve the *current* policy $\pi_\theta$. The importance ratio

$$\rho = \frac{\pi_\theta(y_t \mid \cdot)}{\pi_{\theta_{\text{old}}}(y_t \mid \cdot)}$$

corrects for the difference. It is computed in log space for stability: `ratio = torch.exp(logp_new - logp_old)`.

**The danger.** If $p$ and $q$ are very different, a few weights become huge and the estimate's variance explodes. For a whole sequence, the ratio is a *product* of per-token ratios, so even small per-token differences multiply up over thousands of tokens. This is why PPO **clips** the ratio (Chapter 17), and why "train-inference mismatch" between the rollout engine and the trainer is a real problem (Chapter 43).

> **Worked example.**
> $q$ gives token A probability 0.5 and token B probability 0.5. $p$ gives A 0.9 and B 0.1. Weights: $w(A) = 0.9/0.5 = 1.8$ and $w(B) = 0.1/0.5 = 0.2$. If $f(A) = 1$ and $f(B) = 0$, the true value is $\mathbb{E}_p[f] = 0.9$. Under $q$: $\mathbb{E}_q[wf] = 0.5 \times 1.8 \times 1 + 0.5 \times 0.2 \times 0 = 0.9$. Correct.

## 2.9 The log-derivative trick

This is the single most important identity for RL. It is also called the **score function trick** or the **REINFORCE trick**.

**The problem.** We want the gradient of an expectation where the *distribution* depends on $\theta$:

$$\nabla_\theta \mathbb{E}_{y\sim p_\theta}[f(y)].$$

We cannot just sample $y$ and differentiate $f(y)$, because $f$ (for example, a reward from a unit test) may not depend on $\theta$ at all, and may not even be differentiable. The $\theta$-dependence is hidden inside *which* $y$ gets sampled.

**Notation on this page:** $q$: a second distribution (sampling or proposal) • $p$: a probability distribution (PMF or PDF) • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $y$: an outcome or sequence (a response in LLM context) • $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\theta$: the model's trainable weights • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\rho$: importance ratio $\pi_\theta/\pi_{\theta_{\text{old}}}$ for a token • $t$: token position • $\nabla_\theta$: gradient with respect to $\theta$

**The trick.** From calculus, $\nabla_\theta \log p_\theta(y) = \nabla_\theta p_\theta(y)/p_\theta(y)$. So $\nabla_\theta p_\theta(y) = p_\theta(y)\,\nabla_\theta \log p_\theta(y)$. Now:

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{y\sim p_\theta}[f(y)] &= \nabla_\theta \sum_y p_\theta(y) f(y) \\
&= \sum_y \nabla_\theta p_\theta(y) f(y) && (f \text{ does not depend on } \theta) \\
&= \sum_y p_\theta(y)\,\nabla_\theta \log p_\theta(y) f(y) && (\text{the trick}) \\
&= \mathbb{E}_{y\sim p_\theta}[f(y)\,\nabla_\theta \log p_\theta(y)].
\end{aligned}$$

The gradient of an expectation has become an expectation of something we can compute. Now we can estimate it with Monte Carlo: sample $y_1, \ldots, y_n$ from the model, and average $f(y_i)\,\nabla_\theta \log p_\theta(y_i)$.

**Why this, not that? Why the score-function trick, not the reparameterisation trick?**

**Why.** The reparameterisation trick (used in VAEs) writes a sample as a differentiable function of the parameters plus noise, for example $x = \mu + \sigma\epsilon$, and back-propagates through the sample. Tokens are discrete, so there is no smooth path from $\theta$ to the sampled token. Rewards such as "the unit tests pass" are not differentiable either. The score-function estimator needs neither: it only needs $\nabla_\theta \log p_\theta(y)$, which we always have.
**Trade-off.** Score-function estimates have much higher variance than reparameterised ones, so they need baselines, many samples and small learning rates.
**When the alternative wins.** For continuous actions with a differentiable reward or critic (for example, robot control with SAC or DDPG), reparameterisation gives far lower-variance gradients. Relaxations such as Gumbel-softmax exist for discrete choices, but they are biased and rarely used for LLM RL.

**Intuition.** $\nabla_\theta \log p_\theta(y)$ is the direction in weight space that makes $y$ more likely. We move in that direction, scaled by how good $y$ was. Good samples get pushed up. Bad samples (if $f$ is negative) get pushed down. This is the core of every policy-gradient method: REINFORCE, PPO, GRPO and RLOO.

### 2.9.1 The score has zero mean

The quantity $\nabla_\theta \log p_\theta(y)$ is called the **score**. Its expectation is zero:

$$\mathbb{E}_{y\sim p_\theta}[\nabla_\theta \log p_\theta(y)] = \sum_y p_\theta(y)\,\frac{\nabla_\theta p_\theta(y)}{p_\theta(y)} = \nabla_\theta \sum_y p_\theta(y) = \nabla_\theta 1 = 0.$$

**Consequence: baselines.** For any constant $b$ that does not depend on $y$,

$$\mathbb{E}[(f(y) - b)\,\nabla_\theta \log p_\theta(y)] = \mathbb{E}[f(y)\,\nabla_\theta \log p_\theta(y)] - b \cdot 0.$$

Subtracting a **baseline** $b$ does not change the expected gradient. But it can greatly reduce its variance. If every reward is around 10, then without a baseline every sample gets pushed up a lot, and the useful signal (which samples are *better than usual*) is buried in noise. With $b \approx 10$, only the differences remain. Choosing $b$ as the average reward of a group of samples for the same prompt is exactly what GRPO and RLOO do (Chapters 27 and 28).

**Notation on this page:** $\theta$: the model's trainable weights • $\nabla_\theta$: gradient with respect to $\theta$ • $y$: an outcome or sequence (a response in LLM context) • $p$: a probability distribution (PMF or PDF) • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $n$: number of Monte Carlo samples • $\sigma(z)$: logistic function; $\sigma^2$: a variance • $\mu$: the mean $\mathbb{E}[X]$ • $x$: a value taken by a random variable

**Why this, not that? Why subtract a baseline, not just use more samples?**

**Why.** Variance falls only as $1/n$ with more samples, and each LLM sample can cost thousands of tokens of generation. A good baseline can cut variance by a large factor for almost no cost, because it removes the part of the reward that is the same for every sample.
**Trade-off.** The baseline must not depend on the sample it is applied to, or the gradient becomes biased. A learned baseline (a value network) costs memory and can be wrong; a sample-average baseline needs several samples per prompt.
**When the alternative wins.** If rewards are already centred around zero and the batch is large, a baseline adds little. In the extreme of one sample per prompt, you cannot use a group mean at all, and a learned value function or a running average across prompts is the only option.

The code below checks the trick numerically for a three-way categorical. The exact gradient (from autograd on the exact expectation) matches the Monte Carlo estimate from the trick.

```
import torch

torch.manual_seed(0)
theta = torch.tensor([0.5, -0.2, 0.1], requires_grad=True)  # logits
f = torch.tensor([1.0, 3.0, -2.0])                            # reward per outcome

# Exact gradient of E[f] via autograd
p = torch.softmax(theta, dim=0)
exact = torch.autograd.grad((p * f).sum(), theta)[0]

# Score-function (REINFORCE) estimate from samples
n = 200_000
with torch.no_grad():
    probs = torch.softmax(theta, 0)
    ys = torch.multinomial(probs, n, replacement=True)
    onehot = torch.nn.functional.one_hot(ys, 3).float()
    score = onehot - probs           # grad of log p(y) w.r.t. logits = e_y - p
    baseline = f[ys].mean()          # a baseline lowers variance, same mean
    est = ((f[ys] - baseline).unsqueeze(1) * score).mean(0)

print("exact   ", exact)
print("estimate", est)
```

## 2.10 The logistic function and the Bradley-Terry model

The **logistic** or **sigmoid** function squashes any real number into $(0, 1)$:

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

Useful facts: $\sigma(0) = 0.5$, $\sigma(-z) = 1 - \sigma(z)$, and $\sigma'(z) = \sigma(z)\,(1 - \sigma(z))$. Also, $\log \sigma(z) = -\log(1 + e^{-z})$, which is called the **log-sigmoid**.

**Notation on this page:** $n$: number of Monte Carlo samples • $\sigma(z)$: logistic function; $\sigma^2$: a variance • $z$: logits (raw scores before softmax)

The **Bradley-Terry model** says how likely item A is to beat item B when each has a hidden score (a "strength") $r_A$ and $r_B$:

$$P(A \succ B) = \frac{e^{r_A}}{e^{r_A} + e^{r_B}} = \frac{1}{1 + e^{-(r_A - r_B)}} = \sigma(r_A - r_B).$$

The symbol $A \succ B$ means "A is preferred to B". Notice that this is just a softmax over two items. Like the softmax, only the *difference* in scores matters.

**Worked example.**
If $r_A = 1.5$ and $r_B = 0.5$, then $P(A \succ B) = \sigma(1.0) \approx 0.73$. If $r_A = 2$ and $r_B = -1$, then $P(A \succ B) = \sigma(3) \approx 0.95$.

Fitting the scores by maximum likelihood on a set of human comparisons gives the loss

$$\mathcal{L} = -\log \sigma(r_{\text{chosen}} - r_{\text{rejected}}).$$

This is the **reward model loss** (Chapter 19). In DPO (Chapter 22), the same loss appears with the score replaced by a log-probability ratio. So you will see this equation many times.

**Why this, not that? Why the Bradley-Terry model, not regression on scores?**

**Why.** Preference data only contains comparisons, and Bradley-Terry turns a comparison into a clean likelihood: $P(A \succ B) = \sigma(r_A - r_B)$. It is invariant to adding a constant to all scores, which matches the data (comparisons say nothing about absolute level).
**Trade-off.** It assumes preferences are transitive and that a single number captures quality. It cannot express ties or "both are bad" without extensions, and its scores have no fixed scale, so they must be normalised before use in RL.
**When the alternative wins.** If you have reliable absolute labels (a rubric score, a correctness check, a measured metric), plain regression or classification is simpler and keeps the scale. Pairwise models such as Plackett-Luce (for rankings of more than two) or generative judges are alternatives when the data comes as rankings or written critiques.

## 2.11 Numerical stability: working in log space

Computers store numbers with limited precision. Two problems show up all the time:

- **Overflow**: $e^{1000}$ is too big to store, so it becomes `inf`.
- **Underflow**: the product of many small probabilities becomes exactly 0.

The fix is to work with logs and use stable formulas.

**Log-sum-exp.** We often need $\log \sum_i e^{z_i}$. Subtract the maximum $m = \max_i z_i$ first:

$$\log \sum_i e^{z_i} = m + \log \sum_i e^{z_i - m}.$$

Now every exponent is at most 0, so nothing overflows, and at least one term equals 1, so the sum is never 0.

**Notation on this page:** $r_A$: hidden score (strength) of item $A$ in Bradley-Terry • $\sigma(z)$: logistic function; $\sigma^2$: a variance • $a \succ b$: $a$ is preferred to $b$ • $\mathcal{L}$: a loss to minimise • $z$: logits (raw scores before softmax)

**Worked example.**
For $z = (1000, 1001, 1002)$, the naive formula overflows. The stable one gives $1002 + \log(e^{-2} + e^{-1} + 1) \approx 1002 + 0.408 = 1002.408$.

**In PyTorch**, always prefer the fused, stable functions:

| Do this | Not this |
|---|---|
| `F.log_softmax(z, -1)` | `torch.log(torch.softmax(z, -1))` |
| `F.cross_entropy(logits, target)` | `-torch.log(probs[target])` |
| `F.logsigmoid(z)` | `torch.log(torch.sigmoid(z))` |
| `torch.logsumexp(z, -1)` | `torch.log(torch.exp(z).sum(-1))` |
| `torch.exp(logp_new - logp_old)` | `p_new / p_old` |

**Common mistake.**
Computing logits and softmax in low precision (for example, bfloat16) loses accuracy in the log-probabilities. In RL this matters because the importance ratio $\exp(\log \pi_\theta - \log \pi_{\text{old}})$ should be exactly 1 when the two policies are the same. Small numerical errors make it drift from 1. Many RL codebases compute the final logits and log-softmax in float32 for this reason.

**Interview check**

**Q1. Why do we work with log-probabilities instead of probabilities?** Sequence probabilities are products of many numbers below 1 and underflow to zero. Logs turn products into sums, which are stable and easy to differentiate. Log-space also lets us use stable functions like log-softmax and log-sum-exp.
**Q2. Derive the log-derivative trick.** $\nabla_\theta \mathbb{E}_{p_\theta}[f] = \sum_y \nabla_\theta p_\theta(y) f(y) = \sum_y p_\theta(y) \nabla_\theta \log p_\theta(y) f(y) = \mathbb{E}_{p_\theta}[f(y) \nabla_\theta \log p_\theta(y)]$, using $\nabla p = p \nabla \log p$.
**Q3. Why does subtracting a baseline not bias the policy gradient?** Because the score has zero mean: $\mathbb{E}[\nabla_\theta \log p_\theta(y)] = \nabla_\theta \sum_y p_\theta(y) = 0$. So $\mathbb{E}[b \nabla_\theta \log p_\theta(y)] = 0$ for any $b$ that does not depend on the sampled $y$.
**Q4. What is the gradient of $\log \text{softmax}(z)_i$ with respect to $z$?** $e_i - \text{softmax}(z)$: one-hot minus the probability vector.
**Q5. What is importance sampling and where does it appear in LLM RL?** Estimating an expectation under $p$ with samples from $q$ by weighting each sample by $p(y)/q(y)$. In PPO and GRPO the samples come from the old policy, and the ratio $\pi_\theta / \pi_{\theta_{\text{old}}}$ corrects for the policy having changed since sampling.
**Q6. Why is importance sampling dangerous for long sequences?** The sequence-level ratio is a product of per-token ratios, so its variance grows very quickly with length. A few samples get huge weights. Methods clip the ratio, use per-token ratios, or keep the policies close.
**Q7. Show that the Bradley-Terry model is a two-item softmax.** $P(A \succ B) = e^{r_A} / (e^{r_A} + e^{r_B})$. Divide top and bottom by $e^{r_A}$ to get $1/(1 + e^{-(r_A - r_B)}) = \sigma(r_A - r_B)$.

**Notation on this page:** $z$: logits (raw scores before softmax) • $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\nabla_\theta$: gradient with respect to $\theta$ • $y$: an outcome or sequence (a response in LLM context) • $p$: a probability distribution (PMF or PDF) • $q$: a second distribution (sampling or proposal) • $a \succ b$: $a$ is preferred to $b$ • $r_A$: hidden score (strength) of item $A$ in Bradley-Terry • *and 1 more: see the Notation table*

**Q8. A verifier is right 95% of the time on correct answers and wrongly passes 10% of wrong answers. The model is correct 20% of the time. What is P(correct | pass)?** $0.95 \times 0.2/(0.95 \times 0.2 + 0.1 \times 0.8) = 0.19/0.27 \approx 0.70$.

**Key takeaways**

- A language model is an autoregressive conditional distribution: $\log \pi(y \mid x) = \sum_t \log \pi(y_t \mid x, y_{<t})$.
- MLE equals minimising the token-level NLL, which is the pretraining and SFT loss.
- Monte Carlo estimates are unbiased; their variance shrinks as $1/n$.
- Importance sampling reweights samples by $p/q$; it is the basis of PPO's ratio and becomes unstable when $p$ and $q$ differ.
- The log-derivative trick, $\nabla \mathbb{E}_p[f] = \mathbb{E}_p[f \nabla \log p]$, is the foundation of policy gradients. Baselines reduce variance without bias.
- Bradley-Terry is $\sigma(r_A - r_B)$. It is the base of reward models and DPO.
- Use log-softmax, log-sigmoid and log-sum-exp. Keep log-probabilities in float32 for RL.

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $y_{<t}$: the tokens before position $t$ • $x$: a value taken by a random variable • $y$: an outcome or sequence (a response in LLM context) • $t$: token position • $n$: number of Monte Carlo samples • $p$: a probability distribution (PMF or PDF) • $q$: a second distribution (sampling or proposal) • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\nabla_\theta$: gradient with respect to $\theta$ • $\sigma(z)$: logistic function; $\sigma^2$: a variance • *and 1 more: see the Notation table*

CHAPTER 3

# Information Theory

> **Why this chapter matters.**
> The pretraining loss is a cross-entropy. The RLHF objective has a KL penalty. DPO is derived from a KL-regularised objective. GRPO adds a KL term estimated with a special estimator. Entropy is the number you watch to see if an RL run is collapsing. This chapter gives you all of these tools, with proofs.

## 3.1 Surprise and entropy

Suppose an event has probability $p$. How *surprised* should we be when it happens? We want a measure that:

- is 0 when $p = 1$ (no surprise at a certain event),
- grows as $p$ gets smaller,
- adds up for independent events (the surprise of two independent events is the sum of their surprises).

The only function with these properties (up to a constant factor) is

$$\text{surprise}(p) = -\log p = \log \frac{1}{p}.$$

With the natural log, the unit is the **nat**. With $\log_2$, the unit is the **bit**. One bit is the surprise of a fair coin landing heads.

The **entropy** of a distribution is its *expected* surprise:

$$H(p) = \mathbb{E}_{x\sim p}[-\log p(x)] = -\sum_x p(x) \log p(x).$$

(We treat $0 \log 0$ as 0.) Entropy measures *uncertainty*. It is 0 when one outcome has probability 1. For $K$ outcomes it is largest, $\log K$, when all outcomes are equally likely.

> **Worked example.**
> - Fair coin: $H = -2 \times 0.5 \log_2 0.5 = 1$ bit $= 0.693$ nats.
> - Biased coin with $p = (0.9, 0.1)$: $H \approx 0.469$ bits. Less uncertain.
> - Uniform over 4 outcomes: $H = \log_2 4 = 2$ bits.
> - A language model with a vocabulary of 50,000 tokens that is completely unsure: $H = \log 50{,}000 \approx 10.8$ nats per token.

**Entropy as code length.** Shannon showed that, to send messages drawn from $p$, the best possible average code length is $H(p)$ bits per message (using base 2). Common messages get short codes,

**Notation on this page:** $p$: the true or target distribution • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $H$: entropy (uncertainty) of a distribution • $x$: an outcome (a value of the random variable)

rare ones get long codes. This is the link between language modelling and compression: a model that predicts text well can compress it well.

## 3.2 Cross-entropy

Now suppose the true distribution is $p$, but we use a model $q$ to predict. The **cross-entropy** of $q$ relative to $p$ is the expected surprise *under the model*, averaged over what *really* happens:

$$H(p, q) = \mathbb{E}_{x \sim p}[-\log q(x)] = -\sum_x p(x) \log q(x).$$

In coding terms: if you build your code for $q$ but the messages come from $p$, your average code length is $H(p, q)$.

### 3.2.1 Why cross-entropy is the language-model loss

In language modelling, $p$ is the true data distribution of text. We never know it, but we have samples from it: our training data. For one position, the "true" next token is the one that appears in the data. So the empirical $p$ is a one-hot vector on that token, and the cross-entropy becomes

$$H(p_{\text{one-hot}}, q_\theta) = -\log q_\theta(\text{correct token}).$$

Averaged over all positions in the data, this is exactly the negative log-likelihood from Chapter 2. So:

Minimising **cross-entropy** to the data = minimising **negative log-likelihood** = **maximum likelihood estimation**. For language models, this is the next-token loss used in both pretraining and SFT.

**Why this, not that? Why cross-entropy, not mean squared error?**

**Why.** Cross-entropy is the log-likelihood of a categorical distribution, so minimising it is maximum likelihood. Its gradient with respect to the logits is simply $p - e_{\text{target}}$: large when the model is confidently wrong, small when it is right. MSE on probabilities has gradients that vanish when the softmax saturates, so a confidently wrong model learns slowly.
**Trade-off.** Cross-entropy heavily punishes near-zero probability on observed tokens, which makes the model spread mass over everything in the data (mode covering) and makes it sensitive to label noise.
**When the alternative wins.** MSE is the right loss for real-valued targets: value functions in RL, reward regression, and distillation of continuous scores. For classification with very noisy labels, label smoothing or robust losses can beat plain cross-entropy.

In PyTorch, `F.cross_entropy(logits, targets)` takes raw logits (not probabilities), applies log-softmax inside, and averages $-\log q(\text{target})$ over positions. Targets equal to `ignore_index` (default $-100$) are skipped. This is how SFT masks out prompt tokens (Chapter 9).

```
import torch
import torch.nn.functional as F
```

**Notation on this page:** $p$: the true or target distribution • $q$: the model or approximating distribution • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $H$: entropy (uncertainty) of a distribution • $x$: an outcome (a value of the random variable) • $\theta$: the model's trainable weights

```
logits = torch.tensor([[2.0, 1.0, 0.1],
                       [0.5, 2.5, 0.3]])
targets = torch.tensor([0, -100])   # second position is masked out

loss = F.cross_entropy(logits, targets, ignore_index=-100)
manual = -F.log_softmax(logits[0], -1)[0]
print(loss.item(), manual.item())   # both about 0.417
```

## 3.3 KL divergence

The **Kullback-Leibler (KL) divergence** measures how different a distribution $q$ is from a reference $p$:

$$D_{\mathrm{KL}}(p\,\|\,q) = \sum_x p(x)\log\frac{p(x)}{q(x)} = \mathbb{E}_{x\sim p}\left[\log\frac{p(x)}{q(x)}\right].$$

Expanding the log:

$$D_{\mathrm{KL}}(p\,\|\,q) = \underbrace{-\sum_x p(x)\log q(x)}_{\uparrow\, H(p,q)} \underbrace{-\left(-\sum_x p(x)\log p(x)\right)}_{\uparrow\, H(p)} = H(p,q) - H(p).$$

So KL is the *extra* code length you pay for using $q$ instead of the true $p$. Because $H(p)$ does not depend on the model, **minimising cross-entropy over $q$ is the same as minimising $D_{\mathrm{KL}}(p\,\|\,q)$.**

**Worked example.**
$p = (0.7, 0.2, 0.1)$ and $q = (0.5, 0.3, 0.2)$.

- $H(p) \approx 0.802$ nats.
- $H(p,q) = -(0.7\log 0.5 + 0.2\log 0.3 + 0.1\log 0.2) \approx 0.887$ nats.
- $D_{\mathrm{KL}}(p\,\|\,q) = 0.887 - 0.802 = 0.085$ nats.
- The other direction, $D_{\mathrm{KL}}(q\,\|\,p) \approx 0.092$ nats. Different. KL is not symmetric.

### 3.3.1 KL is never negative

**Claim.** $D_{\mathrm{KL}}(p\,\|\,q) \geq 0$, with equality if and only if $p = q$.

**Proof.** We use **Jensen's inequality**: for a concave function $g$ (one that curves downwards, like log), $\mathbb{E}[g(X)] \leq g(\mathbb{E}[X])$. Then

$$-D_{\mathrm{KL}}(p\,\|\,q) = \sum_x p(x)\log\frac{q(x)}{p(x)} = \mathbb{E}_p\left[\log\frac{q(x)}{p(x)}\right] \leq \log\mathbb{E}_p\left[\frac{q(x)}{p(x)}\right] = \log\sum_{x:p(x)>0} q(x) \leq \log 1 = 0.$$

Equality in Jensen's inequality needs $q(x)/p(x)$ to be the same constant everywhere, which with both summing to 1 means $p = q$. ■

A quicker proof uses the inequality $\log u \leq u - 1$ (the log curve lies below its tangent line at $u = 1$). Set $u = q(x)/p(x)$, multiply by $p(x)$ and sum. This same inequality is behind the k3 estimator later in this chapter.

**Notation on this page:** $q$: the model or approximating distribution • $p$: the true or target distribution • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $x$: an outcome (a value of the random variable) • $H$: entropy (uncertainty) of a distribution • $g$: a gradient

### 3.3.2 Properties to remember

- $D_{\mathrm{KL}}(p \,\|\, q) \geq 0$, and $= 0$ only when $p = q$.
- **Not symmetric**: $D_{\mathrm{KL}}(p \,\|\, q) \neq D_{\mathrm{KL}}(q \,\|\, p)$ in general. So it is not a true distance.
- **Infinite** if $q(x) = 0$ somewhere that $p(x) > 0$.
- **Chain rule for sequences.** For autoregressive models,

$$D_{\mathrm{KL}}(\pi(\cdot \mid x) \,\|\, \pi_{\mathrm{ref}}(\cdot \mid x)) = \mathbb{E}_{y \sim \pi}\left[\sum_{t=1}^{T} \log \frac{\pi(y_t \mid x, y_{<t})}{\pi_{\mathrm{ref}}(y_t \mid x, y_{<t})}\right].$$

  The sequence KL is the expected *sum* of per-token log-ratios. This is why RLHF code computes a KL term for every token and adds them up.

## 3.4 Forward vs reverse KL

Say $p$ is a target distribution and we fit a model $q_\theta$. We can minimise KL in two directions, and they behave very differently.

**Forward KL**, $D_{\mathrm{KL}}(p \,\|\, q_\theta) = \mathbb{E}_{x \sim p}[\log p(x) - \log q_\theta(x)]$.

- The expectation is over the *target* $p$.
- Wherever $p(x) > 0$, the model pays $-\log q_\theta(x)$. If $q_\theta(x) \to 0$ there, the cost goes to infinity.
- So $q_\theta$ must put mass *everywhere* $p$ has mass. It is **mode covering** (also called "mean seeking"). If $p$ has two peaks and $q_\theta$ can only make one, $q_\theta$ spreads out to cover both, putting mass in the valley between them too.
- **This is maximum likelihood.** SFT and pretraining minimise forward KL to the data.

**Reverse KL**, $D_{\mathrm{KL}}(q_\theta \,\|\, p) = \mathbb{E}_{x \sim q_\theta}[\log q_\theta(x) - \log p(x)]$.

- The expectation is over the *model's own samples*.
- Wherever $q_\theta$ puts mass but $p$ does not, the cost is huge. Where $p$ has mass but $q_\theta$ does not, there is *no* cost, because those points are never sampled.
- So $q_\theta$ prefers to sit inside one peak of $p$ and ignore the others. It is **mode seeking** (also "zero forcing").
- **This is what RLHF uses.** The KL penalty is $D_{\mathrm{KL}}(\pi_\theta \,\|\, \pi_{\mathrm{ref}})$, with the expectation over the policy's own samples, because those are the samples we have during RL.

**Why this, not that? Why reverse KL in RLHF, not forward KL?**

**Why.** Reverse KL, $D_{\mathrm{KL}}(\pi_\theta \| \pi_{\mathrm{ref}})$, is an expectation over the policy's own samples, which are exactly what RL produces, so it is cheap to estimate. It also has the behaviour we want: it forbids text the reference would never write, but lets the policy drop the reference's bad modes. Finally, KL-regularised reward maximisation with reverse KL has a clean closed-form optimum, $\pi^* \propto \pi_{\mathrm{ref}} e^{r/\beta}$, which DPO is built on (Chapter 22).
**Trade-off.** Reverse KL is mode seeking, so it reduces diversity and can collapse onto a few safe answers.
**When the alternative wins.** Forward KL is natural in distillation and SFT, where samples come from the teacher or the data and you want the student to cover all of them. Generalised

**Notation on this page:** $D_{\mathrm{KL}}(p \| q)$: KL divergence of $q$ from $p$ • $p$: the true or target distribution • $q$: the model or approximating distribution • $x$: an outcome (a value of the random variable) • $\pi_{\mathrm{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $y_{<t}$: the tokens before position $t$ • $y$: the response (output tokens); $y_t$ is token $t$ • $t$: time step or token position • $T$: number of tokens • *and 4 more: see the Notation table*

KD (Chapter 12) and f-divergence variants of DPO mix the two when diversity matters.

**Why the direction matters for post-training.**
Reverse KL lets the policy drop some behaviours of the reference model (for example, low-quality answers) without penalty, as long as what it *does* produce is plausible under the reference. That is what we want: sharpen the distribution towards good answers, but do not drift into text the reference model would never write. The flip side is that reverse-KL training tends to reduce diversity. This is a real effect in RL-tuned models: they often have lower output diversity and lower pass@k at very large $k$ than their base models (Chapter 26).

```
import torch

# Target p: two peaks at -2 and +2. Model q: a single Gaussian N(mu, s^2).
x = torch.linspace(-6, 6, 1201); dx = x[1] - x[0]
def normal(x, mu, s):
    return torch.exp(-0.5 * ((x - mu) / s) ** 2) / (s * (2 * torch.pi) ** 0.5)
p = 0.5 * normal(x, -2.0, 0.6) + 0.5 * normal(x, 2.0, 0.6)

def fit(direction, steps=2000):
    mu = torch.tensor(1.0, requires_grad=True)       # start a bit to the right
    log_s = torch.tensor(-0.7, requires_grad=True)   # start fairly narrow
    opt = torch.optim.Adam([mu, log_s], lr=0.03)
    for _ in range(steps):
        q = normal(x, mu, log_s.exp()) + 1e-12
        if direction == "forward":   # KL(p || q): expectation under p
            loss = (p * (p.log() - q.log())).sum() * dx
        else:                        # KL(q || p): expectation under q
            loss = (q * (q.log() - (p + 1e-12).log())).sum() * dx
        opt.zero_grad(); loss.backward(); opt.step()
    return round(mu.item(), 2), round(log_s.exp().item(), 2)

print("forward KL (mode covering):", fit("forward"))  # about (0.0, 2.09): wide, sits in
 ↪ the valley
print("reverse KL (mode seeking): ", fit("reverse"))  # about (2.0, 0.6): locks onto one
 ↪ peak
```

Which peak the reverse-KL fit picks depends on where it starts. If you start it very wide, it can even get stuck in the valley, because reverse KL has local optima. Forward KL has no such choice to make: it must cover both peaks, so it ends up wide and centred between them.

## 3.5 Estimating KL from samples

In RLHF and GRPO we need $D_{\mathrm{KL}}(\pi_\theta \,\|\, \pi_{\mathrm{ref}})$. We cannot sum over all responses, so we estimate it from responses sampled from $\pi_\theta$. For one sampled token, define the ratio

$$r = \frac{\pi_{\mathrm{ref}}(y_t \mid \cdot)}{\pi_\theta(y_t \mid \cdot)}.$$

**Notation on this page:** $\pi_{\mathrm{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $y$: the response (output tokens); $y_t$ is token $t$ • $r$: the ratio $\pi_{\mathrm{ref}}/\pi_\theta$ for a sampled token • $t$: time step or token position

Note $\mathbb{E}_{y_t\sim\pi_\theta}[r] = \sum_{y_t}\pi_{\text{ref}}(y_t \mid \cdot) = 1$. The true KL is $\mathbb{E}_{\pi_\theta}[-\log r]$. Three estimators are in common use. They come from John Schulman's short note *Approximating KL Divergence* (2020, on his blog at joschu.net), and the names k1, k2, k3 come from there.

| Name | Formula | Unbiased? | Always $\geq 0$? | Variance |
|---|---|---|---|---|
| k1 | $-\log r = \log\pi_\theta - \log\pi_{\text{ref}}$ | Yes | No | High |
| k2 | $\frac{1}{2}(\log r)^2$ | No (small bias when policies are close) | Yes | Low |
| k3 | $(r-1) - \log r$ | Yes | Yes | Low |

**Why k3 is unbiased.** $\mathbb{E}[r-1] = 1-1 = 0$, so $\mathbb{E}[k3] = \mathbb{E}[-\log r] = D_{\text{KL}}$. Adding $r-1$ is adding a quantity with zero mean (a **control variate**) that is negatively correlated with $-\log r$. It reduces variance without adding bias.

**Why k3 is never negative.** From $\log r \leq r-1$ (the tangent line again), $(r-1) - \log r \geq 0$.

**Why k2 is nearly unbiased.** For $r$ close to 1, write $\log r = u$. Then $\mathbb{E}[k1] = \mathbb{E}[-u]$ and a Taylor expansion shows that $\mathbb{E}[-u] \approx \mathbb{E}[u^2]/2$ when $\mathbb{E}[e^u] = 1$. So k2 matches the KL to second order.

**Where they are used.**

- **Classic PPO-RLHF** (InstructGPT style) uses k1 per token, folded into the *reward*: $r_t = -\beta(\log\pi_\theta(y_t) - \log\pi_{\text{ref}}(y_t))$, plus the reward model score at the last token.
- **GRPO** (DeepSeekMath) uses k3 per token, added directly to the *loss*:

$$\text{KL}_t \approx \frac{\pi_{\text{ref}}(y_t \mid \cdot)}{\pi_\theta(y_t \mid \cdot)} - \log\frac{\pi_{\text{ref}}(y_t \mid \cdot)}{\pi_\theta(y_t \mid \cdot)} - 1.$$

### 3.5.1 A subtle point: estimating KL vs estimating its gradient

An estimator that gives the right *value* does not automatically give the right *gradient* when you put it in the loss and call `backward()`. Let $\ell = \log\pi_\theta(y_t) - \log\pi_{\text{ref}}(y_t)$ with $y_t$ sampled from $\pi_\theta$ and then held fixed. Differentiating each estimator with respect to $\theta$:

- **k1 in the loss:** $\nabla\ell = \nabla\log\pi_\theta(y_t)$. Its expectation is the expected score, which is **zero** (Chapter 2). So k1 as a loss gives no useful gradient on average. That is why PPO-RLHF puts k1 in the *reward*, where the policy gradient turns it into $\ell\cdot\nabla\log\pi_\theta$, the correct reverse-KL gradient.
- **k2 in the loss:** $\nabla\frac{1}{2}\ell^2 = \ell\nabla\log\pi_\theta(y_t)$. In expectation this is exactly the gradient of the reverse KL $D_{\text{KL}}(\pi_\theta\,\|\,\pi_{\text{ref}})$.
- **k3 in the loss:** $\nabla(e^{-\ell} - 1 + \ell) = (1-r)\nabla\log\pi_\theta(y_t)$. In expectation, $\mathbb{E}[(1-r)\nabla\log\pi_\theta] = -\sum_y \pi_{\text{ref}}(y)\nabla\log\pi_\theta(y)$, which is the gradient of the **forward** KL $D_{\text{KL}}(\pi_{\text{ref}}\,\|\,\pi_\theta)$.

So "k3 in the loss", as in the original GRPO, actually pushes on the forward KL, not the reverse KL it estimates. This was pointed out in detail by Yunhao Tang and Rémi Munos in *On a few pitfalls in KL divergence gradient estimation for RL* (arXiv 2506.09477, 2025). Their summary: differentiating an unbiased KL *estimate* does not, in general, give an unbiased KL *gradient*. The estimate depends

**Notation on this page:** $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $y$: the response (output tokens); $y_t$ is token $t$ • $r$: the ratio $\pi_{\text{ref}}/\pi_\theta$ for a sampled token • $t$: time step or token position • $D_{\text{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\nabla_\theta$: gradient with respect to $\theta$

on $\theta$ in two ways, through the samples and through the formula, and back-propagating through the formula alone misses the first part.

> **Watch the naming.**
> Tang and Munos call $D_{\mathrm{KL}}(\pi_{\mathrm{ref}}\|\pi_\theta)$ the "reverse" KL, because it is reversed relative to the target $D_{\mathrm{KL}}(\pi_\theta\|\pi_{\mathrm{ref}})$. This book follows the usual RLHF naming, where $D_{\mathrm{KL}}(\pi_\theta\|\pi_{\mathrm{ref}})$ is the reverse (mode-seeking) KL and $D_{\mathrm{KL}}(\pi_{\mathrm{ref}}\|\pi_\theta)$ is the forward (mode-covering) KL. The maths is the same; only the labels differ.

Three more facts complete the picture:

- **The k3 gradient weight is unbounded.** It is $(1-\pi_{\mathrm{ref}}/\pi_\theta)$. When the policy gives a sampled token much lower probability than the reference, the weight becomes very large, and the update strongly pushes that token up. The DeepSeek-V3.2 report (*DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models*, arXiv 2512.02556) names this as a cause of noisy, unstable updates.
- **The fix used by DeepSeek-V3.2** multiplies k3 by the importance ratio $\pi_\theta/\pi_{\theta_{\mathrm{old}}}$ before back-propagating. On-policy, the ratio's value is 1, but its gradient is not zero. Working it through: $\nabla(\rho\cdot k3) = \rho\, k3\,\nabla\log\pi_\theta + \rho\,(1-r)\,\nabla\log\pi_\theta$, and with $\rho=1$ this is $(-\log r)\,\nabla\log\pi_\theta$, which in expectation is exactly the reverse-KL gradient. TRL's `GRPOConfig` exposes this as `use_bias_correction_kl` (default `True` in TRL 1.16.0, used whenever `beta > 0`).
- **Token-level KL losses only give part of the sequence-level gradient.** Changing token $t$ also changes which tokens come after it, and so changes the later per-token KL terms. A per-token loss ignores that effect. Tang and Munos show how to estimate the full sequence-level gradient; most open-source code still uses the token-level version, which works acceptably when the KL is small.

These are good interview points. In practice, the KL coefficient in reasoning RL is now often small (DeepSeek-R1 used 0.001) or zero (DAPO and many later recipes drop it), which makes the issue less important there. It matters more when the KL term does real work, as in RLHF with a learned reward model or in on-policy distillation. The code below checks all four claims numerically.

```
import torch

torch.manual_seed(0)
V = 6
theta = torch.randn(V, requires_grad=True)   # policy logits
ref = torch.softmax(torch.randn(V), 0)       # fixed reference

def grads_by_estimator(n=400_000):
    pi = torch.softmax(theta, 0)
    ys = torch.multinomial(pi.detach(), n, replacement=True)
    logp = torch.log_softmax(theta, 0)[ys]
    ell = logp - ref.log()[ys]
    out = {}
```

**Notation on this page:** $\theta$: the model's trainable weights • $\pi_{\mathrm{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $\pi_{\theta_{\mathrm{old}}}$: the policy that generated the current samples • $\nabla_\theta$: gradient with respect to $\theta$ • $\rho$: importance ratio $\pi_\theta/\pi_{\theta_{\mathrm{old}}}$ for a token • $r$: the ratio $\pi_{\mathrm{ref}}/\pi_\theta$ for a sampled token • $t$: time step or token position

```
    for name, est in {"k1": ell, "k2": 0.5 * ell**2, "k3": torch.exp(-ell) - 1 +
     ↪  ell}.items():
        out[name] = torch.autograd.grad(est.mean(), theta, retain_graph=True)[0]
    return out

pi = torch.softmax(theta, 0)
rev = torch.autograd.grad((pi * (pi.log() - ref.log())).sum(), theta)[0]
pi = torch.softmax(theta, 0)
fwd = torch.autograd.grad((ref * (ref.log() - pi.log())).sum(), theta)[0]

g = grads_by_estimator()

# DeepSeek-V3.2 fix: multiply k3 by the importance ratio pi_theta / pi_old (value 1
 ↪  on-policy, gradient not 0)
pi = torch.softmax(theta, 0)
ys = torch.multinomial(pi.detach(), 400_000, replacement=True)
logp = torch.log_softmax(theta, 0)[ys]
ell = logp - ref.log()[ys]
ratio = torch.exp(logp - logp.detach())
g_fix = torch.autograd.grad((ratio * (torch.exp(-ell) - 1 + ell)).mean(), theta)[0]

print("k1 grad (about 0):        ", g["k1"].norm().item())
print("k2 grad - reverse KL grad:", (g["k2"] - rev).norm().item())
print("k3 grad - forward KL grad:", (g["k3"] - fwd).norm().item())
print("k3 grad - reverse KL grad:", (g["k3"] - rev).norm().item(), "(not small)")
print("ratio*k3 - reverse KL:    ", (g_fix - rev).norm().item(), "(fixed)")
```

## 3.6 Perplexity and bits per byte

**Perplexity** is the exponential of the average per-token cross-entropy (in nats):

$$\mathrm{PPL} = \exp\left(-\frac{1}{T}\sum_{t=1}^{T}\log q_\theta(y_t \mid y_{<t})\right).$$

Intuition: a perplexity of $k$ means the model is, on average, as unsure as if it were choosing uniformly between $k$ tokens. A loss of 2.0 nats per token gives a perplexity of $e^2 \approx 7.39$.

**Common mistake.**
Perplexity depends on the tokenizer. A tokenizer with bigger tokens has fewer tokens per text, so each token carries more information and per-token loss is higher. You cannot compare perplexities across models with different tokenizers. Use **bits per byte** (or bits per character) instead: total loss over the text in bits, divided by the number of UTF-8 bytes. This is tokenizer independent.

**Why this, not that? Why bits per byte, not perplexity?**

**Why.** Bits per byte divides the total loss by the length of the raw text, so it does not depend on how the text was cut into tokens. Models with different tokenizers become comparable.

**Notation on this page:** $\theta$: the model's trainable weights • $y_{<t}$: the tokens before position $t$ • $y$: the response (output tokens); $y_t$ is token $t$ • $t$: time step or token position • $T$: number of tokens • $q$: the model or approximating distribution

**Trade-off.** It is less familiar, and per-token losses are what training code reports, so you need an extra conversion step.
**When the alternative wins.** Within one model family and one tokenizer, perplexity is fine and easier to read. For downstream quality, neither replaces task evaluations.

$$\text{bits per byte} = \frac{\sum_t -\log_2 q_\theta(y_t \mid y_{<t})}{\text{number of bytes in the text}}.$$

## 3.7 Mutual information and Jensen-Shannon divergence

These appear less often in post-training, but come up in interviews.

**Mutual information** between $X$ and $Y$ measures how much knowing one tells you about the other:

$$I(X;Y) = D_{\mathrm{KL}}(p(x,y) \,\|\, p(x)\,p(y)) = H(X) - H(X \mid Y).$$

It is 0 exactly when $X$ and $Y$ are independent. It is symmetric.

**Jensen-Shannon (JS) divergence** is a symmetric, bounded version of KL. Let $m = \frac{1}{2}(p+q)$. Then

$$\mathrm{JS}(p,q) = \tfrac{1}{2} D_{\mathrm{KL}}(p \,\|\, m) + \tfrac{1}{2} D_{\mathrm{KL}}(q \,\|\, m).$$

It is always between 0 and $\log 2$. Its square root is a true distance. The original GAN objective is closely related to JS divergence, and generalised knowledge distillation (Chapter 12) uses a JS-style family that blends forward and reverse KL.

## 3.8 Entropy of a policy and entropy collapse

For an LLM policy, the **token entropy** at position $t$ is

$$H_t = -\sum_{v \in \text{vocab}} \pi_\theta(v \mid x, y_{<t}) \log \pi_\theta(v \mid x, y_{<t}).$$

RL libraries log the average $H_t$ over all generated tokens. It is one of the most useful health signals of an RL run.

- **High entropy** means the policy is still exploring: it tries different tokens.
- **Low entropy** means the policy is confident and nearly deterministic.

**Entropy collapse** is when entropy drops quickly towards zero early in RL. The model stops exploring. All samples for a prompt become almost the same, so group-based methods like GRPO get identical rewards in each group, the advantages become zero, and learning stalls. Common causes and fixes:

- The clip range in PPO/GRPO is symmetric, which makes it easy to lower the probability of rare tokens but hard to raise them. DAPO's **clip-higher** (a larger upper clip, Chapter 28) addresses this.

**Notation on this page:** $\theta$: the model's trainable weights • $y_{<t}$: the tokens before position $t$ • $y$: the response (output tokens); $y_t$ is token $t$ • $t$: time step or token position • $q$: the model or approximating distribution • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $H$: entropy (uncertainty) of a distribution • $x$: an outcome (a value of the random variable) • $p$: the true or target distribution • $\pi_\theta$: the policy (the model), with weights $\theta$

- An **entropy bonus** adds $+\alpha H$ to the objective (Chapter 17). It is used in classic PPO, but in LLM RL it is often unstable and must be tuned with care.
- Sampling temperature in rollouts affects how much exploration happens (Chapter 7).

> **Tip.**
> Always plot token entropy during RL, next to reward and response length. A steady, slow decline in entropy is normal. A sharp drop in the first few hundred steps, together with a flat reward, is a warning sign.

Computing entropy from logits, stably:

```
import torch
import torch.nn.functional as F

def entropy_from_logits(logits):
    # logits: (..., V). H = logsumexp(z) - sum(p * z)
    logp = F.log_softmax(logits.float(), dim=-1)
    return -(logp.exp() * logp).sum(-1)

z = torch.tensor([[2.0, 1.0, 0.1], [0.0, 0.0, 0.0]])
print(entropy_from_logits(z))   # second row is log(3) = 1.0986
```

**Interview check**

**Q1. Show that minimising cross-entropy is equivalent to minimising KL divergence.** $H(p,q) = H(p) + D_{\mathrm{KL}}(p\,\|\,q)$, and $H(p)$ does not depend on $q$. So both have the same minimiser over $q$.
**Q2. Prove KL is non-negative.** By Jensen: $-D_{\mathrm{KL}}(p\|q) = \mathbb{E}_p[\log(q/p)] \le \log \mathbb{E}_p[q/p] \le \log 1 = 0$. Or use $\log u \le u - 1$ with $u = q/p$.
**Q3. Forward vs reverse KL: which is mode covering, and which does RLHF use?** Forward KL $D_{\mathrm{KL}}(p\|q)$ (as in MLE) is mode covering. Reverse KL $D_{\mathrm{KL}}(q\|p)$ is mode seeking. RLHF penalises the reverse KL $D_{\mathrm{KL}}(\pi_\theta\|\pi_{\mathrm{ref}})$ because the expectation is over the policy's own samples.
**Q4. What is the k3 estimator and why is it used?** $r - 1 - \log r$ with $r = \pi_{\mathrm{ref}}/\pi_\theta$ on samples from $\pi_\theta$. It is unbiased for $D_{\mathrm{KL}}(\pi_\theta\|\pi_{\mathrm{ref}})$, always non-negative, and has lower variance than $-\log r$. GRPO uses it.
**Q5. If you put k1 directly in the loss and backprop, what gradient do you get?** In expectation, zero: the gradient is the score $\nabla \log \pi_\theta(y)$, whose mean is zero. k1 should go in the reward (so the policy gradient multiplies it by the score), or use k2 in the loss.
**Q5b. And if you put k3 in the loss, as the original GRPO does?** In expectation you get the gradient of the forward KL $D_{\mathrm{KL}}(\pi_{\mathrm{ref}}\|\pi_\theta)$, with an unbounded weight on tokens the policy has made rare (Tang and Munos, arXiv 2506.09477). Multiplying k3 by the importance ratio $\pi_\theta/\pi_{\theta_{\mathrm{old}}}$ (DeepSeek-V3.2) restores the reverse-KL gradient.
**Q6. Why can't you compare perplexity across models with different tokenizers?** Per-token loss depends on how much text a token covers. Use bits per byte instead.
**Q7. What is entropy collapse in RL fine-tuning and how do you detect it?** The policy's

**Notation on this page:** $\alpha$: a weighting coefficient (defined in the text) • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $H$: entropy (uncertainty) of a distribution • $p$: the true or target distribution • $q$: the model or approximating distribution • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\pi_{\mathrm{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $r$: the ratio $\pi_{\mathrm{ref}}/\pi_\theta$ for a sampled token • $\nabla_\theta$: gradient with respect to $\theta$ • *and 2 more: see the Notation table*

token entropy drops fast, outputs stop varying, group rewards become identical and advantages vanish. Detect it by logging mean token entropy; fix with clip-higher, entropy bonuses, temperature, or data with the right difficulty.

**Q8. What is the perplexity of a uniform model over a vocabulary of size V?** Exactly $V$. The cross-entropy is $\log V$, and $\exp(\log V) = V$.

**Key takeaways**

- Entropy is expected surprise. Cross-entropy is expected surprise under the model. KL is the gap: $H(p, q) - H(p)$.
- The language-modelling loss is cross-entropy = NLL = forward KL to the data. It is mode covering.
- RLHF penalises reverse KL to a reference model. It is mode seeking and reduces diversity.
- Sequence KL is the expected sum of per-token log-ratios.
- k1, k2 and k3 estimate KL from samples. k3 is unbiased, non-negative and low-variance, but as a loss its gradient matches the forward KL. Multiply it by the importance ratio, or use k2, to get the reverse-KL gradient.
- Perplexity is $e^{\text{loss}}$ and depends on the tokenizer. Bits per byte does not.
- Watch policy entropy during RL. A sudden collapse kills exploration.

**Notation on this page:** $V$: vocabulary size • $H$: entropy (uncertainty) of a distribution • $p$: the true or target distribution • $q$: the model or approximating distribution

CHAPTER 4

# Optimisation and Training Mechanics

> **Why this chapter matters.**
> Most failed fine-tuning and RL runs do not fail because of the algorithm. They fail because the learning rate is wrong, the batch is too small, the loss is normalised badly, or gradients explode. This chapter covers the optimiser and training mechanics that every post-training job uses. Interviewers love to ask you to write the Adam update from memory.

## 4.1 Loss functions and gradients

A **loss function** $\mathcal{L}(\theta)$ is a number that says how bad the model with parameters $\theta$ is on some data. Training means finding $\theta$ that makes the loss small.

The **gradient** $\nabla_\theta \mathcal{L}$ is the vector of partial derivatives $\partial \mathcal{L}/\partial \theta_i$. It points in the direction in which the loss increases fastest. So to lower the loss, we step in the *opposite* direction.

### 4.1.1 Automatic differentiation and backpropagation

A neural network is a long chain of simple operations: matrix multiplies, additions, softmax, and so on. **Automatic differentiation** (**autodiff**) computes exact gradients of such chains by applying the **chain rule** of calculus step by step.

For a chain $\theta \to h_1 \to h_2 \to \mathcal{L}$, the chain rule says

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial h_2}\,\frac{\partial h_2}{\partial h_1}\,\frac{\partial h_1}{\partial \theta}.$$

**Backpropagation** is **reverse-mode** autodiff. It first runs the network forward and saves the intermediate values (the **activations**). Then it goes backwards from the loss, multiplying local derivatives as it goes. Reverse mode is efficient when there are many inputs (billions of parameters) and one output (a scalar loss): one backward pass gives the gradient for every parameter, at a cost of roughly two forward passes.

Two practical facts follow:

- The saved activations use a lot of memory. This is why long sequences are expensive, and why **gradient checkpointing** (saving fewer activations and recomputing them in the backward pass) exists (Chapter 41).
- A forward plus backward pass costs about **3 times** a forward pass alone (1 forward + about 2 backward). This gives the "6ND" rule for training FLOPs in Chapter 5.

In PyTorch, any tensor with `requires_grad=True` records operations. Calling `loss.backward()` fills each parameter's `.grad`. `torch.no_grad()` turns recording off, which is what you want for rollouts and reference-model forward passes in RL.

## 4.2 Gradient descent and SGD

**Gradient descent** updates the parameters with a step size $\eta$, called the **learning rate**:

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta \mathcal{L}(\theta_k).$$

Computing the gradient on the whole dataset is far too expensive. **Stochastic gradient descent (SGD)** uses a small random **mini-batch** of $B$ examples instead:

$$g_k = \frac{1}{B} \sum_{i \in \text{batch}} \nabla_\theta \ell_i(\theta_k), \qquad \theta_{k+1} = \theta_k - \eta\, g_k.$$

The mini-batch gradient is an unbiased Monte Carlo estimate of the full gradient (Chapter 2). Its variance shrinks as $1/B$. A bigger batch gives a less noisy gradient, but costs more compute per step.

## 4.3 Momentum, RMSProp and Adam

Plain SGD has two problems. It zig-zags in narrow valleys, and it uses the same step size for every parameter, even though some parameters get huge gradients and others tiny ones. The optimisers below fix these.

### 4.3.1 Momentum

Keep a running average of past gradients, and step along that average:

$$m_k = \mu\, m_{k-1} + g_k, \qquad \theta_{k+1} = \theta_k - \eta\, m_k.$$

With $\mu \approx 0.9$, noise in random directions cancels out, while consistent directions build up speed. Think of a heavy ball rolling down a hill.

### 4.3.2 RMSProp

Keep a running average of the *squared* gradient per parameter, and divide by its square root:

$$v_k = \beta_2\, v_{k-1} + (1 - \beta_2)\, g_k^2, \qquad \theta_{k+1} = \theta_k - \eta\, \frac{g_k}{\sqrt{v_k} + \epsilon}.$$

(The square and the division are element-wise.) Parameters with big gradients get smaller steps, and parameters with small gradients get bigger steps. Each parameter gets its own effective learning rate.

**Notation on this page:** $\eta$: learning rate • $\theta$: the model's trainable weights • $\nabla_\theta$: gradient with respect to $\theta$ • $\mathcal{L}$: a loss to minimise • $k$: optimiser step number • $B$: mini-batch size • $g_k$: the (mini-batch) gradient at step $k$ • $\mu$: momentum coefficient • $\beta_1, \beta_2$: Adam decay rates for the first and second moments • $\epsilon$: small constant that avoids division by zero

### 4.3.3 Adam

**Adam** combines momentum (the first moment, $m$) and RMSProp (the second moment, $v$), and corrects a start-up bias. For each step $k = 1, 2, \ldots$:

$$
\begin{aligned}
m_k &= \beta_1 m_{k-1} + (1-\beta_1)\, g_k \\
v_k &= \beta_2 v_{k-1} + (1-\beta_2)\, g_k^2 \\
\hat{m}_k &= \frac{m_k}{1-\beta_1^k}, \qquad \hat{v}_k = \frac{v_k}{1-\beta_2^k} \\
\theta_{k+1} &= \theta_k - \eta\, \frac{\hat{m}_k}{\sqrt{\hat{v}_k} + \epsilon}
\end{aligned}
$$

with $m_0 = v_0 = 0$.

**Why the bias correction?** Both averages start at 0, so early on they are too small. Unroll $m_k$:

$$
m_k = (1-\beta_1) \sum_{j=1}^{k} \beta_1^{k-j}\, g_j .
$$

If all $g_j$ had the same mean $\bar{g}$, then $\mathbb{E}[m_k] = (1-\beta_1)\,\bar{g} \sum_{j=1}^{k} \beta_1^{k-j} = (1-\beta_1^k)\,\bar{g}$. Dividing by $(1-\beta_1^k)$ removes this factor. The same argument applies to $v_k$. Without correction, with $\beta_2 = 0.999$, $v$ would be about 1000 times too small at step 1, and the first steps would be far too large.

**Intuition.** The update $\hat{m}/\sqrt{\hat{v}}$ is roughly "average gradient divided by its typical size". It is close to $\pm 1$ when the gradient sign is consistent, and close to 0 when the gradient is noise. So in Adam, the learning rate $\eta$ is roughly the **maximum size of a parameter change per step**, whatever the scale of the gradient. This is why Adam learning rates transfer reasonably well between tasks.

**Typical values.** $\beta_1 = 0.9$. $\beta_2 = 0.999$ is the PyTorch default; LLM pretraining often uses $\beta_2 = 0.95$, which reacts faster to changes in gradient size and is more stable at large scale. $\epsilon = 10^{-8}$.

**Memory.** Adam stores $m$ and $v$ for every parameter. In float32 that is 8 bytes per parameter, on top of the weights and gradients. This is a big part of the memory maths in Chapter 40.

### 4.3.4 AdamW: decoupled weight decay

**Weight decay** shrinks weights towards zero a little every step, which acts as regularisation. In plain SGD, weight decay is the same as adding an **L2 penalty** $\frac{\lambda}{2}\|\theta\|^2$ to the loss. In Adam it is *not* the same: if you add the L2 term to the loss, its gradient $\lambda\theta$ gets divided by $\sqrt{\hat{v}}$ like everything else, so parameters with large gradients get almost no decay.

**AdamW** (Loshchilov and Hutter) **decouples** the decay from the adaptive step:

$$
\theta_{k+1} = \theta_k - \eta \left( \frac{\hat{m}_k}{\sqrt{\hat{v}_k} + \epsilon} + \lambda\, \theta_k \right).
$$

Every weight shrinks by the same fraction $\eta\lambda$ per step. AdamW is the default optimiser for LLM training and fine-tuning. Typical weight decay is 0.0 to 0.1. Biases and normalisation weights are often excluded from decay.

**Notation on this page:** $k$: optimiser step number • $\theta$: the model's trainable weights • $\beta_1, \beta_2$: Adam decay rates for the first and second moments • $\epsilon$: small constant that avoids division by zero • $\eta$: learning rate • $g_k$: the (mini-batch) gradient at step $k$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\lambda$: weight-decay strength

**Why this, not that? Why AdamW, not Adam with L2 regularisation?**

**Why.** With L2 inside Adam, the decay term is divided by $\sqrt{\hat{v}}$, so weights with big gradients barely decay and weights with tiny gradients decay a lot. AdamW applies the same relative shrinkage to every weight, which is what "weight decay" is supposed to mean, and it decouples the best weight decay from the learning rate.
**Trade-off.** AdamW adds one hyperparameter that interacts with the learning-rate schedule (the total shrinkage is roughly $\eta\lambda$ per step). Applying it to norms and biases usually hurts, so you must group parameters.
**When the alternative wins.** For short fine-tuning runs, weight decay matters little and plain Adam (or AdamW with decay 0) works the same. Plain SGD with L2 is still common for some vision models, where it can generalise better.

Here is AdamW written by hand, checked against PyTorch's version:

```
import torch

def adamw_step(p, g, m, v, k, lr=1e-3, b1=0.9, b2=0.999, eps=1e-8, wd=0.01):
    p.mul_(1 - lr * wd)                    # decoupled weight decay
    m.mul_(b1).add_(g, alpha=1 - b1)       # first moment
    v.mul_(b2).addcmul_(g, g, value=1 - b2)  # second moment
    m_hat = m / (1 - b1 ** k)
    v_hat = v / (1 - b2 ** k)
    p.add_(-lr * m_hat / (v_hat.sqrt() + eps))

torch.manual_seed(0)
w_ref = torch.randn(5, requires_grad=True)
w_my = w_ref.detach().clone()
m, v = torch.zeros(5), torch.zeros(5)
opt = torch.optim.AdamW([w_ref], lr=1e-3, weight_decay=0.01)

for k in range(1, 51):
    x = torch.randn(5)
    loss = ((w_ref * x).sum() - 1.0) ** 2
    opt.zero_grad(); loss.backward(); opt.step()
    g = 2 * ((w_my * x).sum() - 1.0) * x   # same gradient, by hand
    adamw_step(w_my, g, m, v, k)

print((w_ref.detach() - w_my).abs().max())   # 0 or tiny: identical
```

### 4.3.5 Other optimisers you may hear about

**Why this, not that? Why Adam-style optimisers, not plain SGD, for transformers?**

**Why.** Transformer gradients have very different scales across parameter types (embeddings, attention, norms, the LM head). Adam's per-parameter scaling copes with this automatically; SGD needs a learning rate small enough for the most sensitive parameters, which is then far too small for the rest.

**Notation on this page:** $\eta$: learning rate • $\lambda$: weight-decay strength

**Trade-off.** Adam stores two extra numbers per parameter (8 bytes in fp32), which is a large part of training memory. It can also be less stable at very large scale without tuning $\beta_2$ and $\epsilon$.
**When the alternative wins.** When memory is the binding constraint, Adafactor, 8-bit Adam or SGD with momentum can be acceptable. Newer optimisers such as Muon have beaten AdamW in some pretraining runs. For fine-tuning an existing model, staying with AdamW is the safe default.

- **Adafactor** stores a factored (row and column) approximation of $v$, saving memory.
- **8-bit Adam** (bitsandbytes) stores $m$ and $v$ in 8 bits. Used with QLoRA (Chapter 10).
- **Lion** uses only the sign of a momentum term.
- **Muon** orthogonalises the update for matrix-shaped parameters. It has been used successfully for pretraining some large models. For fine-tuning a model, it is usually safest to use the same optimiser family it was pretrained with.

## 4.4 Learning-rate schedules

The learning rate is usually not constant. A **schedule** changes it over training.

**Warmup.** Start from (near) zero and increase linearly over the first $W$ steps. Early on, Adam's moment estimates are based on very few gradients and are unreliable, and the model is far from a good region. Large early steps can cause loss spikes or divergence. Typical warmup: 1% to 10% of steps for fine-tuning, or a fixed number such as 100 steps.

After warmup, common shapes are:

**Why this, not that? Why warmup plus decay, not a constant learning rate?**

**Why.** Warmup protects the model while Adam's moment estimates are still unreliable. Decay at the end lets the model settle into a lower-loss region; most of the final loss improvement in a run often happens during the decay.
**Trade-off.** A schedule ties you to a total number of steps decided in advance. Stopping early leaves you with a model that never decayed; continuing past the end means re-warming.
**When the alternative wins.** Constant learning rates (after a short warmup) are common in RL fine-tuning, where data changes as the policy changes and runs are stopped by evaluation, not by a step count. WSD gives the best of both in long pretraining runs: train at a constant rate and decay only when you want a checkpoint.

- **Linear decay** to zero (or to a small value) by the end.
- **Cosine decay**: $\eta_k = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})(1 + \cos(\pi \cdot \text{progress}))$. Smooth, very common.
- **Constant** (after warmup). Common for RL fine-tuning, where the "data" changes as the policy changes and there is no fixed end.
- **WSD (warmup-stable-decay)**: warm up, hold constant for most of training, then decay quickly at the end. Popular for pretraining, because you can branch off and decay from any checkpoint.

**Notation on this page:** $\beta_1, \beta_2$: Adam decay rates for the first and second moments • $\epsilon$: small constant that avoids division by zero • $\eta$: learning rate • $k$: optimiser step number

```python
import math

def lr_at(step, total, peak=2e-5, warmup=100, kind="cosine", min_ratio=0.1):
    if step < warmup:
        return peak * (step + 1) / warmup
    progress = (step - warmup) / max(1, total - warmup)
    if kind == "linear":
        return peak * (1 - progress)
    if kind == "cosine":
        lo = peak * min_ratio
        return lo + 0.5 * (peak - lo) * (1 + math.cos(math.pi * progress))
    return peak  # constant

print([f"{lr_at(s, 1000):.2e}" for s in (0, 50, 99, 300, 600, 999)])
```

**Typical peak learning rates for post-training** (for models of roughly 1B to 70B parameters, with AdamW; always tune for your setting):

| Setting | Typical peak LR |
|---|---|
| Full SFT | $5 \times 10^{-6}$ to $2 \times 10^{-5}$ |
| LoRA SFT | $5 \times 10^{-5}$ to $3 \times 10^{-4}$ |
| Reward model training | $1 \times 10^{-6}$ to $1 \times 10^{-5}$ |
| DPO (full) | $5 \times 10^{-7}$ to $5 \times 10^{-6}$ |
| PPO / GRPO policy (full) | $5 \times 10^{-7}$ to $3 \times 10^{-6}$ |

Notice the pattern. Pretraining uses around $10^{-4}$ to $10^{-3}$. Fine-tuning a pretrained model uses much smaller rates, because we want small changes to an already good model. Preference and RL training use the smallest rates of all, because the training signal is noisy and the model can easily be pushed into strange behaviour. Larger models usually need smaller learning rates.

## 4.5 Gradient clipping

Sometimes one batch gives a huge gradient: a rare token, a bad example, or an unstable RL update. One huge step can ruin a model. **Gradient clipping by global norm** rescales the whole gradient if its norm is too large:

$$\|g\| = \sqrt{\sum_i g_i^2}, \qquad g \leftarrow g \cdot \min\left(1, \frac{c}{\|g\|}\right).$$

The direction stays the same; only the length is capped at $c$. A typical value is $c = 1.0$. In PyTorch: `torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)`, called after `backward()` and before `optimizer.step()`. It returns the norm *before* clipping, which you should log.

**Notation on this page:** $g_k$: the (mini-batch) gradient at step $k$

**Why this, not that? Why clip the global norm, not each value?**

**Why.** Clipping the global norm scales the whole update down, so its direction is unchanged. Clipping each value separately (value clipping) changes the direction, because only the large components are cut.
**Trade-off.** A single huge gradient in one layer shrinks the update for all layers. A fixed threshold also needs tuning: too low and every step is clipped, which is just a hidden lower learning rate.
**When the alternative wins.** Per-value clipping is a cheap safety net against NaN-like spikes in some RL code. Adaptive clipping (based on a running average of recent norms) works better when the typical gradient norm drifts a lot during training.

**Tip.**
The gradient norm is one of the best health signals you have. In a healthy run it settles into a band. Sudden spikes often come just before a loss spike. A steady rise can mean the learning rate is too high. If the norm is clipped on almost every step, your clip threshold or learning rate needs another look.

## 4.6 Batch size, gradient accumulation and the learning rate

The **effective batch size** (also called the **global batch size**) is the number of examples that contribute to one optimiser step:

$$B_{\text{eff}} = \text{per-device batch} \times \text{gradient accumulation steps} \times \text{number of GPUs}.$$

**Gradient accumulation** lets you reach a big effective batch on small hardware. You run several small **micro-batches**, call `backward()` on each (gradients add up in `.grad`), and only then call `optimizer.step()` and `zero_grad()`.

**Batch size and learning rate are linked.** A bigger batch gives a less noisy gradient, so you can usually take bigger steps. For SGD, a common rule is to scale the learning rate linearly with batch size. For Adam, a square-root rule is a better starting guess. Above a certain "critical batch size", bigger batches stop helping. When you change the batch size, retune the learning rate.

**Why this, not that? Why gradient accumulation, not a bigger per-device batch?**

**Why.** Accumulation reaches the same effective batch with the memory of one micro-batch, because activations are freed after each backward pass.
**Trade-off.** It is slower: micro-batches run one after another instead of in parallel, and you must normalise the loss carefully (see the trap below). Batch-norm-style layers would see the small micro-batch, but LLMs use LayerNorm or RMSNorm, so this rarely matters.
**When the alternative wins.** If the bigger batch fits in memory, use it; it is faster. With many GPUs, data parallelism grows the batch without accumulation.

### 4.6.1 The loss-normalisation trap

This is a real bug that has appeared in popular libraries, and a common interview question.

**Notation on this page:** $B$: mini-batch size

Suppose each micro-batch computes its loss as the *mean over its own tokens*, and then the micro-batch losses are averaged. If micro-batches have different numbers of real (non-padding, non-masked) tokens, the result is **not** the mean over all tokens. Tokens in short micro-batches get more weight.

**Worked example.**
Micro-batch A has 10 tokens with total loss 20 (mean 2.0). Micro-batch B has 90 tokens with total loss 90 (mean 1.0).

- True token mean: $(20 + 90)/100 = 1.1$.
- Mean of micro-batch means: $(2.0 + 1.0)/2 = 1.5$.

The gradient is biased towards the 10 tokens in A.

The fix: sum the per-token losses in each micro-batch, and divide by the total number of real tokens in the *whole* effective batch (across all micro-batches and all GPUs). The same question comes back in RL as "sequence-level vs token-level loss aggregation", which is one of the key differences between GRPO and DAPO (Chapter 28).

```
import torch

torch.manual_seed(0)
w = torch.randn(4, requires_grad=True)
micro_batches = [torch.randn(n, 4) for n in (10, 90)]   # 10 and 90 "tokens"

def token_losses(x):            # one loss per token
    return (x @ w - 1.0) ** 2

# Reference: one big batch, mean over all 100 tokens
full = torch.cat(micro_batches)
g_true = torch.autograd.grad(token_losses(full).mean(), w)[0]

# Wrong: mean per micro-batch, then average the micro-batch means
g_wrong = sum(torch.autograd.grad(token_losses(x).mean() / 2, w)[0] for x in
 ↪  micro_batches)

# Right: sum per micro-batch, divide by the total token count
total = sum(len(x) for x in micro_batches)
g_right = sum(torch.autograd.grad(token_losses(x).sum() / total, w)[0] for x in
 ↪  micro_batches)

print("wrong error:", (g_wrong - g_true).norm().item())
print("right error:", (g_right - g_true).norm().item())   # about 0
```

## 4.7 Why deep networks train at all

Three ideas make it possible to train networks with dozens or hundreds of layers.

**Initialisation.** Weights start as small random numbers, scaled so that the size of activations stays roughly the same from layer to layer. For a layer with $n_{\text{in}}$ inputs, a common choice draws weights

with variance about $1/n_{\text{in}}$ (or $2/n_{\text{in}}$ for ReLU-like activations). LLMs often also scale down the initialisation of the output projections of each block, so the residual stream does not grow with depth.

**Residual connections.** Each block adds its output to its input: $h_{l+1} = h_l + f_l(h_l)$. The gradient can then flow straight back through the "+" without passing through every block, so it neither vanishes nor explodes. In a transformer, this sum is called the **residual stream** (Chapter 5).

**Normalisation.** LayerNorm or RMSNorm rescales activations inside each block so their size stays stable. Chapter 5 gives the formulas.

## 4.8 Overfitting, validation and early stopping

**Overfitting** means the model fits the training data too well and does worse on new data. You see it when the training loss keeps falling but the **validation loss** (on held-out data) starts rising.

In fine-tuning, overfitting is common because datasets are small. A typical SFT run uses 1 to 3 **epochs** (passes over the data). More epochs often make the model memorise the training responses and lose general ability.

**Early stopping** keeps the checkpoint with the best validation score. But be careful in post-training: validation *loss* is not the same as *quality*. A DPO model's preference accuracy can keep rising while its generations get worse, and an RL model's reward can rise while it learns to hack the reward. Always check real generations and task metrics, not only losses.

## 4.9 Reading a loss curve

| What you see | Likely cause | What to try |
|---|---|---|
| Loss does not go down at all | LR far too low; labels all masked; bug in data | Check that some labels are not $-100$; print a decoded batch; raise LR |
| Loss goes down, then explodes to NaN | LR too high; fp16 overflow; bad example | Lower LR, add warmup, clip gradients, use bf16 |
| Sudden spikes that recover | Bad batches; LR a bit high; Adam $\epsilon$ or $\beta_2$ issues | Clip gradients, lower LR, find and inspect the batches |
| Training loss falls, validation rises | Overfitting | Fewer epochs, more data, lower LR, regularise |
| Loss falls in a staircase, one step per epoch | The model memorises each example after seeing it | Normal in multi-epoch SFT; fewer epochs if quality drops |
| Loss is suspiciously low from step 0 | Training on the prompt tokens, or data leakage | Check the loss mask; check the eval split |

**Notation on this page:** $\epsilon$: small constant that avoids division by zero • $\beta_1, \beta_2$: Adam decay rates for the first and second moments

**Interview check**

**Q1. Write the Adam update, including bias correction.** $m \leftarrow \beta_1 m + (1-\beta_1)g$; $v \leftarrow \beta_2 v + (1-\beta_2)g^2$; $\hat{m} = m/(1-\beta_1^k)$; $\hat{v} = v/(1-\beta_2^k)$; $\theta \leftarrow \theta - \eta\,\hat{m}/(\sqrt{\hat{v}} + \epsilon)$.
**Q2. What is the difference between L2 regularisation and weight decay in Adam?** L2 adds $\lambda\theta$ to the gradient, which Adam then divides by $\sqrt{\hat{v}}$, so the decay is uneven. AdamW subtracts $\eta\lambda\theta$ directly, so every weight decays by the same fraction.
**Q3. Why do we use warmup?** Early Adam moment estimates are noisy and the model is far from a good region; large early steps can diverge. Warmup ramps the LR up gently.
**Q4. How much memory does AdamW need per parameter in mixed precision training?** Commonly 16 bytes: 2 (bf16 weights) + 2 (bf16 gradients) + 4 (fp32 master weights) + 4 ($m$) + 4 ($v$). Chapter 40 covers variants.
**Q5. You use gradient accumulation and per-micro-batch mean losses. What can go wrong?** If micro-batches have different numbers of real tokens, averaging their means weights tokens unequally. Sum token losses and divide by the total token count of the whole effective batch.
**Q6. What is gradient clipping by global norm, and what value is typical?** Rescale the full gradient vector so its L2 norm is at most $c$, keeping its direction. $c = 1.0$ is typical.
**Q7. Why are RL and DPO learning rates much smaller than SFT learning rates?** The signal is noisier, the objective can be over-optimised, and we want to stay close to a good reference model. Small steps keep the policy stable and the KL to the reference small.
**Q8. Why does $\eta$ in Adam roughly bound the per-step parameter change?** Because $\hat{m}/\sqrt{\hat{v}}$ is a normalised quantity, typically of size at most about 1, so each parameter moves by about $\eta$ or less per step.

**Key takeaways**

- Backprop is reverse-mode autodiff; it saves activations, which costs memory.
- AdamW = Adam (momentum + per-parameter scaling + bias correction) + decoupled weight decay. It stores two extra values per parameter.
- Use warmup, then cosine, linear or constant. Post-training LRs are small, and RL/DPO LRs are smallest.
- Clip gradients at a global norm of about 1.0 and log the gradient norm.
- Effective batch = per-device batch × accumulation steps × GPUs. Normalise the loss by the total number of real tokens.
- Residuals, normalisation and careful initialisation make deep networks trainable.
- Do not trust loss alone in post-training; look at generations and task metrics.

**Notation on this page:** $\beta_1, \beta_2$: Adam decay rates for the first and second moments • $g_k$: the (mini-batch) gradient at step $k$ • $k$: optimiser step number • $\theta$: the model's trainable weights • $\epsilon$: small constant that avoids division by zero • $\eta$: learning rate • $\lambda$: weight-decay strength

CHAPTER 5

# The Transformer and LLM Pretraining Recap

**Why this chapter matters.**
You will not train a transformer from scratch in a post-training job. But you need to know exactly what is inside one. LoRA chooses which matrices to adapt. Memory planning needs parameter counts and KV-cache sizes. Rollout speed depends on attention variants. Chat templates, packing and long-context training all touch positions and masks. Interviewers expect you to write attention from memory and to estimate a model's size and memory on a whiteboard.

## 5.1 The big picture

A **decoder-only transformer** language model does this:

1. Split the text into tokens and turn each token into an integer ID (Chapter 6).
2. Look up a vector for each ID. This vector is the token's **embedding**.
3. Pass the vectors through $L$ identical **blocks**. Each block has an **attention** layer (tokens share information with each other) and a **feed-forward** layer (each token is processed on its own).
4. At the end, map each vector to a score for every token in the vocabulary. These scores are the **logits**. A softmax turns them into next-token probabilities.

Almost every modern open LLM (Llama, Qwen, Mistral, DeepSeek, Gemma, OLMo and others) is a variation on this design. The variations are in the details: the attention type, the position encoding, the normalisation, the feed-forward layer and whether it uses mixture-of-experts.

## 5.2 Embeddings and the residual stream

The **embedding matrix** $W_E$ has shape $V \times d$, where $V$ is the vocabulary size and $d$ (or $d_{\text{model}}$) is the **hidden size**. Token $i$ becomes the row $W_E[i]$, a vector of $d$ numbers.

For a sequence of $T$ tokens we get a matrix $h_0$ of shape $T \times d$. Each block then *adds* to it:

$$h'_l = h_{l-1} + \text{Attn}_l(\text{Norm}(h_{l-1})), \qquad h_l = h'_l + \text{FFN}_l(\text{Norm}(h'_l)).$$

The running sum $h$ is called the **residual stream**. Think of it as a shared working memory, one vector per position. Each layer reads from it and writes an update back into it. Because the updates are added, gradients can flow straight back to early layers (Chapter 4).

**Notation on this page:** $L$: number of layers • $W_Q, W_K, W_V, W_O$: attention projections; $W_E, W_U$: (un)embedding • $V$: value matrix (attention) or vocabulary size • $d$: hidden size ($d_{\text{model}}$) • $T$: sequence length

## 5.3 Scaled dot-product attention

Attention lets each position gather information from other positions. For each position, we compute three vectors from its current hidden state $x$ (a row of the normalised residual stream):

- a **query** $q = xW_Q$: "what am I looking for?"
- a **key** $k = xW_K$: "what do I contain?"
- a **value** $v = xW_V$: "what do I pass on if someone attends to me?"

For all $T$ positions at once, with $X$ of shape $T \times d$, this gives matrices $Q$, $K$ (both $T \times d_k$) and $V$ ($T \times d_v$). Then

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right)V.$$

Step by step:

1. $QK^\top$ is a $T \times T$ matrix of **attention scores**. Entry $(i, j)$ is the dot product $q_i \cdot k_j$: how well position $j$'s key matches position $i$'s query.
2. Divide by $\sqrt{d_k}$ (explained next).
3. Add a **mask** $M$ (explained below).
4. Apply softmax along each row. Row $i$ becomes a probability distribution over positions: the **attention weights** of position $i$.
5. Multiply by $V$. Each output row is a weighted average of value vectors.

### 5.3.1 Why divide by $\sqrt{d_k}$?

Suppose the entries of $q$ and $k$ are independent with mean 0 and variance 1. Then

$$\text{Var}(q \cdot k) = \text{Var}\left(\sum_{i=1}^{d_k} q_i k_i\right) = \sum_{i=1}^{d_k} \text{Var}(q_i k_i) = \sum_{i=1}^{d_k} \mathbb{E}[q_i^2]\,\mathbb{E}[k_i^2] = d_k.$$

So the scores have standard deviation $\sqrt{d_k}$. With $d_k = 128$, scores are around $\pm 11$. A softmax over such large numbers is nearly one-hot: one position gets almost all the weight, and the gradient through the softmax becomes tiny (the softmax **saturates**). Dividing by $\sqrt{d_k}$ brings the variance back to 1, so the softmax stays soft and trainable.

### 5.3.2 The causal mask

A language model must not look at future tokens when predicting the next one. Otherwise it could just copy the answer. The **causal mask** sets

$$M_{ij} = \begin{cases} 0 & j \le i \\ -\infty & j > i \end{cases}$$

so after the softmax, $e^{-\infty} = 0$ and position $i$ only attends to positions $1, \ldots, i$. This is why one forward pass over a sequence of length $T$ gives $T$ next-token predictions at once, each using only

**Notation on this page:** $x$: a token's hidden vector (a row of the residual stream) • $q, k, v$: query, key and value vectors • $W_Q, W_K, W_V, W_O$: attention projections; $W_E, W_U$: (un)embedding • $k$: a key vector; $d_k$: key size; top-$k$: number of experts used • $V$: value matrix (attention) or vocabulary size • $T$: sequence length • $d$: hidden size ($d_{\text{model}}$) • $\mathbb{E}$: expected value (average over the distribution in the subscript)

its past. This is the reason training is parallel, and it is also why we can compute all the log-probabilities of a sampled response in one forward pass during RL.

A second kind of mask is the **padding mask**: when sequences of different lengths share a batch, shorter ones are filled with padding tokens, and attention to those positions is blocked. With **packing** (several examples in one row, Chapter 9), the mask must also stop attention *across* examples.

### 5.3.3 Multi-head attention

One attention operation can only focus on one kind of pattern at a time. **Multi-head attention (MHA)** runs $n_h$ attentions in parallel, each with its own smaller projections of size $d_h = d/n_h$ (the **head dimension**). The head outputs are concatenated and mixed by an **output projection** $W_O$ of shape $d \times d$:

$$\text{MHA}(X) = [\text{head}_1; \dots; \text{head}_{n_h}]W_O, \qquad \text{head}_j = \text{Attention}(XW_Q^{(j)}, XW_K^{(j)}, XW_V^{(j)}).$$

In code, all heads are computed together by reshaping tensors to shape (batch, heads, time, head_dim).

> **Worked example.**
> Llama-3-8B has $d = 4096$ and $n_h = 32$ query heads, so $d_h = 4096/32 = 128$.

### 5.3.4 Attention cost

The score matrix is $T \times T$, so attention costs $O(T^2 d)$ compute and, done naively, $O(T^2)$ memory per head. **FlashAttention** computes the same result exactly, but in tiles that fit in fast on-chip memory, without ever storing the full $T \times T$ matrix. It cuts memory to $O(T)$ and is much faster. It is the default in modern training code (Chapter 41).

## 5.4 MQA, GQA and MLA

During generation, the keys and values of past tokens are stored in the **KV cache** (explained later in this chapter). With standard MHA, every query head has its own key and value head, so the cache is large. Three designs shrink it:

- **Multi-query attention (MQA):** all query heads share *one* key head and *one* value head. Smallest cache, but some quality loss.
- **Grouped-query attention (GQA):** query heads are split into $n_{kv}$ groups, and each group shares one key/value head. With $n_{kv} = n_h$ it is MHA; with $n_{kv} = 1$ it is MQA. Llama-3-8B uses 32 query heads and 8 KV heads, so the cache is 4 times smaller than MHA. GQA is the most common choice today.
- **Multi-head latent attention (MLA):** used in DeepSeek-V2 and V3. Keys and values are compressed into a small shared **latent** vector per token, which is what gets cached. They are expanded back per head when needed. This cuts the cache even more while keeping quality close to MHA. It needs special handling for RoPE (a small separate positional part).

**Why this, not that? Why GQA, not full MHA or MQA?**

**Why.** GQA keeps most of MHA's quality while cutting the KV cache by $n_h/n_{kv}$ (4 times for Llama-3-8B). Decoding is limited by memory bandwidth, so a smaller cache means faster generation and more parallel rollouts in RL.
**Trade-off.** Fewer key/value heads mean slightly less expressive attention. The number of KV heads also limits tensor parallelism: you cannot split 8 KV heads cleanly over 16 GPUs.
**When the alternative wins.** MHA is fine for small models or when inference cost does not matter. MQA wins when the cache is the only thing that matters (very long contexts on small hardware). MLA wins at the largest scales, where its even smaller cache is worth the extra complexity.

## 5.5 Positional information

Attention by itself does not know the order of tokens: if you shuffle the inputs, the outputs shuffle in the same way. So position must be injected.

- **Sinusoidal** (the original transformer): add fixed sine and cosine waves of different frequencies to the embeddings.
- **Learned absolute**: add a trained vector for each position (GPT-2). It cannot handle positions longer than it was trained on.
- **RoPE (rotary position embedding)**: the standard today. Explained below.
- **ALiBi**: add a penalty to attention scores that grows with distance.

### 5.5.1 RoPE

RoPE rotates the query and key vectors by an angle that depends on their position. Split a $d_h$-dimensional vector into $d_h/2$ pairs. Pair $i$ is treated as a 2D point and rotated by angle $m\,\omega_i$ at position $m$, where

$$\omega_i = b^{-2i/d_h}, \qquad i = 0, 1, \ldots, d_h/2 - 1.$$

The **base** $b$ is 10,000 in the original paper. Llama 3 uses 500,000. Early pairs rotate fast (they capture local order), later pairs rotate slowly (they capture long-range position).

For one pair, the rotation at position $m$ is

$$R(m\omega) = \begin{pmatrix} \cos m\omega & -\sin m\omega \\ \sin m\omega & \cos m\omega \end{pmatrix}.$$

**The key property: only relative position matters.** A rotation matrix satisfies $R(\alpha)^\top R(\beta) = R(\beta - \alpha)$. So for a query at position $m$ and a key at position $n$:

$$(R(m\omega)\,q)^\top (R(n\omega)\,k) = q^\top R(m\omega)^\top R(n\omega)\,k = q^\top R((n-m)\omega)\,k.$$

The score depends only on the offset $n - m$, not on the absolute positions. That is a clean way to give attention a sense of distance. RoPE is applied to $q$ and $k$ only, not to $v$, and it is applied inside every attention layer.

**Notation on this page:** $d$: hidden size ($d_{\text{model}}$) • $R(\cdot)$: a 2D rotation matrix • $\beta$: LayerNorm shift (in the RoPE proof: a rotation angle) • $\alpha$: a rotation angle (RoPE proof) • $q, k, v$: query, key and value vectors • $k$: a key vector; $d_k$: key size; top-$k$: number of experts used

**Why this, not that? Why RoPE, not learned absolute positions?**

**Why.** RoPE builds relative position directly into the attention score, adds no parameters, and can be stretched to longer contexts with methods like YaRN. Learned absolute embeddings cannot represent positions past the training length at all.
**Trade-off.** RoPE still degrades beyond its training length without a scaling method, and its pairing convention is a source of conversion bugs.
**When the alternative wins.** ALiBi extrapolates to longer inputs more gracefully with no tuning, and some long-context models drop positional encodings in some layers (NoPE) or mix RoPE and non-RoPE layers to improve length generalisation.

**Common mistake.**
There are two common ways to pair up dimensions: adjacent pairs $(0,1),(2,3),\ldots$ (the original paper) and split halves $(0,d_h/2),(1,d_h/2+1),\ldots$ (the "rotate_half" used in Hugging Face Llama code). They are equivalent up to a fixed permutation of the weights, but mixing them up when converting checkpoints silently breaks a model.

### 5.5.2 Extending the context window

A model trained on 8K-token sequences usually fails on 32K tokens, because it never saw such large rotation angles. Methods to extend context with a little extra training:

- **Position interpolation**: squeeze positions, so position $m$ becomes $m \cdot (L_{\text{train}}/L_{\text{new}})$.
- **NTK-aware scaling**: increase the base $b$, which slows down all rotations, but hurts the fast (local) frequencies less than plain interpolation.
- **YaRN**: treats different frequency bands differently (interpolate the slow ones, keep the fast ones) and adjusts the attention temperature. Widely used.

Long-context extension is usually done during mid-training, with a few billion tokens of long documents.

## 5.6 The feed-forward block

After attention, each position goes through a small two-layer network on its own. The original form is

$$\text{FFN}(x) = \phi(xW_1)\,W_2,$$

with $W_1$ of shape $d \times d_{ff}$, $W_2$ of shape $d_{ff} \times d$, and $d_{ff}$ usually about $4d$. The activation $\phi$ was ReLU, then **GELU** (a smooth version of ReLU).

Most modern LLMs use **SwiGLU**, a *gated* feed-forward layer with three matrices:

$$\text{FFN}_{\text{SwiGLU}}(x) = (\text{SiLU}(xW_{\text{gate}}) \odot (xW_{\text{up}}))\,W_{\text{down}}, \qquad \text{SiLU}(z) = z\,\sigma(z),$$

where $\odot$ is element-wise multiplication. One branch decides how much of the other branch to let through. To keep the parameter count similar to a $4d$ two-matrix FFN, $d_{ff}$ is often set to about $\frac{8}{3}d$, rounded. (Llama-3-8B uses $d_{ff} = 14{,}336 = 3.5d$.) In Hugging Face code these three matrices are

**Notation on this page:** $d$: hidden size ($d_{\text{model}}$) • $L$: number of layers • $\phi$: activation function • $x$: a token's hidden vector (a row of the residual stream) • $W_Q, W_K, W_V, W_O$: attention projections; $W_E, W_U$: (un)embedding • $\sigma$: logistic function (in SiLU); $\sigma^2$: variance • $\odot$: element-wise product

called `gate_proj`, `up_proj` and `down_proj`, and the attention matrices are `q_proj`, `k_proj`, `v_proj` and `o_proj`. You will use these names as LoRA targets (Chapter 10).

**Why this, not that? Why SwiGLU, not a plain GELU feed-forward layer?**

**Why.** Gated feed-forward layers consistently give lower loss at the same parameter count and compute. The gate lets each hidden unit decide, per token, how much of the other branch to pass through.
**Trade-off.** Three matrices instead of two, so $d_{ff}$ is shrunk to about $\frac{8}{3}d$ to keep the size the same. Slightly more memory traffic per parameter.
**When the alternative wins.** For very small models or educational code, a plain GELU MLP is simpler and almost as good.

### 5.6.1 Mixture of experts (MoE)

A **mixture-of-experts** layer replaces the single FFN with $E$ separate FFNs, called **experts**, plus a small **router**. For each token, the router scores all experts and sends the token to only the top $k$ (often $k = 1$ to 8, out of $E = 8$ to 256):

$$s = \text{softmax}(xW_r), \qquad \text{MoE}(x) = \sum_{e \in \text{top-}k(s)} g_e(s)\,\text{FFN}_e(x),$$

where the gates $g_e$ are the router weights (often renormalised over the chosen experts).

- **Total parameters** include all experts. **Active parameters** are those used per token. A model can have hundreds of billions of total parameters but only tens of billions active, so it is cheap to run per token but needs a lot of memory to hold.
- **Load balancing.** Without help, the router sends most tokens to a few experts. An **auxiliary loss** rewards an even spread, or (as in DeepSeek-V3) a per-expert bias is adjusted to balance load without an extra loss.
- **For RL** this adds a problem: the trainer and the rollout engine can route the same token to *different* experts because of tiny numerical differences. This makes log-probabilities disagree and destabilises RL. Some recent RL work on MoE models records the experts chosen during rollout and replays them in training (Chapter 28).

**Why this, not that? Why mixture of experts, not a dense model?**

**Why.** MoE gives the knowledge capacity of a huge model at the per-token compute of a small one. For the same training FLOPs, MoE models usually reach lower loss.
**Trade-off.** All experts must be held in memory. Training needs load balancing and expert parallelism. In RL, routing can differ between the rollout engine and the trainer, which breaks importance ratios unless routes are replayed or the algorithm is made robust (GSPO, Chapter 28).
**When the alternative wins.** Dense models are simpler to fine-tune, serve and debug, and they win when memory, not compute, is the constraint, for example on a single GPU or on-device.

**Notation on this page:** $d$: hidden size ($d_{\text{model}}$) • $k$: a key vector; $d_k$: key size; top-$k$: number of experts used • $x$: a token's hidden vector (a row of the residual stream) • $W_Q, W_K, W_V, W_O$: attention projections; $W_E, W_U$: (un)embedding • $g$: a gradient

## 5.7 Normalisation: LayerNorm vs RMSNorm

**LayerNorm** normalises each vector to mean 0 and variance 1, then rescales:

$$\mathrm{LN}(x) = \gamma \odot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta, \qquad \mu = \frac{1}{d}\sum_i x_i, \quad \sigma^2 = \frac{1}{d}\sum_i (x_i - \mu)^2.$$

**RMSNorm** skips the mean and the bias. It only rescales by the root mean square:

$$\mathrm{RMSNorm}(x) = \gamma \odot \frac{x}{\sqrt{\frac{1}{d}\sum_i x_i^2 + \epsilon}}.$$

It is cheaper and works just as well, so most LLMs use it.

**Why this, not that? Why pre-norm RMSNorm, not post-norm LayerNorm?**

**Why.** Pre-norm leaves the residual path untouched, so gradients flow straight through many layers and training is stable without delicate warmup. RMSNorm drops the mean subtraction and bias, which saves compute and memory traffic with no measurable quality loss.
**Trade-off.** With pre-norm, the residual stream can grow large in deep models, and the last layers may contribute less. Extra norms (QK-norm, a final norm, sometimes post-norms too) are added to fix this.
**When the alternative wins.** Post-norm can give slightly better final quality when it trains at all (the original Transformer and BERT used it), and some recent models put norms both before and after each sub-layer.

**Pre-norm vs post-norm.** The original transformer normalised *after* adding the residual (post-norm). Modern LLMs normalise the *input* of each sub-layer (pre-norm), as in the equations at the start of this chapter. Pre-norm keeps the residual path clean and trains much more stably without careful warmup. A final norm is applied after the last block, before the LM head. Some models also normalise queries and keys (**QK-norm**) to avoid exploding attention scores.

## 5.8 The LM head and logits

The final hidden state at each position (after the final norm) is multiplied by the **unembedding** or **LM head** matrix $W_U$ of shape $d \times V$ to get $V$ logits. Some models **tie** $W_U = W_E^\top$, which saves $V \times d$ parameters. This matters most for small models, where the embedding matrix is a large share of all parameters.

**Logits are big tensors.**
For a batch of 8 sequences of 4,096 tokens and a vocabulary of 128K, the logits tensor has $8 \times 4096 \times 128{,}000 \approx 4.2$ billion entries. In float32 that is about 17 GB. This is why efficient training code computes the loss in chunks or with fused kernels (for example, Liger kernels), and why RL code often computes log-probabilities in small micro-batches.

**Notation on this page:** $\sigma$: logistic function (in SiLU); $\sigma^2$: variance • $\beta$: LayerNorm shift (in the RoPE proof: a rotation angle) • $\epsilon$: small constant for numerical safety • $\gamma, \beta$: LayerNorm learned scale and shift • $\mu$: mean of the vector's entries • $\odot$: element-wise product • $x$: a token's hidden vector (a row of the residual stream) • $d$: hidden size ($d_{\text{model}}$) • $W_Q, W_K, W_V, W_O$: attention projections; $W_E, W_U$: (un)embedding • $V$: value matrix (attention) or vocabulary size

## 5.9 A tiny transformer block in PyTorch

The code below implements one pre-norm block with RMSNorm, grouped-query attention with RoPE, a causal mask and SwiGLU. Real implementations add caching, FlashAttention and fused kernels, but the maths is exactly this.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class RMSNorm(nn.Module):
    def __init__(self, d, eps=1e-6):
        super().__init__()
        self.weight = nn.Parameter(torch.ones(d)); self.eps = eps
    def forward(self, x):
        return self.weight * x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)

def rope(x, base=10000.0):
    # x: (B, heads, T, hd). "rotate_half" convention, as in Hugging Face Llama.
    B, H, T, hd = x.shape
    inv_freq = base ** (-torch.arange(0, hd, 2, dtype=torch.float32) / hd)   # (hd/2,)
    angles = torch.arange(T, dtype=torch.float32)[:, None] * inv_freq[None]   # (T, hd/2)
    cos = torch.cat([angles.cos()] * 2, -1); sin = torch.cat([angles.sin()] * 2, -1)
    x1, x2 = x[..., : hd // 2], x[..., hd // 2 :]
    rotated = torch.cat([-x2, x1], -1)
    return x * cos + rotated * sin

class Block(nn.Module):
    def __init__(self, d=64, n_heads=8, n_kv=2, d_ff=172):
        super().__init__()
        self.h, self.kv, self.hd = n_heads, n_kv, d // n_heads
        self.q_proj = nn.Linear(d, n_heads * self.hd, bias=False)
        self.k_proj = nn.Linear(d, n_kv * self.hd, bias=False)
        self.v_proj = nn.Linear(d, n_kv * self.hd, bias=False)
        self.o_proj = nn.Linear(n_heads * self.hd, d, bias=False)
        self.gate_proj = nn.Linear(d, d_ff, bias=False)
        self.up_proj = nn.Linear(d, d_ff, bias=False)
        self.down_proj = nn.Linear(d_ff, d, bias=False)
        self.norm1, self.norm2 = RMSNorm(d), RMSNorm(d)

    def attn(self, x):
        B, T, _ = x.shape
        q = self.q_proj(x).view(B, T, self.h, self.hd).transpose(1, 2)
        k = self.k_proj(x).view(B, T, self.kv, self.hd).transpose(1, 2)
        v = self.v_proj(x).view(B, T, self.kv, self.hd).transpose(1, 2)
        q, k = rope(q), rope(k)
        rep = self.h // self.kv                          # GQA: share each KV head
        k, v = k.repeat_interleave(rep, 1), v.repeat_interleave(rep, 1)
        scores = q @ k.transpose(-2, -1) / self.hd ** 0.5
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), 1)
```

```
        scores = scores.masked_fill(mask, float("-inf"))
        out = scores.softmax(-1) @ v                    # (B, H, T, hd)
        return self.o_proj(out.transpose(1, 2).reshape(B, T, -1))

    def forward(self, x):
        x = x + self.attn(self.norm1(x))
        h = self.norm2(x)
        return x + self.down_proj(F.silu(self.gate_proj(h)) * self.up_proj(h))

torch.manual_seed(0)
blk = Block()
x = torch.randn(1, 10, 64)
y = blk(x)
# Causality check: changing the last token must not change earlier outputs.
x2 = x.clone(); x2[0, -1] += 1.0
print(y.shape, torch.allclose(blk(x2)[0, :-1], y[0, :-1], atol=1e-6))
# Our attention should match PyTorch's fused version.
B, T = 1, 10
q = rope(blk.q_proj(blk.norm1(x)).view(B, T, 8, 8).transpose(1, 2))
k = rope(blk.k_proj(blk.norm1(x)).view(B, T, 2, 8).transpose(1, 2)).repeat_interleave(4,
 ↪  1)
v = blk.v_proj(blk.norm1(x)).view(B, T, 2, 8).transpose(1, 2).repeat_interleave(4, 1)
ref = F.scaled_dot_product_attention(q, k, v, is_causal=True)
ours = (q @ k.transpose(-2, -1) / 8 ** 0.5).masked_fill(
    torch.triu(torch.ones(T, T, dtype=torch.bool), 1), float("-inf")).softmax(-1) @ v
print(torch.allclose(ref, ours, atol=1e-5))
```

## 5.10 Counting parameters and FLOPs

You should be able to estimate a model's size from its config. For one block with hidden size $d$, $n_h$ query heads, $n_{kv}$ KV heads of size $d_h$, and a SwiGLU FFN of width $d_{ff}$ (ignoring the tiny norm weights):

$$\text{attention} = \underbrace{d \cdot n_h d_h}_{\uparrow W_Q} + \underbrace{2\,d \cdot n_{kv} d_h}_{\uparrow W_K, W_V} + \underbrace{n_h d_h \cdot d}_{\uparrow W_O}, \qquad \text{FFN} = 3\,d\,d_{ff}.$$

Add $Vd$ for the embedding, and another $Vd$ if the LM head is not tied.

For classic MHA with a $4d$ two-matrix FFN, this simplifies to the well-known $12d^2$ per layer ($4d^2$ attention + $8d^2$ FFN).

**Worked example.**
**Llama-3-8B**: $d = 4096$, $L = 32$, $n_h = 32$, $n_{kv} = 8$, $d_h = 128$, $d_{ff} = 14{,}336$, $V = 128{,}256$, untied.

- Attention per layer: $2 \times 4096^2 + 2 \times 4096 \times 1024 \approx 33.6\text{M} + 8.4\text{M} = 41.9\text{M}$.
- FFN per layer: $3 \times 4096 \times 14{,}336 \approx 176.2\text{M}$.
- Per layer: about 218M. Times 32 layers: about 6.98B.
- Embedding and LM head: $2 \times 128{,}256 \times 4096 \approx 1.05\text{B}$.
- **Total: about 8.03B.** This matches the published size.

**FLOPs.** A **FLOP** is one floating-point operation (an add or a multiply). Multiplying a vector by

**Notation on this page:** $d$: hidden size ($d_{\text{model}}$) • $V$: value matrix (attention) or vocabulary size • $W_Q, W_K, W_V, W_O$: attention projections; $W_E, W_U$: (un)embedding • $L$: number of layers

a $d_1 \times d_2$ matrix takes about $2d_1d_2$ FLOPs. So a forward pass costs about $2N$ FLOPs per token for a model with $N$ parameters (ignoring attention's $T^2$ part, which is small for moderate lengths). The backward pass costs about twice the forward pass. So:

$$\text{training FLOPs} \approx 6\,N\,D, \qquad \text{inference FLOPs} \approx 2\,N \text{ per generated token},$$

where $D$ is the number of training tokens.

**Worked example.**
Training an 8B model on 15 trillion tokens costs $6 \times 8 \times 10^9 \times 15 \times 10^{12} = 7.2 \times 10^{23}$ FLOPs. A GPU that sustains $4 \times 10^{14}$ FLOP/s (400 TFLOP/s) would need $1.8 \times 10^9$ seconds, about 57 GPU-years. That is why pretraining runs on thousands of GPUs.

## 5.11 The KV cache and why generation is memory bound

When generating, the model produces one token at a time. At step $t$, the new token's query must attend to the keys and values of *all* previous tokens. Recomputing them every step would waste huge amounts of work. Instead, we store them: this is the **KV cache**.

Generation has two phases:

- **Prefill**: process the whole prompt in one parallel forward pass and fill the cache. This is compute bound, like training.
- **Decode**: generate tokens one at a time. Each step reads *all* the model weights and the whole KV cache from GPU memory, but does very little maths per byte read. So decoding is **memory-bandwidth bound**: speed is limited by how fast data moves, not by FLOPs. Batching many sequences together re-uses each weight read for many tokens, which is why rollout engines batch aggressively (Chapter 43).

**KV-cache size.** For each token, each layer stores one key and one value vector per KV head:

$$\text{KV bytes per token} = 2 \times L \times n_{kv} \times d_h \times \text{bytes per value}.$$

**Worked example.**
Llama-3-8B in bf16 (2 bytes): $2 \times 32 \times 8 \times 128 \times 2 = 131{,}072$ bytes = 128 KiB per token. One sequence of 8,192 tokens needs 1 GiB of cache. A batch of 64 such sequences needs 64 GiB, more than the 16 GB of weights. With full MHA (32 KV heads) it would be 4 times larger. In RL with long reasoning traces, the KV cache, not the weights, often limits how many rollouts you can run at once.

## 5.12 Pretraining in brief

**Objective.** Next-token cross-entropy over all tokens (Chapter 3).

**Data.** A mix of web text, code, books, academic papers, maths and multilingual data, heavily filtered and deduplicated. Data quality and mixture are among the most important choices in the whole pipeline.

**Notation on this page:** $d$: hidden size ($d_{\text{model}}$) • $N$: number of model parameters; $D$: number of training tokens • $T$: sequence length • $t$: token position • $L$: number of layers

**Scaling laws.** The loss falls smoothly, as a power law, as you increase parameters $N$, data $D$ and compute $C$.

- *Kaplan et al.* (*2020*) first measured these laws and suggested putting most extra compute into larger models.
- *Chinchilla* (*Hoffmann et al., 2022*) found that, for a fixed compute budget, $N$ and $D$ should grow together, at roughly **20 tokens per parameter** for a compute-optimal model.
- In practice, modern models are trained on far more tokens than this (Llama-3-8B saw about 15T tokens, almost 2,000 per parameter). Training past the compute-optimal point costs more up front but gives a smaller model for the same quality, and the smaller model is cheaper to *serve* for its whole life.

**Mid-training and annealing.** Near the end of pretraining, the learning rate is decayed while training on a higher-quality mix (maths, code, reasoning, instruction-like data, long documents). This "annealing" phase gives large gains on benchmarks for its cost. Many teams treat it as a separate stage, called mid-training. The line between mid-training and post-training is blurry: some instruction and reasoning data now goes into mid-training.

## 5.13 Base, instruct and reasoning checkpoints

Open model releases usually come in several versions:

- **Base**: after pretraining (and often mid-training). Use it when you want full control over post-training, or when doing "zero" RL experiments in the style of DeepSeek-R1-Zero.
- **Instruct** (or "chat"): after SFT and preference optimisation. Use it when you want to fine-tune a working assistant for a task.
- **Reasoning** (or "thinking"): trained to produce long chains of thought before answering, usually with RL. Some models have a switch to turn thinking on or off through the chat template.

**Tip.**
For a fine-tuning project, starting from the instruct model usually gives the best results with the least data. Starting from the base model makes sense for research on the post-training process itself, or when the instruct model's style or safety tuning gets in the way. Always use the chat template that matches the checkpoint (Chapter 8).

**Interview check**

**Q1. Write scaled dot-product attention and explain the $\sqrt{d_k}$.** $\text{softmax}(QK^\top/\sqrt{d_k} + M)V$. With unit-variance entries, $q \cdot k$ has variance $d_k$; dividing by $\sqrt{d_k}$ keeps scores near unit variance so the softmax does not saturate and gradients do not vanish.
**Q2. What is GQA and why use it?** Query heads are split into groups that share a key/value head. It cuts KV-cache size and memory bandwidth by $n_h/n_{kv}$ with little quality loss, which speeds up generation.
**Q3. Show that RoPE gives relative position.** $(R_m q)^\top (R_n k) = q^\top R_m^\top R_n k = q^\top R_{n-m} k$, since

**Notation on this page:** $N$: number of model parameters; $D$: number of training tokens • $d$: hidden size ($d_{\text{model}}$) • $k$: a key vector; $d_k$: key size; top-$k$: number of experts used • $V$: value matrix (attention) or vocabulary size • $q, k, v$: query, key and value vectors • $R(\cdot)$: a 2D rotation matrix

rotations compose by adding angles and $R_m^\top = R_{-m}$.

**Q4. Estimate the parameter count of a model with $d = 4096$, 32 layers, MHA, 4d FFN, vocab 32K, tied embeddings.** $12d^2L = 12 \times 16.8\text{M} \times 32 \approx 6.4\text{B}$, plus $32\text{K} \times 4096 \approx 0.13\text{B}$: about 6.6B.

**Q5. How big is the KV cache for a 32-layer model with 8 KV heads of size 128, in bf16, for 16 sequences of 4,096 tokens?** 128 KiB per token × 65,536 tokens = 8 GiB.

**Q6. Why is decoding memory bound?** Each decode step reads all weights and the KV cache but does only about 2 FLOPs per weight per sequence. The arithmetic intensity is low, so memory bandwidth limits speed. Batching raises it.

**Q7. RMSNorm vs LayerNorm?** RMSNorm divides by the root mean square, with no mean subtraction and no bias. It is cheaper and works as well.

**Q8. What is the difference between total and active parameters in an MoE model?** Total counts all experts and determines memory. Active counts only what a token uses (shared parts plus the top-$k$ experts) and determines compute per token.

**Q9. Training FLOPs for a 70B model on 2T tokens?** $6 \times 70 \times 10^9 \times 2 \times 10^{12} = 8.4 \times 10^{23}$.

**Key takeaways**

- A decoder-only transformer is embeddings, then $L$ pre-norm blocks of attention + FFN on a residual stream, then an LM head.
- Attention is $\text{softmax}(QK^\top/\sqrt{d_k} + M)V$ with a causal mask. One forward pass gives all next-token predictions.
- GQA, MQA and MLA shrink the KV cache. RoPE gives relative positions. SwiGLU and RMSNorm are standard.
- MoE separates total parameters (memory) from active parameters (compute).
- Parameters ≈ attention $+3dd_{ff}$ per layer + embeddings. Training FLOPs ≈ $6ND$.
- KV cache bytes per token = $2Ln_{kv}d_h\times$ bytes. Decoding is memory-bandwidth bound.
- Modern models are trained far beyond Chinchilla-optimal token counts to make inference cheaper.

**Notation on this page:** $R(\cdot)$: a 2D rotation matrix • $d$: hidden size ($d_{\text{model}}$) • $L$: number of layers • $k$: a key vector; $d_k$: key size; top-$k$: number of experts used • $V$: value matrix (attention) or vocabulary size

CHAPTER 6

# Tokenisation

**Why this chapter matters.**
A surprising number of real post-training bugs are tokenisation bugs: a missing end-of-sequence token, a pad token that is also the EOS token, a prompt that tokenises differently when joined to its response, or a chat template applied twice. In multi-turn RL, decoding text and then re-encoding it can silently change token IDs and corrupt your log-probabilities. This chapter explains how tokenizers work so you can spot these problems.

## 6.1 Why tokenise?

A neural network works on numbers, not text. A **tokenizer** turns text into a sequence of integers, called **token IDs**, and back again. Each ID stands for a **token**, a piece of text from a fixed list called the **vocabulary**.

What should a token be?

- **Characters**: a tiny vocabulary, and no unknown words. But sequences become very long, and attention cost grows with the square of length. Each step also carries little meaning.
- **Words**: short sequences, but the vocabulary becomes huge, and any new word, typo or name is "unknown".
- **Subwords**: the middle ground used by every modern LLM. Common words are single tokens (`"the"`, `" house"`). Rare words are split into pieces (`"tokenisation"` might become `"token"` + `"isation"`). Any string can still be written, because the pieces go all the way down to single bytes.

A typical English text uses about 0.75 words per token, or about 4 characters per token. Vocabulary sizes today are usually 32K to 256K. Larger vocabularies give shorter sequences (faster, more context) but bigger embedding and LM-head matrices.

**Why this, not that? Why a large vocabulary (128K or more), not 32K?**

**Why.** Larger vocabularies cut sequence length, often by 15% or more for code and non-English text. Shorter sequences mean cheaper attention, more effective context and faster generation, which matters a lot for RL rollouts.
**Trade-off.** The embedding and LM-head matrices grow with $V$, the softmax over the vocabulary gets more expensive, and rare tokens get few training examples.
**When the alternative wins.** Small models (under about 1B parameters) can spend a large share of their parameters on the embeddings, so a smaller vocabulary is often better there.

**Notation on this page:** $V$: vocabulary size

## 6.2 Byte-pair encoding (BPE)

**BPE** is the most common algorithm. It *learns* a vocabulary from a training corpus by repeatedly merging the most frequent adjacent pair.

**Training BPE:**

1. Start with a vocabulary of single symbols (characters, or bytes).
2. Split every word in the corpus into those symbols.
3. Count every adjacent pair of symbols, weighted by word frequency.
4. Merge the most frequent pair into a new symbol, add it to the vocabulary, and record the merge.
5. Repeat steps 3 and 4 until the vocabulary reaches the target size.

**Encoding new text** applies the recorded merges in the same order they were learned.

> **Worked example.**
> Corpus word counts: "low" ×5, "lower" ×2, "newest" ×6, "widest" ×3. Start from characters.
>
> - Count pairs. ("e", "s") and ("s", "t") each appear 9 times (6 in "newest" + 3 in "widest"). ("w", "e") appears 8 times. ("l", "o") and ("o", "w") appear 7 times.
> - Ties need a fixed rule. The code below picks the pair that sorts last, so the first merge is "s" + "t" → "st".
> - Now ("e", "st") appears 9 times: merge to "est".
> - Next, ("o", "w") and ("l", "o") tie at 7: merge "o" + "w" → "ow", then "l" + "ow" → "low".
>
> After a few merges, "lowest" (never seen in training) is encoded as "low" + "est".

```
from collections import Counter

def train_bpe(word_counts, num_merges):
    words = {tuple(w): c for w, c in word_counts.items()}  # word -> tuple of symbols
    merges = []
    for _ in range(num_merges):
        pairs = Counter()
        for syms, c in words.items():
            for a, b in zip(syms, syms[1:]):
                pairs[(a, b)] += c
        if not pairs:
            break
        best = max(pairs, key=lambda p: (pairs[p], p))   # most frequent, ties by symbol
        merges.append(best)
        new_words = {}
        for syms, c in words.items():
            out, i = [], 0
            while i < len(syms):
                if i + 1 < len(syms) and (syms[i], syms[i + 1]) == best:
                    out.append(syms[i] + syms[i + 1]); i += 2
```

```
                else:
                    out.append(syms[i]); i += 1
            new_words[tuple(out)] = c
        words = new_words
    return merges

def encode(word, merges):
    syms = list(word)
    for a, b in merges:                    # apply merges in learned order
        i = 0
        while i < len(syms) - 1:
            if syms[i] == a and syms[i + 1] == b:
                syms[i:i + 2] = [a + b]
            else:
                i += 1
    return syms

merges = train_bpe({"low": 5, "lower": 2, "newest": 6, "widest": 3}, 6)
print(merges)
print(encode("lowest", merges))   # ['low', 'est']
```

### 6.2.1 Byte-level BPE

If the base symbols are Unicode characters, you need a huge base vocabulary to cover every script and emoji, and some characters will still be unknown. **Byte-level BPE** (introduced with GPT-2) starts instead from the 256 possible **bytes** of UTF-8. Every string is a sequence of bytes, so *any* text can be encoded and there is never an "unknown" token. GPT-2, GPT-4's tokenizers, Llama 3 and Qwen all use byte-level BPE.

**Why this, not that? Why byte-level BPE, not a Unigram SentencePiece model?**

**Why.** Byte-level BPE can encode any input exactly, with no unknown tokens and no special handling of scripts or emoji. It is fast, deterministic and well supported (for example by tiktoken and Hugging Face tokenizers).
**Trade-off.** Rare scripts are split into many byte tokens, so they cost more. Merges are greedy and can produce odd splits of numbers and words.
**When the alternative wins.** Unigram models give a probabilistic segmentation and can sample several segmentations, which helps robustness and some multilingual settings. They are still common in encoder and translation models (T5 and many multilingual models).

Before BPE runs, text is usually split by a **pre-tokenisation** rule (a regular expression) into chunks such as words, numbers and punctuation, so merges never cross those boundaries. One visible result: in most tokenizers the space belongs to the *following* word. `" Paris"` (with a leading space) and `"Paris"` are different tokens.

## 6.3 WordPiece, Unigram and SentencePiece

- **WordPiece** (BERT): like BPE, but it picks the merge that most increases the likelihood of the training data, rather than the most frequent pair. Word-internal pieces are marked with "##".
- **Unigram language model** (T5, many multilingual models): starts with a large vocabulary and *removes* tokens that least reduce the likelihood of the corpus. Encoding picks the most likely segmentation. It can also sample different segmentations, which is a form of data augmentation.
- **SentencePiece** is a *library*, not an algorithm. It can train BPE or Unigram models directly on raw text, treating the space as a normal symbol (shown as "▁"). Llama 1 and 2 used a SentencePiece BPE model with 32K tokens; Llama 3 moved to a byte-level BPE with about 128K tokens.

## 6.4 Special tokens

Tokenizers reserve some IDs for **special tokens**, which never come from normal text. They mark structure.

| Token | Typical name | Purpose |
|---|---|---|
| Beginning of sequence | BOS, `<s>`, `<\|begin_of_text\|>` | Marks the start of a document |
| End of sequence | EOS, `</s>`, `<\|end_of_text\|>`, `<\|im_end\|>`, `<\|eot_id\|>` | Marks the end; generation stops when it is produced |
| Padding | PAD | Fills short sequences in a batch; ignored by attention and loss |
| Unknown | UNK | For unknown input; not needed with byte-level BPE |
| Chat roles | `<\|im_start\|>`, `<\|start_header_id\|>`, … | Mark where system, user and assistant turns start and end |
| Tool and reasoning markers | e.g. `<tool_call>`, `<think>` (model specific) | Mark tool calls or reasoning sections |

Different model families use different names, and some models use *several* end tokens (one for the end of a turn, one for the end of the text). In Hugging Face, `generation_config.eos_token_id` can be a list. If generation never stops, check that the right stop tokens are set.

**Common mistake.**
**EOS must be in the training labels.** If SFT data does not end each assistant turn with the correct end token, or the end token is masked out of the loss, the model never learns to stop. You will see endless, repeating generations.

## 6.5 Where tokenisation goes wrong

**Numbers.** Tokenizers split numbers in different ways. Some split every digit; some use chunks of up to three digits; older ones split numbers into arbitrary pieces ("12345" → "123", "45"). Inconsistent splits make arithmetic harder to learn. Many modern tokenizers split digits individually or in fixed groups for this reason.

**Whitespace and code.** Runs of spaces and tabs may be one token or many. Python indentation can become expensive. Some tokenizers add special tokens for common whitespace runs.

**Multilingual cost.** A tokenizer trained mostly on English text needs many more tokens for, say, Hindi or Tamil. The same sentence costs more compute and fills the context faster. Vocabulary design is a fairness and cost question, not only an engineering one.

**Boundary effects.** Tokenising two strings separately and joining the IDs can give a different result from tokenising the joined string, because a merge can cross the boundary.

> **Worked example.**
> Suppose a tokenizer has a single token for `" answer:"`. Tokenising the prompt `"The answer"` and the completion `":"` separately gives `[" answer", ":"]`. Tokenising `"The answer:"` in one go gives `[" answer:"]`. The model only ever saw the second form in pretraining. If your training code builds prompt and completion IDs separately, the model trains on token sequences it would never produce on its own.

**Retokenisation drift in RL.** In RL, the model generates token IDs. If your code decodes them to text and later re-encodes the text (for example, in a multi-turn agent loop that rebuilds the conversation), the new IDs can differ from the ones that were actually sampled. Then the log-probabilities you compute in training are for a sequence the policy never sampled, and importance ratios are wrong. The safe rule: **keep the sampled token IDs and train on them directly.**
**Never round-trip through text.** Good RL frameworks pass token IDs, not strings, between the rollout engine and the trainer (Chapter 34).

**Why this, not that? Why pass token IDs between rollout and trainer, not text?**

**Why.** Text-to-token conversion is not one-to-one: decoding and re-encoding can produce different IDs. Passing IDs keeps the training log-probabilities exactly aligned with what was sampled.
**Trade-off.** IDs are tied to one tokenizer and one chat template, so environments and reward functions that work on text need a decode step, and multi-turn agents must append new turns as IDs rather than rebuilding the prompt from strings.
**When the alternative wins.** For single-turn tasks where the prompt is built once and the response is never re-encoded, passing text is harmless and simpler.

## 6.6 Tokenizers in Hugging Face

In `transformers`, `AutoTokenizer.from_pretrained(name)` loads the right tokenizer for a model. The fields you will touch most:

- `tokenizer(text)` returns a dict with `input_ids` and `attention_mask` (1 for real tokens, 0 for padding).
- `tokenizer.decode(ids, skip_special_tokens=True)` turns IDs back into text.
- `tokenizer.pad_token`, `eos_token`, `bos_token` and their `_id` versions.
- `padding_side`: `"right"` or `"left"`.
- `truncation=True, max_length=...` cuts sequences that are too long.
- `add_special_tokens`: whether to add BOS/EOS automatically. Chat templates often add their own, so adding them again causes a **double BOS**.
- `tokenizer.apply_chat_template(messages, ...)` formats a conversation (Chapter 8).

```
from transformers import AutoTokenizer

tok = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B-Instruct")
enc = tok(["Hello world", "Hi"], padding=True, return_tensors="pt")
print(enc["input_ids"], enc["attention_mask"])
print(tok.convert_ids_to_tokens(tok("The answer is 12345.")["input_ids"]))
# ['The', 'Ġanswer', 'Ġis', 'Ġ', '1', '2', '3', '4', '5', '.']
print(tok.padding_side, tok.pad_token, tok.eos_token, tok.bos_token)
# right <|endoftext|> <|im_end|> None
```

The output (checked with `transformers` 5.20.2) shows several things from this chapter at once. `"Ġ"` is how byte-level BPE displays a leading space, so `" answer"` is one token with its space attached. Qwen splits every digit into its own token. Its pad token (`<|endoftext|>`) is different from its end-of-turn token (`<|im_end|>`), and it has no BOS token at all. Other model families make different choices, so always check.

### 6.6.1 Padding side

Batches need rectangular tensors, so shorter sequences are padded.

- **Training** usually pads on the **right**. Positions and the loss mask are simple.
- **Batched generation** with a decoder-only model must pad on the **left**. The model generates from the *last* position of each row. With right padding, the last position of a short prompt is a pad token, and generation continues from padding. With left padding, every row ends with its real last token.

### 6.6.2 Choosing a pad token

Some tokenizers have no pad token. A common quick fix is `tok.pad_token = tok.eos_token`. This is risky: if your data collator masks *all* pad-token positions out of the labels, it also masks the real EOS at the end of each response, and the model never learns to stop (the warning above). Prefer a dedicated pad token if the tokenizer has one (many recent tokenizers include reserved or padding tokens), or make sure your collator masks padding by **position** (from the attention mask), not by token ID. Modern TRL trainers handle this, but you should still check one batch by hand.

**Tip.**
Before any fine-tuning run, print one training example three ways: the raw text, the token strings (`convert_ids_to_tokens`), and the labels with masked positions shown as a placeholder. Check that the template is right, there is exactly one BOS, every assistant turn ends with the end token, and only the tokens you want to train on have labels. This five-minute check prevents the most common SFT bugs.

## 6.7 Measuring tokenizer effects yourself

Three short measurements make the ideas of this chapter concrete. All use the Qwen2.5 tokenizer (vocabulary of 151,643 regular tokens plus added special tokens, 151,665 in total).

**1. Retokenisation drift, concretely.** Suppose the policy sampled the tokens for "hel" and then "lo world". Decoding gives the text "hello world"; encoding that text again gives *different, fewer* token ids, because the tokenizer prefers the merged token "hello". A trainer that re-encodes text would compute log-probabilities for a sequence the policy never produced.

**2. Fertility across languages. Fertility** is the number of tokens per word. Higher fertility means more compute per sentence, a shorter effective context, and (for RL with token-level length limits) less room to reason.

**3. Template overhead.** The chat template adds role markers, newlines and, for Qwen2.5, a default system prompt when none is given. For short prompts this overhead is a large share of the tokens.

```
import os; os.environ["HF_HUB_OFFLINE"] = "1"
import warnings; warnings.filterwarnings("ignore")
from transformers import AutoTokenizer
qtok = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B-Instruct")
sampled = qtok.encode("hel", add_special_tokens=False) + qtok.encode("lo world",
 ↪  add_special_tokens=False)
text = qtok.decode(sampled)
again = qtok.encode(text, add_special_tokens=False)
print(f"sampled ids {sampled} -> text {text!r} -> re-encoded ids {again} | same:
 ↪  {sampled == again}")
samples = {"English": "The weather is nice today and we will go for a walk.",
           "French": "Le temps est beau aujourd'hui et nous irons nous promener.",
          "Hindi": "\u0906\u091c \u092e\u094c\u0938\u092e \u0905\u091a\u094d\u091b\u093e
            ↪  \u0939\u0948 \u0914\u0930 \u0939\u092e \u091f\u0939\u0932\u0928\u0947
            ↪  \u091c\u093e\u090f\u0902\u0917\u0947\u0964",  # "Today the weather is
            ↪  nice and we will go for a walk."
           "Python": "for i in range(10):\n    print(i * i)"}
for lang, s in samples.items():
    n = len(qtok.encode(s, add_special_tokens=False))
    print(f"{lang:8s} {n:3d} tokens, {len(s.split()):2d} words, fertility {n /
     ↪  len(s.split()):.2f}")
user_only = [{"role": "user", "content": "Hi"}]
with_sys = [{"role": "system", "content": "Be brief."}, {"role": "user", "content":
 ↪  "Hi"}]
```

```
for name, m in [("no system message", user_only), ("short system message", with_sys)]:
    ids = qtok.apply_chat_template(m, add_generation_prompt=True, tokenize=True)
    ids = ids["input_ids"] if hasattr(ids, "keys") else ids
    print(f"template tokens for 'Hi' with {name}: {len(ids)}")
```

The sentence in Hindi costs about four times as many tokens per word as English with this tokenizer. Without a system message, the Qwen2.5 template inserts its default system prompt, so a two-character user message becomes about thirty tokens of context; with a short system message it is much smaller. If training data had no system message but production always sends one (or the reverse), the model sees a different context at inference than in training.

## 6.8 Adding special tokens safely

Projects often add tokens: tool-call markers, a new role, a "thinking" delimiter. Three steps, each with a trap.

1. **Add the token to the tokenizer** as a special token, so it is never split and can be used as a stop token: `tokenizer.add_special_tokens({"additional_special_tokens": ["<tool_call>"]})` (or the model's existing reserved tokens, if it has unused ones: better, since their embeddings already exist).
2. **Resize the embedding matrix and the output head** with `model.resize_token_embeddings(len(tokenizer))`. New rows must be initialised sensibly. Random rows can have norms far from the trained ones and produce huge or tiny logits; initialising to the **mean of existing embeddings** (Transformers 5 does this by default, with `mean_resizing=True`) gives the new token an "average" starting point.
3. **Make the new rows trainable.** With LoRA, the embedding and output layers are frozen by default, so the new token would never learn its meaning. Add them to `modules_to_save` (or use `trainable_token_indices` in PEFT to train only the new rows).

```
import torch
from transformers import Qwen2Config, Qwen2ForCausalLM
torch.manual_seed(0)
cfg = Qwen2Config(vocab_size=len(qtok), hidden_size=64, intermediate_size=128,
 ↪  num_hidden_layers=2,
                  num_attention_heads=4, num_key_value_heads=2, tie_word_embeddings=True)
small = Qwen2ForCausalLM(cfg)
t2 = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B-Instruct")
n_added = t2.add_special_tokens({"additional_special_tokens": ["<tool_call_x>"]})
old_rows = small.get_input_embeddings().weight.shape[0]
small.resize_token_embeddings(len(t2))
E = small.get_input_embeddings().weight.detach()
new_id = t2.convert_tokens_to_ids("<tool_call_x>")
print(f"added {n_added} token, id {new_id}; embedding rows {old_rows} -> {E.shape[0]}")
print(f"new row norm {E[new_id].norm():.4f} vs mean-of-old-rows norm
 ↪  {E[:old_rows].mean(0).norm():.4f} "
      f"vs typical row norm {E[:old_rows].norm(dim=1).mean():.4f}")
```

```
print("encoded as one token:", t2.encode("call <tool_call_x> now",
 ↪  add_special_tokens=False))
```

The new row equals (up to small noise) the mean of the old rows, so its norm is much smaller than a typical row's: a neutral starting point that training then moves. Note that this tiny model's vocabulary was sized to the tokenizer, so here the resize adds rows; real checkpoints often have spare rows (Qwen2.5-0.5B's embedding has 151,936 rows for 151,665 tokenizer entries), and then a new token can use an existing row without resizing.

**Why this, not that? Why reuse reserved tokens or add few new ones, not a new special token for every tool and role?**

**Reason.** Every new token starts with no meaning and must be learned from your fine-tuning data alone; reserved tokens and plain-text markers the model already understands (for example JSON or XML-like tags made of ordinary tokens) work from the first step. **Trade-off.** Plain-text markers cost several tokens each and can collide with ordinary text; special tokens are compact, unambiguous and make reliable stop tokens. **When the alternative is better.** When the marker must never appear in normal text (stop sequences, role boundaries) or will be used millions of times (cost per token matters), a dedicated special token, trained with enough data, is the right choice.

## 6.9 Common mistakes

- **Adding special tokens with LoRA but freezing the embeddings**: the new tokens never learn; use `modules_to_save` or trainable token indices.
- **Forgetting to save the modified tokenizer with the model**: inference then splits the new token into pieces.
- **Training without a system message but serving with one** (or the reverse) on templates that insert a default system prompt.

**Interview check**

**Q1. How does BPE training work?** Start from characters or bytes, count adjacent pairs across the corpus weighted by frequency, merge the most frequent pair into a new token, record the merge, and repeat until the vocabulary is the target size. Encoding applies the merges in order.
**Q2. Why byte-level BPE?** The base vocabulary is the 256 bytes, so any UTF-8 text can be encoded with no unknown tokens and a small base alphabet.
**Q3. Why must batched generation use left padding?** Generation continues from the last position of each row. With right padding, short prompts end in pad tokens; left padding puts every prompt's real last token at the end.
**Q4. What goes wrong if you set pad_token = eos_token?** If the collator masks labels by token ID, the real EOS gets masked too, so the model never learns to stop. Mask by position instead, or use a separate pad token.
**Q5. What is retokenisation drift and why does it matter in RL?** Decoding sampled IDs

to text and re-encoding can produce different IDs. Training log-probabilities then refer to a sequence the policy did not sample, which breaks importance ratios. Keep and train on the sampled token IDs.

**Q6. Why can't you compare per-token loss across tokenizers?** Tokens cover different amounts of text. Compare bits per byte instead.

**Q7. What are the trade-offs of a bigger vocabulary?** Shorter sequences, faster inference and more effective context, better multilingual coverage; but bigger embedding and LM-head matrices, more rare tokens with few training examples, and a larger softmax.

**Q8. What is tokenizer fertility and why does it matter?** Tokens per word; higher fertility costs more compute, shortens the effective context and uses up length limits in RL.

**Q9. After adding a special token, what three things must you do?** Resize the embeddings and output head, initialise the new rows sensibly (for example the mean of existing rows), and make them trainable (with LoRA, `modules_to_save` or trainable token indices); then save the tokenizer with the model.

**Q10. Give a concrete example of retokenisation drift.** Sampled tokens "hel" + "lo world" decode to "hello world", which re-encodes as "hello" + " world": different ids and a different log-probability.

**Key takeaways**

- Modern LLMs use subword tokenizers, mostly byte-level BPE, with vocabularies of 32K to 256K.
- BPE learns merges from frequent pairs; encoding replays the merges.
- Special tokens mark structure: BOS, EOS, PAD and chat-role markers. The model must see the correct end token in training labels.
- Joining separately tokenised strings can differ from tokenising the joined string.
- In RL, train on the sampled token IDs; never decode and re-encode.
- Train with right padding, generate with left padding, and never let the EOS be masked as padding.
- Measure: retokenisation drift, fertility per language, template overhead (including default system prompts).
- New special tokens need resizing, mean initialisation, trainable rows and a saved tokenizer; prefer reserved tokens.

CHAPTER 7

# Decoding and Sampling

**Why this chapter matters.**
In RL, the model learns from its own samples. So the way you sample decides what the model can discover, and a mismatch between how you sample and how you compute log-probabilities quietly biases training. In evaluation, the sampling settings can move benchmark scores by several points. And "implement top-p sampling" is a classic coding interview question.

## 7.1 From logits to a distribution

At each step, the model outputs a vector of logits $z \in \mathbb{R}^V$ for the next token. The softmax turns them into probabilities $p_i = e^{z_i} / \sum_j e^{z_j}$ (Chapter 2). A **decoding strategy** is the rule for picking the next token from this distribution. We repeat until we hit a stop condition.

There are two families:

- **Deterministic** search: greedy decoding and beam search. The same prompt always gives the same output.
- **Stochastic** sampling: draw the next token at random from a (possibly modified) distribution. Temperature, top-k, top-p and min-p are modifications.

## 7.2 Greedy decoding and beam search

**Greedy decoding** always picks the most likely token: $y_t = \arg\max_i p_i$. It is fast and deterministic (temperature 0 in most APIs). But the most likely *next token* does not always lead to the most likely *sequence*, and greedy outputs of small models often fall into loops.

**Beam search** keeps the $b$ best partial sequences (the **beam**), ranked by total log-probability. At each step it extends every beam with every token, scores them, and keeps the top $b$. It finds higher-likelihood sequences than greedy. It works well for tasks with one right answer (translation, speech recognition). For open-ended chat it tends to give bland, repetitive text, because high-likelihood text is often generic. It is rarely used for chat LLMs or RL rollouts. A **length penalty** is usually added, because longer sequences always have lower total log-probability.

**Why this, not that? Why sampling, not beam search?**

**Why.** Open-ended text has many good answers. Sampling produces natural, varied text, and RL needs varied samples to explore and to estimate advantages. Beam search concentrates on bland high-likelihood text and returns near-identical beams.

**Notation on this page:** $V$: vocabulary size • $z$: logits • $p$: probabilities (target model in speculative decoding) • $y_t$: the token chosen at step $t$ • $t$: time step or token position

**Trade-off.** Sampling is random, so evaluation needs several runs or pass@k, and a single sample can go off the rails.
**When the alternative wins.** Beam search (or greedy) wins for tasks with one correct output and a good likelihood model: translation, speech recognition, constrained extraction, or short factual answers.

## 7.3 Temperature

**Temperature** $\tau > 0$ divides the logits before the softmax:

$$p_i(\tau) = \frac{e^{z_i/\tau}}{\sum_j e^{z_j/\tau}}.$$

- $\tau = 1$: the model's own distribution.
- $\tau < 1$: sharper. The top token gets more probability. As $\tau \to 0$, this becomes greedy decoding.
- $\tau > 1$: flatter. As $\tau \to \infty$, it becomes uniform over the vocabulary.

**Worked example.**
Logits $z = (2.0,\ 1.0,\ 0.1)$:

| $\tau$ | Probabilities | Entropy (nats) |
|---|---|---|
| 0.5 | (0.864, 0.117, 0.019) | 0.453 |
| 1.0 | (0.659, 0.242, 0.099) | 0.847 |
| 2.0 | (0.502, 0.304, 0.194) | 1.026 |

**Temperature always raises entropy as it goes up.** Write $\beta = 1/\tau$ (the "inverse temperature"), so $p_i \propto e^{\beta z_i}$. A standard calculation (differentiate $H = \log \sum_j e^{\beta z_j} - \beta \sum_i p_i z_i$) gives

$$\frac{dH}{d\beta} = -\beta \operatorname{Var}_p(z) \quad \Longrightarrow \quad \frac{dH}{d\tau} = \frac{\operatorname{Var}_p(z)}{\tau^3} \geq 0,$$

where $\operatorname{Var}_p(z)$ is the variance of the logits under the current distribution. So higher temperature means more randomness, never less.

## 7.4 Truncation: top-k, top-p and min-p

A model's distribution has a long **tail** of thousands of tokens, each with tiny probability. Together they can carry a lot of mass, and sampling one of them can derail a whole response. Truncation methods cut the tail, then renormalise what is left.

**Top-k.** Keep only the $k$ most likely tokens. The problem: a fixed $k$ is too many when the model is confident (one token has 99%) and too few when it is unsure (hundreds of reasonable words).

**Notation on this page:** $\tau$: sampling temperature • $z$: logits • $p$: probabilities (target model in speculative decoding) • $\beta = 1/\tau$: inverse temperature • $d$: hidden size of the model • $k$: number of tokens kept in top-$k$

**Top-p (nucleus sampling).** Sort tokens by probability. Keep the *smallest* set whose total probability is at least $p$. The set adapts: it is small when the model is confident and large when it is unsure. Typical values: 0.9 to 0.95.

**Min-p.** Keep every token whose probability is at least $p_{\min}$ times the probability of the top token. With $p_{\min} = 0.1$, if the top token has 0.6, the threshold is 0.06. This also adapts to confidence, and it behaves well at higher temperatures.

**Why this, not that? Why top-p, not top-k?**

**Why.** Top-p keeps a set of tokens sized to the model's confidence: one or two tokens when it is sure, hundreds when it is not. A fixed $k$ is either too generous in confident steps (letting in junk) or too strict in open steps (cutting good options).
**Trade-off.** Top-p still depends on how flat the tail is. At high temperature the tail fattens, and top-p can let in many poor tokens.
**When the alternative wins.** Top-k is cheaper (no sort) and gives a hard cap, which is useful as a safety bound combined with top-p. Min-p behaves better than top-p at high temperatures, because its cut-off scales with the top token's probability.

**Worked example.**
Sorted probabilities $(0.5, 0.2, 0.15, 0.1, 0.05)$, cumulative $(0.5, 0.7, 0.85, 0.95, 1.0)$.

- **Top-k with $k = 2$:** keep $(0.5, 0.2)$, renormalise to $(0.714, 0.286)$.
- **Top-p with $p = 0.9$:** the cumulative sum first reaches 0.9 at the fourth token $(0.95)$. Keep four tokens and renormalise by 0.95: $(0.526, 0.211, 0.158, 0.105)$.
- **Min-p with $p_{\min} = 0.25$:** the threshold is $0.25 \times 0.5 = 0.125$. Keep $(0.5, 0.2, 0.15)$, renormalise to $(0.588, 0.235, 0.176)$.

**Order of operations.** Libraries usually apply temperature first, then the truncation filters. Since temperature changes the probabilities, it also changes which tokens survive top-p and min-p. Check your library when exact behaviour matters.

Here is a full sampler, written the way you would in an interview:

```
import torch

def sample_next(logits, temperature=1.0, top_k=0, top_p=1.0, min_p=0.0, generator=None):
    """logits: (B, V). Returns sampled token ids (B,)."""
    if temperature == 0:
        return logits.argmax(-1)
    logits = logits / temperature
    if top_k > 0:
        kth = torch.topk(logits, top_k, dim=-1).values[..., -1, None]
        logits = logits.masked_fill(logits < kth, float("-inf"))
    probs = logits.softmax(-1)
    if min_p > 0:
        thresh = min_p * probs.max(-1, keepdim=True).values
        logits = logits.masked_fill(probs < thresh, float("-inf"))
        probs = logits.softmax(-1)
```

```
    if top_p < 1.0:
        sorted_p, idx = probs.sort(-1, descending=True)
        cum = sorted_p.cumsum(-1)
        remove = cum - sorted_p >= top_p      # mass before this token already reaches
↪   top_p
        sorted_p = sorted_p.masked_fill(remove, 0.0)
        probs = torch.zeros_like(probs).scatter(-1, idx, sorted_p)
        probs = probs / probs.sum(-1, keepdim=True)
    return torch.multinomial(probs, 1, generator=generator).squeeze(-1)

# Check top-p on the worked example: 4 tokens survive with p = 0.9.
p = torch.tensor([[0.5, 0.2, 0.15, 0.1, 0.05]])
g = torch.Generator().manual_seed(0)
draws = torch.stack([sample_next(p.log(), top_p=0.9, generator=g) for _ in
↪   range(20000)])
print(torch.bincount(draws.flatten(), minlength=5) / 20000)  # about [0.526, 0.211,
↪   0.158, 0.105, 0]
```

The line `cum - sorted_p >= top_p` keeps a token if the mass *before* it has not yet reached $p$. This always keeps at least the top token, and it includes the token that crosses the threshold. Getting this boundary wrong by one token is the most common bug in hand-written top-p.

## 7.5 Repetition, presence and frequency penalties

These change the logits of tokens that have already appeared, to reduce loops.

- **Repetition penalty** (from the CTRL paper; used in Hugging Face and vLLM): for every token already in the context, divide its logit by $\alpha > 1$ if the logit is positive, or multiply by $\alpha$ if negative. Either way the token becomes less likely. Typical values are 1.0 (off) to about 1.2.
- **Presence penalty** (OpenAI style): subtract a fixed amount from the logit of any token that has appeared at least once.
- **Frequency penalty**: subtract an amount proportional to how many times the token has appeared.

> **Common mistake.**
> Penalties change the distribution you sample from. For RL rollouts that breaks the link between "the tokens we sampled" and "the policy we are training" (see below). Some models ship with a non-default penalty in their `generation_config.json`, so you may be using one without knowing it.

## 7.6 Stop conditions, length limits and log-probabilities

Generation stops when one of these happens:

- The model emits a **stop token** (EOS or end-of-turn). A model may have several.
- A **stop string** appears, for example `"</answer>"` or `"\nUser:"`.

- The output reaches the **maximum number of new tokens**.

Know the difference between *max new tokens* (`max_new_tokens` in Hugging Face, `max_tokens` in vLLM and OpenAI-style APIs) and a limit on the *total* length including the prompt (`max_length` in Hugging Face). In RL, a response that hits the limit is **truncated**: it has no final answer, so a correctness reward is usually 0. How you treat truncated samples matters: DAPO, for example, masks or softly penalises overlong samples (Chapter 28).

Most engines can also return **log-probabilities** of the sampled tokens (and of the top alternatives). RL frameworks use these "rollout log-probs" as $\log \pi_{\theta_{\text{old}}}$, or compare them with the trainer's own log-probs to detect mismatch (Chapter 43).

## 7.7 Defaults you did not choose

Sampling settings can come from three places: the library's built-in defaults, the model's `generation_config.json`, and your own arguments. They silently stack.

**Worked example.**
Checked with `transformers` 5.20.2 and `trl` 1.16.0 (October 2026):

- In `transformers`, fields you leave unset fall back to built-in defaults, which include `do_sample=False`, `max_length=20`, `temperature=1.0`, `top_k=50` and `top_p=1.0`. So if you turn on sampling and say nothing else, you get **top-k = 50**, not the model's full distribution. And without `max_new_tokens`, outputs are cut at 20 tokens in total.
- The `generation_config.json` of `Qwen/Qwen2.5-0.5B-Instruct` sets `do_sample=true`, `temperature=0.7`, `top_k=20`, `top_p=0.8` and `repetition_penalty=1.1`. These are applied whenever you call `generate()` without overriding them.
- TRL's `GRPOConfig` defaults to `temperature=1.0`, `top_p=1.0`, `top_k=0` (top-k off) and `repetition_penalty=1.0` for its rollouts.

Always set sampling parameters explicitly, and print the final generation config once at the start of a run.

## 7.8 Sampling for RL rollouts

This section is where sampling meets the theory of Part III.

**The policy-gradient estimator assumes the samples come from the policy being trained.** The log-derivative trick (Chapter 2) gives $\nabla J = \mathbb{E}_{y\sim\pi_\theta}[r(y)\,\nabla \log \pi_\theta(y)]$. If the rollouts actually come from a modified distribution $q$ (with top-k, top-p, penalties or a different temperature), then:

- The samples are from $q$, not $\pi_\theta$. Without importance weights $\pi_\theta/q$ the gradient is biased.
- If you compute $\log \pi_\theta$ in the trainer with *plain* logits but sampled with temperature $\tau$, then your "old" and "new" log-probabilities describe different distributions, so the PPO/GRPO importance ratio is wrong even on the very first step.

Common practice follows from this:

**Notation on this page:** $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\theta$: the model's trainable weights • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\nabla_\theta$: gradient with respect to $\theta$ • $y_t$: the token chosen at step $t$ • $q$: draft-model distribution • $\tau$: sampling temperature

1. **Sample rollouts at temperature 1.0 with no truncation** (top-p 1.0, top-k off, no penalties). Then the sampling distribution *is* $\pi_\theta$.
2. **If you do use a temperature** $\tau \neq 1$, compute the training log-probabilities from `logits / tau` as well, so both sides describe the same distribution. TRL's GRPO trainer, for example, divides logits by the configured temperature when computing log-probabilities.
3. **Avoid truncation** (top-p, top-k, min-p) in rollouts unless you know how your framework handles it. Truncated tokens have zero probability under the sampler but non-zero under the policy, so the two distributions do not match.
4. **Temperature controls exploration.** Higher temperature gives more diverse samples and more varied rewards within a group, which helps group-based methods like GRPO. Too high and samples become incoherent.

**Why this, not that? Why temperature 1.0 for RL rollouts, not a lower temperature?**

**Why.** At temperature 1.0 with no truncation, the samples come from exactly the policy being trained, so the policy gradient and the importance ratios are correct without any correction. It also gives the most exploration.
**Trade-off.** Some samples will be poor, which costs generation compute, and very long responses at high entropy can drift into nonsense.
**When the alternative wins.** A slightly higher temperature (1.0 to 1.2) can help when entropy collapses. A lower temperature can be used if the trainer computes log-probabilities with the same temperature (TRL does this), and recipes such as DeepSeek-V3.2 even keep the top-p sampling mask during training so both sides use the same truncated distribution.

**Evaluation sampling is a separate choice.** For evaluation you use whatever settings the model is meant to be served with. For example, DeepSeek recommended a temperature around 0.6 with top-p 0.95 for its R1 reasoning models, partly to avoid endless repetition that greedy decoding can cause. Report the sampling settings with every evaluation number, and when you sample, average over several seeds or use pass@k (Chapter 37).

## 7.9 Speculative decoding

Decoding is memory bound (Chapter 5): each step reads all the weights to produce one token. **Speculative decoding** uses a small, fast **draft** model $q$ to propose $k$ tokens. The big **target** model $p$ then checks all $k$ in one forward pass, which costs about the same as one normal step.

Each proposed token $x$ is accepted with probability $\min(1, p(x)/q(x))$. At the first rejection, a replacement token is sampled from the leftover distribution

$$p'(x) = \frac{\max(0,\ p(x) - q(x))}{\sum_{x'} \max(0,\ p(x') - q(x'))}$$

and the rest of the draft is thrown away. This rule is exact: the output has *exactly* the target distribution $p$, whatever the draft model is. A good draft is accepted more often, so you get more tokens per expensive step. Variants use extra prediction heads (Medusa, EAGLE) or the model's own early layers instead of a separate draft model. Rollout engines can use speculative decoding to speed up RL generation, as long as the output distribution is unchanged.

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\tau$: sampling temperature • $q$: draft-model distribution • $k$: number of tokens kept in top-$k$ • $p$: probabilities (target model in speculative decoding) • $x$: a token

```
import torch

torch.manual_seed(0)
p = torch.tensor([0.5, 0.3, 0.2])   # target
q = torch.tensor([0.2, 0.2, 0.6])   # draft

def speculative_one_token():
    x = torch.multinomial(q, 1).item()
    if torch.rand(()) < torch.clamp(p[x] / q[x], max=1.0):
        return x                                    # accept the draft token
    residual = torch.clamp(p - q, min=0.0)
    return torch.multinomial(residual / residual.sum(), 1).item()   # resample

counts = torch.bincount(torch.tensor([speculative_one_token() for _ in range(30000)]),
 ↪  minlength=3)
print(counts / counts.sum())   # about [0.5, 0.3, 0.2] = p, not q
```

## 7.10 What truncation does to the probabilities RL sees

Top-k, top-p and min-p change the distribution you sample from. For RL this matters, because the policy gradient and importance ratios assume we know the probability of each sampled token under the policy that produced it.

Let $p$ be the full softmax distribution at temperature $T$ and let $S$ be the kept set (the nucleus). Top-p sampling draws from the **truncated, renormalised** distribution

$$q(a) = \frac{p(a)\,\mathbf{1}[a \in S]}{M}, \qquad M = \sum_{a \in S} p(a),$$

where $M$ is the probability mass of the nucleus. Three consequences follow.

1. **The sampled tokens have** $\log q(a) = \log p(a) - \log M$**.** If the trainer recomputes $\log p$ (the full softmax) but the data came from $q$, every token's ratio is off by the factor $M$, which changes from step to step.
2. $\mathrm{KL}(q \,\|\, p) = -\log M$**.** With $M = 0.9$ this is about 0.105 nats per token, already larger than the KL budget many RL runs target.
3. **Tokens outside** $S$ **are never sampled**, so the policy gradient never pushes them down (or up). The training signal is biased toward the head of the distribution, and exploration of rare continuations disappears.

**Worked example.** A next-token distribution over 50 tokens; top-p 0.9 keeps a subset with mass $M$ just above 0.9. The snippet measures $M$, the size of the kept set, the per-token KL, and the constant log-ratio offset.

```
import torch
torch.manual_seed(0)
logits = 2.0 * torch.randn(50)
p = torch.softmax(logits, -1)
```

**Notation on this page:** $p$: probabilities (target model in speculative decoding) • $T$: number of steps or tokens in a sequence • $q$: draft-model distribution • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$

```
def top_p_dist(p, top_p):
    sp, idx = p.sort(descending=True)
    keep = (sp.cumsum(0) - sp) < top_p          # keep tokens until the mass before them
 ↪  reaches top_p
    q = torch.zeros_like(p); q[idx[keep]] = sp[keep]
    return q / q.sum(), sp[keep].sum().item(), int(keep.sum())
for tp in [1.0, 0.95, 0.9, 0.7]:
    q, M, k = top_p_dist(p, tp)
    S = q > 0
    kl = (q[S] * (q[S].log() - p[S].log())).sum().item()
    print(f"top_p {tp:4.2f}: kept {k:2d}/50 tokens, mass M {M:.3f}, KL(q||p)
     ↪  {abs(kl):.4f} = -log M {abs(torch.log(torch.tensor(M)).item()):.4f}, "
          f"log q - log p on sampled tokens {abs((q[S].log() -
           ↪  p[S].log())[0].item()):.4f}")
```

The equality $\mathrm{KL}(q\|p) = -\log M$ holds exactly, and the log-ratio offset is the same constant for every kept token. The practical rules: sample RL rollouts at temperature 1 with top-p 1.0, top-k off and min-p off (TRL 1.16 GRPOConfig defaults: `temperature=1.0`, `top_p=1.0`, `top_k=0`, `min_p=None`); if you must truncate, make the trainer use the same truncated distribution, or ask the engine for **processed** log-probabilities (vLLM's `logprobs_mode="processed_logprobs"`; the default `raw_logprobs` is before temperature and truncation) and correct with importance sampling (Chapter 43).

**Why this, not that? Why sample RL rollouts from the untruncated distribution, not with the chat defaults?**

**Reason.** RL maths assumes the samples come from the policy being trained; truncation creates a different behaviour policy with a step-dependent offset and zero mass on the tail, which biases gradients and kills exploration. **Trade-off.** Untruncated sampling at temperature 1 sometimes produces junk tokens from the tail, which wastes rollouts, especially for small or weak models. **When the alternative is better.** For evaluation and serving, where only output quality matters and no gradient is computed, chat defaults with top-p or min-p are fine and often better. Some recipes also truncate in RL deliberately and correct for it with the matching processed log-probs.

## 7.11 Many samples per question: self-consistency and best-of-n

Sampling several answers and combining them is the simplest way to spend more inference compute. **Self-consistency** (Wang et al., 2203.11171) samples $n$ reasoning paths and takes the **majority vote** over the final answers. **Best-of-n** picks the answer with the highest reward-model score. **pass@n** asks whether *any* of the $n$ samples is correct; it needs a verifier and is an upper bound on what selection can achieve.

When does majority voting help? Let the correct answer have probability $c$ per sample, and the most likely wrong answer probability $w$. As $n$ grows, the vote converges to whichever answer has the highest probability. So voting helps when $c > w$, even if $c < 0.5$, and it hurts when one wrong answer is more likely than the right one.

**Notation on this page:** $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $p$: probabilities (target model in speculative decoding) • $q$: draft-model distribution • $n$: number of samples

```
import torch
torch.manual_seed(1)
def vote_acc(probs, n, trials=20000):
    draws = torch.multinomial(torch.tensor(probs).repeat(trials, 1), n,
 ↪  replacement=True)
    counts = torch.nn.functional.one_hot(draws, len(probs)).sum(1).float()
    counts += 1e-3 * torch.rand_like(counts)                  # random tie-break
    return (counts.argmax(1) == 0).float().mean().item()      # answer 0 is the correct
     ↪  one
spread = [0.4] + [0.1] * 6           # correct 0.4, six wrong answers at 0.1 each
concentrated = [0.4, 0.45, 0.15]     # one wrong answer is more likely than the right one
for n in [1, 4, 16, 64]:
    pass_n = 1 - (1 - 0.4) ** n
    print(f"n={n:2d}: vote acc (spread) {vote_acc(spread, n):.3f} | vote acc
     ↪  (concentrated) {vote_acc(concentrated, n):.3f} | pass@n {pass_n:.3f}")
```

With spread-out errors, voting climbs from 0.40 toward 1.0. With a concentrated wrong answer, it does not improve and then falls (0.33 at $n = 64$, toward 0 as $n$ grows) even though pass@n climbs toward 1. This is why RL that sharpens the model can raise majority-vote accuracy (maj@n) while pass@n at large $n$ does not improve or even drops (Chapter 30): RL moves mass onto answers the model already favoured.

## 7.12 Reproducibility and seeds

A per-request seed (`SamplingParams.seed` in vLLM, `torch.manual_seed` with `generate` in Transformers) makes sampling repeatable *in principle*. In practice, results can still change between runs, because batched GPU kernels are not **batch-invariant**: the same request computes slightly different logits depending on what else is in the batch, and sampling turns small logit differences into different tokens. For RL this is one of the sources of the mismatch between rollout and trainer log-probabilities. For evaluation, report the mean over several seeds rather than trusting one run.

## 7.13 Common mistakes

- **Leaving top-p, top-k or a repetition penalty on for RL rollouts** while the trainer computes plain softmax log-probs: a silent behaviour-policy mismatch.
- **Reading vLLM's default `raw_logprobs` as the sampling distribution** when temperature or truncation was applied.
- **Reporting one greedy or one sampled run** as a model's accuracy; use several samples or seeds and report the spread.

**Interview check**

**Q1. Implement top-p sampling.** Sort probabilities in descending order, take the cumulative sum, drop tokens where the cumulative mass *before* them is already at least $p$, renormalise, and sample. (See the code above.)

**Notation on this page:** $n$: number of samples • $p$: probabilities (target model in speculative decoding)

**Q2. What does temperature do mathematically? What happens as it goes to 0 and to infinity?** Divides logits by $\tau$ before the softmax. $\tau \to 0$ gives argmax (greedy); $\tau \to \infty$ gives uniform. Entropy increases with $\tau$, since $dH/d\tau = \mathrm{Var}_p(z)/\tau^3$.
**Q3. Why might top-p be better than top-k?** Top-p adapts the number of kept tokens to the model's confidence; a fixed $k$ is too many when the model is sure and too few when it is not.
**Q4. You sample GRPO rollouts with temperature 0.7 but compute log-probs from raw logits. What goes wrong?** The rollout distribution and the trained distribution differ, so importance ratios are not 1 even before any update, and the gradient is biased. Divide logits by 0.7 in the trainer too, or sample at 1.0.
**Q5. Why is beam search rarely used for chat models?** It finds high-likelihood text, which tends to be generic and repetitive for open-ended tasks; it is also costly and gives no diversity. Sampling gives more natural text and is required for RL exploration.
**Q6. Why does speculative decoding not change the output distribution?** Accepting with probability $\min(1, p/q)$ and resampling rejections from the normalised $\max(0, p - q)$ makes the total probability of each token exactly $p(x)$.
**Q7. Name three sources of sampling settings that can stack without you noticing.** Library defaults (for example top-k 50 in Hugging Face), the model's `generation_config.json`, and the arguments you pass (or your framework passes) at call time.
**Q8. Top-p keeps nucleus mass M. What is KL(q‖p) between the truncated and full distributions?** Exactly $-\log M$; the sampled tokens' log-probs differ by the constant $-\log M$ from the full softmax.
**Q9. When does majority voting hurt?** When one wrong answer is more likely than the correct one; the vote then converges to the wrong answer as $n$ grows.
**Q10. Why can identical seeds give different samples in a serving engine?** Batched kernels are not batch-invariant: logits change slightly with batch composition, and sampling amplifies the difference.

**Key takeaways**

- Greedy and beam search are deterministic; sampling with temperature, top-k, top-p and min-p is stochastic.
- Temperature rescales logits; higher temperature always means higher entropy.
- Top-p and min-p adapt to the model's confidence; top-k does not.
- Penalties and truncation change the sampling distribution.
- For RL, sample from the true policy (temperature 1, no truncation) or make the trainer's log-probs match the sampler exactly.
- Always set sampling parameters explicitly; library and model defaults stack silently.
- Speculative decoding speeds up generation without changing the output distribution.
- Truncation gives a different behaviour policy: $q = p\mathbf{1}_S/M$, $\mathrm{KL}(q\|p) = -\log M$; sample RL rollouts untruncated or correct for it.
- Majority voting converges to the most likely answer: good when errors are spread, bad when one wrong answer dominates.
- Seeds do not guarantee reproducibility under batching; average over several runs.

**Notation on this page:** $\tau$: sampling temperature • $z$: logits • $p$: probabilities (target model in speculative decoding) • $d$: hidden size of the model • $k$: number of tokens kept in top-$k$ • $q$: draft-model distribution • $x$: a token • $n$: number of samples • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$

PART II

# Supervised Fine-Tuning

CHAPTER 8

# Instruction Tuning and Chat Data

**Why this chapter matters.**
Supervised fine-tuning (SFT) is the first post-training stage in almost every pipeline, and it is the stage most teams run most often. Most SFT failures are data failures: the wrong format, a broken chat template, a missing end token, or data that teaches the wrong habits. This chapter covers what SFT data looks like, how chat templates turn conversations into tokens, and how to build a good dataset.

## 8.1 What SFT changes in a model

**Supervised fine-tuning** trains a pretrained model on examples of the behaviour we want. Each example is a prompt and a good response. The loss is the same next-token cross-entropy as pretraining (Chapter 3), usually computed only on the response tokens (Chapter 9).

A useful way to think about it: **SFT mostly teaches format and style, not new knowledge.** The base model already knows a lot from pretraining. SFT teaches it:

- to answer instead of continuing the text,
- the chat format: where a turn starts and ends, and when to stop,
- a house style: tone, length, use of lists, refusal style,
- skills such as calling tools in a fixed format or writing a chain of thought before an answer.

This idea is sometimes called the **superficial alignment hypothesis**. It was made popular by *LIMA: Less Is More for Alignment* (Zhou et al., arXiv 2305.11206), which fine-tuned a 65B model on only 1,000 carefully chosen examples and got a strong assistant. The lesson: a small set of excellent, diverse examples can beat a large set of average ones.

SFT can still add knowledge, but it does so inefficiently and with side effects. Training on facts the model does not know tends to teach it to *state things confidently whether or not it knows them*, which increases hallucination. New knowledge is better added in pretraining or mid-training, or given at inference time (retrieval).

**Why this, not that? Why fine-tune, not just prompt the model?**

**Why.** A fine-tuned model follows the format every time without a long prompt, which cuts inference cost and latency. It can learn behaviours that are hard to describe in words (a house style, a tool format, a refusal policy), and it can be smaller than a prompted general model with the same task quality.
**Trade-off.** Fine-tuning needs data, compute, evaluation and a serving pipeline for the new weights. It can damage general abilities (catastrophic forgetting, Chapter 9), and every base-

model upgrade means training again.
**When the alternative wins.** Prompting (with few-shot examples or retrieval) wins when the task changes often, when you have little data, when you need the newest base model the day it is released, or when the base model already does the task well enough.

## 8.2 Data formats

Training libraries accept a few standard formats. TRL (the Hugging Face post-training library) uses these names, and most other libraries use something similar.

**Language modelling (plain text).** One field of raw text. The loss covers every token. Used for continued pretraining on domain text.

```
{"text": "The mitochondria is the powerhouse of the cell. ..."}
```

**Prompt-completion.** A prompt and the desired completion. The loss usually covers only the completion.

```
{"prompt": "Translate to French: Good morning", "completion": "Bonjour"}
```

**Conversational ("messages").** A list of turns, each with a `role` and `content`. Roles are usually `system`, `user`, `assistant` and `tool`. This is the standard format for chat models.

```
{"messages": [
  {"role": "system", "content": "You are a concise assistant."},
  {"role": "user", "content": "What is 2+2?"},
  {"role": "assistant", "content": "4"}
]}
```

**Tool calls.** The assistant turn contains a structured call, and a `tool` turn carries the result back. The exact field names depend on the model's template, but the common shape is:

```
{"messages": [
  {"role": "user", "content": "Weather in Pune?"},
  {"role": "assistant", "tool_calls": [{"type": "function",
     "function": {"name": "get_weather", "arguments": {"city": "Pune"}}}]},
  {"role": "tool", "name": "get_weather", "content": "{\"temp_c\": 29}"},
  {"role": "assistant", "content": "It is 29 °C in Pune right now."}
]}
```

Tools are described to the model with a JSON schema (name, description, parameters), which the template inserts into the system prompt.

**Preference data** (for DPO and reward models, Parts IV and V) adds two responses: `{"prompt": ..., "chosen": ..., "rejected": ...}`. **Unpaired** preference data (for KTO) has a single completion and a boolean `label`.

## 8.3 Chat templates

A model does not see JSON. It sees one long string of tokens. A **chat template** is the rule that turns a list of messages into that string. Templates are written in **Jinja** (a small templating language) and stored with the tokenizer (`tokenizer.chat_template`). You apply them with `tokenizer.apply_chat_template(messages, ...)`.

Here is the real output for Qwen2.5-0.5B-Instruct (checked with `transformers` 5.20.2). Its template uses the **ChatML** style:

```
from transformers import AutoTokenizer
tok = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B-Instruct")
msgs = [{"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "What is 2+2?"},
        {"role": "assistant", "content": "4"}]
print(tok.apply_chat_template(msgs, tokenize=False))
```

```
<|im_start|>system
You are a helpful assistant.<|im_end|>
<|im_start|>user
What is 2+2?<|im_end|>
<|im_start|>assistant
4<|im_end|>
```

Three options matter most:

- `tokenize=False` returns the string; `tokenize=True` returns token IDs (with `return_dict=True` you also get the attention mask).
- `add_generation_prompt=True` appends the start of an assistant turn (`<|im_start|>assistant\n`) so the model knows it should answer now. Use it at **inference** and for **RL prompts**. Do not use it when you format a full training conversation that already contains the answer.
- `continue_final_message=True` leaves the last message open, so the model continues it (useful for prefilling a response).

Other families use other markers. Llama 3 uses `<|start_header_id|>user<|end_header_id|>` and ends turns with `<|eot_id|>`. Reasoning models often add a `<think>` block before the answer, and some templates prefill it, so the model's output *starts inside* the reasoning block.

### 8.3.1 Template behaviours that surprise people

These were checked on real templates and are common sources of bugs:

1. **Hidden default system prompts.** If you leave out the system message, Qwen2.5's template inserts its own: "You are Qwen, created by Alibaba Cloud. You are a helpful assistant." Your training data and your inference prompts must agree on this.
2. **Reasoning is dropped from earlier turns.** Some reasoning-model templates (for example, the DeepSeek-R1-Distill template shipped with those checkpoints) delete the `<think>...</think>` part of *previous* assistant turns when they render a multi-turn conversation. If

you SFT on multi-turn reasoning data with such a template, the earlier reasoning silently disappears from the training text.

3. **Templates may not be prefix-preserving.** A template is **prefix-preserving** if adding a message never changes how earlier messages are rendered. Templates that strip old reasoning are not. Multi-turn RL code that builds the conversation turn by turn as token IDs (Chapter 34) needs a prefix-preserving template, or the tokens it trains on will not match what the model saw. TRL ships patched "training" versions of common templates for this reason.
4. **Assistant masks need special tags.** `apply_chat_template(..., return_assistant_tokens_mask=True)` only works if the template wraps assistant output in `{% generation %}...{% endgeneration %}`. Most templates shipped with models do not. With Qwen2.5's stock template the returned mask is all zeros, and `transformers` only prints a warning. TRL's `SFTTrainer` swaps in a patched template when you set `assistant_only_loss=True`, and warns if the end-of-turn token would fall outside the mask.

**Common mistake.**
**Use the template the model was trained with, exactly.** Using Llama's template on a Qwen model, adding your own "### Instruction:" format to an instruct model, or forgetting `add_generation_prompt=True` at inference all give a model that seems strangely worse. For RL, the prompts you sample from must be formatted exactly as they will be at inference.

### 8.3.2 The usual template bugs

| Symptom | Likely cause |
|---|---|
| Model never stops; rambles into a fake next "User:" turn | End-of-turn token missing from labels, or wrong EOS in `generation_config` |
| Two BOS tokens at the start | Template adds BOS and the tokenizer adds it again (`add_special_tokens=True`) |
| Model replies with the role name, e.g. "assistant: ..." | `add_generation_prompt` missing at inference, or mismatched template |
| Fine-tuned model ignores the system prompt | Training data never had system prompts, but inference does (or the other way round) |
| Good loss, bad multi-turn behaviour | Template rewrote earlier turns (dropped reasoning) differently in training and inference |

## 8.4 Where SFT data comes from

**Human-written data.** Expert annotators write responses to prompts. High quality, but slow and expensive. Usually used for small, high-value sets (safety, style, hard domains).

**Synthetic data from a stronger model.** The most common source today. A strong "teacher" model writes responses; filters keep the good ones. This is a form of distillation (Chapter 12). Check the teacher model's licence: some forbid using outputs to train other models.

**Self-generated data.**

- *Self-Instruct* (Wang et al., arXiv 2212.10560) starts from a few seed tasks and has the model write new instructions and answers, then filters them.
- *Evol-Instruct*, introduced in *WizardLM* (Xu et al., arXiv 2304.12244), asks a model to rewrite instructions to be harder (add constraints, deepen, add reasoning steps) or broader, building a dataset that climbs in difficulty.
- **Rejection sampling** (also called best-of-$n$ SFT or RFT): sample many answers from the current model, keep only those that pass a checker (correct maths answer, passing tests) or score highly with a reward model, and fine-tune on them. This is a cheap, stable cousin of RL and is used as a stage in the DeepSeek-R1 recipe (Chapter 30).

### 8.4.1 What makes SFT data good

- **Diversity of prompts**: many task types, topics, lengths and difficulty levels. Duplicates and near-duplicates waste compute and cause memorisation.
- **Quality of responses**: correct, well-formatted, consistent style. A few bad examples teach bad habits.
- **Matching the target distribution**: if users will send long messy prompts, train on long messy prompts.
- **Decontamination**: remove anything that overlaps with your evaluation sets (Chapter 39).
- **Balance**: the mix of domains decides what the model gets good at. Too much of one domain (say, maths) can hurt chat quality.

**Why this, not that? Why a small, high-quality dataset, not a large average one?**

**Why.** SFT mostly teaches format and style, which a few thousand clean examples can do. Every low-quality example also teaches its flaws: wrong facts, verbosity, bad formatting. Smaller sets are cheaper to check by hand and to train on.
**Trade-off.** Small sets cover fewer skills and domains, and they can overfit quickly. Curating them takes expert time.
**When the alternative wins.** Large datasets win for broad skills that need many examples: code across many languages, maths with many problem types, multilingual chat, tool use with many APIs. Large synthetic sets with strong filtering (correctness checks, reward-model scoring) are how most current open models do SFT.

**Why this, not that? Why synthetic data from a strong model, not human-written data?**

**Why.** It is far cheaper and faster, it scales to millions of examples, and a strong model often writes better and more consistent answers than average annotators, especially for code and maths where outputs can be checked automatically.
**Trade-off.** The student inherits the teacher's mistakes, biases and style. Training repeatedly on model outputs can reduce diversity. Licences may restrict use.
**When the alternative wins.** Human data wins for new domains no model handles well, for evaluation sets, for safety policy examples, for capturing real user needs, and for preference labels where you want human judgement rather than a model's.

**Notation on this page:** $n$: number of samples

## 8.5 Multi-turn and system prompts

Real conversations have many turns. Train on multi-turn data if you want multi-turn ability: later turns must use information from earlier turns, follow corrections, and keep the persona consistent.

In multi-turn SFT, the loss is usually applied to **every assistant turn**, not only the last one (Chapter 9 shows how). Each assistant turn is then a training target, with all earlier turns as context.

**System prompts** set the persona, rules, tools and output format. If the deployed model will receive system prompts, train with a variety of them, including ones that conflict with the user. Some models are trained with an explicit **instruction hierarchy**: system instructions beat user instructions, which beat instructions found inside tool outputs or documents. This matters a lot for agents, where a web page or tool result might contain an injected instruction.

## 8.6 What actually changes: token distribution shift

If SFT mostly teaches style, the token-level evidence should show it. *The Unlocking Spell on Base LLMs: Rethinking Alignment via In-Context Learning* (Lin et al., arXiv 2312.01552, the URIAL paper) compared base and aligned models token by token: at each position of an aligned model's answer, they asked where that token ranked under the base model given the same prefix. For most positions the base model's top choice was the same token; the positions where the two models disagreed were mostly **stylistic**: discourse markers ("Hello", "Sure", "However"), formatting, transition phrases and safety disclaimers, and they were concentrated early in the response. The paper also showed that a base model prompted with a few carefully written examples and a system-style preamble (no fine-tuning, "URIAL") gets close to aligned models on many prompts.

What this means in practice:

- **SFT changes few tokens, but they are the ones that steer.** The first tokens of a response decide its format and approach; once the model has written "Let's solve this step by step", the base model's knowledge does the rest.
- **Knowledge is not added cheaply.** Facts absent from pre-training are learned slowly by SFT and can increase hallucination if forced (the model learns to produce confident answers it cannot ground).
- **Long reasoning SFT is the exception.** Distilled chains of thought change thousands of tokens per response and teach genuinely new behaviour (long, self-checking reasoning), which is why reasoning SFT needs far more data and compute than chat SFT.

## 8.7 The data mixture is a hyperparameter

Real SFT sets combine many sources: general chat, maths, code, safety, multilingual, tool use. How much of each the model sees matters as much as any learning rate. A common way to set proportions is **temperature-scaled sampling**: with $n_i$ examples in source $i$, sample from it with

**Notation on this page:** $n$: number of samples

probability

$$p_i = \frac{n_i^{1/\tau}}{\sum_j n_j^{1/\tau}}.$$

$\tau = 1$ samples in proportion to size (large sources dominate); $\tau \to \infty$ samples every source equally (small sources are repeated many times); $\tau$ between 2 and 5 is a typical compromise. The number of **epochs** each source receives is then $p_i \cdot N_{\text{total}}/n_i$, and that is what to watch: a small source repeated 20 times will be memorised.

**Worked example.** Three sources: chat 900k, maths 90k, safety 10k examples, and a budget of 1M training examples. With $\tau = 1$: proportions 90%, 9%, 1%, each source seen once. With $\tau = 3$: $n^{1/3} = 96.5, 44.8, 21.5$, so proportions 59%, 28%, 13%: chat is seen 0.66 times (subsampled), maths 3.1 times, safety 13 times. That much repetition of 10k safety examples risks memorised refusals and over-refusal; capping at a few epochs per source is a sensible extra rule.

```
def mixture(sizes, tau, total, max_epochs=None):
    w = {k: n ** (1 / tau) for k, n in sizes.items()}
    p = {k: v / sum(w.values()) for k, v in w.items()}
    if max_epochs:                                  # cap repetition, give the excess
     ↪ back to the others
        for _ in range(10):
            capped = {k: min(p[k], max_epochs * sizes[k] / total) for k in p}
            free = 1 - sum(capped.values())
            open_ = [k for k in p if capped[k] < max_epochs * sizes[k] / total]
            if free < 1e-9 or not open_: p = capped; break
            share = sum(capped[k] for k in open_)
           p = {k: capped[k] + (free * capped[k] / share if k in open_ else 0) for k in p}
    return {k: (round(p[k], 3), round(p[k] * total / sizes[k], 2)) for k in p}
sizes = {"chat": 900_000, "maths": 90_000, "safety": 10_000}
for tau, cap in [(1, None), (3, None), (3, 4)]:
    print(f"tau={tau} cap={cap}: (proportion, epochs) =", mixture(sizes, tau, 1_000_000,
     ↪ cap))
```

With the cap, safety is limited to 4 epochs and the freed budget goes back to the other sources.

**Why this, not that? Why temperature-scaled mixing with an epoch cap, not proportional to dataset size?**

**Reason.** Proportional sampling lets the largest source (usually generic chat) drown out small, valuable ones (maths, safety, tool use); temperature scaling raises small sources, and the epoch cap prevents memorising them. **Trade-off.** It adds hyperparameters ($\tau$, caps) that interact with evaluation results, and the best mixture depends on the model and the target uses; finding it takes ablations. **When the alternative is better.** When sources are already curated to the desired balance, or when one domain is the whole point (a maths-only model), proportional or hand-set weights are simpler.

## 8.8 Choosing examples: from heuristics to influence

Beyond deduplication and quality filters (Chapter 39), two selection ideas recur in interviews:

**Notation on this page:** $\tau$: a trajectory (states, actions, rewards of one episode) • $p$: a probability distribution (often the target or data) • $n$: number of samples • $N$: number of examples

- **Quality and diversity scores.** Score each example for complexity and quality with a model, embed examples, and pick high-scoring examples that are far from those already chosen. *What Makes Good Data for Alignment?* (Liu et al., arXiv 2312.15685) is a careful study of this approach.
- **Gradient-based influence.** Estimate how much training on an example would reduce the loss on a small target set by comparing gradients (a first-order Taylor approximation: the loss change is about $-\eta \nabla \ell_{\text{target}} \cdot \nabla \ell_{\text{example}}$). *LESS: Selecting Influential Data for Targeted Instruction Tuning* (Xia et al., arXiv 2402.04333) does this with LoRA gradients projected to a low dimension, and reports that a small selected fraction of a large instruction set can match or beat training on all of it for the targeted tasks.

The second idea explains the first: an example is useful when its gradient points the same way as the gradients of the behaviour you want.

## 8.9 Common mistakes

- **Letting one large source dominate the mixture** without checking per-source epochs.
- **Repeating small sources many times**: memorised safety refusals become over-refusal.
- **Expecting SFT to add knowledge reliably**: it mostly changes style and format; forcing unknown facts can increase hallucination.

**Interview check**

**Q1. What does SFT mainly teach a pretrained model?** Format and behaviour: answering instead of continuing, the chat template and when to stop, style, tool-call formats and step-by-step reasoning. Most knowledge comes from pretraining.
**Q2. What does `add_generation_prompt=True` do and when should you use it?** It appends the opening of an assistant turn so the model starts answering. Use it at inference and when formatting RL prompts; not when formatting a complete training conversation.
**Q3. Your fine-tuned model never stops generating. What do you check?** That every assistant turn ends with the correct end-of-turn token, that this token is inside the loss mask, that the pad token is not masking it, and that `generation_config.eos_token_id` includes it.
**Q4. What did LIMA show?** A strong base model fine-tuned on only 1,000 carefully curated examples can be a competitive assistant: data quality and diversity matter more than quantity for alignment-style SFT.
**Q5. What is rejection-sampling fine-tuning?** Sample several responses per prompt from the current model, keep only those that pass a verifier or score highly with a reward model, and SFT on them. It is cheap, stable and often a strong baseline for RL.
**Q6. Why does a prefix-preserving template matter for multi-turn RL?** Multi-turn RL builds the conversation turn by turn as token IDs. If the template re-renders earlier turns differently when a new turn is added (for example, by dropping old reasoning), the trained tokens no longer match what the model saw when it generated them.
**Q7. Why can SFT on new facts increase hallucination?** The model learns to produce confident answers to questions it cannot answer from its own knowledge, so it learns the habit of answering confidently without knowing.

**Q8. What did the URIAL paper find about base versus aligned models?** They agree on the top token at most positions; the differences are mostly stylistic tokens early in the response, and a well-prompted base model gets close to aligned models.
**Q9. How does temperature-scaled data mixing work?** Sample source $i$ with probability proportional to $n_i^{1/\tau}$; $\tau = 1$ is proportional to size, large $\tau$ approaches uniform.
**Q10. What is the first-order influence of a training example on a target loss?** About $-\eta \nabla \ell_{\text{target}} \cdot \nabla \ell_{\text{example}}$: examples whose gradients align with the target's gradients reduce its loss.

**Key takeaways**

- SFT trains on prompt-response examples with next-token loss on the response; it mostly teaches format and behaviour.
- Standard formats: plain text, prompt-completion, conversational messages (with tool calls), and preference pairs.
- Chat templates turn messages into tokens. Use the model's own template and `add_generation_prompt=True` at inference.
- Watch for hidden default system prompts, templates that drop old reasoning, non-prefix-preserving templates and missing assistant masks.
- Good SFT data is diverse, high-quality, matched to the target use, balanced and decontaminated. Rejection sampling is a strong, cheap data source.
- SFT shifts few tokens (mostly style, early in the response), except for long reasoning SFT.
- Set mixtures with temperature scaling plus per-source epoch caps; watch repetition of small sources.
- Data selection: quality and diversity scores, or gradient influence (LESS).

**Notation on this page:** $\tau$: a trajectory (states, actions, rewards of one episode) • $n$: number of samples • $\nabla_\theta$: gradient with respect to $\theta$ • $\eta$: learning rate

CHAPTER 9

# The SFT Loss in Detail

**Why this chapter matters.**
The SFT loss is one line of maths, but implementing it correctly needs care: which tokens to train on, how to average over tokens and sequences, how to pack many short examples into one row without them leaking into each other, and how to avoid destroying what the model already knows. The same questions (masking and normalisation) come back in DPO and in RL, so getting them right here pays off later.

## 9.1 Token-level cross-entropy with a mask

Let a training example be a token sequence $s = (s_1, \dots, s_T)$ made from the formatted conversation. Let $m_t \in \{0, 1\}$ be a **loss mask**: $m_t = 1$ if token $t$ should be learned, and 0 otherwise. The SFT loss for one sequence is

$$\mathcal{L}(\theta; s) = -\frac{1}{\sum_t m_t} \sum_{t=2}^{T} m_t \log \pi_\theta(s_t \mid s_{<t}).$$

Every masked-in token contributes $-\log$ (probability of the correct token), exactly as in pretraining. Masked-out tokens are still in the input (the model reads them as context); they simply do not contribute to the loss.

In PyTorch and Hugging Face, the mask is stored inside the **labels**: the labels are a copy of the input IDs with masked positions set to $-100$, the default `ignore_index` of `F.cross_entropy`. The model shifts labels by one internally, so you pass `labels` aligned with `input_ids`.

```
import torch
import torch.nn.functional as F

def sft_loss(logits, labels):
    """logits: (B, T, V); labels: (B, T) with -100 on masked positions.
    Mean over all real target tokens in the batch (token-level average)."""
    logits = logits[:, :-1].reshape(-1, logits.size(-1))   # position t predicts t+1
    targets = labels[:, 1:].reshape(-1)
    return F.cross_entropy(logits.float(), targets, ignore_index=-100, reduction="mean")

torch.manual_seed(0)
B, T, V = 2, 6, 11
logits = torch.randn(B, T, V)
input_ids = torch.randint(0, V, (B, T))
labels = input_ids.clone()
labels[:, :3] = -100           # pretend the first 3 tokens are the prompt
```

**Notation on this page:** $s$: a training sequence $(s_1, \dots, s_T)$ of tokens • $T$: sequence length • $t$: time step or token position • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathcal{L}$: a loss to minimise

```
print(sft_loss(logits, labels))
```

## 9.2 What the SFT loss really minimises

**SFT is maximum likelihood, and maximum likelihood is forward KL.** Let $p_{\mathcal{D}}(y \mid x)$ be the (unknown) distribution of responses in the data. The expected SFT loss over the data is

$$\mathbb{E}_{y\sim p_{\mathcal{D}}}[-\log \pi_\theta(y \mid x)] = \underbrace{\mathbb{E}_{p_{\mathcal{D}}}[-\log p_{\mathcal{D}}(y \mid x)]}_{\uparrow\, H(p_{\mathcal{D}})\text{: fixed}} + \mathrm{KL}(p_{\mathcal{D}} \,\|\, \pi_\theta).$$

The entropy of the data does not depend on $\theta$, so minimising cross-entropy is the same as minimising the **forward KL** from data to model. Forward KL is **mode-covering** (Chapter 3): the model is punished heavily wherever the data has mass and the model does not, so it spreads probability to cover every behaviour in the data, including rare and low-quality ones. This is the deep reason SFT copies the *average* quality of its data, and why RL and preference methods (which optimise something closer to a reverse KL with a reward) can then sharpen the model.

The loss also has a floor: it can never go below the data's entropy. A validation loss that stops falling may simply mean the model has reached the irreducible randomness of the responses, not that it has stopped learning.

**SFT as a policy gradient.** The SFT gradient for one target sequence $y^*$ is $\nabla \log \pi_\theta(y^*)$. We can write this as an on-policy expectation:

$$\nabla_\theta \log \pi_\theta(y^*) = \mathbb{E}_{y\sim\pi_\theta}\Big[\frac{\mathbb{1}[y = y^*]}{\pi_\theta(y)} \nabla_\theta \log \pi_\theta(y)\Big],$$

because the expectation picks out the single term $\pi_\theta(y^*) \cdot \frac{1}{\pi_\theta(y^*)} \nabla \log \pi_\theta(y^*)$. So SFT is REINFORCE (Chapter 15) with a "reward" of $1/\pi_\theta(y^*)$ on the target and 0 elsewhere. That reward is **huge when the model finds the target unlikely**, which makes SFT gradients large and unstable on surprising tokens. DFT multiplies by $\pi_\theta$ to cancel exactly this factor, turning the weight back into a constant reward of 1.

## 9.3 The gradient with respect to the logits

For one position with logits $z \in \mathbb{R}^V$, probabilities $p = \mathrm{softmax}(z)$ and target token $c$, the loss is $\ell = -\log p_c = -z_c + \log \sum_v e^{z_v}$. Differentiate:

$$\frac{\partial \ell}{\partial z_v} = p_v - \mathbb{1}[v = c], \qquad \text{i.e.} \quad \nabla_z \ell = p - e_c.$$

*Intuition.* Every wrong token's logit is pushed down in proportion to its probability; the correct token's logit is pushed up by $1 - p_c$. When the model is already confident and right ($p_c \to 1$), the gradient vanishes. This is the most commonly asked "derive it on the whiteboard" question about cross-entropy.

**Notation on this page:** $x$: the prompt • $y^*$: the target sequence; $y$ a sampled sequence • $p = \mathrm{softmax}(z)$; $p_{\mathcal{D}}$: data distribution • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $H$: entropy (uncertainty) of a distribution • $\nabla_\theta$: gradient with respect to $\theta$ • $\mathbb{1}[\cdot]$: 1 if the condition is true, else 0 • $V$: vocabulary size • *and 5 more: see the Notation table*

```
import torch, torch.nn.functional as F
torch.manual_seed(0)

# (1) Gradient of cross-entropy w.r.t. the logits is softmax(z) - onehot(target).
z = torch.tensor([2.0, 1.0, 0.1, -1.0], requires_grad=True)
target = torch.tensor(1)
loss = F.cross_entropy(z[None], target[None])
loss.backward()
p = torch.softmax(z.detach(), -1)
print("loss            ", round(loss.item(), 4), " = -log p[target] =",
 ↪  round(-p[1].log().item(), 4))
print("autograd grad   ", [round(v, 4) for v in z.grad.tolist()])
print("softmax - onehot", [round(v, 4) for v in (p - F.one_hot(target, 4)).tolist()])
```

## 9.4 Which tokens to train on

**Full-sequence loss.** Train on every token, including the prompt. This is what pretraining does.

**Completion-only loss.** For prompt-completion data, mask the prompt and train only on the completion.

**Assistant-only loss.** For multi-turn conversations, train only on the assistant turns (including their end-of-turn tokens), not on system, user or tool turns.

In TRL 1.16.0, `SFTConfig(completion_only_loss=None)` (the default) means "completion-only if the dataset is prompt-completion, full sequence otherwise", and `assistant_only_loss=False` is the default for conversational data. So for a "messages" dataset, **TRL trains on the whole conversation unless you set `assistant_only_loss=True`.**

**Why this, not that? Why train only on the assistant tokens, not the whole conversation?**

**Why.** We want the model to learn to *write responses*, not to write user prompts or tool outputs. Training on user turns teaches the model to imitate users (and to produce fake "User:" turns). Training on tool outputs teaches it to hallucinate tool results instead of calling the tool. Masking also focuses the gradient on the tokens that matter.
**Trade-off.** Fewer tokens carry a learning signal, so each example teaches less, and with short answers to long prompts the signal can be thin. Masks also need template support and are a source of bugs.
**When the alternative wins.** Training on all tokens can help when prompts are high-quality and similar to what the model should produce (for example, domain documents in continued pretraining), or when data is scarce and you want every token to count. Some studies find a small gain from including prompts for short-answer datasets; it is worth testing, not assuming.

In multi-turn RL with tools (Chapter 34), the same idea appears as **masking tool outputs**: tokens returned by the environment are not the policy's actions, so they get no policy-gradient loss.

## 9.5 Loss normalisation

There are several ways to turn per-token losses into one number. They give different gradients.

Let sequence $i$ have $n_i = \sum_t m_{i,t}$ trainable tokens and per-token losses $\ell_{i,t}$.

**Token-level mean (over the batch).**

$$\mathcal{L}_{\text{tok}} = \frac{\sum_i \sum_t m_{i,t}\,\ell_{i,t}}{\sum_i n_i}.$$

Every token counts equally. Long responses contribute more in total, in proportion to their length.

**Sequence-level mean.**

$$\mathcal{L}_{\text{seq}} = \frac{1}{B}\sum_i \frac{1}{n_i}\sum_t m_{i,t}\,\ell_{i,t}.$$

Every *sequence* counts equally. A token in a short response gets more weight than a token in a long one.

**Sum.** $\sum_i \sum_t m_{i,t}\,\ell_{i,t}$, sometimes divided by a constant. Its scale grows with batch size and length, so the learning rate would have to change with them.

The token-level mean is the standard for SFT. As Chapter 4 showed, with gradient accumulation and multiple GPUs the denominator must be the number of trainable tokens in the **whole effective batch**, not in each micro-batch. In late 2024, a widely discussed bug in popular training code averaged per micro-batch, so results changed with the accumulation setting; `transformers` and TRL now pass a global token count (`num_items_in_batch`) to the loss, and TRL's `average_tokens_across_devices` extends this across GPUs.

**Why this, not that? Why a token-level mean, not a sequence-level mean?**

**Why.** With a token-level mean, the gradient does not depend on how the tokens happen to be grouped into sequences or micro-batches, so results are reproducible across batch settings. It also matches the maximum-likelihood objective: every target token is one observation.
**Trade-off.** Long examples dominate: one 4,000-token answer outweighs forty 100-token answers. If long examples are lower quality (or just more verbose), the model drifts towards them.
**When the alternative wins.** A sequence-level mean is better when each example should count once regardless of length, for example when a dataset mixes very short classification-style answers with long essays and you care equally about both. In RL the same choice reappears, with important length effects (GRPO vs DAPO vs Dr. GRPO, Chapter 28).

## 9.6 Packing and padding-free batching

Batches of sequences with different lengths need padding, and padding wastes compute. With chat data, lengths vary a lot, and over half of all tokens in a padded batch can be padding.

**Packing** concatenates several examples into one row of a fixed length (for example 4,096 tokens). Done naively, this has two problems:

**Notation on this page:** $i$: index of a sequence in the batch • $t$: time step or token position • $n_i$: number of trainable tokens in sequence $i$ • $\ell_{i,t}$: loss on token $t$ of sequence $i$ • $\mathcal{L}$: a loss to minimise • $B$: number of sequences in the batch

1. **Cross-contamination.** With a normal causal mask, tokens of example 2 can attend to example 1 in the same row. The model learns from unrelated context.
2. **Wrong positions.** Position IDs keep counting across examples, so example 2 starts at position 1,500 instead of 0.

The correct version uses a **block-diagonal causal mask** (each example attends only to itself) and **resets position IDs** at each example boundary.

```
import torch

def packed_mask_and_positions(lengths):
    """Block-diagonal causal mask and position ids for examples packed into one row."""
    T = sum(lengths)
    mask = torch.zeros(T, T, dtype=torch.bool)   # True = may attend
    pos, start = [], 0
    for n in lengths:
        mask[start:start + n, start:start + n] = torch.tril(torch.ones(n, n,
↪   dtype=torch.bool))
        pos.extend(range(n)); start += n
    return mask, torch.tensor(pos)

mask, pos = packed_mask_and_positions([3, 2, 4])
print(pos.tolist())          # [0, 1, 2, 0, 1, 0, 1, 2, 3]
print(mask.int())
```

**Padding-free batching** goes one step further: all sequences in the batch are flattened into one long sequence with no padding at all, and the attention kernel is told where each sequence starts. FlashAttention's "variable length" kernels take a list of cumulative sequence lengths (often called `cu_seqlens`) for this. No $T \times T$ mask is ever built.

In TRL 1.16.0, `SFTConfig(packing=True)` uses `packing_strategy="bfd"` by default: **best-fit decreasing** bin packing, which places whole examples into rows like packing boxes into bins, and truncates any example longer than `max_length`. `"bfd_split"` splits long examples instead of truncating them, and `"wrapped"` simply cuts the stream every `max_length` tokens (fast, but it splits examples). With `"bfd"`, TRL enables padding-free mode, which TRL's documentation says is only supported with FlashAttention 2 or 3.

**Why this, not that? Why packing, not plain padding?**

**Why.** Packing removes padding waste, often doubling throughput on chat data, and gives batches with a near-constant number of tokens, which makes memory use and step time predictable.
**Trade-off.** It needs correct masks and position IDs (or a variable-length attention kernel), and it changes how many examples are in each step, which interacts with sequence-level loss averaging. Bugs are silent: a model trained with cross-contamination looks fine on loss curves.
**When the alternative wins.** Plain padding is simpler and safe when lengths are similar, when you lack FlashAttention support, or when each step must contain a fixed number of

**Notation on this page:** $T$: sequence length

examples (for example, when debugging or when the loss is defined per sequence, as in DPO).

## 9.7 Truncation

Examples longer than `max_length` must be cut or dropped. Cutting the *end* of a response removes the end-of-turn token, so the model sees responses that never finish; with many such examples it can learn to ramble. Prefer dropping over-long examples, raising `max_length`, or (for very long prompts) truncating the *start* of the context while keeping the full response. In TRL 1.16.0, `truncation_mode` accepts only `"keep_start"` (the default, which cuts the end; `"keep_end"` is deprecated), so the safe options there are filtering long examples out beforehand or raising `max_length`.

## 9.8 Catastrophic forgetting

**Catastrophic forgetting** is when fine-tuning on a new task damages abilities the model had before: general knowledge, maths, coding, multilingual skill, or safety behaviour. SFT on a narrow dataset for many epochs is the classic cause.

Ways to reduce it:

- **Lower learning rate and fewer epochs.** The single most effective fix.
- **Mix in general data** ("replay"): add a share of general instruction data (for example 10 to 30%) to the task data.
- **Parameter-efficient fine-tuning** (Chapter 10): LoRA changes fewer directions in weight space and tends to forget less, though it also learns less.
- **Regularise towards the original model**, for example with a KL term to the base model's outputs (this is what the KL penalty in RLHF does).
- **Model merging**: average the fine-tuned weights with the original weights, or merge several task-specific fine-tunes. Simple linear interpolation often recovers general ability at a small cost to the new task.

A related research finding: on-policy methods (RL, on-policy distillation) tend to forget less than SFT on the same task, because they train on the model's own outputs and so make smaller, more targeted changes to its distribution.

## 9.9 Variants of the SFT loss

**Label smoothing** replaces the one-hot target with $(1 - \epsilon)$ on the correct token and $\epsilon$ spread over the rest. It discourages over-confidence. It is rarely used for LLM SFT.

**Dynamic Fine-Tuning (DFT)**, from *On the Generalization of SFT: A Reinforcement Learning Perspective with Reward Rectification* (Wu et al., arXiv 2508.05629), argues that the SFT gradient behaves like a policy gradient with a reward of $1/\pi_\theta(\text{token})$, which gives huge weight to tokens the model

finds unlikely. DFT multiplies each token's loss by its (stop-gradient) probability:

$$\mathcal{L}_{\text{DFT}} = -\sum_t m_t \, \text{sg}(\pi_\theta(s_t \mid s_{<t})) \log \pi_\theta(s_t \mid s_{<t}),$$

where sg means "stop gradient" (treated as a constant). TRL implements this as `SFTConfig(loss_type="dft")`; the code multiplies the per-token negative log-probability by the detached probability. The paper reports better generalisation on maths reasoning; as with any new method, test it on your task.

**Weighting by quality.** Some pipelines weight each example by a quality score, or train on positive examples and lightly penalise known-bad ones (a step towards preference optimisation).

## 9.10 Hyperparameters that matter for SFT

| Hyperparameter | Typical range | Notes |
|---|---|---|
| Learning rate (full) | $5 \times 10^{-6}$ to $2 \times 10^{-5}$ | Smaller for bigger models; TRL default $2 \times 10^{-5}$ |
| Learning rate (LoRA) | $5 \times 10^{-5}$ to $3 \times 10^{-4}$ | About 10x the full fine-tuning rate |
| Epochs | 1 to 3 | More epochs: memorisation and forgetting |
| Effective batch | 32 to 512 sequences (or 0.1M to 2M tokens) | Bigger for large, noisy datasets |
| Schedule | warmup 3 to 10%, then cosine or linear | |
| `max_length` | covers most examples | Check the length histogram first |
| Weight decay | 0 to 0.1 | Small effect in short runs |
| Precision | bf16 | fp16 needs loss scaling (Chapter 41) |

**Tip.**
Hold out a small set of prompts and *read the generations* every few hundred steps. Validation loss can keep falling while answers get worse (too long, too repetitive, losing the end token). Also track a few general benchmarks to catch forgetting early.

## 9.11 Reading loss numbers: perplexity

**Perplexity** is exp(mean token loss): the effective number of tokens the model is "choosing between" at each step. A loss of 0.7 nats is a perplexity of about 2.0; a loss of 2.0 is about 7.4. Typical numbers:

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathcal{L}$: a loss to minimise • $\text{sg}(\cdot)$: stop-gradient (treated as a constant in backprop) • $s$: a training sequence $(s_1, \ldots, s_T)$ of tokens • $t$: time step or token position

- A pretrained model on general web text: loss around 2 (perplexity about 7 to 10, depending on tokenizer).
- Chat SFT on assistant tokens: the loss often starts around 1.0 to 1.5 and falls to 0.5 to 1.0 within the first epoch, because assistant responses are more predictable than web text.
- A training loss that keeps falling towards 0.1 in later epochs while the validation loss rises is memorisation.

Perplexities are only comparable with the **same tokenizer** and the **same masking**: a model with a bigger vocabulary has fewer, harder tokens per sentence, and including easy prompt tokens lowers the average.

## 9.12 Gradient accumulation and the token count

With gradient accumulation, one optimiser step sums gradients from several micro-batches. If each micro-batch averages over its own tokens and the results are averaged again, a micro-batch with 2 tokens gets as much weight as one with 10. The fix is to divide each micro-batch's **sum** of token losses by the **total number of target tokens in the whole step**. Hugging Face Transformers and TRL do this by passing `num_items_in_batch` to the loss, and `average_tokens_across_devices=True` (TRL default) extends the count across GPUs. A demonstration:

```
V = 5
w = torch.randn(V, V, requires_grad=True)                # a toy "model": logits = W[x]
def per_token_nll(x, y):
    return F.cross_entropy(w[x], y, reduction="none")
mb1 = (torch.randint(0, V, (2,)), torch.randint(0, V, (2,)))     # micro-batch with 2
↪ target tokens
mb2 = (torch.randint(0, V, (10,)), torch.randint(0, V, (10,)))   # micro-batch with 10
↪ target tokens

full = torch.cat([per_token_nll(*mb1), per_token_nll(*mb2)]).mean()
g_full = torch.autograd.grad(full, w)[0]
naive = (per_token_nll(*mb1).mean() + per_token_nll(*mb2).mean()) / 2
g_naive = torch.autograd.grad(naive, w)[0]
n_items = 12                                              # total target tokens in the
↪ whole step
fixed = per_token_nll(*mb1).sum() / n_items + per_token_nll(*mb2).sum() / n_items
g_fixed = torch.autograd.grad(fixed, w)[0]
print(f"naive accumulation error {(g_naive - g_full).norm() / g_full.norm():.2%}, "
      f"num_items_in_batch error {(g_fixed - g_full).norm() / g_full.norm():.2%}")
```

The naive version gives a very different gradient from training on the whole batch at once; dividing by the step's total token count matches it exactly. This bug existed in many trainers until late 2024 and still appears in custom code.

## 9.13 Teacher forcing and exposure bias

SFT always conditions on the **gold** prefix $s_{<t}$ from the data (**teacher forcing**). At inference the model conditions on its **own** previous tokens. Small early mistakes put it in states it never saw in training, and errors can compound: this is **exposure bias**. It is one reason long generations drift and one argument for on-policy training (RL, on-policy distillation in Chapter 12), which trains on the model's own prefixes.

**Why this, not that? Why teacher forcing, not training on the model's own samples?**

**Why.** Teacher forcing turns training into one parallel forward pass per sequence (all positions at once), gives a dense, low-variance signal at every token, and needs no reward or teacher at training time. It is what makes SFT cheap and stable.
**Trade-off.** Exposure bias: the model never practises recovering from its own mistakes, and it learns to imitate the data's distribution rather than to achieve a goal.
**When the alternative wins.** On-policy methods (RL with a reward, on-policy distillation with a teacher's per-token log-probs, DAgger-style correction) win when the model must handle its own errors, for long reasoning or agent trajectories, and to reduce forgetting.

**Why this, not that? Why cross-entropy (forward KL to the data), not a reverse-KL or reward-weighted objective?**

**Why.** Forward KL needs only samples from the data, not the data's probabilities, and gives a convex, well-behaved loss per token. Reverse KL would need $\log p_{\mathcal{D}}$, which we do not have for human-written text.
**Trade-off.** Mode-covering: the model learns to cover everything in the data, including mistakes and low-quality styles; it cannot do better than the data's average.
**When the alternative wins.** When a teacher model provides probabilities (distillation, where reverse KL is possible and mode-seeking), or when a reward is available (RL), objectives that focus on high-quality modes beat plain imitation.

## 9.14 Noise in the embeddings: NEFTune

*NEFTune: Noisy Embeddings Improve Instruction Finetuning* (Jain et al., arXiv 2310.05914) adds uniform random noise to the input embeddings during SFT, scaled by $\alpha/\sqrt{Ld}$ for sequence length $L$ and embedding size $d$. Fine-tuning Llama-2-7B on Alpaca improved AlpacaEval from 29.79% to 64.69%, with smaller gains (around 8 to 10%) on other datasets. It acts as a regulariser against overfitting to the exact wording of the training answers; part of the AlpacaEval gain comes from longer, more detailed answers. In TRL: `SFTConfig(neftune_noise_alpha=5)` (default `None`, which disables it).

## 9.15 TRL's defaults, checked

Fields of `SFTConfig` in TRL 1.16.0 worth knowing: `learning_rate=2e-5`, `num_train_epochs=3.0`, `lr_scheduler_type="linear"`, `warmup_steps=0`, `max_length=1024`, `packing=False` (strategy

**Notation on this page:** $s$: a training sequence $(s_1, \ldots, s_T)$ of tokens • $t$: time step or token position • $p = \text{softmax}(z)$; $p_{\mathcal{D}}$: data distribution • $\alpha$: NEFTune noise scale • $d$: embedding size • $L$: sequence length (NEFTune)

`"bfd"` when enabled), `padding_free=False`, `loss_type="nll"` (or `"dft"`), `completion_only_loss=None`, `assistant_only_loss=False`, `gradient_checkpointing=True`, `average_tokens_across_devices=True`, `neftune_noise_alpha=None`. Two defaults deserve attention: **three epochs** is often too many for large chat datasets, and **no warmup** can make the first steps unstable at higher learning rates.

## 9.16 Common mistakes

- Averaging micro-batch means under gradient accumulation instead of dividing by the step's total token count.
- Comparing perplexities across different tokenizers or masks.
- Keeping TRL's default 3 epochs and no warmup without checking.

**Interview check**

**Q1. How do you implement completion-only loss in Hugging Face?** Copy `input_ids` into `labels` and set the prompt (and padding) positions to $-100$; the loss uses `ignore_index=-100`. In TRL, use a prompt-completion dataset, or `assistant_only_loss=True` for conversations.
**Q2. Token-level vs sequence-level mean: what is the difference in gradient?** Token-level gives each trainable token weight $1/\sum_i n_i$; sequence-level gives tokens in sequence $i$ weight $1/(B\,n_i)$, so tokens in short sequences count more.
**Q3. What goes wrong with naive packing?** Examples attend to each other across boundaries and position IDs do not reset. Fix with a block-diagonal mask and per-example position IDs, or padding-free varlen attention.
**Q4. Name four ways to reduce catastrophic forgetting.** Lower LR and fewer epochs; replay general data; LoRA; KL or weight regularisation to the base model; model merging.
**Q5. Why might truncating long examples hurt?** Truncated responses lose their end token and their conclusion, teaching the model to produce responses that do not finish.
**Q6. What is DFT?** Dynamic Fine-Tuning: weight each token's NLL by its stop-gradient probability, which removes the implicit $1/p$ weighting that SFT gives to unlikely tokens.
**Q7. Why must the loss denominator be the global token count with gradient accumulation?** Otherwise micro-batches with fewer tokens get more weight per token, and the gradient depends on how data was split.
**Q8. Show that minimising SFT loss minimises a KL divergence. Which one?** $\mathbb{E}_{p_{\mathcal{D}}}[-\log \pi_\theta] = H(p_{\mathcal{D}}) + \mathrm{KL}(p_{\mathcal{D}} \| \pi_\theta)$: the forward KL from data to model, which is mode-covering.
**Q9. Derive the gradient of cross-entropy with respect to the logits.** $\ell = -z_c + \log \sum_v e^{z_v}$, so $\partial \ell / \partial z_v = p_v - \mathbb{1}[v = c]$.
**Q10. Write SFT as a policy gradient. What is the implied reward?** $\nabla \log \pi(y^*) = \mathbb{E}_{y\sim\pi}[\mathbb{1}[y = y^*]/\pi(y)\,\nabla \log \pi(y)]$: reward $1/\pi(y^*)$ on the target, which explodes for unlikely targets; DFT cancels it.
**Q11. What is exposure bias?** Training on gold prefixes but generating from the model's own prefixes; errors compound at inference. On-policy training reduces it.
**Q12. What is perplexity and when can you compare perplexities?** exp of the mean token

**Notation on this page:** $n_i$: number of trainable tokens in sequence $i$ • $i$: index of a sequence in the batch • $B$: number of sequences in the batch • $p = \mathrm{softmax}(z)$; $p_{\mathcal{D}}$: data distribution • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $H$: entropy (uncertainty) of a distribution • $z$: logits at one position • $\ell_{i,t}$: loss on token $t$ of sequence $i$ • *and 6 more: see the Notation table*

loss; only with the same tokenizer, data and masking.
**Q13. What does NEFTune do?** Adds scaled uniform noise to input embeddings during SFT as a regulariser; big AlpacaEval gains on Alpaca (29.8% → 64.7% for Llama-2-7B).
**Q14. Why can the validation loss plateau even if the model is fine?** The loss is bounded below by the entropy of the data's responses; a plateau may be the irreducible randomness.

**Key takeaways**

- SFT loss = masked token-level cross-entropy; masked labels are $-100$.
- Train on assistant tokens (and their end token) for chat; TRL trains on everything in "messages" data unless `assistant_only_loss=True`.
- Normalise by the number of trainable tokens in the whole effective batch.
- Packing needs block-diagonal masks and reset positions; padding-free varlen attention does this without masks.
- Fight forgetting with low LR, few epochs, replay data, LoRA, KL regularisation or merging.
- DFT reweights tokens by their probability; label smoothing is rarely used.
- SFT = MLE = forward KL (mode-covering); loss floor = data entropy.
- $\nabla_z \ell = p - e_c$; SFT is REINFORCE with reward $1/\pi$ on the target.
- Divide by the step's total token count under gradient accumulation (`num_items_in_batch`).

**Notation on this page:** $\nabla_\theta$: gradient with respect to $\theta$ • $z$: logits at one position • $p = \text{softmax}(z)$; $p_{\mathcal{D}}$: data distribution • $\ell_{i,t}$: loss on token $t$ of sequence $i$ • $c$: the correct (target) token • $e_c$: one-hot vector for token $c$

CHAPTER 10

# Parameter-Efficient Fine-Tuning

**Why this chapter matters.**
Full fine-tuning of a 7B model needs over 100 GB of GPU memory once you count gradients and optimizer states. **Parameter-efficient fine-tuning (PEFT)** trains a tiny number of new parameters instead and leaves the original weights frozen. LoRA and QLoRA make it possible to fine-tune large models on one GPU, to keep many task adapters for one base model, and, increasingly, to run RL cheaply. Interviewers expect you to derive LoRA, count its parameters, and explain each hyperparameter.

## 10.1 Where the memory goes

For full fine-tuning with AdamW in mixed precision, each parameter needs roughly:

- 2 bytes for the bf16 weight,
- 2 bytes for its bf16 gradient,
- 4 bytes for an fp32 master copy of the weight (common in mixed precision),
- 8 bytes for Adam's two fp32 moment estimates.

That is about **16 bytes per parameter**, so a 7B model needs about 112 GB before activations. With LoRA, only the adapter parameters need gradients and optimizer state; the frozen base needs just its 2 bytes per parameter (14 GB for 7B), or about 0.5 bytes with 4-bit quantisation (QLoRA).

## 10.2 LoRA: Low-Rank Adaptation

*LoRA: Low-Rank Adaptation of Large Language Models* (Hu et al., arXiv 2106.09685) is based on one observation: the *change* in weights during fine-tuning tends to have low "intrinsic rank". So instead of learning a full update $\Delta W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$, LoRA learns it as the product of two thin matrices:

$$W' = W_0 + \Delta W = W_0 + \frac{\alpha}{r} BA, \qquad B \in \mathbb{R}^{d_{\text{out}} \times r},\ A \in \mathbb{R}^{r \times d_{\text{in}}},\ r \ll \min(d_{\text{in}}, d_{\text{out}}).$$

Here $W_0$ is the frozen pretrained weight, $r$ is the **rank** of the adapter, and $\alpha$ is a scaling constant (`lora_alpha`). The forward pass for an input $x$ is

$$h = W_0 x + \frac{\alpha}{r} B(Ax).$$

Computing $B(Ax)$ never forms the full $d_{\text{out}} \times d_{\text{in}}$ matrix, so it is cheap.

**Initialisation.** $A$ starts random (Kaiming-uniform in the reference code) and $B$ starts at **zero**. So $BA = 0$ at the start, and the adapted model is *exactly* the base model before training. Training starts from the pretrained behaviour with no sudden jump.

**Notation on this page:** $d_{\text{in}}, d_{\text{out}}$: input and output sizes of the layer • $W_0$: frozen pretrained weight; $W'$: adapted weight • $\Delta W$: the learned change to the weight • $\alpha$: LoRA scaling numerator (`lora_alpha`) • $r$: LoRA rank • $B$: LoRA up-projection ($d_{\text{out}} \times r$), zero at start • $A$: LoRA down-projection ($r \times d_{\text{in}}$), random at start • $x$: input vector to the layer • $h$: output of the adapted layer

**Parameter count.** A full update has $d_{\text{out}}d_{\text{in}}$ parameters. LoRA has $r(d_{\text{in}} + d_{\text{out}})$.

*Worked example.* A 4096×4096 projection has 16.8M parameters. LoRA with $r = 16$ has $16\times(4096+4096) = 131{,}072$, which is 0.78%. Applied to the 7 linear layers in each of 32 transformer blocks of a Llama-style 7B model (with MLP layers of size $4096 \times 11008$), it adds about 40M trainable parameters, or about 0.6% of the model.

**Merging.** After training, $W' = W_0 + \frac{\alpha}{r}BA$ can be computed once and saved as an ordinary weight. The merged model has **no extra inference cost**. Or you can keep adapters separate and swap them per request: serving engines such as vLLM can serve many LoRA adapters on one base model at once.

```
import torch, torch.nn as nn, math

class LoRALinear(nn.Module):
    def __init__(self, base: nn.Linear, r=8, alpha=16, rslora=False):
        super().__init__()
        self.base = base
        for p in self.base.parameters():
            p.requires_grad_(False)                          # freeze W0
        self.A = nn.Parameter(torch.empty(r, base.in_features))
        self.B = nn.Parameter(torch.zeros(base.out_features, r))   # B = 0 at start
        nn.init.kaiming_uniform_(self.A, a=math.sqrt(5))
        self.scale = alpha / math.sqrt(r) if rslora else alpha / r

    def forward(self, x):
        return self.base(x) + self.scale * (x @ self.A.T) @ self.B.T

    def merged_weight(self):
        return self.base.weight + self.scale * self.B @ self.A

torch.manual_seed(0)
base = nn.Linear(64, 32, bias=False)
lora = LoRALinear(base, r=4, alpha=8)
x = torch.randn(5, 64)
print("same as base at init:", torch.allclose(lora(x), base(x)))
with torch.no_grad():
    lora.B.normal_()                                         # pretend we trained
print("merge matches:", torch.allclose(lora(x), x @ lora.merged_weight().T, atol=1e-5))
trainable = sum(p.numel() for p in lora.parameters() if p.requires_grad)
print("trainable:", trainable, "vs full:", 64 * 32)
```

**Why this, not that? Why a low-rank update, not full fine-tuning?**

**Why.** LoRA cuts optimizer and gradient memory by over 99%, so large models fit on small hardware. Adapters are tiny files (megabytes), easy to store, share and swap. The frozen base also limits how far the model can drift, which reduces forgetting.
**Trade-off.** A rank-$r$ update cannot express every change. For tasks that need a lot of new capacity (large-scale SFT on lots of new data, continued pretraining) LoRA learns less than full fine-tuning. It also adds hyperparameters (rank, alpha, target modules).

**Notation on this page:** $d_{\text{in}}, d_{\text{out}}$: input and output sizes of the layer • $r$: LoRA rank • $\alpha$: LoRA scaling numerator (`lora_alpha`) • $W_0$: frozen pretrained weight; $W'$: adapted weight

**When the alternative wins.** Full fine-tuning wins when you have the hardware and large datasets, when the task is far from the base model's abilities, or when you are building a flagship model and every point of quality counts.

## 10.3 The LoRA hyperparameters

In Hugging Face PEFT (version 0.21.2), `LoraConfig` has these main fields. The defaults are shown.

| Field | Default | Meaning |
|---|---|---|
| `r` | 8 | Rank of the update |
| `lora_alpha` | 8 | Scaling numerator $\alpha$ |
| `lora_dropout` | 0.0 | Dropout on the input to the adapter path |
| `target_modules` | `None` | Which modules get adapters; `"all-linear"` = every linear layer except the output head |
| `use_rslora` | `False` | Use scale $\alpha/\sqrt{r}$ instead of $\alpha/r$ |
| `use_dora` | `False` | Use DoRA (magnitude and direction) |
| `init_lora_weights` | `True` | `True` = $B = 0$; also `"gaussian"`, `"pissa"`, `"pissa_niter_[n]"`, `"olora"`, `"loftq"`, `"eva"`, `"corda"`, `"orthogonal"` and others |
| `bias` | `"none"` | Also train biases: `"none"`, `"all"`, `"lora_only"` |
| `modules_to_save` | `None` | Extra modules trained fully and saved (e.g. a new classification head) |
| `rank_pattern`, `alpha_pattern` | `{}` | Per-layer rank or alpha overrides by regex |
| `layers_to_transform` | `None` | Only adapt these layer indices |

**Rank $r$.** 8 to 64 is common. Higher rank gives more capacity and more memory. For RL, very low ranks can work, because RL updates carry little information per episode (see below).

**Alpha.** The effective step size of the adapter scales with $\alpha/r$. A common rule is $\alpha = 2r$ or $\alpha = r$. If you change $r$, the meaning of the learning rate changes too, unless you use rsLoRA.

**Target modules.** The original paper adapted only the attention query and value projections. Later practice, and the experiments in Thinking Machines' "LoRA Without Regret" blog post (2025), found that applying LoRA to **all linear layers, especially the MLP layers**, is much closer

to full fine-tuning than attention-only LoRA. That post also reported that LoRA matched full fine-tuning for RL even at very low rank, and that the best LoRA learning rate was about 10 times the best full fine-tuning rate.

**Learning rate.** LoRA usually needs a higher learning rate than full fine-tuning: often $1 \times 10^{-4}$ to $3 \times 10^{-4}$ for SFT, versus about $1 \times 10^{-5}$ for full fine-tuning.

## 10.4 rsLoRA: fixing the scale

*A Rank Stabilization Scaling Factor for Fine-Tuning with LoRA* (Kalajdzievski, arXiv 2312.03732) shows that with the scale $\alpha/r$, the adapter's output and gradients shrink as $r$ grows, so high-rank adapters learn slowly and you gain little from raising $r$. The paper derives that the scale should be $\alpha/\sqrt{r}$ to keep learning stable across ranks.

*Intuition.* $BAx$ is a sum of $r$ terms. If they behave roughly like independent random terms, the size of the sum grows like $\sqrt{r}$, not $r$. Dividing by $r$ over-corrects; dividing by $\sqrt{r}$ keeps the size constant.

**Why this, not that? Why scale by alpha over root r (rsLoRA), not alpha over r?**

**Why.** With $\alpha/\sqrt{r}$, raising the rank actually adds useful capacity, and you can change $r$ without re-tuning the learning rate.
**Trade-off.** At the same $\alpha$ and large $r$, the update is much larger than with $\alpha/r$, so old learning rates may be too high. Many published recipes assume $\alpha/r$.
**When the alternative wins.** At low ranks (8 to 16) the difference is small, and the standard scale with a well-tuned learning rate works fine. Use the standard scale when reproducing a published recipe.

## 10.5 LoRA+: different learning rates for A and B

*LoRA+: Efficient Low Rank Adaptation of Large Models* (Hayou et al., arXiv 2402.12354) argues that $A$ and $B$ play different roles and, for wide models, should not share one learning rate. It sets $\eta_B = \lambda\,\eta_A$ with $\lambda > 1$ (the paper suggests about $2^4 = 16$), and reports faster training and small quality gains.

## 10.6 PiSSA: start from the important directions

*PiSSA: Principal Singular Values and Singular Vectors Adaptation of Large Language Models* (Meng et al., arXiv 2404.02948) initialises the adapter from the **singular value decomposition** (SVD, writing a matrix as $USV^\top$ with orthonormal $U$, $V$ and non-negative diagonal $S$) of $W_0$. The top $r$ singular directions go into the adapter, and the rest stays frozen as a residual:

$$W_0 = \underbrace{U_{[:r]}S_{[:r]}^{1/2}}_{\uparrow B}\,\underbrace{S_{[:r]}^{1/2}V_{[:r]}^\top}_{\uparrow A} + W_{\text{res}}.$$

The model is unchanged at the start, but training updates the most important directions first, which the paper reports converges faster than standard LoRA. In PEFT, `init_lora_weights=`

**Notation on this page:** $\alpha$: LoRA scaling numerator (`lora_alpha`) • $r$: LoRA rank • $A$: LoRA down-projection ($r \times d_{\text{in}}$), random at start • $B$: LoRA up-projection ($d_{\text{out}} \times r$), zero at start • $\eta_A, \eta_B$: learning rates for $A$ and $B$ • $\lambda$: LoRA+ learning-rate ratio $\eta_B/\eta_A$ • $V$: right singular vectors (SVD) • $S$: diagonal matrix of singular values • $U, S, V$: SVD factors (left vectors, singular values, right vectors) • $W_0$: frozen pretrained weight; $W'$: adapted weight

`"pissa"` (or `"pissa_niter_16"` for a fast approximate SVD).

## 10.7 DoRA: magnitude and direction

*DoRA: Weight-Decomposed Low-Rank Adaptation* (Liu et al., arXiv 2402.09353) splits each weight into a magnitude and a direction. Writing $\|\cdot\|_c$ for the norm of each column,

$$W' = m \odot \frac{W_0 + BA}{\|W_0 + BA\|_c},$$

where $m$ is a trainable vector of column magnitudes (initialised to $\|W_0\|_c$), $\odot$ multiplies each column by its magnitude, and LoRA updates only the direction. The authors found that full fine-tuning changes magnitude and direction in a different pattern from LoRA, and that DoRA's pattern is closer to full fine-tuning. It usually gives a small gain over LoRA at the same rank, at extra compute cost. In PEFT, `use_dora=True`.

## 10.8 QLoRA: LoRA on a 4-bit base

*QLoRA: Efficient Finetuning of Quantized LLMs* (Dettmers et al., arXiv 2305.14314) stores the frozen base weights in 4 bits and trains bf16 LoRA adapters on top. Three ideas make it work:

1. **NF4 (4-bit NormalFloat).** A 4-bit data type whose 16 levels are placed at the quantiles of a normal distribution, which suits weights that are roughly normally distributed.
2. **Double quantisation.** The per-block scale constants are themselves quantised, saving about 0.37 bits per parameter.
3. **Paged optimizers.** Optimizer states can move to CPU memory when GPU memory spikes.

During the forward pass, each 4-bit block is **dequantised** to bf16 just in time, multiplied, and thrown away. Gradients flow through the frozen dequantised weights into the adapters.

In `transformers`, QLoRA uses `BitsAndBytesConfig(load_in_4bit=True, bnb_4bit_quant_type="nf4", bnb_4bit_use_double_quant=True, bnb_4bit_compute_dtype=torch.bfloat16)`. A 65B model can be fine-tuned on a single 48 GB GPU this way.

**Why this, not that? Why QLoRA, not plain LoRA?**

**Why.** Storing the base model in 4 bits cuts its memory by about 4 times versus bf16, so much larger models fit on one GPU, with quality close to 16-bit LoRA in the paper's experiments.
**Trade-off.** Training is slower (dequantising every step), quantisation adds a small error, and merging adapters into a 4-bit base is not exact: you usually merge into a bf16 copy of the base instead, which may behave slightly differently from what was trained.
**When the alternative wins.** If the bf16 model fits, plain LoRA is faster and simpler. For RL, where rollouts must be fast and match the training model, a 4-bit training model complicates the train/inference match (Chapter 33).

**Notation on this page:** $\odot$: element-wise product • $W_0$: frozen pretrained weight; $W'$: adapted weight • $m$: learned column magnitudes (DoRA)

## 10.9 LoRA for RL

LoRA is increasingly used for RL post-training. The argument (made in "LoRA Without Regret") is about information: a policy-gradient update gets roughly one scalar reward per episode, so the total information learned in RL is small, and even a rank-1 adapter can hold it. For SFT on large datasets, every token carries information and LoRA capacity can become the bottleneck. In practice:

- LoRA RL saves memory, and the reference model for the KL penalty is free: just turn the adapter off.
- The rollout engine must apply the current adapter, which serving engines support.
- Use a higher learning rate than for full RL fine-tuning.

**Why this, not that? Why turn the adapter off for the reference model, not keep a separate copy?**

**Why.** With LoRA, $\pi_{\text{ref}}$ is the base model, which is the same network with the adapter disabled. You save a full copy of the model in GPU memory.
**Trade-off.** Each reference log-prob needs an extra forward pass through the same weights (no parallelism with the policy pass), and the reference must be the base model, not an earlier checkpoint.
**When the alternative wins.** If you want to periodically reset the reference to the current policy (as in the DeepSeek-R1 run), or the reference is a different model, keep a separate reference model, or precompute reference log-probs.

## 10.10 The gradients of LoRA

Let $G = \partial\mathcal{L}/\partial W$ be the gradient that full fine-tuning would apply to the weight matrix (shape $d_{\text{out}} \times d_{\text{in}}$), and $s = \alpha/r$ the scale. Since $W = W_0 + sBA$, the chain rule gives

$$\frac{\partial\mathcal{L}}{\partial A} = s\,B^\top G, \qquad \frac{\partial\mathcal{L}}{\partial B} = s\,GA^\top.$$

Three consequences follow, and each is a common interview question:

1. **At initialisation $B = 0$, so $\partial\mathcal{L}/\partial A = 0$.** Only $B$ moves on the first step. $A$ starts learning once $B$ is non-zero. LoRA therefore has a built-in "warm-up": its effective learning rate grows as $B$ grows. This is why the best learning rate is higher for short runs.
2. **The first update lives in $A$'s row space.** After one SGD step, $B = -\eta sGA^\top$, so $\Delta W = sBA = -\eta s^2\,GA^\top A$: the full gradient projected onto the $r$-dimensional subspace spanned by $A$'s rows. LoRA is like full fine-tuning restricted to a random (then learned) low-dimensional subspace.
3. **$A$ and $B$ see different gradient scales**, which motivates giving them different learning rates (LoRA+).

```
import torch
torch.manual_seed(0)
```

**Notation on this page:** $\mathcal{L}$: a loss to minimise • $G$: group size (samples per prompt) • $W_0$: frozen pretrained weight; $W'$: adapted weight • $d_{\text{in}}, d_{\text{out}}$: input and output sizes of the layer • $\alpha$: LoRA scaling numerator (`lora_alpha`) • $r$: LoRA rank • $s = \alpha/r$: LoRA scale • $B$: LoRA up-projection ($d_{\text{out}} \times r$), zero at start • $A$: LoRA down-projection ($r \times d_{\text{in}}$), random at start • $\eta_A, \eta_B$: learning rates for $A$ and $B$ • $\Delta W$: the learned change to the weight

```
# Gradients of LoRA factors: dL/dA = s * B^T G and dL/dB = s * G A^T, where G = dL/dW.
d_out, d_in, r, s = 6, 8, 2, 2.0
W0 = torch.randn(d_out, d_in)
A = torch.randn(r, d_in, requires_grad=True)
B = torch.zeros(d_out, r, requires_grad=True)
x, y = torch.randn(16, d_in), torch.randn(16, d_out)
W = W0 + s * B @ A
loss = ((x @ W.T - y) ** 2).mean()
G = torch.autograd.grad(loss, W, retain_graph=True)[0]          # gradient a full
 ↪ fine-tune would see
gA, gB = torch.autograd.grad(loss, (A, B))
print("dL/dA == s B^T G :", torch.allclose(gA, s * B.T @ G), "| norm of dL/dA at init:",
 ↪ gA.norm().item())
print("dL/dB == s G A^T :", torch.allclose(gB, s * G @ A.T, atol=1e-6))
```

**Why the $1/r$ scale makes the learning rate nearly rank-independent.** Write $BA = \sum_{i=1}^{r} b_i a_i^\top$, a sum of $r$ rank-1 terms. At the first step every term receives the same expected update (each $a_i$ is drawn from the same distribution), so $\frac{1}{r}\sum_i b_i a_i^\top$ is an *average* of $r$ terms with the same expectation, and its expected size does not depend on $r$. The "LoRA Without Regret" post (Schulman and Thinking Machines, 2025) observed that learning curves for different ranks are almost identical early in training, and that the best learning rate changed by less than a factor of 2 between ranks 4 and 512. (rsLoRA's $1/\sqrt{r}$ argument is about the *size of random fluctuations* in the sum; it matters more for long training at high rank.)

**Redundant hyperparameters.** The same post showed that with Adam, four knobs ($\alpha$, the initial scale of $A$, and the learning rates of $A$ and $B$) collapse to two effective ones: the size of the initial updates ($\alpha \cdot \text{init}_A \cdot \text{LR}_B$) and how quickly $A$ moves away from its random start ($\text{init}_A/\text{LR}_A$). LoRA+ and "use a large $\alpha$ at high rank" are both ways of slowing $A$ relative to $B$.

## 10.11 How much rank does a task need?

A weight update needs rank $r$ if most of its "energy" lies in $r$ directions. You can measure this with the singular values $\sigma_1 \geq \sigma_2 \geq \ldots$ of a full fine-tuning update: the share of $\sum \sigma_i^2$ captured by the top $k$ tells you how well a rank-$k$ adapter could represent it.

```
torch.manual_seed(1)
d = 256
U, _ = torch.linalg.qr(torch.randn(d, d)); V, _ = torch.linalg.qr(torch.randn(d, d))
spectrum = torch.cat([torch.tensor([10., 6., 4., 3., 2., 1.5, 1., 1.]), 0.05 *
 ↪ torch.ones(d - 8)])
dW = U @ torch.diag(spectrum) @ V.T                          # a synthetic update: 8 strong
 ↪ directions + noise
S = torch.linalg.svdvals(dW)
energy = (S ** 2).cumsum(0) / (S ** 2).sum()
for k in (1, 4, 8, 16, 64):
    print(f"rank {k:3d} captures {energy[k-1]:.1%} of the update's squared norm")
```

Real updates are less tidy. *LoRA Learns Less and Forgets Less* (Biderman et al., arXiv 2405.09673)

**Notation on this page:** $r$: LoRA rank • $b_i$: column $i$ of $B$ • $\alpha$: LoRA scaling numerator (`lora_alpha`) • $A$: LoRA down-projection ($r \times d_{\text{in}}$), random at start • $B$: LoRA up-projection ($d_{\text{out}} \times r$), zero at start • $\sigma_i$: singular values • $k$: input size of a layer; top-$k$ singular directions

compared LoRA with full fine-tuning on code and maths, both for instruction tuning (about 100k examples) and continued pretraining (20B tokens). In standard low-rank settings LoRA **substantially underperformed** full fine-tuning on the target domain, but **forgot less** of the base model's abilities outside it (better than weight decay or dropout as a regulariser) and kept generations more diverse. They found full fine-tuning learns perturbations with **10 to 100 times higher rank** than typical LoRA configurations. "LoRA Without Regret" refined the picture: on post-training-sized datasets, with LoRA on all layers and a well-chosen learning rate, LoRA matched full fine-tuning; it fell behind only when the dataset exceeded the adapter's capacity, and it was **less tolerant of large batch sizes** (a property of the $BA$ product that higher rank did not fix).

## 10.12 The capacity argument for RL, with numbers

How much information does training have to store? For SFT, roughly the total log-loss over the first epoch: about one bit per token, so millions to billions of bits for real datasets. For policy-gradient RL with one scalar reward per episode, at most about **one bit per episode** (the advantage is the only part of the update that depends on the reward). "LoRA Without Regret" gave concrete numbers: RL on MATH with about 10,000 problems and 32 samples each is about 320,000 bits, while rank-1 LoRA on Llama-3.1-8B already has about 3 million parameters. DeepSeek-R1-Zero's 10,400 steps of 32 questions with 16 samples is about 5.3 million episodes, which is less than a low-rank adapter's parameter count. In their experiments, LoRA RL matched full fine-tuning even at rank 1, with a wider range of good learning rates.

**Why this, not that? Why apply LoRA to all linear layers, not just the attention projections (as in the original paper)?**

**Why.** Most parameters, and most of the model's "knowledge", sit in the MLP (or MoE expert) layers. Both Biderman et al. and "LoRA Without Regret" found that attention-only LoRA underperforms, even when given the same number of parameters by using a higher rank, and that adding attention on top of MLP-only LoRA gives little extra.
**Trade-off.** More adapted modules mean more adapter parameters, memory and compute, and larger adapter files; with MoE models, per-expert adapters complicate parallelism.
**When the alternative wins.** Attention-only (or q/v-only) LoRA is enough for light style or format changes on a tight memory budget, and is the right choice when reproducing older recipes.

## 10.13 Compute and memory, worked out

**FLOPs.** For one $N \times N$ weight, a full training step costs about $3N^2$ multiply-adds per token (forward, input gradient, weight gradient). With LoRA, the weight gradient for $W_0$ is skipped, and the adapter adds about $6Nr$, so the total is $2N^2 + 6Nr$: **slightly more than two thirds** of full fine-tuning (derived in "LoRA Without Regret"). LoRA saves memory a lot, but compute only modestly.

**Memory for a 7B model with LoRA ($r = 16$, all linear layers, about 40M adapter parameters):**

| Item | Full fine-tuning (bf16 + fp32 Adam) | LoRA (bf16 base) | QLoRA (NF4 base) |
|---|---|---|---|
| Base weights | 14 GB (+ 28 GB fp32 master) | 14 GB | about 3.5 to 4 GB |
| Gradients | 14 GB | 0.08 GB | 0.08 GB |
| Adam states | 56 GB | 0.32 GB | 0.32 GB |
| Activations | depends on batch and length; gradient checkpointing cuts it a lot | same | same |
| **Total before activations** | **about 112 GB** | **about 15 GB** | **about 4 to 5 GB** |

Activations often dominate for long sequences, which is why `gradient_checkpointing=True` is the default in TRL configs.

## 10.14 Other parameter-efficient methods

- **Adapters** (*Parameter-Efficient Transfer Learning for NLP*, Houlsby et al., arXiv 1902.00751): small bottleneck MLPs inserted after attention and MLP blocks. They add inference latency because they cannot be merged.
- **Prefix tuning** (*Prefix-Tuning: Optimizing Continuous Prompts for Generation*, Li and Liang, arXiv 2101.00190): learn key/value vectors prepended at every layer.
- **Prompt tuning** (*The Power of Scale for Parameter-Efficient Prompt Tuning*, Lester et al., arXiv 2104.08691): learn only a few "soft prompt" embeddings at the input; becomes competitive only for very large models.
- **(IA)$^3$** (*Few-Shot Parameter-Efficient Fine-Tuning is Better and Cheaper than In-Context Learning*, Liu et al., arXiv 2205.05638): learn per-channel scaling vectors for keys, values and MLP activations; extremely few parameters.

**Why this, not that? Why LoRA, not prefix tuning, prompt tuning or adapters?**

**Why.** LoRA merges into the weights (zero inference overhead), works at all model sizes, does not consume context length, and is supported everywhere (PEFT, TRL, vLLM, SGLang multi-adapter serving).
**Trade-off.** It has more trainable parameters than prompt tuning or (IA)$^3$, and unmerged multi-adapter serving still costs extra matmuls.
**When the alternative wins.** Prompt tuning or (IA)$^3$ when you need thousands of tiny task-specific variants with very little storage; adapters when the architecture or framework already depends on them.

## 10.15 Using PEFT in code

```
from transformers import AutoModelForCausalLM
from peft import LoraConfig, get_peft_model

model = AutoModelForCausalLM.from_pretrained("Qwen/Qwen2.5-1.5B-Instruct",
 ↪  dtype="bfloat16")
config = LoraConfig(r=16, lora_alpha=32, lora_dropout=0.0, target_modules="all-linear",
                    task_type="CAUSAL_LM")
model = get_peft_model(model, config)
model.print_trainable_parameters()      # e.g. "trainable params: ... || all params: ...
 ↪  || trainable%: ..."

with model.disable_adapter():           # the base model: a free reference policy for KL
 ↪  or DPO
    ref_logits = model(**batch).logits

merged = model.merge_and_unload()      # W0 + (alpha/r) B A baked in; a plain model again
merged.save_pretrained("merged-model")
```

With TRL you normally pass `peft_config=LoraConfig(...)` to the trainer instead of wrapping the model yourself; the trainer then uses the adapter-disabled model as the reference for DPO and GRPO.

## 10.16 Common mistakes

- Expecting $A$ to change on the first step (its gradient is zero while $B = 0$).
- Using attention-only LoRA and concluding "LoRA does not work for this task".
- Using very large batch sizes with LoRA without checking against a smaller batch.
- Merging a QLoRA adapter into the 4-bit weights instead of into a bf16 copy of the base.

**Interview check**

**Q1. Write the LoRA forward pass and explain the initialisation.** $h = W_0x + \frac{\alpha}{r}BAx$ with $W_0$ frozen; $A$ random, $B = 0$, so the model starts exactly at the base model.
**Q2. How many parameters does LoRA add to a $d \times k$ matrix?** $r(d + k)$.
**Q3. Does LoRA add inference latency?** Not after merging $W_0 + \frac{\alpha}{r}BA$. Unmerged, it adds a small extra matmul.
**Q4. What does rsLoRA change and why?** The scale from $\alpha/r$ to $\alpha/\sqrt{r}$, so adapter outputs and gradients do not shrink as rank grows.
**Q5. What are the three key ideas of QLoRA?** NF4 4-bit quantisation of the frozen base, double quantisation of the scales, and paged optimizers.
**Q6. Which modules should LoRA target?** All linear layers, including MLP, usually give results closest to full fine-tuning; attention-only LoRA underperforms.
**Q7. What is DoRA?** It decomposes weights into column magnitudes and directions, trains the magnitudes directly and the direction with LoRA.

**Notation on this page:** $A$: LoRA down-projection ($r \times d_{\text{in}}$), random at start • $B$: LoRA up-projection ($d_{\text{out}} \times r$), zero at start • $\alpha$: LoRA scaling numerator (`lora_alpha`) • $x$: input vector to the layer • $r$: LoRA rank • $W_0$: frozen pretrained weight; $W'$: adapted weight • $h$: output of the adapted layer • $d_{\text{in}}, d_{\text{out}}$: input and output sizes of the layer • $k$: input size of a layer; top-$k$ singular directions

**Q8. Why does LoRA work well for RL even at low rank?** RL gives little information per episode (a scalar reward), so the total change needed is small.
**Q9. Why does LoRA need a higher learning rate than full fine-tuning?** Because $B$ starts at zero and the update is scaled by $\alpha/r$, the effective change in the weights per step is small at the same nominal learning rate.
**Q10. Derive the gradients of LoRA's $A$ and $B$.** With $G = \partial\mathcal{L}/\partial W$ and $s = \alpha/r$: $\partial\mathcal{L}/\partial A = sB^\top G$, $\partial\mathcal{L}/\partial B = sGA^\top$.
**Q11. What happens to $A$ on the first step, and why does it matter?** Nothing: $B = 0$ so its gradient is zero. LoRA's effective learning rate grows as $B$ grows, an implicit warm-up; short runs want a higher LR.
**Q12. After the first step, what subspace does the LoRA update live in?** $\Delta W = -\eta s^2 GA^\top A$: the full gradient projected onto $A$'s row space.
**Q13. What did "LoRA Learns Less and Forgets Less" find?** LoRA underperformed full fine-tuning on code/maths targets but forgot less and kept diversity; full fine-tuning updates have 10–100x higher rank than typical LoRA.
**Q14. How much compute does LoRA save per step?** About one third: $2N^2 + 6Nr$ vs $3N^2$ multiply-adds per weight; the main saving is memory.
**Q15. Estimate the information learned in an RL run and compare it to adapter size.** About one bit per episode: e.g. 10k problems × 32 samples ≈ 320k bits, below rank-1 LoRA's ~3M parameters on an 8B model.
**Q16. Why might LoRA lag full fine-tuning at large batch sizes?** The $BA$ product has different optimisation dynamics; "LoRA Without Regret" found the gap grows with batch size and is not fixed by higher rank.

**Key takeaways**

- Full fine-tuning with AdamW costs about 16 bytes per parameter; LoRA trains about 0.1 to 1% of parameters.
- LoRA: $W' = W_0 + \frac{\alpha}{r}BA$, $B = 0$ at init, mergeable with no inference cost.
- Target all linear layers; use a learning rate about 10x full fine-tuning.
- rsLoRA ($\alpha/\sqrt{r}$), LoRA+ (bigger LR for $B$), PiSSA (SVD init) and DoRA (magnitude/direction) are refinements.
- QLoRA: NF4 base + double quantisation + paged optimizers.
- LoRA works well for RL; the reference model is the base with the adapter off.
- $\partial\mathcal{L}/\partial A = sB^\top G$, $\partial\mathcal{L}/\partial B = sGA^\top$; $A$ is frozen at step 1.
- LoRA learns less and forgets less; on post-training-sized data with all layers it can match full fine-tuning.
- RL needs little capacity (~1 bit/episode); LoRA saves memory, but only ~1/3 compute.

**Notation on this page:** $B$: LoRA up-projection ($d_{\text{out}} \times r$), zero at start • $\alpha$: LoRA scaling numerator (`lora_alpha`) • $r$: LoRA rank • $A$: LoRA down-projection ($r \times d_{\text{in}}$), random at start • $G$: group size (samples per prompt) • $W_0$: frozen pretrained weight; $W'$: adapted weight • $L$: number of layers • $s = \alpha/r$: LoRA scale • $\eta_A, \eta_B$: learning rates for $A$ and $B$ • $\Delta W$: the learned change to the weight • $N$: layer width (FLOP count)

CHAPTER 11

# Number Formats and Quantisation

**Why this chapter matters.**
Post-training touches number formats everywhere: bf16 training, fp32 master weights, 4-bit QLoRA bases, FP8 rollouts, and quantised models in production. In RL, a precision mismatch between the rollout engine and the trainer is a known cause of unstable training (Chapter 33). This chapter explains floating-point formats, how quantisation works, and the main methods (LLM.int8, GPTQ, AWQ, SmoothQuant, FP8).

## 11.1 Floating-point formats

A floating-point number has a **sign** bit, **exponent** bits (which set the range, how big or small a number can be) and **mantissa** bits (which set the precision, how many significant digits it has). The value is roughly $(-1)^{\text{sign}} \times 1.\text{mantissa} \times 2^{\text{exponent}-\text{bias}}$.

| Format | Exponent | Mantissa | Max value | Notes |
|---|---|---|---|---|
| fp32 | 8 | 23 | $3.4 \times 10^{38}$ | Default for optimizer states |
| fp16 | 5 | 10 | 65,504 | Small range: needs loss scaling |
| bf16 | 8 | 7 | $3.4 \times 10^{38}$ | Same range as fp32, low precision |
| FP8 E4M3 | 4 | 3 | 448 | Forward pass (weights, activations) |
| FP8 E5M2 | 5 | 2 | 57,344 | Gradients (needs range) |

Below 8 bits, formats come with **block scales**. **NF4** (Chapter 10) uses 16 levels at normal-distribution quantiles with one scale per 64 weights. **MXFP4** (an OCP "microscaling" format) stores blocks of 32 values in FP4 (E2M1: 1 sign, 2 exponent, 1 mantissa bit) that share one 8-bit power-of-two scale; OpenAI's gpt-oss models were released with their MoE weights in MXFP4. **INT4/INT8** are plain integers with separate scales (next section).

The FP8 formats were standardised in *FP8 Formats for Deep Learning* (Micikevicius et al., arXiv 2209.05433).

**bf16** ("brain float 16") keeps fp32's 8 exponent bits, so it almost never overflows, but has only 7 mantissa bits: about 2 to 3 significant decimal digits. Near 1.0, the gap between neighbouring bf16 numbers is $2^{-7} \approx 0.0078$. Adding a tiny update to a large weight in bf16 can do nothing at all:

```
import torch
w = torch.tensor(1.0, dtype=torch.bfloat16)
print(w + 1e-3)                    # tensor(1., dtype=torch.bfloat16): the update
 ↪  vanished
print(torch.tensor(1.0) + 1e-3)    # fp32 keeps it
print(torch.finfo(torch.bfloat16).eps, torch.finfo(torch.float16).max)
```

This is why mixed-precision training keeps an **fp32 master copy** of the weights for the optimizer update, while the forward and backward passes run in bf16.

**Why this, not that? Why bf16, not fp16, for training?**

**Why.** bf16 has the same range as fp32, so activations and gradients almost never overflow or underflow, and you do not need loss scaling. Training "just works" with the same hyperparameters.
**Trade-off.** bf16 has 3 fewer mantissa bits than fp16, so each number is less precise. For RL, this can matter: the same computation in a rollout engine and a trainer may round differently and give different log-probabilities.
**When the alternative wins.** fp16 wins on hardware without bf16 support (older GPUs), and some recent work argues that fp16's extra precision reduces the train/inference mismatch in RL fine-tuning. With careful loss scaling, fp16 training is stable.

## 11.2 Quantisation basics

**Quantisation** stores numbers with fewer bits by mapping a range of real values onto a small set of integer levels. For $b$-bit integers:

**Symmetric (absmax) quantisation.** Use a scale $s = \max|x|/(2^{b-1}-1)$ and

$$q = \text{round}(x/s), \qquad \hat{x} = s\,q.$$

Here $q$ is the stored integer, $\hat{x}$ is the dequantised (approximate) value, and round rounds to the nearest integer.

**Asymmetric (zero-point) quantisation.** For data that is not centred on zero, use the full range with an integer **zero-point** $z$:

$$s = \frac{x_{\max} - x_{\min}}{2^b - 1}, \quad z = \text{round}\left(-\frac{x_{\min}}{s}\right), \quad q = \text{clamp}\left(\text{round}(x/s) + z,\, 0,\, 2^b - 1\right), \quad \hat{x} = s\,(q - z).$$

**Granularity.** One scale for a whole tensor (per-tensor) is crude: one large value forces a large scale and wastes levels on everything else. **Per-channel** (one scale per output row) and **per-group** (one scale per block of, say, 64 or 128 weights) are much more accurate, at the cost of storing more scales.

**Notation on this page:** $b$: number of bits • $x$: real value to quantise; $\hat{x}$: dequantised approximation • $s$: quantisation scale (real value per integer step) • $q$: the stored integer • $z$: zero-point (integer that represents real 0)

*Worked example.* Quantise $x = (0.1, -0.5, 0.9, 2.0)$ to 4-bit symmetric. $s = 2.0/7 \approx 0.2857$. Then $x/s = (0.35, -1.75, 3.15, 7.0)$, so $q = (0, -2, 3, 7)$ and $\hat{x} = (0, -0.571, 0.857, 2.0)$. The small value 0.1 became 0: small weights are crushed when one large value sets the scale.

```
import torch

def quant_dequant(x, bits=4, group=None):
    """Symmetric absmax quantisation; group=None means one scale per tensor."""
    shape = x.shape
    xg = x.reshape(-1, group) if group else x.reshape(1, -1)
    qmax = 2 ** (bits - 1) - 1
    s = xg.abs().amax(dim=1, keepdim=True).clamp(min=1e-12) / qmax
    q = torch.clamp(torch.round(xg / s), -qmax - 1, qmax)
    return (q * s).reshape(shape)

torch.manual_seed(0)
W = torch.randn(256, 256)
W[0, 0] = 40.0                                  # one outlier weight
for g in (None, 128, 32):
    err = (quant_dequant(W, 4, g) - W).pow(2).mean().sqrt()
    print(f"group={g}: RMS error {err:.4f}")
```

The outlier ruins per-tensor quantisation; smaller groups confine its damage to one group.

## 11.3 Outliers and LLM.int8

*LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale* (Dettmers et al., arXiv 2208.07339) found that once models pass about 6.7B parameters, a few hidden dimensions carry **outlier activations** up to 20 times larger than the rest. Quantising them with everything else destroys quality. LLM.int8 splits the matmul: the few outlier dimensions run in fp16, and the rest in int8. This keeps quality at 8 bits, at some speed cost.

*SmoothQuant* (Xiao et al., arXiv 2211.10438) takes another route for quantising both weights and activations: it divides the activations by a per-channel factor and multiplies the weights by the same factor. The product is unchanged, but the hard-to-quantise outliers move from activations into weights, which are easier to quantise.

## 11.4 Weight-only post-training quantisation: GPTQ and AWQ

For inference, it is common to quantise only the weights to 4 bits and keep activations in 16 bits. Two well-known methods use a small calibration dataset:

**GPTQ** (*GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*, Frantar et al., arXiv 2210.17323) quantises weights one column at a time and updates the remaining unquantised columns to compensate for the error, using second-order (Hessian) information from calibration activations. It minimises the change in each layer's *output*, $\|WX - \hat{W}X\|^2$, not the change in the weights.

**AWQ** (*AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*, Lin et al.,

**Notation on this page:** $x$: real value to quantise; $\hat{x}$: dequantised approximation • $s$: quantisation scale (real value per integer step) • $q$: the stored integer • $W$: weight matrix; $\hat{W}$: its quantised version • $X$: calibration activations (inputs to the layer)

arXiv 2306.00978) observes that about 1% of weight channels matter most: those that meet large activations. It scales these salient channels up before quantising (and the activations down to compensate) so they lose less precision. No back-propagation or reconstruction is needed.

**Why this, not that? Why calibration-based methods (GPTQ, AWQ), not plain rounding?**

**Why.** Plain round-to-nearest at 4 bits damages large models noticeably. GPTQ and AWQ use a few hundred calibration examples to protect what matters for the layer outputs, and recover most of the lost quality.
**Trade-off.** They need calibration data (whose domain affects the result), extra time to quantise, and specific kernels to run fast.
**When the alternative wins.** At 8 bits, plain rounding with per-channel scales is usually good enough. For training (QLoRA), NF4 with simple per-block scales is used because the base must be quantised quickly and the adapters can absorb errors.

## 11.5 Quantisation-aware training

**Post-training quantisation** (PTQ, like GPTQ and AWQ) quantises a finished model. **Quantisation-aware training** (QAT) instead simulates quantisation during training ("fake quantisation": quantise then immediately dequantise in the forward pass) so the weights learn to sit where rounding hurts least. Rounding has zero gradient almost everywhere, so QAT uses the **straight-through estimator**: in the backward pass, treat the rounding step as the identity. QAT costs a training run but recovers more quality at very low bit-widths (4 bits and below), and some labs now apply it during the final post-training stage so the released low-bit model is the one that was trained.

```
import torch
def fake_quant(x, bits=4):
    s = x.detach().abs().max() / (2 ** (bits - 1) - 1)
    q = torch.round(x / s) * s
    return x + (q - x).detach()          # forward: q; backward: identity (STE)
w = torch.tensor([0.31, -0.72, 1.0], requires_grad=True)
fake_quant(w).sum().backward()
print(fake_quant(w).detach(), w.grad)    # gradient passes straight through
```

## 11.6 FP8 training and inference

Recent GPUs run FP8 matmuls about twice as fast as bf16. **FP8 training** keeps master weights and optimizer states in higher precision and casts matmul inputs to FP8 with fine-grained scales. DeepSeek-V3 was trained largely in FP8, using block-wise scaling (small tiles of activations and weights each get their own scale). FP8 is also increasingly used for **RL rollouts**, to make generation faster.

## 11.7 Quantisation and RL

In RL, rollouts are generated by a fast inference engine (vLLM or SGLang), and the policy gradient is computed by the trainer. If the two use different precisions, kernels or quantisation, then the probability of a sampled token under the rollout engine, $\pi_{\text{infer}}$, differs from its probability under the trainer, $\pi_{\text{train}}$. The rollouts become slightly **off-policy**. MiniMax-M1 (arXiv 2506.13585) reported that this mismatch blocked reward growth until they computed the LM head in fp32. Chapter 33 covers the fixes (importance-sampling correction, masking). TRL enables `vllm_importance_sampling_correction=True` by default in `GRPOConfig`.

## 11.8 Quantisation error: the numbers behind the methods

A **uniform quantiser** with $b$ bits maps a range $[-a, a]$ onto $2^b$ evenly spaced levels with step $\Delta = 2a/(2^b - 1)$. If values are spread evenly within a step, the rounding error is uniform on $[-\Delta/2, \Delta/2]$, with variance

$$\sigma^2_{\text{err}} = \frac{\Delta^2}{12}.$$

Each extra bit halves $\Delta$ and so quarters the error variance: about 6 dB of signal-to-noise per bit. Going from 8 to 4 bits multiplies the error variance by $2^8 = 256$.

The range $a$ is the real problem. With **absmax** scaling, $a$ is the largest magnitude in the group of values that shares one scale. One outlier ten times larger than the rest makes $\Delta$ ten times larger for every value in its group. Three granularities trade storage for error:

- **Per-tensor**: one scale for the whole matrix. Cheapest, worst with outliers.
- **Per-channel** (per output row): one scale per row. Standard for INT8 weights.
- **Group-wise** (blocks of 32–128 consecutive values): one scale per block. Standard for 4-bit weights (GPTQ, AWQ, bitsandbytes NF4 with block size 64).

```
import torch
torch.manual_seed(0)
W = torch.randn(256, 1024) * 0.02
W[3, 17] = 0.6; W[100, 900] = -0.5                    # two outliers, like real LLM weights
def quant_absmax(x, bits, group=None, per_row=False):
    qmax = 2 ** (bits - 1) - 1
    if group:
        xs = x.reshape(-1, group); s = xs.abs().amax(1, keepdim=True) / qmax
        return (torch.round(xs / s).clamp(-qmax, qmax) * s).reshape(x.shape)
    s = (x.abs().amax(1, keepdim=True) if per_row else x.abs().max()) / qmax
    return torch.round(x / s).clamp(-qmax, qmax) * s
def rel_err(a, b): return ((a - b).norm() / b.norm()).item()
for bits in (8, 4):
    print(f"{bits}-bit  per-tensor {rel_err(quant_absmax(W, bits), W):.3f}  "
          f"per-row {rel_err(quant_absmax(W, bits, per_row=True), W):.3f}  "
          f"group-64 {rel_err(quant_absmax(W, bits, group=64), W):.3f}")
# NF4-style levels: quantiles of a normal distribution vs evenly spaced levels, for
↪ normal weights
normal = torch.distributions.Normal(0, 1)
```

**Notation on this page:** $b$: number of bits • $\sigma(z)$: the logistic (sigmoid) function $1/(1 + e^{-z})$

```
probs = torch.linspace(0.5 / 16, 1 - 0.5 / 16, 16)
nf_levels = normal.icdf(probs); nf_levels = nf_levels / nf_levels.abs().max()
uni_levels = torch.linspace(-1, 1, 16)
def snap(x, levels):
    xs = x.reshape(-1, 64); s = xs.abs().amax(1, keepdim=True)
    idx = ((xs / s).unsqueeze(-1) - levels).abs().argmin(-1)
    return (levels[idx] * s).reshape(x.shape)
Wn = torch.randn(256, 1024)
print("4-bit, group 64, normal weights: uniform levels %.3f  normal-quantile levels
 ↪  %.3f" %
      (rel_err(snap(Wn, uni_levels), Wn), rel_err(snap(Wn, nf_levels), Wn)))
```

The outliers ruin per-tensor 4-bit quantisation: the step becomes so large that most ordinary weights round to zero (95% relative error). Per-row scales repair most of the damage, because only two rows contain outliers, and group-wise scales do best by confining each outlier to one block of 64. The last line shows why NF4 places its 16 levels at quantiles of a normal distribution: for normally distributed weights it puts levels where values are dense, cutting the error by about a tenth compared with evenly spaced levels at the same bit width. (This simple version uses equal-probability quantiles; the real NF4 is constructed so that zero is represented exactly and the levels are asymmetric, but the principle is the same.)

## 11.9 How many bits does "4-bit" really cost?

The scales are stored too. With blocks of $B$ values and one $s$-bit scale per block, the cost is $4 + s/B$ bits per parameter. QLoRA's numbers (*QLoRA: Efficient Finetuning of Quantized LLMs*, arXiv 2305.14314): block size 64 with 32-bit scales costs $32/64 = 0.5$ extra bits per parameter. **Double quantisation** quantises those scales to 8 bits in blocks of 256, with one 32-bit second-level constant per block:

$$\frac{8}{64} + \frac{32}{64 \times 256} \approx 0.127 \text{ bits per parameter},$$

a saving of about 0.373 bits per parameter, roughly 3 GB for a 65B model.

**Worked example.** An 8B model with 7B quantised parameters (embeddings and output head stay in 16-bit, about 1B parameters): $7 \times 10^9 \times 4.127/8 \approx 3.6$ GB for the quantised part, plus $1 \times 10^9 \times 2 = 2$ GB for 16-bit embeddings and head, about 5.6 GB in total, which matches Project 5's budget.

**Why this, not that? Why group-wise scales, not one scale per tensor or per row?**

**Reason.** LLM weights have rare large values; a scale is only as good as the largest value it must cover. Small groups confine each outlier to its own block, so 4-bit quantisation stays accurate. **Trade-off.** Each group needs a stored scale (0.125–0.5 extra bits per parameter) and kernels must apply many scales, which costs some speed. **When the alternative is better.** At 8 bits the error is small enough that per-channel scales suffice and are simpler and faster; for activations, per-token scales computed on the fly are the usual choice.

## 11.10 Quantising the KV cache

During long generation the KV cache can be larger than the weights (Chapter 43). Storing it in 8-bit floating point halves its size compared with bf16, so twice as many tokens fit, which raises throughput for long responses. vLLM exposes this as the `kv_cache_dtype` engine argument (`auto` by default, meaning the model's dtype; `fp8` variants available), and the related `CacheConfig.cache_dtype` field is listed in Chapter 53. Scales can be computed per tensor or calibrated; accuracy loss is usually small, but check it on your task, because errors in keys change attention patterns for every later token. For RL rollouts a quantised KV cache is another source of train–inference mismatch: the sampler's probabilities differ more from the trainer's, so watch the mismatch metrics of Chapter 43 when you turn it on.

## 11.11 Common mistakes

- **Quantising embeddings and the output head to 4 bits** by default: most libraries keep them in 16-bit because they are sensitive and their size is a small share of a large model.
- **Forgetting the scale overhead** when estimating memory: "4-bit" is about 4.1–4.5 bits per quantised parameter.
- **Turning on an FP8 KV cache in RL rollouts** without checking the trainer–engine log-probability mismatch.

**Interview check**

**Q1. bf16 vs fp16?** Both 16 bits. bf16 has 8 exponent bits (fp32's range) and 7 mantissa bits; fp16 has 5 and 10. bf16 avoids overflow and loss scaling; fp16 is more precise.
**Q2. Why keep fp32 master weights?** Small updates added to bf16 weights round away; fp32 accumulates them.
**Q3. Write symmetric int8 quantisation.** $s = \max|x|/127$, $q = \text{round}(x/s)$, $\hat{x} = sq$.
**Q4. Why per-group quantisation?** One outlier sets the scale for its whole group; smaller groups limit the damage and lower error.
**Q5. What do GPTQ and AWQ do differently?** GPTQ quantises column by column and corrects remaining weights using second-order information; AWQ protects salient weight channels (those with large activations) by scaling them before quantising.
**Q6. Why does quantisation matter for RL?** If rollout and training precision differ, token probabilities differ and training is off-policy; this can destabilise RL unless corrected.
**Q7. What is the straight-through estimator?** In QAT, the backward pass treats the non-differentiable rounding as the identity, so gradients reach the full-precision weights.
**Q8. What are E4M3 and E5M2?** Two FP8 formats: 4 exponent / 3 mantissa bits (more precision, used forwards) and 5/2 (more range, used for gradients).
**Q9. What is the rounding-error variance of a uniform quantiser with step $\Delta$?** $\Delta^2/12$; each extra bit halves $\Delta$ and quarters the variance.
**Q10. Why do outliers hurt absmax quantisation so much?** The scale is set by the largest magnitude in the group, so one outlier enlarges the step for every value sharing that scale.
**Q11. How much does double quantisation save in QLoRA?** About 0.373 bits per parameter: scale overhead falls from 0.5 bits (32-bit scales per 64 values) to about 0.127 bits.

**Notation on this page:** $x$: real value to quantise; $\hat{x}$: dequantised approximation • $s$: quantisation scale (real value per integer step) • $q$: the stored integer

**Q12. Why is a quantised KV cache attractive for long generations, and what is the risk?** It halves cache memory compared with bf16, fitting more or longer sequences; it adds numerical error to attention and increases train–inference mismatch in RL.

**Key takeaways**

- Exponent bits give range; mantissa bits give precision. bf16 = fp32 range, low precision.
- Quantisation: $q = \text{round}(x/s)$ (plus zero-point if asymmetric); per-group scales handle outliers.
- LLM.int8 keeps outlier dimensions in fp16; SmoothQuant moves outliers from activations into weights.
- GPTQ (second-order error correction) and AWQ (protect salient channels) are standard 4-bit weight methods.
- FP8 speeds up training and rollouts; precision mismatch between rollout and trainer makes RL off-policy.
- Uniform quantisation error variance is $\Delta^2/12$; outliers set $\Delta$, so use small groups at 4 bits.
- "4-bit" costs $4 + s/B$ bits; QLoRA double quantisation brings the overhead to about 0.127 bits.
- FP8 KV caches double usable context per GPU; check accuracy and RL mismatch.

**Notation on this page:** $x$: real value to quantise; $\hat{x}$: dequantised approximation • $s$: quantisation scale (real value per integer step) • $q$: the stored integer • $B$: batch size

CHAPTER 12

# Knowledge Distillation

**Why this chapter matters.**
Most small open models are trained largely by **distillation**: a large "teacher" model teaches a small "student". The DeepSeek-R1 distilled models, the Qwen3 small models and many others were made this way. Distillation also connects SFT and RL: on-policy distillation looks almost exactly like RL with a dense, per-token reward. This chapter covers forward and reverse KL, sequence-level vs token-level distillation, and on-policy distillation (GKD, MiniLLM).

## 12.1 The idea

*Distilling the Knowledge in a Neural Network* (Hinton et al., arXiv 1503.02531) trained a small model to match a large model's full output distribution, not just the correct label. The teacher's probabilities on the "wrong" answers carry extra information: that a picture of a 2 looks a bit like a 7. For LLMs, the teacher's next-token distribution shows which other words would also be fine.

Let $p_T(\cdot \mid c)$ be the teacher's next-token distribution in context $c$, and $q_\theta(\cdot \mid c)$ the student's. Token-level distillation minimises a divergence between them at each position.

## 12.2 Three kinds of LLM distillation

**1. Sequence-level distillation (SFT on teacher outputs).** Generate responses with the teacher, then SFT the student on them. Only needs the teacher's text, so it works with closed-model APIs. This is what "distillation" usually means for reasoning models: DeepSeek-R1's distilled Qwen and Llama models were plain SFT on 800k teacher-generated samples (*DeepSeek-R1*, arXiv 2501.12948).

**2. Token-level (logit) distillation, off-policy.** On fixed sequences (from a dataset or the teacher), match the student's full next-token distribution to the teacher's at every position. Needs the teacher's logits (so both models must share a tokenizer, or you need a vocabulary mapping). Gives far more signal per token than text alone.

**3. On-policy distillation.** The *student* generates responses, and the teacher scores every token of them. The student learns on its own mistakes, the same way RL does.

## 12.3 Forward vs reverse KL

The choice of divergence matters a lot, as in Chapter 3.

**Notation on this page:** subscript $T$: teacher; $T$ alone: softmax temperature • $p_T$: the teacher's next-token distribution • $c$, $c_t$: the context (tokens before position $t$) • $\theta$: the model's trainable weights • $q_\theta$: the student's next-token distribution

**Forward KL** $\mathrm{KL}(p_T \,\|\, q_\theta) = \sum_y p_T(y) \log \frac{p_T(y)}{q_\theta(y)}$. Minimising it is the same as cross-entropy with the teacher's soft labels: $-\sum_y p_T(y) \log q_\theta(y)$ plus a constant. It is **mass-covering**: the student is punished heavily wherever the teacher has probability but the student does not, so it spreads probability over all the teacher's options.

**Reverse KL** $\mathrm{KL}(q_\theta \,\|\, p_T) = \sum_y q_\theta(y) \log \frac{q_\theta(y)}{p_T(y)}$. It is **mode-seeking**: the student is punished for putting probability where the teacher has little, so it picks one of the teacher's good options and commits to it.

A small student cannot copy everything a big teacher does. With forward KL it spreads itself thin and puts probability on places between the teacher's modes, which in text means incoherent outputs. With reverse KL it concentrates on what it can do well. This is the argument of *MiniLLM: On-Policy Distillation of Large Language Models* (Gu et al., arXiv 2306.08543), which uses reverse KL with policy-gradient optimisation.

**Generalised Jensen-Shannon divergence** interpolates between the two. With mixture $m = \beta p_T + (1-\beta) q_\theta$,

$$\mathrm{JSD}_\beta(p_T \,\|\, q_\theta) = \beta\, \mathrm{KL}(p_T \,\|\, m) + (1-\beta)\, \mathrm{KL}(q_\theta \,\|\, m).$$

Here $\beta \in [0, 1]$ is the interpolation weight. In TRL's implementation, $\beta$ near 0 behaves like forward KL and $\beta$ near 1 like reverse KL.

```python
import torch

def kl(p, q):        # KL(p || q) for probability vectors
    return (p * (p / q).log()).sum()

teacher = torch.tensor([0.49, 0.02, 0.49])       # two good modes, bad middle
# a small (unimodal) student must either sit between the modes or pick one
cands = {"spread": torch.tensor([0.25, 0.50, 0.25]),
         "mode A": torch.tensor([0.90, 0.08, 0.02])}
for name, q in cands.items():
    print(f"{name:7s} forward KL={kl(teacher, q):.3f}  reverse KL={kl(q, teacher):.3f}")
```

Forward KL prefers the spread student, even though it puts half its probability on the bad middle option. Reverse KL prefers the student that commits to one good mode.

**Why this, not that? Why reverse KL for distilling into a small student, not forward KL?**

**Why.** A small student cannot cover all the teacher's modes. Reverse KL lets it pick modes it can do well instead of averaging them into nonsense. Combined with on-policy samples, it trains the student exactly on the text it will produce.
**Trade-off.** Reverse KL loses diversity (mode collapse) and may ignore valid answers. Computing it on-policy needs student generation, which is slower than training on fixed data.
**When the alternative wins.** Forward KL (or plain SFT) is better when the student is nearly as capable as the teacher, when diversity matters, or early in training when the student's own samples are poor and would give little useful signal.

**Notation on this page:** $\theta$: the model's trainable weights • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $y$: a token, or a generated response $y_1, y_2, \ldots$ • subscript $T$: teacher; $T$ alone: softmax temperature • $p_T$: the teacher's next-token distribution • $q_\theta$: the student's next-token distribution • $\beta$: generalised-JSD weight (towards reverse KL as it grows) • $m$: the mixture $\beta p_T + (1-\beta) q_\theta$

## 12.4 GKD: on-policy distillation

*On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes* (Agarwal et al., arXiv 2306.13649) introduced **Generalised Knowledge Distillation (GKD)**. The core problem it addresses is **exposure bias** (also called train/inference mismatch): in off-policy distillation the student only ever trains on prefixes written by the teacher, but at inference it must continue its *own* prefixes, which may contain mistakes it never learned to recover from.

GKD trains on a mix: with probability $\lambda$ the sequences come from the student (on-policy), otherwise from a fixed dataset; the teacher then gives token-level targets on those sequences, and the loss is the generalised JSD with weight $\beta$. The objective is

$$\mathcal{L}_{\mathrm{GKD}}(\theta) = \mathbb{E}_x \, \mathbb{E}_{y \sim \mathrm{mix}_\lambda}\Big[ \sum_t D_\beta(p_T(\cdot \mid x, y_{<t}) \,\|\, q_\theta(\cdot \mid x, y_{<t}))\Big],$$

where gradients do *not* flow through the sampling of $y$ (unlike RL, which uses the score function).

In TRL 1.16.0:

- The top-level `DistillationTrainer` with `DistillationConfig` generates completions **on-policy** from the student and uses the generalised JSD with `beta=1.0` by default (reverse KL; 0.0 = forward KL, 0.5 = JSD). It supports vLLM for student generation; `temperature` (default 1.0) is used for both sampling and the loss; the default learning rate is $1 \times 10^{-6}$.
- `trl.experimental.gkd.GKDConfig` keeps the original GKD knobs: `lmbda=0.5` (fraction of on-policy student data), `beta=0.5`, `seq_kd=False` (set True to also do sequence-level distillation on teacher outputs), `temperature=0.9`, `max_new_tokens=128`.

## 12.5 On-policy distillation as dense-reward RL

Write the per-token reverse KL on student samples as a reward:

$$r_t = -(\log q_\theta(y_t \mid c_t) - \log p_T(y_t \mid c_t)) = \log p_T(y_t \mid c_t) - \log q_\theta(y_t \mid c_t).$$

Here $c_t$ is the context before token $t$. Summing over tokens gives minus the single-sample estimate of the sequence reverse KL (the $k_1$ estimator of Chapter 3). So on-policy distillation is RL where *every token gets a reward* telling the student how much the teacher agrees with that choice. Compare RL with verifiable rewards, where a whole response of thousands of tokens gets one bit of reward at the end. This density makes on-policy distillation far more sample-efficient than RL. The *Qwen3 Technical Report* (arXiv 2505.09388) describes "strong-to-weak distillation" for its small models (an off-policy stage then an on-policy stage), and reports that on-policy distillation reached better results than RL at about a tenth of the GPU hours. Thinking Machines' "On-Policy Distillation" blog post (2025) popularised the per-token reverse-KL-as-reward view.

**Why this, not that? Why on-policy distillation, not RL, for a small model?**

**Why.** It gives a learning signal at every token instead of one scalar per episode, it needs no reward function or verifier, and it trains on the student's own distribution, so it avoids exposure bias.

**Notation on this page:** $\lambda$: fraction of on-policy (student-generated) sequences • $\beta$: generalised-JSD weight (towards reverse KL as it grows) • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\mathcal{L}$: a loss to minimise • $y_{<t}$: the tokens before position $t$ • $x$: the prompt • $y$: a token, or a generated response $y_1, y_2, \ldots$ • $t$: time step or token position • subscript $T$: teacher; $T$ alone: softmax temperature • $p_T$: the teacher's next-token distribution • *and 5 more: see the Notation table*

**Trade-off.** You need a good teacher with the *same tokenizer* (or a mapping), and its logits on every student token, which costs a teacher forward pass per sample. The student cannot exceed the teacher.
**When the alternative wins.** RL wins when you want to go beyond any teacher, when no strong teacher exists for the task, or when the target is defined by a reward (pass the tests, win the game) rather than by imitation. A common recipe is distillation first, RL after.

## 12.6 Practical points

- **Tokenizer match.** Logit distillation needs the same vocabulary. Across families, use sequence-level distillation or a cross-tokenizer method.
- **Temperature.** Dividing teacher and student logits by $T > 1$ softens the distributions and gives more weight to the "dark knowledge" in small probabilities. Hinton et al. multiply the loss by $T^2$ to keep gradient sizes comparable.
- **Top-$k$ logits.** Storing a full 150k-wide distribution per token is expensive; many pipelines store only the teacher's top-$k$ (say 20 to 100) log-probabilities and renormalise.
- **Reasoning traces.** Distilling long chains of thought from a reasoning teacher gives a small model far more reasoning ability than RL on the small model alone. DeepSeek-R1 reports that distilling R1 into Qwen2.5-32B beat running large-scale RL on Qwen2.5-32B directly (arXiv 2501.12948).

## 12.7 Token-level distillation: the gradients

Let $z$ be the student's logits at one position, $q = \text{softmax}(z)$ the student's distribution and $p$ the teacher's. The two KL directions give very different gradients with respect to the student's logits.

**Forward KL** (teacher first), $D_{\text{KL}}(p\|q) = \sum_v p_v \log p_v - \sum_v p_v \log q_v$. Only the cross-entropy part depends on $z$, and Chapter 2's result gives

$$\nabla_z D_{\text{KL}}(p\|q) = q - p.$$

This is exactly the SFT gradient with a soft target $p$ instead of a one-hot target. Every vocabulary entry gets a push toward the teacher's probability: the student learns the teacher's whole ranking, including which wrong tokens are "nearly right" (Hinton's "dark knowledge").

**Reverse KL** (student first), $D_{\text{KL}}(q\|p) = \sum_v q_v(\log q_v - \log p_v)$. Differentiating (using $\partial q_v / \partial z_k = q_v(\mathbb{1}[v = k] - q_k)$) gives

$$\frac{\partial}{\partial z_k} D_{\text{KL}}(q\|p) = q_k\Big[(\log q_k - \log p_k) - D_{\text{KL}}(q\|p)\Big].$$

Tokens where the student puts more log-probability than the teacher (relative to the average gap) are pushed down, weighted by how much the student currently uses them. A token the student ignores ($q_k \approx 0$) receives almost no gradient, however much the teacher likes it: this is the mode-seeking behaviour of Chapter 3, seen at the level of one position.

**Temperature.** Hinton et al.'s classic recipe softens both distributions with a temperature $T$ and multiplies the loss by $T^2$, because the gradient of the softened loss scales as $1/T^2$; the factor keeps

**Notation on this page:** subscript $T$: teacher; $T$ alone: softmax temperature • $k$: an index or count (see text) • $z$: logits (raw scores before softmax) • $q_\theta$: the student's next-token distribution • $p_T$: the teacher's next-token distribution • $D_{\text{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $D_\beta$: the generalised JSD divergence • $\nabla_\theta$: gradient with respect to $\theta$ • $\mathbb{1}[\cdot]$: 1 if the condition is true, else 0

gradient magnitudes comparable when $T$ changes. TRL's `DistillationConfig.temperature` (default 1.0) applies the temperature to both distributions; for LLM distillation $T = 1$ is the common choice, since the student should match the teacher's actual sampling distribution.

```
import torch
torch.manual_seed(0)
V = 7
z = torch.randn(V, requires_grad=True); p = torch.softmax(torch.randn(V) * 2, 0)
q = torch.softmax(z, 0)
fwd = (p * (p.log() - q.log())).sum(); g_fwd, = torch.autograd.grad(fwd, z)
print("forward KL grad == q - p:", torch.allclose(g_fwd, (q - p).detach(), atol=1e-6))
q = torch.softmax(z, 0)
rev = (q * (q.log() - p.log())).sum(); g_rev, = torch.autograd.grad(rev, z)
formula = (q * ((q.log() - p.log()) - rev)).detach()
print("reverse KL grad == q*(log q - log p - KL):", torch.allclose(g_rev, formula,
 ↪  atol=1e-6))
# A token the teacher likes but the student ignores: compare the push it gets
z2 = torch.tensor([4.0, 0.0, -6.0], requires_grad=True); p2 = torch.tensor([0.5, 0.2,
 ↪  0.3])
q2 = torch.softmax(z2, 0)
gf, = torch.autograd.grad((p2 * (p2.log() - q2.log())).sum(), z2)
q2 = torch.softmax(z2, 0)
gr, = torch.autograd.grad((q2 * (q2.log() - p2.log())).sum(), z2)
print("token 2 (teacher 0.30, student %.4f): forward-KL push %.3f, reverse-KL push %.5f"
 ↪  %
      (q2[2].item(), -gf[2].item(), -gr[2].item()))
```

## 12.8 Inside TRL's distillation loss

The generalised JSD above is what TRL's `DistillationTrainer` computes. In the TRL 1.16 source (`trl/kernels/chunked_divergence.py`), with student $q$, teacher $p$ and mixture $m = (1-\beta)\,q + \beta\,p$:

$$\mathrm{JSD}_\beta(p, q) = \beta\, D_{\mathrm{KL}}(p \| m) + (1-\beta)\, D_{\mathrm{KL}}(q \| m), \qquad 0 < \beta < 1,$$

with the endpoints defined as the two KLs: `beta=0.0` is forward KL $D_{\mathrm{KL}}(p\|q)$ and `beta=1.0` (the default) is reverse KL $D_{\mathrm{KL}}(q\|p)$. The mixture keeps every term finite, which is why the JSD is more stable than either KL when the two distributions have very different supports. The same source projects only the unmasked completion positions through the language-model heads in tiles, so the full vocabulary-sized logits are never stored, and it allows teacher and student to have different hidden sizes: only the vocabulary must match.

**Worked example (memory).** A batch of 16 sequences × 2,048 completion tokens with a 152k vocabulary has $16 \times 2048 \times 152{,}000 \approx 5 \times 10^9$ logits per model. In float32 that is 20 GB for the student and 20 GB for the teacher, before gradients. Computing the divergence in tiles and discarding each tile after use is what makes on-policy distillation fit on one GPU.

**Notation on this page:** subscript $T$: teacher; $T$ alone: softmax temperature • $q_\theta$: the student's next-token distribution • $p_T$: the teacher's next-token distribution • $\beta$: generalised-JSD weight (towards reverse KL as it grows) • $m$: the mixture $\beta p_T + (1-\beta) q_\theta$ • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $D_\beta$: the generalised JSD divergence

**Why this, not that? Why store only the teacher's top-k log-probabilities, not the full distribution, for offline token-level distillation?**

**Reason.** Full distributions cost vocabulary-size floats per token (hundreds of kilobytes); the top 20–100 tokens usually hold almost all the probability, so storing them cuts storage by thousands of times with little loss of signal. **Trade-off.** The tail is lost: the student gets no information about which rare tokens the teacher considers impossible versus merely unlikely, and the truncated distribution must be renormalised (or the remainder treated as one "other" bucket), which biases the target slightly. **When the alternative is better.** On-policy distillation with a live teacher computes the full distribution on the fly (no storage), and small vocabularies or short datasets make full storage affordable.

## 12.9 What distillation costs: a worked comparison

Distil a 32B teacher into a 1.5B student on 100k maths problems, 1,000 tokens per solution. Use about $2N$ FLOPs per token for a forward pass and $6N$ for training.

- **Sequence-level.** Teacher generation: $10^8$ tokens $\times 2{\times}32{\times}10^9 = 6.4{\times}10^{18}$ FLOPs (generation is memory-bound, so wall-clock is worse than FLOPs suggest). Student SFT: $10^8 \times 6 \times 1.5 \times 10^9 = 9 \times 10^{17}$ FLOPs. Done once; the data can be reused.
- **On-policy (one pass over the prompts).** Student generation $10^8 \times 2 \times 1.5 \times 10^9 = 3 \times 10^{17}$; teacher scoring (a prefill, not generation) $6.4{\times}10^{18}$; student training $9{\times}10^{17}$. Similar FLOPs, but the teacher work is a parallel prefill, which runs at high GPU utilisation, and it must be repeated every epoch because the student's samples change.

So on-policy distillation is not more expensive in FLOPs per pass, but it needs the teacher online during training (memory and serving), while sequence-level distillation needs it only once.

## 12.10 Common mistakes

- **Mismatched tokenizers in token-level distillation.** Logit matching needs the same vocabulary; with different tokenizers use sequence-level distillation or a cross-tokenizer alignment method.
- **Forgetting the $T^2$ factor** when using a softmax temperature in classic KD: gradients shrink as $1/T^2$.
- **Comparing distillation runs at different temperatures** for sampling the student and computing the loss: the student learns a distribution different from the one it samples from.

**Interview check**

**Q1. Sequence-level vs token-level distillation?** Sequence-level: SFT on teacher-generated text (needs only text). Token-level: match the full next-token distribution at each position (needs teacher logits and a shared tokenizer).
**Q2. Forward or reverse KL for a small student, and why?** Usually reverse KL: mode-seeking, so the small student commits to modes it can represent instead of spreading prob-

ability between modes.
**Q3. What problem does on-policy distillation solve?** Exposure bias: the student learns on its own generated prefixes, including its mistakes, as it will at inference.
**Q4. What do `lmbda` and `beta` control in GKD?** `lmbda` is the fraction of student-generated (on-policy) sequences; `beta` interpolates the generalised JSD between forward-KL-like and reverse-KL-like behaviour.
**Q5. How is on-policy distillation related to RL?** It is RL with a dense per-token reward $\log p_T - \log q_\theta$, i.e. minimising reverse KL to the teacher on student samples.
**Q6. Why did DeepSeek distil R1 into small models instead of running RL on them?** Distillation from the strong reasoner gave better small models than large-scale RL on the small models, at much lower cost.
**Q7. What is the gradient of forward-KL distillation with respect to the student's logits?** $q - p$: the SFT gradient with the teacher's distribution as a soft target.
**Q8. Why does reverse KL give little gradient to tokens the student ignores?** Its gradient for token $k$ is proportional to $q_k$, so a token with near-zero student probability barely moves, even if the teacher likes it.
**Q9. What does TRL's DistillationConfig `beta` select?** 0.0 forward KL (teacher‖student), 1.0 reverse KL (student‖teacher, the default), in between a generalised JSD with mixture $(1-\beta)q + \beta p$.
**Q10. Why is the generalised JSD numerically safer than KL?** Both distributions are compared to a mixture that is non-zero wherever either is, so no term involves the log of zero probability.

**Key takeaways**

- Distillation transfers a teacher's behaviour: sequence-level (SFT on teacher text), token-level (match logits), on-policy (student samples, teacher scores).
- Forward KL is mass-covering; reverse KL is mode-seeking and usually better for small students.
- GKD mixes on-policy and fixed data and uses the generalised JSD. TRL: `DistillationTrainer` (on-policy, `beta=1.0` reverse KL by default); experimental `GKDConfig` with `lmbda`, `beta`, `seq_kd`.
- On-policy distillation = RL with a dense per-token reward; far more sample-efficient than sparse-reward RL.
- Forward-KL gradient is $q - p$ (soft-target SFT); reverse-KL gradient is $q_k[(\log q_k - \log p_k) - \mathrm{KL}]$, mode-seeking.
- TRL's DistillationTrainer: generalised JSD with `beta` (default 1.0 = reverse KL), tiled logits, same vocabulary required.

**Notation on this page:** $\theta$: the model's trainable weights • subscript $T$: teacher; $T$ alone: softmax temperature • $p_T$: the teacher's next-token distribution • $q_\theta$: the student's next-token distribution • $k$: an index or count (see text) • $\beta$: generalised-JSD weight (towards reverse KL as it grows) • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$

PART III

# Reinforcement Learning from the Ground Up

CHAPTER 13

# The Reinforcement Learning Problem

> **Why this chapter matters.**
> Every post-training method after SFT (RLHF, GRPO, DPO, agent training) is built on reinforcement learning (RL) ideas. Interviewers often start with the basics: what is a Markov decision process, what is a return, why discount, and how does an LLM fit the RL picture? This chapter sets up the vocabulary used for the rest of the book.

## 13.1 Agent, environment, reward

In **reinforcement learning**, an **agent** learns by acting in an **environment** and receiving **rewards**. There is no correct answer to copy, only a score for what the agent did. The loop is:

1. The environment is in some **state** $s_t$ at time step $t$.
2. The agent picks an **action** $a_t$.
3. The environment moves to a new state $s_{t+1}$ and gives a **reward** $r_t$ (a number).
4. Repeat until the **episode** ends (the game is over, the answer is finished, or a time limit is hit).

The sequence $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$ is called a **trajectory** (or rollout, or episode). The agent's goal is to choose actions that make the total reward large.

Compare with supervised learning:

| | Supervised learning | Reinforcement learning |
|---|---|---|
| Feedback | The correct answer | A score for the chosen action |
| Data | Fixed dataset | Generated by the agent's own actions |
| Credit | Per example | Delayed: which earlier action earned the reward? |
| Main difficulty | Generalisation | Exploration, credit assignment, unstable data |

The last two rows are what make RL hard. Because the agent creates its own data, a bad policy produces bad data, which can make the policy worse. And because rewards can come long after the action that caused them, the agent must work out which actions deserve credit, the **credit assignment problem**.

**Notation on this page:** $s$: a state • $t$: time step or token position • $a$: an action • $r_t$: reward at step $t$ • $\tau$: a trajectory (states, actions, rewards of one episode)

## 13.2 Markov decision processes

A **Markov decision process** (MDP) is the standard mathematical model of an RL problem. It is a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma, \rho_0)$:

- $\mathcal{S}$: the set of possible states.
- $\mathcal{A}$: the set of possible actions.
- $P(s' \mid s, a)$: the **transition probability**, the chance of moving to state $s'$ after taking action $a$ in state $s$.
- $R(s, a)$: the reward function (it may also depend on $s'$, or be random).
- $\gamma \in [0, 1]$: the **discount factor** (below).
- $\rho_0$: the distribution of the starting state.

The key assumption is the **Markov property**: the next state and reward depend only on the current state and action, not on the earlier history:

$$P(s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0) = P(s_{t+1} \mid s_t, a_t).$$

In words, the state contains everything that matters for the future. If it does not (a card game where you cannot see the other hands, a robot with a noisy camera), the problem is a **partially observable MDP** (POMDP): the agent receives **observations** that give only partial information about the true state. A standard fix is to treat the whole history of observations as the state. This is exactly what an LLM does: it conditions on all tokens so far.

## 13.3 Policies

A **policy** is the agent's rule for choosing actions.

- A **deterministic** policy maps each state to one action: $a = \mu(s)$.
- A **stochastic** policy gives a probability distribution over actions: $a \sim \pi(\cdot \mid s)$.

We write $\pi_\theta$ for a policy with trainable parameters $\theta$ (the weights of a neural network). An LLM is a stochastic policy: given the text so far, it outputs a probability for every token in the vocabulary.

Stochastic policies are useful for two reasons. They **explore** naturally, because they try different actions. And they make the objective **smooth** in $\theta$: a small change to the weights makes a small change to the action probabilities, so we can use gradients (Chapter 15).

## 13.4 Returns and discounting

The **return** $G_t$ is the total reward collected from step $t$ onwards. With discounting,

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}.$$

The **discount factor** $\gamma$ says how much the agent cares about future rewards compared with immediate ones. With $\gamma = 0.99$, a reward 100 steps away is worth $0.99^{100} \approx 0.37$ of an immediate one.

**Notation on this page:** $\gamma$: discount factor (0 to 1) • $\rho_0$: distribution of the start state • $R(s, a)$: reward function • $\mathcal{S}$: the set of states • $\mathcal{A}$: the set of actions • $P(s' \mid s, a)$: transition probability • $s$: a state • $a$: an action • $t$: time step or token position • $\mu(s)$: a deterministic policy • $\pi_\theta$: the policy (the model), with weights $\theta$ • *and 5 more: see the Notation table*

Returns satisfy a simple recursion, which we will use constantly:

$$G_t = r_t + \gamma G_{t+1}.$$

*Worked example.* Rewards $r_0, r_1, r_2 = 0, 0, 1$ with $\gamma = 0.9$. Working backwards: $G_2 = 1$, $G_1 = 0 + 0.9 \times 1 = 0.9$, $G_0 = 0 + 0.9 \times 0.9 = 0.81$.

```python
def returns(rewards, gamma):
    """Discounted return G_t for every step, computed backwards."""
    G, out = 0.0, []
    for r in reversed(rewards):
        G = r + gamma * G
        out.append(G)
    return out[::-1]

print(returns([0, 0, 1], 0.9))      # [0.81, 0.9, 1.0]
print(returns([1, 1, 1, 1], 1.0))   # [4.0, 3.0, 2.0, 1.0]
```

The objective of RL is to find parameters that maximise the **expected return** from the start:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[G_0] = \mathbb{E}_{\tau \sim \pi_\theta}\Big[\sum_t \gamma^t r_t\Big],$$

where $\tau \sim \pi_\theta$ means trajectories are generated by starting from $\rho_0$, choosing actions with $\pi_\theta$ and following the environment's transitions.

**Why this, not that? Why discount future rewards, not just add them up?**

**Why.** Discounting keeps returns finite in never-ending tasks, makes the maths of value functions work (Chapter 14), and reduces variance: far-future rewards are noisy and only loosely connected to the current action, so shrinking them helps learning.
**Trade-off.** Discounting changes the problem: the agent becomes short-sighted and may prefer a quick small reward to a large delayed one. It also biases gradient estimates.
**When the alternative wins.** For short, fixed-length episodes where only the final outcome matters, such as an LLM writing one answer that gets one reward at the end, $\gamma = 1$ is standard. Discounting would penalise long correct reasoning for no good reason. Almost all LLM RL (PPO-RLHF, GRPO) uses $\gamma = 1$.

## 13.5 The LLM as a policy

How does text generation fit into an MDP? There are two common views.

**Token-level MDP.** The state is the prompt plus the tokens generated so far: $s_t = (x, y_{<t})$. The action is the next token $a_t = y_t$, chosen from the vocabulary. The transition is deterministic: append the token, $s_{t+1} = (x, y_{\le t})$. The episode ends at the end-of-sequence token or a length limit. Rewards are usually zero for every token except the last, which gets the score of the whole response (from a reward model or a verifier). PPO-RLHF often also adds a small per-token KL penalty (Chapter 20).

**Notation on this page:** $\gamma$: discount factor (0 to 1) • $G_t$: return from step $t$ (discounted sum of future rewards) • $r_t$: reward at step $t$ • $t$: time step or token position • $G$: group size (samples per prompt) • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\tau$: a trajectory (states, actions, rewards of one episode) • $J(\theta)$: objective to maximise (expected reward) • $\rho_0$: distribution of the start state • *and 5 more: see the Notation table*

**Sequence-level bandit.** The whole response is one action. The prompt $x$ is the "state" (often called the context), the action is the complete response $y \sim \pi_\theta(\cdot \mid x)$, and the reward is $r(x, y)$. There is one step per episode. This is a **contextual bandit**. Its objective is

$$J(\theta) = \mathbb{E}_{x \sim \mathcal{D}} \, \mathbb{E}_{y \sim \pi_\theta(\cdot|x)}[r(x, y)].$$

The probability of a full response factorises over tokens: $\pi_\theta(y \mid x) = \prod_t \pi_\theta(y_t \mid x, y_{<t})$, so the two views share the same gradient when $\gamma = 1$ and the only reward is at the end.

The choice matters for *how credit is assigned*. In the token view, a value function can estimate how good each prefix is and give different tokens different advantages (PPO with a critic). In the bandit view, every token in the response gets the same advantage (GRPO, RLOO, REINFORCE).

**Multi-turn agents** (Chapter 34) bring back a true multi-step MDP: the model acts (writes a tool call), the environment responds (tool output), and the episode continues over many turns. Here, unlike token generation, transitions are not deterministic and not under the model's control.

**Why this, not that? Why treat a whole response as one action (bandit view), not each token as an action?**

**Why.** The reward is only given for the complete response, so per-token credit must be guessed by a learned critic, which is hard to train for LLMs and costs a second large model. The bandit view gives one advantage per response, which is simple, needs no critic and works well with a group baseline (GRPO).
**Trade-off.** Every token in a response gets the same credit, good tokens in a bad answer are pushed down along with the bad ones, and learning signal per token is weak for very long responses.
**When the alternative wins.** When rewards arrive during the episode (tool use, multi-turn tasks, per-step process rewards), or when responses are extremely long and fine-grained credit assignment matters, the token- or step-level view with a value function or process reward can learn faster.

## 13.6 Exploration and exploitation

The agent must balance **exploitation** (choosing actions known to give good reward) and **exploration** (trying actions whose value is uncertain, which might be better). Too little exploration, and the agent gets stuck on the first decent strategy. Too much, and it wastes time on bad actions.

Classic ideas:

- **$\epsilon$-greedy**: with probability $\epsilon$ take a random action, otherwise the best known one.
- **Entropy bonus**: add the policy's entropy to the objective, so the policy is rewarded for staying random (Chapter 17).
- **Optimism**: treat untried actions as good until shown otherwise (as in upper confidence bound methods for bandits).

For LLMs, exploration comes mostly from **sampling temperature** and from sampling **several responses per prompt**. A key limit: the model can only explore answers it gives some probability to. If the correct reasoning path has near-zero probability, sampling will never find it, and RL cannot

**Notation on this page:** $x$: the prompt • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $y$: the response; $y_t$ its token $t$ • $r_t$: reward at step $t$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $J(\theta)$: objective to maximise (expected reward) • $\mathcal{D}$: a dataset • $y_{<t}$: the tokens before position $t$ • $t$: time step or token position • $\gamma$: discount factor (0 to 1) • *and 1 more: see the Notation table*

reinforce it. This is why RL on LLMs depends so heavily on a strong base model (Chapter 26), and why **entropy collapse**, where the policy becomes nearly deterministic and stops exploring, is a key failure mode (Chapter 31).

## 13.7 Worked example: one LLM episode, written out as an MDP

Take the prompt "2+2=" and a policy that writes "4", then the end-of-sequence token. Written as an MDP with $\gamma = 1$:

| $t$ | State $s_t$ (prompt + tokens so far) | Action $a_t$ | $\pi_\theta(a_t \mid s_t)$ | Reward $r_t$ |
|---|---|---|---|---|
| 0 | "2+2=" | "4" | 0.90 | 0 |
| 1 | "2+2=4" | <eos> | 0.95 | +1 (verifier: correct) |

The probability of the whole response is the product of the per-token probabilities, $0.90 \times 0.95 = 0.855$, and its log-probability is the sum, $\log 0.90 + \log 0.95 = -0.157$. The return from $t = 0$ is $G_0 = 0 + 1 = 1$. Two facts make LLMs a special MDP:

1. **Transitions are deterministic and known.** The next state is the current state with the chosen token appended. All randomness is in the policy. This is why model-based planning tricks from games are not needed for the dynamics, and why $Q(s_t, a_t) = V(s_{t+1})$ (Chapter 14).
2. **The state is the whole history.** The Markov property holds trivially because the state *is* the full prefix. The cost is that the state space is astronomically large ($|V|^t$ prefixes), so everything must be learned with function approximation (the network itself).

**KL as a per-token reward.** In PPO-based RLHF (Chapter 20) the KL penalty is usually folded into the reward at every token:

$$r_t = -\beta \log \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\text{ref}}(a_t \mid s_t)} \;+\; \mathbb{1}[t = T]\, R(x, y).$$

Summing over the episode, the return is $R(x,y) - \beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}$, whose expectation is exactly the KL-regularised objective. So the "sparse" RLHF reward is really dense: a KL term at every token plus the reward model's score at the end.

The snippet below builds a tiny "language model" over three tokens with an end token, enumerates every possible response up to length 3, and checks the policy-gradient formula of Chapter 15 against the exact gradient.

```
import itertools, torch
torch.manual_seed(0)
VOC = ["a", "b", "c", "<eos>"]; EOS = 3; MAXLEN = 3
theta = torch.randn(len(VOC), len(VOC), requires_grad=True)          # bigram logits:
 ↪  row = previous token
def seq_logprob(seq, prev=0):
```

**Notation on this page:** $\gamma$: discount factor (0 to 1) • $t$: time step or token position • $s$: a state • $a$: an action • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $r_t$: reward at step $t$ • $G$: group size (samples per prompt) • $V(s)$: state value (expected future return from $s$) • $Q(s,a)$: action value (expected return after taking $a$ in $s$) • $V$: vocabulary size • *and 7 more: see the Notation table*

```
    lp = 0.0
    for tok in seq:
        lp = lp + torch.log_softmax(theta[prev], 0)[tok]; prev = tok
    return lp
def reward(seq):
    text = "".join(VOC[t] for t in seq if t != EOS)
    return {"ba": 1.0, "c": 0.5, "cb": 0.3}.get(text, 0.0) if seq[-1] == EOS else 0.0
     ↪ # truncated = 0
# every response: up to MAXLEN tokens, ending with <eos> (or cut at MAXLEN without it)
seqs = [list(p) + [EOS] for n in range(MAXLEN) for p in itertools.product(range(3),
 ↪ repeat=n)]
seqs += [list(p) for p in itertools.product(range(3), repeat=MAXLEN)]       # truncated
 ↪ responses
probs = torch.stack([seq_logprob(s).exp() for s in seqs])
print("number of responses %d, total probability %.6f" % (len(seqs),
 ↪ probs.sum().item()))
J = sum(p * reward(s) for p, s in zip(probs, seqs))
g_exact, = torch.autograd.grad(J, theta)
# REINFORCE estimate: sample responses, average R * grad log pi
def reinforce(N):
    g_est = torch.zeros_like(theta)
    for i in torch.multinomial(probs.detach(), N, replacement=True).tolist():
        s = seqs[i]
        if reward(s):
            g, = torch.autograd.grad(seq_logprob(s), theta); g_est += reward(s) * g
    return g_est / N
for N in (20, 200, 2000):
    cos = torch.nn.functional.cosine_similarity(g_exact.flatten(),
 ↪ reinforce(N).flatten(), 0)
    print("N = %4d samples: cosine(REINFORCE estimate, exact gradient) = %.3f" % (N,
     ↪ cos.item()))
print("exact J = %.4f" % J.item())
```

The probabilities of all complete and truncated responses sum to one, so the enumeration is a valid distribution over episodes, and the sampled REINFORCE gradient approaches the exact gradient's direction as the number of samples grows; with few samples it is noisy, which is the variance problem baselines address (Chapter 15). With a real vocabulary, enumeration is impossible and sampling is the only option.

## 13.8 Partial observability: when the prefix is not the whole state

In multi-turn agent tasks the environment has state the model does not see: the contents of a database, a hidden goal of a simulated user, the result of a web search not yet run. Formally this is a **partially observable MDP (POMDP)**: the agent receives observations $o_t$ that depend on a hidden state $s_t$. The model's context (all observations and actions so far) acts as its **belief state**, a summary of what it knows. Practical consequences:

- **Two episodes with the same context can have different outcomes**, so rewards are noisier than in single-turn tasks; more samples per prompt are needed for a good baseline.

**Notation on this page:** $t$: time step or token position • $s$: a state

- **Information-gathering actions have value.** Asking a clarifying question or calling a search tool earns no reward directly but changes the belief state; credit assignment must reach back to them (Chapter 34).
- **Environment randomness must be controlled** (seeds, cached tool results) so that groups of samples for one prompt see the same hidden state; otherwise group-relative advantages compare different problems.

## 13.9 Horizon, discounting and the "effective horizon"

With discount $\gamma < 1$ a reward $k$ steps away is weighted by $\gamma^k$. The weights sum to $1/(1-\gamma)$, called the **effective horizon**: $\gamma = 0.99$ looks about 100 steps ahead. For token-level LLM RL with responses of thousands of tokens, even $\gamma = 0.999$ (horizon 1,000) would shrink the final reward at the first token of a 4,000-token answer by $0.999^{4000} \approx 0.018$. That is why LLM RL uses $\gamma = 1$ and handles variance with baselines instead of discounting. In multi-turn agent RL, discounting is sometimes applied **per turn** rather than per token, to encourage finishing in fewer turns.

**Why this, not that? Why a sparse outcome reward, not hand-designed per-token rewards, in most LLM RL?**

**Reason.** Outcome rewards (correct or not, preferred or not) measure what we actually want and are hard to game; hand-designed intermediate rewards encode guesses about *how* to solve the task, which the policy then exploits or which block better strategies. **Trade-off.** Sparse rewards make credit assignment hard: every token of a long response gets the same signal, so learning needs many samples per prompt and good baselines. **When the alternative is better.** Long multi-turn agent tasks where success is rare, tasks with natural verified milestones (tests passing one by one, sub-goals reached), or process reward models validated against outcomes (Chapter 29); and the KL term, which is already a dense per-token reward.

## 13.10 Common mistakes

- Confusing the *reward* (one step) with the *return* (sum of future rewards).
- Forgetting that in RL the data distribution depends on the policy: a dataset collected by an old policy is **off-policy** for the new one.
- Assuming the token-level and sequence-level views give different optimal policies. With $\gamma = 1$ and a terminal reward, they optimise the same objective; they differ in how gradients are estimated.
- **Assuming the context is the full state in agent tasks.** Hidden environment state makes it a POMDP; control seeds so groups are comparable.
- **Using $\gamma < 1$ per token** for long responses: early tokens get almost no signal.

**Interview check**

**Q1. Define an MDP.** A tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma, \rho_0)$: states, actions, transition probabilities, reward function, discount and start-state distribution, with the Markov property.

**Notation on this page:** $\gamma$: discount factor (0 to 1) • $k$: steps into the future • $\rho_0$: distribution of the start state • $R(s,a)$: reward function • $\mathcal{S}$: the set of states • $\mathcal{A}$: the set of actions • $P(s' \mid s,a)$: transition probability

**Q2. What is the Markov property?** The next state and reward depend only on the current state and action, not on earlier history.
**Q3. What is the return and how is it computed?** The discounted sum of future rewards, $G_t = \sum_k \gamma^k r_{t+k}$, computed backwards with $G_t = r_t + \gamma G_{t+1}$.
**Q4. How does LLM text generation map onto an MDP?** State = prompt + tokens so far; action = next token; deterministic transition (append); reward usually only at the end. Or as a contextual bandit with the whole response as the action.
**Q5. Why is $\gamma = 1$ usual in LLM RL?** Episodes are finite and the reward is at the end; discounting would just penalise longer correct answers.
**Q6. What is the credit assignment problem?** Working out which earlier actions caused a later reward.
**Q7. Why can RL not discover a reasoning strategy the base model never samples?** Policy-gradient methods only reinforce sampled behaviour; actions with near-zero probability are never sampled and get no gradient.
**Q8. Why are LLM transitions deterministic, and what follows?** The next state is the prefix with the chosen token appended, so all randomness is in the policy; hence $Q(s_t, a_t) = V(s_{t+1})$ and no dynamics model is needed.
**Q9. Show that a per-token KL reward plus a final score gives the KL-regularised objective.** Summing $r_t = -\beta \log \frac{\pi_\theta(a_t|s_t)}{\pi_{\text{ref}}(a_t|s_t)}$ over tokens gives $-\beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}$; adding $R$ and taking the expectation gives $\mathbb{E}[R] - \beta D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})$.
**Q10. What is a POMDP and where does it show up in LLM agents?** An MDP where the agent sees observations of a hidden state; in agents, hidden databases, user goals and unseen tool results.
**Q11. What is the effective horizon of a discount factor $\gamma$?** $1/(1-\gamma)$ steps; $\gamma = 0.99$ looks about 100 steps ahead.

**Key takeaways**

- RL: an agent acts, gets rewards and learns to maximise expected return $J(\theta)$.
- MDP = states, actions, transitions, rewards, discount, start distribution, with the Markov property.
- Return $G_t = r_t + \gamma G_{t+1}$; LLM RL usually uses $\gamma = 1$.
- An LLM is a stochastic policy: token-level MDP or sequence-level contextual bandit.
- Exploration in LLMs comes from sampling; RL can only reinforce what the model already samples.
- An LLM episode is an MDP with deterministic transitions and history as state; per-token KL plus a final score is the KL-regularised objective.
- Agents face POMDPs: the context is a belief state; control hidden randomness.
- Use $\gamma = 1$ per token; the effective horizon of $\gamma$ is $1/(1-\gamma)$.

**Notation on this page:** $\gamma$: discount factor (0 to 1) • $G_t$: return from step $t$ (discounted sum of future rewards) • $r_t$: reward at step $t$ • $t$: time step or token position • $G$: group size (samples per prompt) • $k$: steps into the future • $V(s)$: state value (expected future return from $s$) • $Q(s,a)$: action value (expected return after taking $a$ in $s$) • $s$: a state • $a$: an action • $V$: vocabulary size • *and 10 more: see the Notation table*

CHAPTER 14

# Value Functions and Bellman Equations

**Why this chapter matters.**
Value functions answer "how good is this situation?" They are the critic in PPO, the baseline in policy gradients, and the source of the advantage that tells each action how much better it was than average. The Bellman equations behind them are a favourite interview derivation, and temporal-difference learning is the idea behind GAE (Chapter 16).

## 14.1 Three value functions

For a policy $\pi$:

**State value.** The expected return from state $s$ if we follow $\pi$:

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s].$$

**Action value.** The expected return if we take action $a$ in state $s$ and then follow $\pi$:

$$Q^\pi(s,a) = \mathbb{E}_\pi[G_t \mid s_t = s, a_t = a].$$

**Advantage.** How much better action $a$ is than the policy's average action in $s$:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s).$$

The two are linked: $V^\pi(s) = \sum_a \pi(a \mid s)\, Q^\pi(s,a)$, the average of $Q$ over the policy's actions. So the advantage averages to zero under the policy: $\sum_a \pi(a \mid s) A^\pi(s,a) = 0$. A positive advantage means "better than usual, do it more"; negative means "worse than usual, do it less". This is exactly the signal a policy gradient uses.

*LLM example.* Prompt: a maths question. The model has a 30% chance of solving it, so $V(x) = 0.3$ with a 0/1 reward. A sampled response that is correct has return 1 and advantage $1 - 0.3 = 0.7$. An incorrect one has advantage $-0.3$. GRPO estimates $V(x)$ by averaging the rewards of several responses to the same prompt (Chapter 27).

## 14.2 The Bellman expectation equations

Start from $G_t = r_t + \gamma G_{t+1}$ and take expectations under $\pi$:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[r_t + \gamma G_{t+1} \mid s_t = s] \\ &= \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s,a)\big[R(s,a) + \gamma\, \mathbb{E}_\pi[G_{t+1} \mid s_{t+1} = s']\big] \\ &= \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s,a)\big[R(s,a) + \gamma V^\pi(s')\big]. \end{aligned}$$

**Notation on this page:** $s$: a state • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $V(s)$: state value (expected future return from $s$) • $G_t$: return from step $t$ (discounted sum of future rewards) • $t$: time step or token position • $G$: group size (samples per prompt) • $V$: vocabulary size • $a$: an action • $Q(s,a)$: action value (expected return after taking $a$ in $s$) • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • *and 5 more: see the Notation table*

The step from the second to third line uses the Markov property: once we know $s_{t+1} = s'$, the expected future return is $V^\pi(s')$, whatever happened before. Similarly,

$$Q^\pi(s,a) = \sum_{s'} P(s' \mid s,a)\Big[R(s,a) + \gamma \sum_{a'} \pi(a' \mid s')\, Q^\pi(s',a')\Big].$$

In words: **the value of a state is the immediate reward plus the discounted value of where you land next.** This self-consistency is what lets us learn values from single steps.

## 14.3 The Bellman optimality equations

The **optimal** value functions are the best achievable by any policy: $V^*(s) = \max_\pi V^\pi(s)$ and $Q^*(s,a) = \max_\pi Q^\pi(s,a)$. They satisfy

$$V^*(s) = \max_a \sum_{s'} P(s' \mid s,a)[R(s,a) + \gamma V^*(s')],$$

$$Q^*(s,a) = \sum_{s'} P(s' \mid s,a)[R(s,a) + \gamma \max_{a'} Q^*(s',a')].$$

Once you have $Q^*$, an optimal policy is simply greedy: $\pi^*(s) = \arg\max_a Q^*(s,a)$.

## 14.4 Dynamic programming

If we know $P$ and $R$, we can solve these equations directly.

**Policy evaluation.** Start with any $V$ and repeatedly apply the expectation equation as an update. It converges to $V^\pi$ because the update is a **contraction** for $\gamma < 1$: each sweep shrinks the largest error by at least a factor $\gamma$.

**Policy iteration.** Alternate: evaluate the current policy, then make it greedy with respect to its own values. Each round gives a policy at least as good, and it converges to the optimum in a finite number of rounds.

**Value iteration.** Apply the optimality equation as an update: $V(s) \leftarrow \max_a \sum_{s'} P(s' \mid s,a)[R(s,a) + \gamma V(s')]$. It converges to $V^*$.

```
import numpy as np

# A 1-D corridor of 5 cells; reward +1 for stepping into the right end (terminal).
n, gamma = 5, 0.9
actions = (-1, +1)                    # left, right

def step(s, a):
    s2 = min(max(s + a, 0), n - 1)
    return s2, (1.0 if s2 == n - 1 else 0.0)

V = np.zeros(n)
for sweep in range(100):
    V_new = V.copy()
    for s in range(n - 1):            # cell n-1 is terminal, value 0
```

**Notation on this page:** $s$: a state • $t$: time step or token position • $\pi_\theta$: the policy (the model), with weights $\theta$ • $V(s)$: state value (expected future return from $s$) • $V$: vocabulary size • $\gamma$: discount factor (0 to 1) • $Q(s,a)$: action value (expected return after taking $a$ in $s$) • $a$: an action • $R(s,a)$: reward function • $P(s' \mid s,a)$: transition probability • $\pi^*$: the optimal policy

```
        V_new[s] = max(r + gamma * V[s2] for s2, r in (step(s, a) for a in actions))
    if np.abs(V_new - V).max() < 1e-10:
        break
    V = V_new
print("sweeps:", sweep, "V* =", np.round(V, 3))   # 1, 0.9, 0.81, 0.729 from right to
 ↪ left
```

Real problems rarely give us $P$ and $R$, and LLM state spaces are astronomically large. So we must learn from samples.

## 14.5 Monte Carlo vs temporal-difference learning

Suppose we want to learn $V^\pi$ from experience.

**Monte Carlo (MC).** Run full episodes, compute the actual return $G_t$ for each visited state, and move $V(s_t)$ towards it:

$$V(s_t) \leftarrow V(s_t) + \eta\,(G_t - V(s_t)).$$

Here $\eta$ is a step size. MC targets are **unbiased** (their average is the true value) but have **high variance**, because $G_t$ adds up the randomness of every future step.

**Temporal difference (TD(0)).** After one step, use the Bellman equation with the current estimate as the target:

$$V(s_t) \leftarrow V(s_t) + \eta\,\delta_t, \qquad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t).$$

The quantity $\delta_t$ is the **TD error**. Using an estimate inside the target is called **bootstrapping**. TD targets have **low variance** (only one step of randomness) but are **biased** while $V$ is wrong. TD can learn before an episode ends, and from incomplete episodes.

**TD($\lambda$) and $n$-step returns** sit in between. The $n$-step target uses $n$ real rewards then bootstraps: $r_t+\gamma r_{t+1}+\cdots+\gamma^{n-1}r_{t+n-1}+\gamma^n V(s_{t+n})$. TD($\lambda$) averages all $n$-step targets with weights $(1-\lambda)\lambda^{n-1}$. $\lambda = 0$ gives TD(0) and $\lambda = 1$ gives Monte Carlo. GAE (Chapter 16) applies the same averaging to advantages.

**Why this, not that? Why bootstrap (TD), not wait for the full return (Monte Carlo)?**

**Why.** One-step targets have far lower variance, so values are learned with fewer samples; learning can happen mid-episode; and it works for never-ending tasks.
**Trade-off.** The target uses the current, possibly wrong, estimate, so errors feed back into themselves (bias). With function approximation and off-policy data, bootstrapping can even diverge.
**When the alternative wins.** Monte Carlo wins when episodes are short, rewards come only at the end and you can sample several episodes per start state. That is the LLM case: critic-free methods such as GRPO and RLOO use Monte Carlo returns with a sampled baseline instead of a bootstrapped critic.

**Notation on this page:** $P(s' \mid s,a)$: transition probability • $R(s,a)$: reward function • $V(s)$: state value (expected future return from $s$) • $V$: vocabulary size • $G_t$: return from step $t$ (discounted sum of future rewards) • $t$: time step or token position • $G$: group size (samples per prompt) • $s$: a state • $\eta$: step size of the update • $\gamma$: discount factor (0 to 1) • $\delta_t$: TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$ • *and 3 more: see the Notation table*

## 14.6 Q-learning and DQN

**Q-learning** learns $Q^*$ directly with a TD update on the optimality equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)].$$

It is **off-policy**: the data can come from any behaviour policy (for example $\epsilon$-greedy), because the target uses the greedy action $\max_{a'}$, not the action the agent actually took next.

**DQN** (*Playing Atari with Deep Reinforcement Learning*, Mnih et al., arXiv 1312.5602) used a neural network for $Q$ and added two stabilisers: an **experience replay** buffer (train on random old transitions, which breaks correlations) and a **target network** (a slowly updated copy used in the TD target).

Value-based methods like DQN need $\max_a Q(s, a)$ over all actions. With a vocabulary of 150,000 tokens and sequences of thousands of tokens, learning accurate $Q$ values for every token is impractical, and the greedy policy they produce is not a good generator of diverse text. So LLM post-training uses **policy-gradient methods** (next chapter), sometimes with a learned $V$ as a helper.

**Why this, not that? Why policy gradients for LLMs, not value-based methods like Q-learning?**

**Why.** Policy gradients directly improve the token distribution we already have (the pre-trained model), work with huge discrete action spaces, keep the policy stochastic, and only need log-probabilities of sampled tokens.
**Trade-off.** They are on-policy and sample-inefficient: data must be regenerated after each update, and gradient estimates are noisy.
**When the alternative wins.** Value-based methods win in small discrete action spaces where off-policy reuse of data matters (games with replay buffers), and value-like ideas still appear in LLM work: the critic in PPO, token-level value models, and process reward models that estimate how likely a partial solution is to succeed.

## 14.7 A worked example: three ways to get the same values

Take a chain of five states $s_0, \ldots, s_4$. The policy moves right with probability 0.7 and left with probability 0.3 (from $s_0$ a left move stays in place). Entering $s_4$ ends the episode with reward +1; every other step gives 0. Use $\gamma = 0.9$.

**Exact values by linear algebra.** For a fixed policy the Bellman expectation equation is linear: $V = r_\pi + \gamma P_\pi V$, where $P_\pi$ is the state-to-state transition matrix under the policy and $r_\pi$ the expected immediate reward. So

$$V = (I - \gamma P_\pi)^{-1} r_\pi.$$

This is **policy evaluation** in closed form. It costs a matrix inverse, which is fine for 5 states and impossible for the astronomically many states of a language model.

**Monte Carlo (MC).** Run episodes and average the observed discounted returns from each visited state. Unbiased, but each return depends on many random moves, so the estimate is noisy.

**Notation on this page:** $Q(s, a)$: action value (expected return after taking $a$ in $s$) • $\eta$: step size of the update • $\gamma$: discount factor (0 to 1) • $r_t$: reward at step $t$ • $s$: a state • $a$: an action • $t$: time step or token position • $\epsilon$: clip range for the probability ratio • $V$: vocabulary size • $P(s' \mid s, a)$: transition probability

**Temporal difference, TD(0).** After each step update $V(s) \leftarrow V(s) + \eta\,[r + \gamma V(s') - V(s)]$. The target uses the current estimate of the next state (bootstrapping): lower variance, but biased while $V(s')$ is still wrong.

```
import torch, random
random.seed(0)
nS, gamma, p_right = 5, 0.9, 0.7
P = torch.zeros(nS, nS); r = torch.zeros(nS)
for s in range(nS - 1):                                  # s4 is terminal (absorbing,
 ↪  value 0)
    P[s, s + 1] += p_right; P[s, max(s - 1, 0)] += 1 - p_right
    r[s] = p_right * (1.0 if s + 1 == nS - 1 else 0.0)  # expected reward of the step
 ↪  from s
V_exact = torch.linalg.solve(torch.eye(nS) - gamma * P, r)

def episode(start=0):
    s, traj = start, []
    while s != nS - 1:
        s2 = s + 1 if random.random() < p_right else max(s - 1, 0)
        traj.append((s, 1.0 if s2 == nS - 1 else 0.0, s2)); s = s2
    return traj

def mc(n_eps):
    tot, cnt = torch.zeros(nS), torch.zeros(nS)
    for _ in range(n_eps):
        G, seen = 0.0, []
        for s, rew, _ in reversed(episode()):
            G = rew + gamma * G; seen.append((s, G))
        for s, G in seen: tot[s] += G; cnt[s] += 1          # every-visit MC
    return tot / cnt.clamp_min(1)

def td0(n_eps, lr=0.1):
    V = torch.zeros(nS)
    for _ in range(n_eps):
        for s, rew, s2 in episode():
            V[s] += lr * (rew + gamma * V[s2] - V[s])
    return V

print("exact V:", [round(v, 3) for v in V_exact[:4].tolist()])
for n in (10, 100, 1000):
    e_mc = sum((mc(n) - V_exact)[:4].abs().mean().item() for _ in range(20)) / 20
    e_td = sum((td0(n) - V_exact)[:4].abs().mean().item() for _ in range(20)) / 20
    print(f"{n:5d} episodes: mean abs error  MC {e_mc:.3f}   TD(0) {e_td:.3f}")
```

In this small chain Monte Carlo wins at every budget. The episodes are short, so a return is not very noisy, while TD(0) starts from zero and must pass the reward backward one state per visit (after 10 episodes its error is still 0.45). With a constant step size TD(0) also keeps a residual error (about 0.03) from its own noisy updates, which a decaying step size would remove. TD's advantage shows up when episodes are long and returns are noisy, because then the variance of a full return dominates; that is the bias–variance trade-off that $\lambda$ in GAE tunes (Chapter 16). Do

**Notation on this page:** $\eta$: step size of the update • $\gamma$: discount factor (0 to 1) • $V(s)$: state value (expected future return from $s$) • $r_t$: reward at step $t$ • $s$: a state • $V$: vocabulary size • $\lambda$: TD($\lambda$) mixing weight (0 = TD(0), 1 = Monte Carlo)

not memorise "TD is better": say *when* each is better.

## 14.8 $n$-step returns and TD($\lambda$)

Between one-step TD and Monte Carlo there is a family of targets. The **$n$-step return** uses $n$ real rewards and then bootstraps:

$$G_t^{(n)} = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}).$$

$n = 1$ is TD(0); $n = \infty$ (to the end of the episode) is Monte Carlo. The **$\lambda$-return** averages all of them with geometric weights:

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}.$$

The weights $(1-\lambda)\lambda^{n-1}$ sum to one. With $\lambda = 0$ only the one-step return counts; with $\lambda \to 1$ the weight moves to long returns. GAE is exactly this average applied to advantages: $\hat{A}_t^{\text{GAE}} = G_t^\lambda - V(s_t)$ (Chapter 16). So when an interviewer asks "what is $\lambda$ in GAE?", the precise answer is: the geometric weight in an average of $n$-step advantage estimates.

**Worked example.** With $\lambda = 0.9$, the one-step return gets weight 0.1, the two-step 0.09, the ten-step $0.1 \times 0.9^9 \approx 0.039$. The weights halve about every $\ln 2/(-\ln 0.9) \approx 6.6$ steps, so the effective horizon is about $1/(1-\lambda) = 10$ steps. For a 2,000-token response with a reward only at the end and $\gamma = 1$, a critic-based method with $\lambda = 0.95$ (horizon about 20 tokens) relies almost entirely on the critic for early tokens. This is why VAPO makes $\lambda$ grow with response length (Chapter 16).

## 14.9 What a value means for a language model

For an LLM with a single terminal reward $R \in \{0, 1\}$ (a verifier) and $\gamma = 1$, the state value of a prefix has a concrete meaning:

$$V^\pi(x, y_{<t}) = \mathbb{E}_{y_{\geq t} \sim \pi}[R(x, y)] = P_\pi(\text{the final answer is correct} \mid x, y_{<t}).$$

It is the **probability that the current policy will finish this partial solution correctly**. Three consequences:

1. **Values can be estimated by sampling.** From a prefix, sample $k$ continuations and count successes: $\hat{V} = c/k$. This is unbiased and needs no critic network. *VinePPO* (arXiv 2410.01679) uses exactly this to compute advantages for PPO on maths, and *Math-Shepherd* (arXiv 2312.08935) uses the same idea to label reasoning steps for a process reward model without human annotation (Chapter 29).
2. **A step's advantage is a change in success probability.** If $\hat{V}$ is 0.40 before a reasoning step and 0.65 after it, that step's advantage is about +0.25: it made success more likely. A step after which $\hat{V}$ drops to 0.05 is probably an error.
3. **The critic has to learn a hard function.** Predicting "will this half-written proof end up correct?" from hidden states is as hard as verifying reasoning. That is the deeper reason critics for long reasoning are hard to train, and why critic-free methods with group baselines became popular (Chapter 27).

**Notation on this page:** $n$: number of real rewards before bootstrapping • $\gamma$: discount factor (0 to 1) • $V(s)$: state value (expected future return from $s$) • $G_t$: return from step $t$ (discounted sum of future rewards) • $r_t$: reward at step $t$ • $s$: a state • $t$: time step or token position • $G$: group size (samples per prompt) • $V$: vocabulary size • $\lambda$: TD($\lambda$) mixing weight (0 = TD(0), 1 = Monte Carlo) • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • *and 8 more: see the Notation table*

**Worked example.** A prompt has $\hat{V}(x) = 0.5$ (half of the samples are correct). After the first line "Let $n$ be the number of apples", 8 continuations give 5 successes: $\hat{V} = 0.625$, advantage +0.125. After a second line that contains an arithmetic slip, 8 continuations give 1 success: $\hat{V} = 0.125$, advantage −0.5. A single outcome reward would give both lines the same credit; Monte Carlo values separate them, at the cost of 16 extra generations for two steps.

**Why this, not that? Why estimate step values by sampling continuations (VinePPO-style), not with a learned critic?**

**Reason.** Sampled values are unbiased and need no extra network; a critic for long reasoning is slow to train, biased early in training and can mislead every advantage. **Trade-off.** Every value estimate costs $k$ extra generations from each chosen prefix; with long responses and many steps this multiplies rollout cost several times, and the estimate is noisy for small $k$. **When the alternative is better.** When generation is the bottleneck (it usually is), when the episode is long and has many decision points (agents), or when intermediate rewards exist: a learned critic amortises the estimate across states for one forward pass each.

## 14.10 Function approximation and the "deadly triad"

With a neural network $V_\phi$ instead of a table, TD learning minimises

$$\mathcal{L}(\phi) = \mathbb{E}\Big[\big(r + \gamma V_{\bar{\phi}}(s') - V_\phi(s)\big)^2\Big],$$

where the target uses a copy $\bar{\phi}$ that is not differentiated (the **semi-gradient**). Sutton and Barto call the combination of **function approximation**, **bootstrapping** and **off-policy data** the *deadly triad*: together they can make values diverge, even when each alone is safe. DQN's target network and replay buffer were engineering answers to it (Chapter 14's Q-learning section). For LLM post-training the lesson is practical: critics are trained on-policy, with Monte Carlo or high-$\lambda$ targets (VC-PPO uses $\lambda_{\text{critic}} = 1$), and are clipped like the policy (PPO's value clipping) to avoid the triad.

## 14.11 Common mistakes

- **Confusing $V$ and $Q$ in an LLM.** $V(s_t)$ is the value of a prefix before choosing the next token; $Q(s_t, a_t)$ is the value after choosing it, which equals $V(s_{t+1})$ when transitions are deterministic (appending the token). So for LLMs $A_t = V(s_{t+1}) - V(s_t)$ when there is no intermediate reward and $\gamma = 1$.
- **Bootstrapping across a truncation as if it were terminal.** If an episode stops because of a length limit, the next state's value is not zero (Chapter 32).
- **Reading a low value loss as a good critic.** A critic that predicts the batch mean everywhere has a low loss when rewards are rare; check explained variance, not just loss.

**Interview check**

**Q1. Define $V^\pi$, $Q^\pi$ and $A^\pi$.** Expected return from $s$; from $s$ after taking $a$; and $Q^\pi - V^\pi$.
**Q2. Derive the Bellman expectation equation for $V^\pi$.** $G_t = r_t + \gamma G_{t+1}$; take expectations;

**Notation on this page:** $x$: the prompt (input) • $V$: vocabulary size • $n$: number of real rewards before bootstrapping • $k$: steps into the future • $\phi$: weights of a second network (reward model or critic) • $V(s)$: state value (expected future return from $s$) • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\gamma$: discount factor (0 to 1) • $\mathcal{L}$: a loss to minimise • $r_t$: reward at step $t$ • $s$: a state • *and 7 more: see the Notation table*

by the Markov property $\mathbb{E}[G_{t+1} \mid s_{t+1}] = V^\pi(s_{t+1})$.
**Q3. What is the expected advantage under the policy?** Zero: $\sum_a \pi(a|s)A^\pi(s,a) = V^\pi(s) - V^\pi(s) = 0$.
**Q4. MC vs TD?** MC: unbiased, high variance, needs full episodes. TD: biased (bootstraps), low variance, learns online.
**Q5. What is the TD error?** $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$.
**Q6. Why is Q-learning off-policy?** Its target uses $\max_{a'} Q(s', a')$, independent of the behaviour policy's next action.
**Q7. What two tricks made DQN stable?** Experience replay and a target network.
**Q8. Why does value iteration converge?** The Bellman optimality operator is a $\gamma$-contraction in the max norm, so repeated application converges to its unique fixed point $V^*$.
**Q9. What does $V(s_t)$ mean for an LLM with a binary verifier reward?** The probability that the current policy, continuing from this prefix, ends with a correct answer.
**Q10. How can you estimate a prefix's value without a critic?** Sample several continuations from the prefix and take the fraction that succeed (VinePPO, Math-Shepherd). Unbiased, but it costs extra generation.
**Q11. What is the $\lambda$-return?** A geometrically weighted average of $n$-step returns with weights $(1-\lambda)\lambda^{n-1}$; GAE is the $\lambda$-return minus $V(s_t)$.
**Q12. What is the deadly triad?** Function approximation, bootstrapping and off-policy learning together, which can make value estimates diverge.
**Q13. Why is $Q(s_t, a_t) = V(s_{t+1})$ for a language model without intermediate rewards?** Transitions are deterministic (the token is appended), so taking action $a_t$ in $s_t$ leads to exactly one next state, and $\gamma = 1$, $r_t = 0$ before the end.

**Key takeaways**

- $V^\pi(s)$, $Q^\pi(s,a)$ and $A^\pi = Q^\pi - V^\pi$; advantages average to zero under $\pi$.
- Bellman: value = immediate reward + discounted value of the next state.
- With a known model: policy evaluation, policy iteration, value iteration.
- MC: unbiased, high variance. TD: bootstraps, biased, low variance. TD($\lambda$) interpolates.
- Q-learning/DQN are off-policy value methods; LLMs use policy gradients instead.
- Policy evaluation is linear: $V = (I - \gamma P_\pi)^{-1} r_\pi$; MC is unbiased and noisy, TD is biased and steadier; the $\lambda$-return interpolates.
- For an LLM with a 0/1 reward, $V$ of a prefix is the probability of finishing correctly; it can be estimated by sampling continuations.

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $V(s)$: state value (expected future return from $s$) • $s$: a state • $t$: time step or token position • $G$: group size (samples per prompt) • $V$: vocabulary size • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $a$: an action • $\gamma$: discount factor (0 to 1) • $\delta_t$: TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$ • *and 5 more: see the Notation table*

CHAPTER 15

# Policy Gradient Methods

**Why this chapter matters.**
The policy gradient theorem is the single most important equation in LLM RL. PPO, GRPO, RLOO, REINFORCE++, DAPO, CISPO and even on-policy distillation are all ways of estimating it with less noise or more stability. You should be able to derive it on a whiteboard, explain why baselines do not add bias, and write REINFORCE from memory.

## 15.1 The goal

We want to maximise the expected return

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)], \qquad R(\tau) = \sum_{t=0}^{T-1} \gamma^t r_t,$$

by gradient ascent: $\theta \leftarrow \theta + \eta\nabla_\theta J(\theta)$. The difficulty: $\theta$ affects the *distribution* of trajectories, and the environment (and the reward function) may not be differentiable. We cannot backpropagate through "the environment gave reward 1". The **score-function** trick of Chapter 2 solves this.

## 15.2 Deriving the policy gradient

**Step 1: probability of a trajectory.** Under the policy and the environment,

$$p_\theta(\tau) = \rho_0(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t \mid s_t)\, P(s_{t+1} \mid s_t, a_t).$$

**Step 2: the log-derivative trick.** For any probability $p_\theta$, $\nabla_\theta p_\theta = p_\theta \nabla_\theta \log p_\theta$. So

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_\tau p_\theta(\tau) R(\tau) = \sum_\tau p_\theta(\tau)\, \nabla_\theta \log p_\theta(\tau)\, R(\tau) = \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log p_\theta(\tau)\, R(\tau)].$$

(The sum becomes an integral for continuous trajectories; nothing else changes.)

**Step 3: the environment drops out.** Take the log of Step 1:

$$\log p_\theta(\tau) = \log \rho_0(s_0) + \sum_t \log \pi_\theta(a_t \mid s_t) + \sum_t \log P(s_{t+1} \mid s_t, a_t).$$

Only the middle term depends on $\theta$, so

$$\nabla_\theta \log p_\theta(\tau) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t).$$

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\gamma$: discount factor (0 to 1) • $\tau$: a trajectory (states, actions, rewards of one episode) • $J(\theta)$: objective to maximise (expected reward) • $r_t$: reward at step $t$ • $t$: time step or token position • $T$: episode length • $R(\tau)$: total (discounted) reward of trajectory $\tau$ • $\nabla_\theta$: gradient with respect to $\theta$ • *and 6 more: see the Notation table*

We never need the environment's transition probabilities or the reward's gradient. We only need the log-probabilities of the actions we took. For an LLM, those are the log-probabilities of the sampled tokens, which any forward pass gives us.

**Result (REINFORCE form).**

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\Big[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\; R(\tau)\Big].$$

*Intuition.* $\nabla_\theta \log \pi_\theta(a_t \mid s_t)$ is the direction in weight space that makes action $a_t$ more likely. We push in that direction, scaled by how good the whole trajectory was. Good trajectories have all their actions made more likely; bad ones (negative return) less likely.

## 15.3 Reward-to-go: using causality

An action at time $t$ cannot affect rewards that came before it. Rewards before $t$ only add noise to that action's gradient. Formally, for $t' < t$, $\mathbb{E}[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\, r_{t'}] = 0$ (by the same argument as the baseline proof below, since $r_{t'}$ is fixed once we condition on $s_t$). Dropping those terms gives the **reward-to-go** form:

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau\Big[\sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\; \hat{G}_t\Big], \qquad \hat{G}_t = \sum_{t'\geq t} \gamma^{t'-t} r_{t'}.$$

(Strictly, the discounted objective gives a factor $\gamma^t$ in front of each term; in practice it is dropped, and with $\gamma = 1$ it does not exist.)

For LLMs with a single terminal reward and $\gamma = 1$, the reward-to-go of every token equals the final reward, so reward-to-go changes nothing. It matters for multi-step tasks with intermediate rewards.

## 15.4 Baselines

**Claim.** For any function $b(s_t)$ that does not depend on the action $a_t$,

$$\mathbb{E}_{a_t\sim\pi_\theta(\cdot\mid s_t)}[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\; b(s_t)] = 0.$$

**Proof.**

$$\sum_a \pi_\theta(a \mid s)\, \nabla_\theta \log \pi_\theta(a \mid s)\, b(s) = b(s) \sum_a \nabla_\theta \pi_\theta(a \mid s) = b(s)\, \nabla_\theta \sum_a \pi_\theta(a \mid s) = b(s)\, \nabla_\theta 1 = 0.$$

So we may subtract a **baseline** from the return without changing the expected gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau\Big[\sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\; (\hat{G}_t - b(s_t))\Big].$$

**Why bother?** The *variance* changes a lot. If every return is positive (CartPole gives +1 per step), plain REINFORCE makes *every* sampled action more likely and relies on small differences between large numbers to learn which ones are better. Subtracting a baseline near the

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\nabla_\theta$: gradient with respect to $\theta$ • $\tau$: a trajectory (states, actions, rewards of one episode) • $J(\theta)$: objective to maximise (expected reward) • $s$: a state • $a$: an action • $t$: time step or token position • $T$: episode length • $R(\tau)$: total (discounted) reward of trajectory $\tau$ • *and 4 more: see the Notation table*

average return makes good actions positive and bad ones negative. The best common choice is $b(s_t) \approx V^\pi(s_t)$, which turns $\hat{G}_t - b(s_t)$ into an estimate of the **advantage** $A^\pi(s_t, a_t)$.

The baseline must not depend on the action being scored. In GRPO and RLOO, the baseline for response $i$ is computed from the *other* responses to the same prompt (RLOO), or from all responses including $i$ (GRPO). Including $i$'s own reward introduces a small bias, which is why RLOO uses "leave one out" (Chapter 28).

*Worked example.* Two actions with $\pi(a_1) = \pi(a_2) = 0.5$, rewards 10 and 11. Without a baseline, the gradient estimate from a single sample is $\nabla \log \pi(a) \times 10$ or $\times 11$: large and nearly cancelling. With baseline $b = 10.5$, the weights are $-0.5$ and $+0.5$: the same expected gradient, much less noise.

**Why this, not that? Why subtract a baseline, not use the raw return?**

**Why.** It leaves the expected gradient unchanged (proved above) and can cut its variance by orders of magnitude, especially when all rewards share a sign or a large offset.
**Trade-off.** A good baseline must be estimated: a learned value function costs a second network (the critic), and a sample-based baseline (group mean) costs extra samples per prompt. A baseline that depends on the current action, even slightly, adds bias.
**When the alternative wins.** Almost never omitted in practice. With rewards that are already centred (for example $\pm 1$ with a 50% success rate), the gain is small, and the simplest code may skip it.

## 15.5 Advantage normalisation

Many implementations also normalise advantages within a batch: subtract the mean and divide by the standard deviation. Subtracting the batch mean is a (slightly biased) baseline. Dividing by the standard deviation rescales the gradient, which acts like an adaptive learning rate. GRPO does this per prompt group; Dr. GRPO argues the division by the standard deviation adds a difficulty bias (Chapter 28).

## 15.6 REINFORCE

**REINFORCE** (Williams, 1992) is the Monte Carlo policy gradient algorithm:

1. Run the current policy to collect a batch of episodes.
2. Compute the reward-to-go $\hat{G}_t$ for each step, and subtract a baseline.
3. Form the **surrogate loss** $\mathcal{L}(\theta) = -\frac{1}{N} \sum_{\text{episodes}} \sum_t \log \pi_\theta(a_t \mid s_t) \hat{A}_t$ with the advantages treated as constants, and take a gradient step.
4. Throw the data away and repeat (it is now off-policy).

The surrogate loss is not "the objective"; its value means nothing on its own. It is a function whose gradient equals (minus) the policy gradient estimate. This is a common interview trap.

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $V(s)$: state value (expected future return from $s$) • $s$: a state • $t$: time step or token position • $V$: vocabulary size • $b(s_t)$: baseline (must not depend on the action) • $G$: group size (samples per prompt) • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $a$: an action • $\nabla_\theta$: gradient with respect to $\theta$ • $\theta$: the model's trainable weights • *and 2 more: see the Notation table*

**Why this, not that? Why throw away data after one step (on-policy), not reuse it?**

**Why.** The policy gradient formula is an expectation under the *current* policy. After an update, old samples come from a different distribution, and reusing them as if they were fresh gives a biased gradient that can push the policy in the wrong direction.
**Trade-off.** Sample inefficiency: for LLMs, generating rollouts is usually the most expensive part of training, and using each rollout once wastes compute.
**When the alternative wins.** With importance-sampling corrections and a trust region (PPO, Chapter 17), you can safely take several gradient steps on the same batch. Fully off-policy methods (Q-learning, Chapter 14; asynchronous RL, Chapter 28) reuse data much more.

## 15.7 Code: REINFORCE on CartPole

**CartPole** is the classic first RL task: push a cart left or right to keep a pole balanced. The state is 4 numbers (cart position and velocity, pole angle and angular velocity), there are 2 actions, and the reward is +1 per step until the pole falls or 500 steps pass. Gymnasium's `CartPole-v1` is the standard version; to keep this book's code free of extra installs, here is a version with the same physics.

```
import math, random, torch, torch.nn as nn

class CartPole:
    """Same dynamics as Gymnasium's CartPole-v1 (Euler integration)."""
    def __init__(self, seed=0, max_steps=500):
        self.rng = random.Random(seed); self.max_steps = max_steps
    def reset(self):
        self.s = [self.rng.uniform(-0.05, 0.05) for _ in range(4)]; self.t = 0
        return torch.tensor(self.s)
    def step(self, action):
        x, x_dot, th, th_dot = self.s
        force = 10.0 if action == 1 else -10.0
        g, mc, mp, l, dt = 9.8, 1.0, 0.1, 0.5, 0.02
        cos, sin = math.cos(th), math.sin(th)
        temp = (force + mp * l * th_dot**2 * sin) / (mc + mp)
        th_acc = (g * sin - cos * temp) / (l * (4/3 - mp * cos**2 / (mc + mp)))
        x_acc = temp - mp * l * th_acc * cos / (mc + mp)
        x, x_dot = x + dt * x_dot, x_dot + dt * x_acc
        th, th_dot = th + dt * th_dot, th_dot + dt * th_acc
        self.s = [x, x_dot, th, th_dot]; self.t += 1
        fell = abs(x) > 2.4 or abs(th) > 12 * 2 * math.pi / 360
        done = fell or self.t >= self.max_steps
        return torch.tensor(self.s), 1.0, done
```

The REINFORCE loop. Each update uses 8 episodes; the advantage is the reward-to-go normalised over the batch (the batch mean acts as the baseline).

```
torch.manual_seed(0)
env = CartPole(seed=0)
policy = nn.Sequential(nn.Linear(4, 32), nn.Tanh(), nn.Linear(32, 2))
opt = torch.optim.Adam(policy.parameters(), lr=1e-2)

def run_episode():
    s, logps, rewards, done = env.reset(), [], [], False
    while not done:
        dist = torch.distributions.Categorical(logits=policy(s))
        a = dist.sample()
        logps.append(dist.log_prob(a))
        s, r, done = env.step(a.item())
        rewards.append(r)
    return torch.stack(logps), rewards

for it in range(60):
    batch_logp, batch_G, lens = [], [], []
    for _ in range(8):
        logps, rewards = run_episode()
        G = torch.tensor(rewards).flip(0).cumsum(0).flip(0)    # reward-to-go, gamma = 1
        batch_logp.append(logps); batch_G.append(G); lens.append(len(rewards))
    G = torch.cat(batch_G)
    adv = (G - G.mean()) / (G.std() + 1e-8)                    # baseline + scaling
    loss = -(torch.cat(batch_logp) * adv).sum() / len(lens)    # surrogate loss
    opt.zero_grad(); loss.backward(); opt.step()
    if it % 10 == 0 or it == 59:
        print(f"update {it:2d}: mean episode length {sum(lens) / len(lens):.0f}")
```

On a laptop CPU this takes about 20 seconds; the mean episode length climbs from about 20 to over 400 (the maximum is 500). Try removing the normalisation line (use `adv = G`) and see how much slower and noisier learning becomes.

## 15.8 Variance and its sources

Policy-gradient estimates are noisy for three reasons:

1. **Few samples.** Each update sees a tiny fraction of possible trajectories.
2. **Long horizons.** The return sums many random rewards.
3. **Credit assignment.** Every action in a trajectory gets the same return (minus the baseline), whether or not it mattered.

The rest of Part III and much of Part VI is about reducing this variance: baselines and critics (Chapter 16), trust regions so we can reuse samples (Chapter 17), group baselines (GRPO, RLOO) and careful normalisation (Chapter 28).

## 15.9 The score function of a softmax policy

For a softmax policy $\pi_\theta(a) = e^{\theta_a} / \sum_b e^{\theta_b}$ over logits $\theta$, the log-probability is $\log \pi_\theta(a) = \theta_a - \log \sum_b e^{\theta_b}$, so

$$\frac{\partial \log \pi_\theta(a)}{\partial \theta_b} = \mathbb{1}[a = b] - \pi_\theta(b), \qquad \nabla_\theta \log \pi_\theta(a) = e_a - \pi_\theta.$$

This is the same "one-hot minus probabilities" vector as the cross-entropy gradient (Chapter 9) with the sign flipped. A REINFORCE update with reward $R$ moves the logits by $\eta R(e_a - \pi_\theta)$: up for the sampled action, down for all others in proportion to their probability. For an LLM, this happens at the output layer for every generated token, and back-propagation carries it into the network.

## 15.10 The optimal baseline

Any baseline $b$ keeps the gradient unbiased; which one minimises the variance? For one component $j$ of the gradient, with $g_j = \partial \log \pi / \partial \theta_j$, the estimator is $(R - b)\, g_j$. Its mean does not depend on $b$, so minimising the variance means minimising the second moment:

$$\frac{d}{db}\, \mathbb{E}[(R-b)^2 g_j^2] = -2\, \mathbb{E}[(R-b)\, g_j^2] = 0 \quad \Longrightarrow \quad b_j^* = \frac{\mathbb{E}[g_j^2\, R]}{\mathbb{E}[g_j^2]}.$$

The optimal baseline is a reward average **weighted by the squared score**. If $g_j^2$ were unrelated to $R$, it would reduce to the mean reward, which is why the mean (or a learned $V(s)$) is the practical choice. A numerical check on a 4-action problem with rewards 10 to 13 (a large shared offset, as with many reward models):

```
import torch
torch.manual_seed(0)

theta = torch.tensor([0.0, 0.2, -0.1, 0.1])
rew = torch.tensor([10.0, 11.0, 12.0, 13.0])
p = torch.softmax(theta, -1)
true_grad = p * (rew - (p * rew).sum())                    # exact gradient of E[r] for a
 ↪ softmax policy

def estimates(baseline, n=200000):
    a = torch.multinomial(p, n, replacement=True)
    score = torch.nn.functional.one_hot(a, 4).float() - p  # grad of log softmax at the
 ↪ sampled action
    if callable(baseline):                                 # per-component baseline, shape
     ↪ (4,)
        return (rew[a][:, None] - baseline(score, a)) * score
    return (rew[a] - baseline)[:, None] * score

def optimal_b(score, a):
    """Per-component optimal constant baseline b* = E[g^2 R] / E[g^2] (estimated from
     ↪ the samples)."""
    g2 = score ** 2
```

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $a$: an action • $b(s_t)$: baseline (must not depend on the action) • $\nabla_\theta$: gradient with respect to $\theta$ • $\mathbb{1}[\cdot]$: 1 if the condition is true, else 0 • $R(\tau)$: total (discounted) reward of trajectory $\tau$ • $\eta$: learning rate • $g$: a gradient • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $d$: hidden size of the model • *and 3 more: see the Notation table*

```
    return ((g2 * rew[a][:, None]).mean(0) / g2.mean(0))

for name, b in [("no baseline", 0.0), ("mean-reward baseline", (p * rew).sum().item()),
 ↪  ("optimal baseline", optimal_b)]:
    g = estimates(b)
    print(f"{name:>22}: mean {[round(v, 3) for v in g.mean(0).tolist()]}, total variance
     ↪  {g.var(0).sum():8.3f}")
print(f"{'true gradient':>22}: {[round(v, 3) for v in true_grad.tolist()]}")
```

All three estimators have the same mean (the true gradient), but the variance drops from about 99 with no baseline to 0.6 with the mean reward and 0.2 with the optimal baseline: a **150x** reduction from one subtraction. With an offset of 10 and no baseline, every sampled action looks "good" and the gradient is dominated by noise.

**Why this, not that? Why the mean reward (or a learned V) as baseline, not the variance-optimal baseline?**

**Why.** The optimal baseline needs per-parameter estimates of $\mathbb{E}[g^2 R]$ and $\mathbb{E}[g^2]$, which is impossible for billions of parameters; the mean reward gets most of the benefit and is one number per prompt (or per state).
**Trade-off.** It leaves some variance on the table, especially when the score magnitude correlates with reward (for example, long responses having both large scores and different rewards).
**When the alternative wins.** In small problems, or with per-sample control variates in research code, optimal or learned state-action baselines can cut variance further; group baselines (GRPO, RLOO) are the LLM-friendly middle ground.

## 15.11 Common mistakes

- **Not detaching the advantage.** If the advantage is computed from the same network (a shared critic head), gradients flow into it through the policy loss. Advantages must be constants.
- **Wrong sign.** Optimisers minimise, so the loss is *minus* log-prob times advantage.
- **Summing vs averaging.** Summing log-probs over tokens and averaging over episodes (as above) differ from averaging over tokens; these choices create length biases in LLM RL (Chapter 28).
- **Reading the surrogate loss as progress.** Watch the reward, not the loss.

**Interview check**

**Q1. Derive the policy gradient.** $\nabla J = \mathbb{E}[\nabla \log p_\theta(\tau) R(\tau)]$ by the log-derivative trick; $\log p_\theta(\tau)$ is a sum of policy log-probs plus terms independent of $\theta$, so $\nabla J = \mathbb{E}[\sum_t \nabla \log \pi_\theta(a_t|s_t) R(\tau)]$.
**Q2. Why does the environment's dynamics not appear in the gradient?** Transition probabilities do not depend on $\theta$, so their log-gradients are zero.
**Q3. Prove that a baseline does not bias the gradient.** $\mathbb{E}_a[\nabla \log \pi(a|s) b(s)] =$

**Notation on this page:** $\mathbb{E}$: expected value (average over the distribution in the subscript) • $g$: a gradient • $R(\tau)$: total (discounted) reward of trajectory $\tau$ • $\theta$: the model's trainable weights • $\nabla_\theta$: gradient with respect to $\theta$ • $\tau$: a trajectory (states, actions, rewards of one episode) • $p_\theta(\tau)$: probability of a trajectory under the policy • $\pi_\theta$: the policy (the model), with weights $\theta$ • $s$: a state • $a$: an action • $t$: time step or token position

$b(s)\nabla \sum_a \pi(a|s) = b(s)\nabla 1 = 0$.

**Q4. What is reward-to-go and why use it?** Use only rewards from $t$ onward for action $a_t$; earlier rewards cannot be affected by $a_t$ and only add variance.

**Q5. What is the best simple baseline?** The state value $V^\pi(s)$, making the weight an advantage estimate.

**Q6. Why is REINFORCE on-policy?** The gradient is an expectation under the current policy; after an update, old samples come from a different distribution.

**Q7. What does the value of the REINFORCE surrogate loss tell you?** Little: its gradient is what matters. Track rewards instead.

**Q8. Why might all-positive rewards slow REINFORCE down?** Every sampled action is pushed up; learning relies on small differences between large weights, which is high variance. A baseline fixes it.

**Q9. What is $\nabla_\theta \log \pi_\theta(a)$ for a softmax policy over logits?** $e_a - \pi_\theta$: one-hot of the sampled action minus the probability vector.

**Q10. Derive the variance-optimal constant baseline.** Minimise $\mathbb{E}[(R-b)^2 g^2]$: $b^* = \mathbb{E}[g^2 R]/\mathbb{E}[g^2]$, a score-weighted mean reward.

**Q11. Why does a constant reward offset hurt REINFORCE without a baseline?** It adds $c \cdot \nabla \log \pi$ to every sample: zero in expectation but large variance; every action looks good.

**Key takeaways**

- $\nabla J = \mathbb{E}[\sum_t \nabla \log \pi_\theta(a_t|s_t)\, \hat{A}_t]$: only log-probs of taken actions are needed.
- Reward-to-go and baselines reduce variance without adding bias (if the baseline does not depend on the action).
- REINFORCE: sample, compute advantages, minimise $-\sum \log \pi \cdot \hat{A}$, discard data.
- The surrogate loss value is not a progress metric; advantages must be detached.
- Softmax score $= e_a - \pi_\theta$.
- Optimal baseline $b^* = \mathbb{E}[g^2 R]/\mathbb{E}[g^2]$; the mean reward gets most of the gain.

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\nabla_\theta$: gradient with respect to $\theta$ • $s$: a state • $a$: an action • $b(s_t)$: baseline (must not depend on the action) • $t$: time step or token position • $V(s)$: state value (expected future return from $s$) • $V$: vocabulary size • $\theta$: the model's trainable weights • $g$: a gradient • *and 2 more: see the Notation table*

CHAPTER 16

# Actor-Critic Methods and GAE

**Why this chapter matters.**
PPO for RLHF is an **actor-critic** method: the policy (actor) is trained with advantages computed by **Generalised Advantage Estimation** (GAE) from a learned value function (critic). Interviewers often ask you to write GAE, explain what $\lambda$ does, and explain why many LLM teams dropped the critic. Recent work (VC-PPO, VAPO) shows that a well-trained critic can still beat critic-free methods on long reasoning, so both sides of the argument matter.

## 16.1 From REINFORCE to actor-critic

REINFORCE uses the Monte Carlo return $\hat{G}_t$, which is unbiased but noisy. An **actor-critic** method learns a value function $V_\phi(s)$ (the **critic**, with its own weights $\phi$) and uses it to build lower-variance advantage estimates for the policy (the **actor**).

The simplest actor-critic advantage is the one-step TD error from Chapter 14:

$$\hat{A}_t = \delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t).$$

If $V_\phi = V^\pi$ exactly, then $\mathbb{E}[\delta_t \mid s_t, a_t] = Q^\pi(s_t, a_t) - V^\pi(s_t) = A^\pi(s_t, a_t)$: the TD error is an unbiased advantage estimate. With an imperfect critic it is biased, but its variance is much lower than the Monte Carlo return's.

The critic is trained by regression on value targets $\hat{R}_t$ (for example Monte Carlo returns or TD($\lambda$) returns):

$$\mathcal{L}_V(\phi) = \frac{1}{2}\,\mathbb{E}_t[(V_\phi(s_t) - \hat{R}_t)^2].$$

**A2C and A3C.** *Asynchronous Methods for Deep Reinforcement Learning* (Mnih et al., arXiv 1602.01783) introduced A3C: many workers collect experience in parallel and update a shared actor-critic asynchronously. A2C is the synchronous version (wait for all workers, then update), which is simpler and uses GPUs better. Both use $n$-step returns and an entropy bonus.

## 16.2 Generalised Advantage Estimation

*High-Dimensional Continuous Control Using Generalized Advantage Estimation* (Schulman et al., arXiv 1506.02438) defines a family of advantage estimators that smoothly trade bias for variance.

**$k$-step advantages.** Using $k$ real rewards and then the critic:

$$\hat{A}_t^{(k)} = r_t + \gamma r_{t+1} + \cdots + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k}) - V(s_t).$$

**Notation on this page:** $\lambda$: GAE parameter (0 = one-step TD, 1 = Monte Carlo) • $t$: time step or token position • $G$: group size (samples per prompt) • $\phi$: critic weights • $V(s)$: state value (expected future return from $s$) • $s$: a state • $V$: vocabulary size • $\gamma$: discount factor (0 to 1) • $\delta_t$: TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$ • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $r_t$: reward at step $t$ • *and 8 more: see the Notation table*

A neat identity: this equals a discounted sum of TD errors, because the intermediate value terms cancel (telescope):

$$\hat{A}_t^{(k)} = \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}.$$

*Check for $k = 2$:* $\delta_t + \gamma\delta_{t+1} = r_t + \gamma V(s_{t+1}) - V(s_t) + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - \gamma V(s_{t+1}) = r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t)$.

**GAE** takes an exponentially weighted average of all $k$-step advantages, with weights $(1-\lambda)\lambda^{k-1}$:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = (1-\lambda)(\hat{A}_t^{(1)} + \lambda\hat{A}_t^{(2)} + \lambda^2\hat{A}_t^{(3)} + \dots) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \, \delta_{t+l}.$$

(To get the last form, substitute the TD-error sums and collect the coefficient of each $\delta_{t+l}$: it is $\gamma^l(1-\lambda)(\lambda^l + \lambda^{l+1} + \dots) = (\gamma\lambda)^l$.)

**The two extremes.**

- $\lambda = 0$: $\hat{A}_t = \delta_t$, the one-step TD advantage. Low variance, biased by critic errors.
- $\lambda = 1$: $\hat{A}_t = \sum_l \gamma^l \delta_{t+l} = \hat{G}_t - V(s_t)$, the Monte Carlo return minus a baseline. Unbiased, high variance.

**Computation.** GAE is computed backwards with the recursion

$$\hat{A}_t = \delta_t + \gamma\lambda\,\hat{A}_{t+1}, \qquad \hat{A}_T = 0,$$

and $V(s_T) = 0$ at a terminal state. The value targets for the critic are usually $\hat{R}_t = \hat{A}_t + V(s_t)$ (the TD($\lambda$) return).

```
import torch

def gae(rewards, values, gamma=1.0, lam=0.95):
    """rewards: (T,), values: (T,) = V(s_0..s_{T-1}); episode ends after step T-1."""
    T = rewards.shape[0]
    adv = torch.zeros(T)
    next_value, running = 0.0, 0.0
    for t in reversed(range(T)):
        delta = rewards[t] + gamma * next_value - values[t]
        running = delta + gamma * lam * running
        adv[t] = running
        next_value = values[t]
    return adv, adv + values          # advantages, value targets

r = torch.tensor([0.0, 0.0, 0.0, 1.0])        # LLM-style: reward only at the end
V = torch.tensor([0.5, 0.4, 0.6, 0.7])
print("lam=1:", gae(r, V, 1.0, 1.0)[0])        # = G_t - V(s_t) = 1 - V
print("lam=0:", gae(r, V, 1.0, 0.0)[0])        # = delta_t = V(s_{t+1}) - V(s_t) (+ r at
 ↪  end)
print("lam=.95:", gae(r, V, 1.0, 0.95)[0])
```

*Reading the output.* With $\lambda = 1$, each token's advantage is "final reward minus my value estimate". With $\lambda = 0$, each token's advantage is "how much did the value estimate go up after this token": a fine-grained credit signal, but only as good as the critic.

**Notation on this page:** $\gamma$: discount factor (0 to 1) • $\delta_t$: TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$ • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $t$: time step or token position • $k$: number of real rewards before bootstrapping • $l$: steps ahead; in VAPO, the response length • $V(s)$: state value (expected future return from $s$) • $r_t$: reward at step $t$ • $s$: a state • $V$: vocabulary size • $\lambda$: GAE parameter (0 = one-step TD, 1 = Monte Carlo) • *and 3 more: see the Notation table*

**Why this, not that? Why GAE with lambda below 1, not plain Monte Carlo advantages?**

**Why.** In long episodes, Monte Carlo returns are very noisy. GAE with $\lambda \approx 0.95$ shrinks the weight on far-future TD errors and lets a good critic give each step its own credit, which speeds up learning in classic control tasks.
**Trade-off.** Lower $\lambda$ means more trust in the critic, so critic errors become advantage bias. In LLM RL with one reward at the end of thousands of tokens, $(\gamma\lambda)^l$ with $\lambda = 0.95$ makes the final reward's influence on early tokens vanish ($0.95^{1000} \approx 5 \times 10^{-23}$), so early tokens learn only from the (biased) critic.
**When the alternative wins.** For long chain-of-thought with a terminal reward, $\lambda = 1$ (or the length-adaptive rule below) is safer, and critic-free Monte Carlo methods (GRPO, RLOO) avoid the critic entirely.

## 16.3 The critic in LLM RL

In PPO-RLHF (Chapter 20), the critic is usually a copy of the reward model or the policy with a scalar **value head** (a linear layer from the last hidden state to one number), giving a value for every token position. This has costs and pitfalls:

- **Memory and compute.** A second full-size model to store, run and train.
- **Initialisation bias.** A critic initialised from the reward model was trained to score *complete* responses, not partial ones. *What's Behind PPO's Collapse in Long-CoT? Value Optimization Holds the Secret* (Yuan et al., VC-PPO, arXiv 2503.01491) found this bias makes early tokens get systematically wrong advantages, and fixed it with **value pretraining**: train the critic on rollouts from the fixed initial policy before starting PPO.
- **Reward signal decay.** With $\lambda < 1$, the terminal reward barely reaches early tokens. VC-PPO **decouples** GAE: the critic's targets use $\lambda_{\text{critic}} = 1.0$ (unbiased returns), while the policy's advantages use $\lambda_{\text{actor}}$ (0.95 in its baseline comparison, with values from 0.9 to 1.0 studied).
- **Variable lengths.** *VAPO: Efficient and Reliable Reinforcement Learning for Advanced Reasoning Tasks* (Yue et al., arXiv 2504.05118) adds **length-adaptive GAE**, setting $\lambda_{\text{policy}} = 1 - \frac{1}{\alpha l}$ with $\alpha = 0.05$ for a response of length $l$, so the effective horizon $1/(1-\lambda)$ grows with the response. Together with DAPO-style tricks (clip-higher with $\epsilon_{\text{high}} = 0.28$, token-level loss), VAPO reports 60.4 on AIME 2024 with a Qwen 32B base model, more than 10 points above DeepSeek-R1-Zero-Qwen-32B and DAPO under the same settings.

**Why this, not that? Why critic-free methods (GRPO, RLOO) for LLMs, not an actor-critic?**

**Why.** No second large model to train and store; no critic initialisation or value-bias problems; the per-prompt group baseline is an unbiased (or nearly unbiased) Monte Carlo estimate of $V(x)$ that is easy to get when you sample several responses per prompt anyway.
**Trade-off.** All tokens in a response share one advantage, so credit assignment within a response is crude; you need several samples per prompt; and a prompt where all samples get the same reward gives zero signal.
**When the alternative wins.** VC-PPO and VAPO show that a carefully trained critic can be more sample-efficient and stable for long reasoning. Critics also help in multi-turn agent

**Notation on this page:** $\lambda$: GAE parameter (0 = one-step TD, 1 = Monte Carlo) • $\gamma$: discount factor (0 to 1) • $l$: steps ahead; in VAPO, the response length • $\alpha$: VAPO length-adaptive constant (0.05) • $\epsilon$: clip range for the probability ratio • $x$: the prompt (input) • $V$: vocabulary size

tasks with intermediate rewards, where one response-level advantage is too coarse.

## 16.4 Practical details of critics

- **Value loss clipping.** PPO implementations often clip the new value prediction to stay within $\epsilon_v$ of the old one, and take the larger of the clipped and unclipped squared errors. Studies of PPO details (*Implementation Matters in Deep Policy Gradients*, Engstrom et al., arXiv 2005.12729; *What Matters In On-Policy Reinforcement Learning?*, Andrychowicz et al., arXiv 2006.05990) found value clipping does not help and can hurt in standard benchmarks; it is still common in RLHF code.
- **Return / reward normalisation.** Scaling rewards so returns have roughly unit variance keeps the value loss in a sensible range.
- **Separate or shared networks.** Sharing the trunk saves memory but couples the two losses; LLM implementations usually use separate models.
- **Value loss coefficient.** With a shared network, the total loss is $\mathcal{L}_\pi + c_v\mathcal{L}_V - c_H H$ with $c_v \approx 0.5$ and a small entropy coefficient $c_H$.

## 16.5 Seeing the bias-variance trade-off

What matters for the policy gradient is the quantity that multiplies $\nabla \log \pi$. Since $-V(s_t)$ is a baseline (it does not bias the gradient, Chapter 15), the relevant estimate is $\hat{A}_t + V(s_t)$, an estimate of $Q(s_t, a_t)$. With $\lambda = 1$ this is the Monte Carlo return: unbiased whatever the critic's errors, but it carries the noise of every future reward. With $\lambda = 0$ it is $r_t + \gamma V(s_{t+1})$: low noise, but any error in the critic at $s_{t+1}$ goes straight into the gradient. A simulation with a 20-step episode, noisy rewards and a critic with fixed random errors:

```
import torch
torch.manual_seed(0)

T, n_eps = 20, 4000
mean_r = torch.linspace(0.0, 1.0, T)                    # expected reward at each step
V_true = mean_r.flip(0).cumsum(0).flip(0)               # V(t) = sum of expected future
 ↪  rewards (gamma = 1)

def gae(rewards, values, lam, gamma=1.0):
    adv = torch.zeros_like(rewards); running = torch.zeros(rewards.size(0))
    for t in reversed(range(rewards.size(1))):
        nv = values[:, t + 1] if t + 1 < rewards.size(1) else torch.zeros(rewards.size(0))
        running = rewards[:, t] + gamma * nv - values[:, t] + gamma * lam * running
        adv[:, t] = running
    return adv

rewards = mean_r + 1.0 * torch.randn(n_eps, T)          # noisy rewards
critic = V_true + 1.0 * torch.randn(T)                  # an imperfect critic (same
 ↪  errors in every episode)
```

**Notation on this page:** $\epsilon$: clip range for the probability ratio • $\mathcal{L}$: a loss to minimise • $V$: vocabulary size • $c_v, c_H$: value-loss and entropy coefficients • $\nabla_\theta$: gradient with respect to $\theta$ • $V(s)$: state value (expected future return from $s$) • $s$: a state • $t$: time step or token position • $A, \hat{A}$: advantage (how much better an action is than average); hat = estimate • $Q(s, a)$: action value (expected return after taking $a$ in $s$) • $a$: an action • *and 3 more: see the Notation table*

```
for lam in (0.0, 0.5, 0.9, 0.95, 1.0):
    a = gae(rewards, critic.expand(n_eps, T), lam)
    q_hat = a + critic                                  # the part that multiplies grad
↪   log pi (V(s_t) is a baseline)
    bias = (q_hat.mean(0) - V_true).abs().mean().item()   # true Q = V here (one action
↪   per state)
    var = q_hat.var(0).mean().item()
    print(f"lambda={lam:4.2f}: |bias| {bias:.3f}, variance {var:.2f}")
```

Bias falls and variance rises as $\lambda$ goes from 0 to 1 (the small bias left at $\lambda = 1$ is sampling noise). The variance grows with the number of future noisy rewards, which is why **long horizons** push you towards smaller $\lambda$ (or a better critic), and why **single terminal rewards** (most LLM tasks, where intermediate rewards are zero apart from the KL) change the picture: with only one noisy reward at the end, $\lambda = 1$ adds little variance per token.

## 16.6 Length-adaptive $\lambda$ and decoupled $\lambda$

In LLM RL a "trajectory" is a response of anywhere from 50 to 30,000 tokens. With $\gamma = 1$, a fixed $\lambda = 0.95$ means the terminal reward's weight at a token $k$ positions before the end is $0.95^k$: after 100 tokens it is 0.006, so for long responses the early tokens get almost no signal from the actual reward and depend entirely on the critic.

- **Decoupled GAE** (VC-PPO, arXiv 2503.01491): use $\lambda_{\text{critic}} = 1.0$ to compute the **value targets** (unbiased Monte Carlo targets, so the critic learns the true values) and $\lambda_{\text{actor}} = 0.95$ for the **policy advantages** (lower variance).
- **Length-adaptive GAE** (VAPO, arXiv 2504.05118): set the actor's $\lambda$ per response from its length $l$,

$$\lambda_{\text{policy}} = 1 - \frac{1}{\alpha l}, \qquad \alpha = 0.05.$$

  *Worked example.* $l = 100$: $\lambda = 1 - 1/5 = 0.8$. $l = 1{,}000$: $\lambda = 0.98$. $l = 10{,}000$: $\lambda = 0.998$. Their motivation: the sum of GAE weights $\sum_k \lambda^k \approx 1/(1-\lambda) = \alpha l$ grows in proportion to length, so short and long responses get a comparable balance of bias and variance.

**Why this, not that? Why γ = 1 for LLM responses, not a discount below 1?**

**Why.** A response is a short, finite episode with one reward at the end. Discounting would make early tokens care less about the final outcome, which is the opposite of what we want: the first tokens of a proof matter as much as the last. With $\gamma = 1$ the return of every token is the final reward minus future KL.
**Trade-off.** No discount means higher variance for long responses, which must then be controlled by $\lambda$, a good critic, or critic-free baselines.
**When the alternative wins.** Discounting is natural for long or infinite-horizon control tasks, for multi-turn agents with intermediate rewards where you want to prefer *faster* success, and as a mild variance-reduction tool.

**Notation on this page:** $\lambda$: GAE parameter (0 = one-step TD, 1 = Monte Carlo) • $\gamma$: discount factor (0 to 1) • $k$: number of real rewards before bootstrapping • $l$: steps ahead; in VAPO, the response length • $\alpha$: VAPO length-adaptive constant (0.05)

## 16.7 Bootstrapping at truncation

When an episode is cut by a time or length limit rather than ending naturally, the state is **not terminal**: the true return would continue. Two options:

1. **Bootstrap:** treat $V(s_T)$ as the value of the cut-off state and use $\delta_{T-1} = r_{T-1} + \gamma V(s_T) - V(s_{T-1})$.
2. **Treat as terminal** (value 0), which is wrong in control tasks and biases values downwards near the limit.

In LLM RL the convention is different: a response that hits the maximum length has **no final answer**, so it is treated as terminal with a penalty or a zero reward (the EOS trick, DAPO's overlong shaping), or its tokens are **masked out** of the loss entirely (DAPO's overlong filtering, TRL `mask_truncated_completions=True`). Bootstrapping would ask the critic to guess the value of an unfinished answer, which is rarely reliable.

**Why this, not that? Why mask truncated responses, not give them a zero reward?**

**Why.** A truncated response might have been heading to a correct answer; a zero reward punishes it as if it were wrong, adding noise to the advantages and teaching the model that long reasoning is bad.
**Trade-off.** Masking throws away the samples, wastes generation compute, and removes any pressure to keep responses under the limit, so lengths can drift up until most samples are truncated.
**When the alternative wins.** A penalty (soft overlong punishment) is better when you *want* shorter responses or a hard length budget; zero reward is the simplest choice when truncation is rare.

## 16.8 Common mistakes

- Computing value targets with the same small $\lambda$ as the advantages for very long responses (the critic never sees the real reward).
- Bootstrapping from the critic at the length limit in LLM RL.
- Using $\gamma < 1$ for single-response tasks without a reason.

**Interview check**

**Q1. What is an actor-critic method?** A policy-gradient method where a learned value function (critic) provides lower-variance advantage estimates for the policy (actor).
**Q2. Write the GAE formula and its recursion.** $\hat{A}_t = \sum_l (\gamma\lambda)^l \delta_{t+l}$; recursively $\hat{A}_t = \delta_t + \gamma\lambda\hat{A}_{t+1}$, with $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$.
**Q3. What do $\lambda = 0$ and $\lambda = 1$ give?** $\lambda = 0$: TD(0) advantage $\delta_t$ (low variance, biased). $\lambda = 1$: $\hat{G}_t - V(s_t)$ (unbiased, high variance).
**Q4. Why is $\lambda = 0.95$ problematic for long chain-of-thought?** The terminal reward is discounted by $(\gamma\lambda)^l$ on its way back; after thousands of tokens it vanishes, so early tokens rely on the biased critic.

**Notation on this page:** $V(s)$: state value (expected future return from $s$) • $s$: a state • $T$: number of steps or tokens in a sequence • $V$: vocabulary size • $\gamma$: discount factor (0 to 1) • $\delta_t$: TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$ • $r_t$: reward at step $t$ • $\lambda$: GAE parameter (0 = one-step TD, 1 = Monte Carlo) • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $t$: time step or token position • $l$: steps ahead; in VAPO, the response length • *and 1 more: see the Notation table*

**Q5. What did VC-PPO change?** Value pretraining to remove critic initialisation bias, and decoupled GAE ($\lambda_{\text{critic}} = 1$, separate $\lambda_{\text{actor}}$).
**Q6. What is length-adaptive GAE?** VAPO's $\lambda_{\text{policy}} = 1 - 1/(\alpha l)$, making the effective horizon proportional to response length.
**Q7. How is the critic usually built for LLM PPO?** A copy of the reward model or policy with a scalar value head that outputs a value per token.
**Q8. Which quantity should be unbiased for the policy gradient, $\hat{A}$ or $\hat{A} + V(s_t)$?** $\hat{A} + V(s_t)$ (the Q estimate): $V(s_t)$ is a baseline. $\lambda = 1$ makes it the unbiased Monte Carlo return.
**Q9. What is decoupled GAE?** Different $\lambda$ for the critic targets (1.0, unbiased) and the actor advantages (e.g. 0.95, lower variance), from VC-PPO.
**Q10. Compute VAPO's length-adaptive $\lambda$ for a 200-token response.** $1 - 1/(0.05 \cdot 200) = 0.9$.
**Q11. How should truncated responses be handled in LLM RL?** Usually as terminal with a penalty, or masked out of the loss; not bootstrapped from the critic.
**Q12. With $\gamma = 1$ and $\lambda = 0.95$, how much of the final reward reaches a token 100 positions before the end?** $0.95^{100} \approx 0.006$ of it; the rest comes through the critic.

**Key takeaways**

- Actor-critic: a learned $V_\phi$ supplies advantages; the TD error is the simplest one.
- GAE: $\hat{A}_t = \sum_l (\gamma\lambda)^l \delta_{t+l}$; $\lambda$ trades bias (low $\lambda$) for variance (high $\lambda$).
- For long LLM outputs with terminal rewards, $\lambda$ near 1 (or length-adaptive) matters.
- Critics cost memory and have initialisation bias; GRPO/RLOO avoid them, VC-PPO/VAPO show how to fix them.
- $\lambda \to 1$: unbiased, high variance; $\lambda \to 0$: low variance, critic bias.
- Long responses: decouple $\lambda$ (critic 1.0) or make it length-adaptive (VAPO).
- Truncated LLM responses: penalise or mask, don't bootstrap.

**Notation on this page:** $\lambda$: GAE parameter (0 = one-step TD, 1 = Monte Carlo) • $\alpha$: VAPO length-adaptive constant (0.05) • $l$: steps ahead; in VAPO, the response length • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $V(s)$: state value (expected future return from $s$) • $s$: a state • $t$: time step or token position • $V$: vocabulary size • $\gamma$: discount factor (0 to 1) • $\phi$: critic weights • $\delta_t$: TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$

CHAPTER 17

# Trust Regions: TRPO and PPO

**Why this chapter matters.**
**PPO** (Proximal Policy Optimization) is the algorithm behind InstructGPT and the original ChatGPT RLHF, and its clipped objective survives inside GRPO, DAPO and almost every LLM RL method. You must be able to derive the clipped objective, explain the four cases of the clip, say why the importance ratio appears, and list the implementation details that make PPO work.

## 17.1 Why big policy steps break training

In supervised learning, a bad step is corrected by the next batch from the same fixed dataset. In RL, the policy *creates its own data*. A step that is too large can make the policy much worse, and then it collects worse data, which can make it worse again: a collapse that is hard to recover from. Policy-gradient estimates are also noisy, so any single step may point the wrong way.

Two ideas fix this:

1. **Reuse data safely.** Use importance sampling so data from the old policy can be used for several gradient steps.
2. **Limit the step.** Keep the new policy close to the old one, measured by how much the action *distributions* change (KL divergence), not how much the weights change.

## 17.2 The performance difference lemma

For two policies $\pi$ (old) and $\pi'$ (new), the exact difference in their returns is

$$J(\pi') - J(\pi) = \frac{1}{1-\gamma}\, \mathbb{E}_{s\sim d^{\pi'}}\, \mathbb{E}_{a\sim\pi'(\cdot|s)}[A^{\pi}(s,a)],$$

where $d^{\pi'}$ is the (normalised, discounted) distribution of states visited by the *new* policy, and $A^{\pi}$ is the *old* policy's advantage.

*Proof sketch.* Write $A^{\pi}(s_t,a_t) = \mathbb{E}[r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)]$. Sum $\gamma^t A^{\pi}(s_t,a_t)$ along a trajectory of $\pi'$: the value terms telescope to $-V^{\pi}(s_0)$, leaving $\sum_t \gamma^t r_t - V^{\pi}(s_0)$, whose expectation under $\pi'$ is $J(\pi') - J(\pi)$.

*Intuition.* The new policy is better exactly when, in the states it visits, it picks actions that the old policy's advantage function rates above average.

**The problem:** we cannot sample states from $d^{\pi'}$ before we have $\pi'$. So we approximate by using

**Notation on this page:** $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\gamma$: discount factor (0 to 1) • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $J(\theta)$: objective to maximise (expected reward) • $s$: a state • $a$: an action • $d^{\pi}$: discounted state-visit distribution of policy $\pi$; $d_{\text{targ}}$: target KL • $\pi_\theta$: the policy (the model), with weights $\theta$ • $V(s)$: state value (expected future return from $s$) • $r(x,y)$: the reward for response $y$ to prompt $x$ • $t$: time step or token position • *and 1 more: see the Notation table*

the old state distribution and importance-weight the actions:

$$L_\pi(\pi') = \mathbb{E}_{s\sim d^\pi}\,\mathbb{E}_{a\sim\pi}\Big[\frac{\pi'(a\mid s)}{\pi(a\mid s)}\,A^\pi(s,a)\Big].$$

This **surrogate objective** matches the true improvement to first order: $L_\pi(\pi) = 0$, and its gradient at $\pi' = \pi$ is the ordinary policy gradient. So it is accurate only while $\pi'$ stays close to $\pi$.

The *Trust Region Policy Optimization* paper (Schulman et al., arXiv 1502.05477) proved a lower bound of the form

$$J(\pi') - J(\pi) \geq L_\pi(\pi') - C\cdot\max_s \mathrm{KL}(\pi(\cdot\mid s)\,\|\,\pi'(\cdot\mid s)),$$

for a constant $C$ that depends on $\gamma$ and the size of the advantages. Maximising the right side guarantees improvement. This justifies "improve the surrogate, but stay close in KL."

## 17.3 TRPO

TRPO turns the bound into a constrained problem with a practical average KL:

$$\max_\theta\ \mathbb{E}\Big[\frac{\pi_\theta(a\mid s)}{\pi_{\theta_{\text{old}}}(a\mid s)}\hat{A}\Big]\quad\text{subject to}\quad \mathbb{E}_s\Big[\mathrm{KL}(\pi_{\theta_{\text{old}}}(\cdot\mid s)\,\|\,\pi_\theta(\cdot\mid s))\Big]\leq\delta.$$

Approximating the objective to first order (gradient $g$) and the KL to second order (the **Fisher information matrix** $F$, the Hessian of the KL) gives the **natural gradient** step:

$$\theta_{\text{new}} = \theta_{\text{old}} + \sqrt{\frac{2\delta}{g^\top F^{-1} g}}\,F^{-1}g.$$

For a network with millions of weights, $F$ cannot be stored, so TRPO solves $Fx = g$ with the **conjugate gradient** method using only Fisher-vector products, then does a **backtracking line search** to make sure the KL constraint holds and the surrogate actually improves.

TRPO works but is complicated, needs second-order computations, and does not fit easily with shared actor-critic networks, dropout, or very large models. PPO keeps the idea and drops the machinery.

## 17.4 PPO-Clip

*Proximal Policy Optimization Algorithms* (Schulman et al., arXiv 1707.06347) defines the **probability ratio**

$$\rho_t(\theta) = \frac{\pi_\theta(a_t\mid s_t)}{\pi_{\theta_{\text{old}}}(a_t\mid s_t)},$$

which is 1 before any update, and the clipped objective (to *maximise*)

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\Big[\min\big(\rho_t(\theta)\,\hat{A}_t,\ \text{clip}(\rho_t(\theta),\,1-\epsilon,\,1+\epsilon)\,\hat{A}_t\big)\Big],$$

with $\epsilon$ usually 0.1 to 0.2.

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $s$: a state • $a$: an action • $d^\pi$: discounted state-visit distribution of policy $\pi$; $d_{\text{targ}}$: target KL • $L^{\text{CLIP}}$: PPO clipped surrogate objective; $L_\pi(\pi')$: surrogate • $D_{\text{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $J(\theta)$: objective to maximise (expected reward) • $C$: constant in the TRPO bound • $\gamma$: discount factor (0 to 1) • *and 10 more: see the Notation table*

### 17.4.1 The four cases

| Advantage | Ratio | Which term is the min? | Gradient |
|---|---|---|---|
| $\hat{A} > 0$ | $\rho \le 1+\epsilon$ | unclipped $\rho\hat{A}$ | pushes $\rho$ up (normal) |
| $\hat{A} > 0$ | $\rho > 1+\epsilon$ | clipped $(1+\epsilon)\hat{A}$, a constant | **zero**: already increased enough |
| $\hat{A} < 0$ | $\rho \ge 1-\epsilon$ | unclipped $\rho\hat{A}$ | pushes $\rho$ down (normal) |
| $\hat{A} < 0$ | $\rho < 1-\epsilon$ | clipped $(1-\epsilon)\hat{A}$, a constant | **zero**: already decreased enough |

The important asymmetry: the clip only stops the policy from moving *further in the direction the advantage wants*. If an update has gone the *wrong* way (say $\hat{A} > 0$ but $\rho$ fell below $1-\epsilon$), the min picks the unclipped term, which is smaller, and the gradient still pulls the ratio back. The objective is a **pessimistic** (lower) bound on the unclipped surrogate.

*Worked example.* $\hat{A} = 2$, $\epsilon = 0.2$. If $\rho = 1.1$: $\min(2.2, 2.2) = 2.2$, gradient flows. If $\rho = 1.5$: $\min(3.0, 2.4) = 2.4$, a constant, so no gradient. If $\rho = 0.5$: $\min(1.0, 1.6) = 1.0$, the unclipped term, so the gradient pushes $\rho$ back up.

```
import torch

def ppo_clip_loss(logp_new, logp_old, adv, eps=0.2):
    ratio = (logp_new - logp_old).exp()
    unclipped = ratio * adv
    clipped = ratio.clamp(1 - eps, 1 + eps) * adv
    return -torch.min(unclipped, clipped).mean()            # minimise the negative

for adv, rho in [(2.0, 1.1), (2.0, 1.5), (2.0, 0.5), (-2.0, 0.5), (-2.0, 1.5)]:
    logp = torch.tensor(float(torch.tensor(rho).log()), requires_grad=True)
    loss = ppo_clip_loss(logp, torch.tensor(0.0), torch.tensor(adv))
    loss.backward()
    print(f"A={adv:+.0f} ratio={rho}: loss={loss.item():+.2f}  d loss/d
    ↪  logp={logp.grad.item():+.2f}")
```

**Why this, not that? Why clip the ratio (PPO), not enforce a KL constraint (TRPO)?**

**Why.** Clipping is a first-order method: one extra line in the loss, compatible with Adam, mini-batches, shared networks and huge models. It performs as well as or better than TRPO on standard benchmarks, at a fraction of the complexity.
**Trade-off.** Clipping does not actually bound the KL: the policy can still drift far over many epochs, and clipped tokens give zero gradient, which wastes data (CISPO and others revisit this for rare but important tokens, Chapter 28). The guarantee of the TRPO bound is lost.
**When the alternative wins.** When you need a hard guarantee on step size, or when the clip throws away too much signal: a KL penalty (PPO-Penalty), natural-gradient methods, or

**Notation on this page:** $A, \hat{A}$: advantage (how much better an action is than average); hat = estimate • $\epsilon$: clip range for the probability ratio • $\rho_t(\theta)$: probability ratio $\pi_\theta / \pi_{\theta_{\text{old}}}$ for the taken action

soft alternatives (CISPO's clipped importance weights, SAPO's soft gates) may be better.

### 17.4.2 Dual clip

With a very negative advantage and a very large ratio, $\rho\hat{A}$ can be huge and negative, giving a huge gradient. **Dual-clip PPO** adds a floor for $\hat{A} < 0$: $\max\big(\min(\rho\hat{A}, \text{clip}(\rho)\hat{A}),\, c\,\hat{A}\big)$ with $c$ around 3. Several LLM RL frameworks implement it (in veRL it is `clip_ratio_c`).

## 17.5 PPO-Penalty and adaptive KL

The PPO paper also proposed using a KL penalty instead of the clip:

$$L^{\text{KLPEN}}(\theta) = \mathbb{E}_t[\rho_t(\theta)\hat{A}_t] - \beta\,\mathbb{E}_t\Big[\text{KL}(\pi_{\theta_{\text{old}}}(\cdot \mid s_t) \,\|\, \pi_\theta(\cdot \mid s_t))\Big],$$

with $\beta$ adapted after each update towards a target KL $d_{\text{targ}}$: if the measured KL $d > 1.5\,d_{\text{targ}}$, double $\beta$; if $d < d_{\text{targ}}/1.5$, halve it. In the paper's experiments this worked worse than clipping.

**Do not confuse** this penalty (to the *old* policy, for step size) with the RLHF KL penalty (to the *reference* model, to stop drift from the SFT model, Chapter 20). InstructGPT-style RLHF uses an adaptive controller of the same shape for the reference-model KL coefficient.

## 17.6 Entropy bonus

Adding $c_H\,\mathbb{E}_t[H(\pi_\theta(\cdot \mid s_t))]$ to the objective rewards randomness and slows the collapse of exploration. Typical $c_H$ is 0.01 in control tasks. In LLM RL it is usually zero or tiny, because the vocabulary is huge and an entropy bonus can quickly produce gibberish; entropy is better managed by clip-higher, data selection and temperature (Chapter 31).

## 17.7 The full PPO algorithm

For iteration $k = 1, 2, \ldots$:

1. **Rollout.** Run $\pi_{\theta_{\text{old}}}$ to collect a batch of trajectories. Store states, actions, rewards and the old log-probs $\log \pi_{\theta_{\text{old}}}(a_t \mid s_t)$.
2. **Advantages.** Compute $V_\phi(s_t)$ with the critic; compute GAE advantages $\hat{A}_t$ and value targets $\hat{R}_t$.
3. **Optimise.** For $K$ epochs, shuffle the batch into mini-batches. For each mini-batch, minimise

$$\mathcal{L} = -L^{\text{CLIP}}(\theta) + c_v\,\mathcal{L}_V(\phi) - c_H\,H.$$

4. Set $\theta_{\text{old}} \leftarrow \theta$.

In LLM RL, step 3 with $K = 1$ and one mini-batch makes $\rho_t = 1$ exactly, so the clip never activates and PPO reduces to REINFORCE with advantages. The clip only matters when one batch of rollouts is split into several mini-batch updates (or reused for several epochs), or when rollouts are stale (asynchronous training).

**Notation on this page:** $\rho_t(\theta)$: probability ratio $\pi_\theta/\pi_{\theta_{\text{old}}}$ for the taken action • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $\text{clip}(u, a, b)$: limit $u$ to the range $[a, b]$ • $c$: dual-clip floor; $c_v$, $c_H$: value and entropy coefficients • $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $D_{\text{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $\beta$: KL-penalty coefficient (PPO-Penalty) • $s$: a state • *and 12 more: see the Notation table*

## 17.8 Implementation details that matter

PPO's results depend heavily on details. *Implementation Matters in Deep Policy Gradients* (Engstrom et al., arXiv 2005.12729) showed that code-level choices explained much of PPO's advantage over TRPO. *What Matters In On-Policy Reinforcement Learning?* (Andrychowicz et al., arXiv 2006.05990) tested over 50 choices at scale. For LLMs, *The N+ Implementation Details of RLHF with PPO: A Case Study on TL;DR Summarization* (Huang et al., arXiv 2403.17031) reproduced OpenAI's RLHF work and documented the details. Key ones:

- **Advantage normalisation** per mini-batch or batch (mean 0, std 1).
- **Gradient clipping** by global norm (0.5 to 1.0).
- **Learning-rate annealing** and Adam with a small learning rate.
- **Observation and reward normalisation** (control tasks); reward whitening (RLHF).
- **Orthogonal initialisation** and a small final-layer init for the policy (control tasks).
- **Separate value-function handling**: value clipping (often not helpful), value loss coefficient.
- **Correct handling of time limits**: an episode cut by a time limit is not a true terminal state; ideally bootstrap from $V(s_T)$ instead of using 0. (The code below, like many simple implementations, ignores this.)
- **Monitoring**: approximate KL between old and new policy, the **clip fraction** (share of samples where the clip is active), entropy, explained variance of the critic.

## 17.9 Code: PPO from scratch on CartPole

This reuses the `CartPole` class from Chapter 15. It collects 1,024 steps per iteration, computes GAE, and runs 4 epochs of mini-batch PPO with a critic and an entropy bonus. It also logs a k3 estimate of the KL from the old to the new policy and the clip fraction.

```
import math, random, torch, torch.nn as nn

class CartPole:
    """Same dynamics as Gymnasium's CartPole-v1 (Euler integration)."""
    def __init__(self, seed=0, max_steps=500):
        self.rng = random.Random(seed); self.max_steps = max_steps
    def reset(self):
        self.s = [self.rng.uniform(-0.05, 0.05) for _ in range(4)]; self.t = 0
        return torch.tensor(self.s)
    def step(self, action):
        x, x_dot, th, th_dot = self.s
        force = 10.0 if action == 1 else -10.0
        g, mc, mp, l, dt = 9.8, 1.0, 0.1, 0.5, 0.02
        cos, sin = math.cos(th), math.sin(th)
        temp = (force + mp * l * th_dot**2 * sin) / (mc + mp)
        th_acc = (g * sin - cos * temp) / (l * (4/3 - mp * cos**2 / (mc + mp)))
        x_acc = temp - mp * l * th_acc * cos / (mc + mp)
        x, x_dot = x + dt * x_dot, x_dot + dt * x_acc
        th, th_dot = th + dt * th_dot, th_dot + dt * th_acc
```

**Notation on this page:** $V(s)$: state value (expected future return from $s$) • $s$: a state • $T$: number of steps or tokens in a sequence • $V_\phi$: critic

```
        self.s = [x, x_dot, th, th_dot]; self.t += 1
        fell = abs(x) > 2.4 or abs(th) > 12 * 2 * math.pi / 360
        done = fell or self.t >= self.max_steps
        return torch.tensor(self.s), 1.0, done

torch.manual_seed(0)
env = CartPole(seed=1)
actor = nn.Sequential(nn.Linear(4, 64), nn.Tanh(), nn.Linear(64, 2))
critic = nn.Sequential(nn.Linear(4, 64), nn.Tanh(), nn.Linear(64, 1))
opt = torch.optim.Adam([*actor.parameters(), *critic.parameters()], lr=3e-3)
clip_eps, epochs, minibatch, gamma, lam = 0.2, 4, 256, 0.99, 0.95

def collect(n_steps=1024):
    S, A, LP, R, D, lengths = [], [], [], [], [], []
    s, ep_len = env.reset(), 0
    for _ in range(n_steps):
        with torch.no_grad():
            dist = torch.distributions.Categorical(logits=actor(s))
            a = dist.sample()
        s2, r, done = env.step(a.item())
        S.append(s); A.append(a); LP.append(dist.log_prob(a)); R.append(r);
 ↪  D.append(done)
        s, ep_len = s2, ep_len + 1
        if done:
            lengths.append(ep_len); s, ep_len = env.reset(), 0
    S = torch.stack(S)
    with torch.no_grad():
        V = critic(S).squeeze(-1); last_V = critic(s).item()
    adv, running = torch.zeros(n_steps), 0.0
    for t in reversed(range(n_steps)):                    # GAE over the whole buffer
        next_v = 0.0 if D[t] else (last_V if t == n_steps - 1 else V[t + 1].item())
        if D[t]: running = 0.0
        delta = R[t] + gamma * next_v - V[t].item()
        running = delta + gamma * lam * running
        adv[t] = running
    return S, torch.stack(A), torch.stack(LP), adv, adv + V, lengths

for it in range(60):
    S, A, old_logp, adv, ret, lengths = collect()
    for _ in range(epochs):
        for idx in torch.randperm(len(S)).split(minibatch):
            dist = torch.distributions.Categorical(logits=actor(S[idx]))
            logp = dist.log_prob(A[idx])
            ratio = (logp - old_logp[idx]).exp()
            a_mb = (adv[idx] - adv[idx].mean()) / (adv[idx].std() + 1e-8)
            pg_loss = -torch.min(ratio * a_mb, ratio.clamp(1 - clip_eps, 1 + clip_eps) *
 ↪  a_mb).mean()
            v_loss = 0.5 * (critic(S[idx]).squeeze(-1) - ret[idx]).pow(2).mean()
            loss = pg_loss + 0.5 * v_loss - 0.01 * dist.entropy().mean()
            opt.zero_grad(); loss.backward()
            nn.utils.clip_grad_norm_([*actor.parameters(), *critic.parameters()], 0.5)
```

```
            opt.step()
    if it % 10 == 0 or it == 59:
        with torch.no_grad():
            r_ = (actor(S).log_softmax(-1).gather(1, A[:, None]).squeeze(1) - old_logp)
            approx_kl = (r_.exp() - 1 - r_).mean()            # k3 estimate of KL(old ||
↪   new)
            clip_frac = ((r_.exp() - 1).abs() > clip_eps).float().mean()
        mean_len = sum(lengths) / max(len(lengths), 1)
        print(f"iter {it:2d}: mean episode length {mean_len:5.0f}  KL {approx_kl:.4f}
         ↪  clip frac {clip_frac:.2f}")
```

It reaches the 500-step maximum in under a minute on a laptop CPU. Note the KL and clip fraction: healthy PPO runs keep the KL around 0.01 and the clip fraction around 0.1 to 0.2. A clip fraction near 0.5 means the step size is too big; near 0, the clip is doing nothing.

## 17.10 The gradient of the clipped objective, written out

It helps to write the per-sample gradient explicitly. Using $\nabla_\theta \rho_t = \rho_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)$,

$$\nabla_\theta L_t^{\text{CLIP}} = \begin{cases} \rho_t \hat{A}_t \nabla_\theta \log \pi_\theta(a_t \mid s_t) & \text{if } (\hat{A}_t > 0,\ \rho_t \le 1+\epsilon) \text{ or } (\hat{A}_t < 0,\ \rho_t \ge 1-\epsilon), \\ 0 & \text{otherwise.} \end{cases}$$

So PPO is "importance-weighted policy gradient, with the sample switched off once it has moved far enough in the helpful direction". Two consequences you should be able to state:

1. At $\rho_t = 1$ (the first mini-batch after a rollout) the gradient is exactly the vanilla policy gradient $\hat{A}_t \nabla \log \pi$.
2. A sample's gradient is either fully on (weighted by $\rho_t$) or fully off. There is no partial down-weighting. Soft variants (SAPO's sigmoid gates, CISPO's clipped-but-kept weights, Chapter 28) change exactly this.

You can define the **clip indicator** $\mathbb{1}[\text{clipped}]_t$ from the cases above; its batch mean is the **clip fraction** that PPO implementations log. A healthy run usually shows a clip fraction of a few percent to about 20%; much higher means the steps are too big or the data too stale.

## 17.11 Why the clip does not bound the KL

The clip removes a sample's own gradient once it leaves $[1-\epsilon, 1+\epsilon]$ in the helpful direction, but **other samples' gradients move shared parameters**, and those changes can push the clipped sample's ratio further away. Over many epochs and mini-batches the ratios, and the KL, keep drifting. A small experiment with a softmax policy over six actions and 512 samples from one rollout:

```
import torch
torch.manual_seed(0)

# A one-state "policy" over 6 actions. One batch of 512 samples from the old policy.
```

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\nabla_\theta$: gradient with respect to $\theta$ • $\rho_t(\theta)$: probability ratio $\pi_\theta / \pi_{\theta_{\text{old}}}$ for the taken action • $s$: a state • $a$: an action • $t$: time step or token position • $\epsilon$: clip range for the probability ratio • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $L^{\text{CLIP}}$: PPO clipped surrogate objective; $L_\pi(\pi')$: surrogate • $\mathbb{1}[\cdot]$: 1 if the condition is true, else 0

```
n_act, N, eps = 6, 512, 0.2
old_logits = torch.zeros(n_act)
a = torch.multinomial(torch.softmax(old_logits, -1), N, replacement=True)
adv = torch.randn(n_act)[a] + 0.3 * torch.randn(N)       # noisy per-sample advantages
adv = (adv - adv.mean()) / adv.std()
logp_old = torch.log_softmax(old_logits, -1)[a]

def train(epochs, target_kl=None, lr=0.1):
    logits = old_logits.clone().requires_grad_(True)
    opt = torch.optim.SGD([logits], lr=lr)
    for ep in range(epochs):
        for mb in torch.randperm(N).split(64):              # 8 mini-batches per epoch
            logp = torch.log_softmax(logits, -1)[a[mb]]
            ratio = (logp - logp_old[mb]).exp()
            loss = -torch.min(ratio * adv[mb], ratio.clamp(1 - eps, 1 + eps) *
 ↪  adv[mb]).mean()
            opt.zero_grad(); loss.backward(); opt.step()
        with torch.no_grad():
            log_r = torch.log_softmax(logits, -1)[a] - logp_old
            approx_kl = ((log_r.exp() - 1) - log_r).mean().item()   # k3 estimate of
 ↪  KL(old || new)
        if target_kl is not None and approx_kl > 1.5 * target_kl:
            return ep + 1, log_r.exp(), approx_kl
    return epochs, log_r.exp(), approx_kl

for epochs in (1, 4, 10, 40):
    n, r, kl = train(epochs)
    print(f"{epochs:2d} epochs: max ratio {r.max():.2f}, min ratio {r.min():.2f}, "
          f"share outside clip {((r > 1 + eps) | (r < 1 - eps)).float().mean():.2f},
           ↪  approx KL {kl:.3f}")
n, r, kl = train(40, target_kl=0.01)
print(f"with target_kl=0.01: stopped after {n} epochs, max ratio {r.max():.2f}, approx
 ↪  KL {kl:.3f}")
```

With more epochs, ratios go well below $1 - \epsilon = 0.8$ and the KL keeps rising, even though every sample's own gradient was switched off at the boundary. Practical PPO code therefore adds an explicit **early stop on a target KL** (for example Stable-Baselines3's `target_kl`): measure the approximate KL after each epoch and stop when it exceeds about 1.5× the target. *Implementation Matters in Deep Policy Gradients* (arXiv 2005.12729) found that PPO's clipping does not by itself keep the policy inside a trust region, and *Revisiting Design Choices in Proximal Policy Optimization* (Hsu et al., arXiv 2009.10897) described further failure modes of the clipped objective.

**Why this, not that? Why several epochs over each batch (K > 1), not a single pass?**

**Why.** Collecting rollouts is the expensive part. Reusing each batch for several mini-batch updates squeezes more learning out of it, and the ratio and clip make that reuse safe enough. In control tasks $K = 10$ is common.

**Trade-off.** Every extra epoch makes the data more off-policy: ratios drift, more samples are clipped (wasted), and the KL grows (the experiment above). Too many epochs can collapse

**Notation on this page:** $\epsilon$: clip range for the probability ratio • $K$: optimisation epochs per batch

the policy.
**When the alternative wins.** In LLM RL, generation is costly but so is each training pass, rewards are noisy, and stability matters most, so $K = 1$ to 2 with a few mini-batches is typical (InstructGPT used one inner epoch). With strictly on-policy single updates, the clip is unnecessary.

## 17.12 The value loss and value clipping

The critic $V_\phi$ is trained to regress the return targets $\hat{R}_t = \hat{A}_t + V_{\phi_{\text{old}}}(s_t)$ (GAE advantages plus the old value):

$$\mathcal{L}_V(\phi) = \tfrac{1}{2}\,\mathbb{E}_t[(V_\phi(s_t) - \hat{R}_t)^2].$$

Many implementations (OpenAI Baselines, and the PPO-RLHF code that copied it) also **clip the value update**, mirroring the policy clip:

$$\mathcal{L}_V^{\text{clip}} = \tfrac{1}{2}\,\mathbb{E}_t\Big[\max\big((V_\phi(s_t) - \hat{R}_t)^2,\ (V_{\phi_{\text{old}}}(s_t) + \text{clip}(V_\phi(s_t) - V_{\phi_{\text{old}}}(s_t), -\epsilon_v, \epsilon_v) - \hat{R}_t)^2\big)\Big].$$

Engstrom et al. and Andrychowicz et al. (arXiv 2006.05990) found value clipping does not help and can hurt in control tasks; it is kept in many RLHF codebases mainly for compatibility. Note that $\epsilon_v$ is in the units of the reward, so the value 0.2 copied from the policy clip may be far too small or too large.

**Shared vs separate networks.** In control, actor and critic often share a trunk with two heads, and the losses are added with $c_v \approx 0.5$. For LLMs, the critic is usually a **separate** model (initialised from the reward model in InstructGPT), so the value loss cannot interfere with the policy's representations and each can have its own learning rate.

**Why this, not that? Why a separate critic model for LLMs, not a value head on the policy?**

**Why.** The value regression and the policy objective pull the shared weights in different directions; in LLMs a strong value-loss gradient can damage language ability. A separate critic also allows a different initialisation (from the reward model, which already understands quality) and learning rate.
**Trade-off.** Twice the trainable parameters and optimiser state, and an extra forward/backward per step: this cost is the main reason critic-free methods (GRPO, RLOO) became popular.
**When the alternative wins.** A shared value head is fine for small models and control tasks, and with memory limits; LoRA-based RLHF sometimes puts a value head on the same base with separate adapters.

## 17.13 PPO hyperparameters: control vs LLMs

**Notation on this page:** $K$: optimisation epochs per batch • $\phi$: critic weights • $V(s)$: state value (expected future return from $s$) • $V_\phi$: critic • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $s$: a state • $t$: time step or token position • $\hat{R}_t$: value target • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\mathcal{L}$: a loss to minimise • $\epsilon$: clip range for the probability ratio • *and 2 more: see the Notation table*

| Setting | Classic control (PPO paper / common defaults) | LLM RLHF / RLVR (typical) |
|---|---|---|
| Learning rate | $3 \times 10^{-4}$ (Adam) | $10^{-6}$ to $3 \times 10^{-6}$ policy; critic often higher |
| Clip $\epsilon$ | 0.2 | 0.2 (DAPO: $\epsilon_{\text{high}} = 0.28$) |
| Epochs $K$ | 10 | 1 to 4 |
| Discount $\gamma$ | 0.99 | 1.0 (single episode = one response) |
| GAE $\lambda$ | 0.95 | 0.95, or 1.0 (Open-Reasoner-Zero; VC-PPO for the critic) |
| Entropy coefficient | 0.0 to 0.01 | 0 |
| Value coefficient $c_v$ | 0.5 | separate critic loss |
| Gradient clip (norm) | 0.5 | 1.0 |
| KL to reference | none | $\beta$ 0.01 to 0.05 in RLHF; 0 to 0.001 in RLVR |

## 17.14 PPO as approximate natural-gradient or mirror descent

Seen from a distance, TRPO and PPO both implement **"maximise the surrogate, minus a penalty for distribution change"**. With a KL penalty, the update

$$\theta_{k+1} = \arg\max_{\theta} \; \mathbb{E}[\rho_t(\theta)\hat{A}_t] - \frac{1}{\eta}\,\mathbb{E}_s[\mathrm{KL}(\pi_{\theta_k} \| \pi_\theta)]$$

is a step of **mirror descent** in policy space, where the KL replaces the Euclidean distance of ordinary gradient descent. Its second-order approximation gives the natural gradient $F^{-1}g$ (TRPO). PPO-Clip replaces the penalty by a hard, per-sample, first-order limit. This view explains why the same "stay close in distribution" idea reappears in every LLM method: the KL to the reference in RLHF and DPO is the same kind of proximity term, just anchored to a fixed policy instead of the previous iterate.

## 17.15 Common mistakes

- Computing `old_logp` with gradients enabled, or recomputing it after the first update. It must be fixed when the rollout is collected.
- Using a different sampling distribution for rollouts (temperature, top-p) from the one used to compute log-probs in training. The ratio is then wrong from the start (Chapter 28).
- Expecting the clip to bound the KL. It does not; monitor the KL directly.
- Mixing up the two KLs: old-vs-new (step size) and reference-vs-policy (RLHF regulariser).
- Assuming the clip bounds the KL; monitor approximate KL and stop epochs early when needed.
- Copying value-clip $\epsilon_v = 0.2$ without considering the reward scale.

**Notation on this page:** $\epsilon$: clip range for the probability ratio • $K$: optimisation epochs per batch • $\gamma$: discount factor (0 to 1) • $\lambda$: GAE parameter (0 to 1), trades bias for variance • $c$: dual-clip floor; $c_v, c_H$: value and entropy coefficients • $\beta$: KL-penalty coefficient (PPO-Penalty) • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $\eta$: mirror-descent step size • *and 7 more: see the Notation table*

- Computing $\rho$ with log-probs from a different numerical path (e.g. vLLM vs trainer) and reading the mismatch as policy change (Chapter 28).

**Interview check**

**Q1. Write the PPO clipped objective.** $\mathbb{E}[\min(\rho\hat{A}, \text{clip}(\rho, 1-\epsilon, 1+\epsilon)\hat{A})]$ with $\rho = \pi_\theta / \pi_{\theta_{\text{old}}}$.
**Q2. When is the PPO gradient zero for a sample?** When $\hat{A} > 0$ and $\rho > 1+\epsilon$, or $\hat{A} < 0$ and $\rho < 1-\epsilon$.
**Q3. Why the min?** It makes the objective a pessimistic bound: the clip removes the incentive to move too far, but never hides a change that made things worse.
**Q4. Why is there an importance ratio at all?** Data comes from $\pi_{\theta_{\text{old}}}$, but we optimise $\pi_\theta$; the ratio corrects the action distribution so we can take several steps per batch.
**Q5. What is the performance difference lemma?** $J(\pi') - J(\pi) = \frac{1}{1-\gamma}\mathbb{E}_{s\sim d^{\pi'}, a\sim\pi'}[A^\pi(s,a)]$.
**Q6. How does TRPO enforce the trust region?** KL constraint; natural gradient via conjugate gradient with Fisher-vector products, then a backtracking line search.
**Q7. With one epoch and one mini-batch, what does PPO reduce to?** Vanilla policy gradient with advantages: the ratio is exactly 1 and the clip never activates.
**Q8. What is the clip fraction and what values are healthy?** Share of samples where the clip is active; about 0.05 to 0.2. Much higher means steps are too large.
**Q9. What is the adaptive KL rule from the PPO paper?** If KL $> 1.5\,d_{\text{targ}}$, double $\beta$; if KL $< d_{\text{targ}}/1.5$, halve $\beta$.
**Q10. Name five PPO implementation details that matter.** Advantage normalisation, gradient clipping, LR annealing, value loss handling, reward normalisation, orthogonal init, time-limit bootstrapping.
**Q11. Write the per-sample gradient of PPO-Clip.** $\rho_t \hat{A}_t \nabla \log \pi_\theta(a_t|s_t)$ when the sample is not clipped in the helpful direction, otherwise zero.
**Q12. Does PPO's clip bound the KL between old and new policies? Why not?** No: clipped samples lose their own gradient, but shared parameters still move due to other samples, so ratios and KL drift over epochs; use target-KL early stopping.
**Q13. What is the clip fraction and what values are healthy?** The share of samples where the clip is active; a few percent to about 20%. Higher means steps too big or data stale.
**Q14. What is value clipping and does it help?** Limits the critic's change per update like the policy clip; studies found it usually does not help.
**Q15. Why do LLM PPO setups use a separate critic?** To avoid value gradients harming the policy's language ability, and to initialise from the reward model; at the cost of double the trainable memory.
**Q16. How does PPO relate to mirror descent?** Maximising the surrogate minus a KL to the previous policy is mirror descent with KL geometry; its second-order form is the natural gradient (TRPO); PPO-Clip is a first-order, per-sample approximation.

**Key takeaways**

- Big policy steps cause collapse because the policy generates its own data.
- Performance difference lemma → surrogate $L_\pi(\pi')$ → trust region (TRPO, KL constraint).

**Notation on this page:** $\rho_t(\theta)$: probability ratio $\pi_\theta/\pi_{\theta_{\text{old}}}$ for the taken action • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\epsilon$: clip range for the probability ratio • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $\text{clip}(u,a,b)$: limit $u$ to the range $[a,b]$ • $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\gamma$: discount factor (0 to 1) • $J(\theta)$: objective to maximise (expected reward) • $s$: a state • *and 6 more: see the Notation table*

- PPO-Clip: $\min(\rho\hat{A}, \text{clip}(\rho)\hat{A})$; zero gradient once a sample has moved $\epsilon$ in the advantage's direction.
- The clip does not bound KL; monitor KL and clip fraction.
- Details (normalisation, clipping, schedules) matter as much as the objective.
- Per-sample gradient: on (weighted by $\rho$) or off; no soft down-weighting.
- The clip does not bound KL; use target-KL early stopping and watch the clip fraction.
- LLM PPO: low LR, few epochs, $\gamma = 1$, separate critic, KL to reference.

**Notation on this page:** $\rho_t(\theta)$: probability ratio $\pi_\theta / \pi_{\theta_{\text{old}}}$ for the taken action • $A, \hat{A}$: advantage (how much better an action is than average); hat = estimate • $\text{clip}(u, a, b)$: limit $u$ to the range $[a, b]$ • $\epsilon$: clip range for the probability ratio • $\gamma$: discount factor (0 to 1)

CHAPTER 18

# Off-Policy, Offline and Other RL Ideas

**Why this chapter matters.**
Modern LLM RL systems are rarely perfectly on-policy: rollouts are generated asynchronously, by a different inference engine, or reused across several updates. Preference optimisation (DPO) is a form of offline RL. Best-of-$n$ sampling is a bandit idea. This short chapter collects the RL ideas outside the PPO family that come up in LLM interviews.

## 18.1 Off-policy correction with importance sampling

Data collected by a **behaviour policy** $\mu$ can be used to estimate expectations under a **target policy** $\pi$ by weighting each sample with the ratio $\pi/\mu$ (Chapter 2):

$$\mathbb{E}_{a\sim\pi}[f(a)] = \mathbb{E}_{a\sim\mu}\Big[\frac{\pi(a)}{\mu(a)}f(a)\Big].$$

For a whole trajectory, the weight is the product of per-step ratios, $\prod_t \pi(a_t \mid s_t)/\mu(a_t \mid s_t)$. Products of many ratios have **exploding variance**: a few trajectories get enormous weights and the rest almost none. For an LLM response of 1,000 tokens, even small per-token differences multiply into extreme sequence weights.

Practical fixes all **truncate** or **reshape** the weights:

- **Per-token ratios** instead of the product (as in PPO): biased, but stable.
- **Clipping** or **truncating** the weight at a maximum (truncated importance sampling, TIS).
- **Geometric mean** over the sequence (GSPO's length-normalised sequence ratio, Chapter 28).
- **Masking** samples whose weight is out of range.

## 18.2 V-trace

*IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures* (Espeholt et al., arXiv 1802.01561) separates many **actors** that generate experience from a central **learner**. By the time data reaches the learner, the policy has moved on, so the data is slightly off-policy. **V-trace** corrects the value targets with truncated ratios $c_t = \min(\bar{c}, \pi/\mu)$ and $\rho_t = \min(\bar{\rho}, \pi/\mu)$:

$$v_s = V(s_s) + \sum_{t\geq s} \gamma^{t-s}\Big(\prod_{i=s}^{t-1} c_i\Big)\rho_t\,(r_t + \gamma V(s_{t+1}) - V(s_t)).$$

**Notation on this page:** $n$: number of samples (best-of-$n$); $n_a$: times arm $a$ was tried • $\mu$: behaviour policy (generated the data) • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $a$: an action • $f$: any function whose average we want • $s$: a state • $t$: time step or token position • $c_t = \min(\bar{c}, \pi/\mu)$: truncated trace weight • $\rho_t, c_t$: truncated importance ratios in V-trace • $\gamma$: discount factor (0 to 1) • *and 4 more: see the Notation table*

With $\bar{\rho} = \bar{c} = 1$ (the usual choice) the weights never exceed 1. The same architecture (separate generation and training, slightly stale data, truncated corrections) is now standard for **asynchronous LLM RL** (Chapters 28 and 44).

**Why this, not that? Why asynchronous, slightly off-policy training, not strictly on-policy?**

**Why.** In LLM RL, generation dominates the time. If the trainer waits for every rollout and the generators wait for every update, GPUs sit idle. Letting generation run one or more steps ahead keeps all hardware busy, often doubling throughput; long-tail responses (a few very long chains of thought) no longer block everyone.
**Trade-off.** The data is stale, so you need importance corrections, clipping or masking, and staleness limits; too much staleness destabilises training.
**When the alternative wins.** Small-scale experiments, debugging, and research where exact on-policy behaviour matters for clean comparisons. Synchronous training is simpler and is the right default until generation idle time becomes the bottleneck.

## 18.3 SAC and DDPG, briefly

For continuous actions (robotics), two off-policy actor-critic methods are standard:

- **DDPG** learns a deterministic policy $\mu(s)$ and a Q-function, updating the policy along $\nabla_a Q(s, a)$.
- **SAC** (*Soft Actor-Critic*, Haarnoja et al., arXiv 1801.01290) maximises reward plus an **entropy bonus**, $\mathbb{E}[\sum_t r_t + \alpha H(\pi(\cdot \mid s_t))]$, with a replay buffer. The entropy term keeps exploration alive and makes training robust. The "maximum entropy RL" view also explains the KL-regularised RLHF objective: a KL penalty to a reference model is an entropy bonus relative to that reference.

Neither is used directly for LLMs (actions are discrete tokens and $\nabla_a Q$ makes no sense for them), but interviewers like to check that you know what they are.

## 18.4 Offline RL

**Offline RL** learns from a fixed dataset with no new interaction. The main danger is **distribution shift**: the learned policy chooses actions that are rare in the data, where value estimates are unreliable and usually too optimistic. Offline methods therefore keep the policy close to the data (behaviour-regularised methods) or are pessimistic about unseen actions (conservative Q-learning).

**Preference optimisation is offline RL.** DPO (Chapter 22) trains on a fixed dataset of preferred and rejected responses, with an implicit KL constraint to a reference model, which plays the role of behaviour regularisation. Its known failure modes (rewarding responses far from the data, likelihood of both chosen and rejected dropping) are offline-RL problems. Online and iterative DPO (Chapter 24) add fresh samples to reduce distribution shift.

**Notation on this page:** $\rho_t, c_t$: truncated importance ratios in V-trace • $c_t = \min(\bar{c}, \pi/\mu)$: truncated trace weight • $\mu$: behaviour policy (generated the data) • $s$: a state • $\nabla_\theta$: gradient with respect to $\theta$ • $Q(s, a)$: action value (expected return after taking $a$ in $s$) • $a$: an action • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\alpha$: entropy weight (SAC) • $H$: entropy (uncertainty) of a distribution • *and 2 more: see the Notation table*

**Why this, not that? Why online RL, not offline learning from a fixed preference dataset?**

**Why.** Online methods score the policy's *own* current outputs, so they correct the mistakes the policy actually makes and do not suffer distribution shift. Several comparisons (for example *Is DPO Superior to PPO for LLM Alignment?*, Xu et al., arXiv 2404.10719) found well-tuned online PPO beating offline DPO.
**Trade-off.** Online RL needs a reward function or reward model to score fresh samples, a generation engine, and far more compute and engineering.
**When the alternative wins.** Offline methods win when you have good preference data but no reliable reward model, limited compute, or need a simple, stable pipeline; they are also a good first stage before online RL.

## 18.5 Bandits and best-of-$n$

A **multi-armed bandit** is an MDP with one state: pick an arm, get a reward, repeat. The challenge is pure exploration vs exploitation, measured by **regret**: the total reward lost compared with always picking the best arm. **Upper confidence bound** (UCB) methods pick the arm with the highest optimistic estimate, mean $+c\sqrt{\log t / n_a}$, where $n_a$ counts how often arm $a$ was tried; **Thompson sampling** samples a plausible value for each arm from its posterior and picks the best.

LLM generation for one prompt is a **contextual bandit** (Chapter 13). The simplest way to "optimise" it is **best-of-$n$** (rejection sampling): sample $n$ responses and return the one with the highest reward. If rewards are continuous and the best-of-$n$ policy is compared with the base policy, its KL divergence is known in closed form under mild assumptions: $\mathrm{KL} = \log n - (n-1)/n$. So best-of-16 moves about $\log 16 - 15/16 \approx 1.84$ nats from the base model. Best-of-$n$ is a strong baseline that RL methods must beat, and it is also used to make SFT data (rejection-sampling fine-tuning, Chapter 8).

```
import math, torch
torch.manual_seed(0)
# Best-of-n on a toy "reward = quality score" distribution: expected best reward grows
 ↪  like sqrt(log n).
for n in (1, 4, 16, 64):
    best = torch.randn(20000, n).max(dim=1).values.mean()
    print(f"n={n:3d}: mean best reward {best:.2f}   KL from base = {math.log(n) - (n -
     ↪  1) / n:.2f} nats")
```

The reward gain grows slowly (roughly like $\sqrt{2\log n}$ for normally distributed rewards) while the cost grows linearly in $n$, which is why we train the policy instead of sampling more at inference.

## 18.6 How bad does importance sampling get? A measurement

Importance sampling (IS) is unbiased in theory, but its variance depends on how far apart the two policies are. A clean way to see it: estimate the expected reward of a target policy $\pi$ from samples of a behaviour policy $\mu$, with three estimators.

**Notation on this page:** $a$: an action • $t$: time step or token position • $n$: number of samples (best-of-$n$); $n_a$: times arm $a$ was tried • $c_t = \min(\bar{c}, \pi/\mu)$: truncated trace weight • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $\mu$: behaviour policy (generated the data)

- **Ordinary IS:** $\hat{J} = \frac{1}{N}\sum_i w_i r_i$ with $w_i = \pi(a_i)/\mu(a_i)$. Unbiased.
- **Self-normalised (weighted) IS:** $\hat{J} = \sum_i w_i r_i / \sum_i w_i$. Slightly biased for finite $N$ (the bias vanishes as $N$ grows), often far lower variance.
- **Truncated IS:** $\hat{J} = \frac{1}{N}\sum_i \min(w_i, C)\, r_i$. Biased (it under-weights samples the target likes much more), bounded variance.

The variance of ordinary IS is governed by $\mathbb{E}_\mu[w^2] = \sum_a \pi(a)^2/\mu(a) = \exp(D_2(\pi\|\mu))$, where $D_2$ is the Rényi divergence of order 2. It can be huge even when the KL divergence looks modest, because it is dominated by actions that $\pi$ likes and $\mu$ rarely samples. A useful single number is the **effective sample size**,

$$\mathrm{ESS} = \frac{\left(\sum_i w_i\right)^2}{\sum_i w_i^2},$$

which is $N$ when all weights are equal and approaches 1 when one sample dominates.

```
import torch
torch.manual_seed(0)
K, N = 50, 256
reward = torch.rand(K)
mu = torch.softmax(torch.randn(K), 0)                          # behaviour policy
def estimates(shift, trials=400, C=2.0):
    pi = torch.softmax(mu.log() + shift * reward * 4, 0)       # target leans toward high
 ↪  reward
    true = (pi * reward).sum().item()
    kl = (pi * (pi.log() - mu.log())).sum().item()
    res = {"ois": [], "wis": [], "tis": [], "ess": []}
    for _ in range(trials):
        a = torch.multinomial(mu, N, replacement=True)
        w = pi[a] / mu[a]; r = reward[a]
        res["ois"].append((w * r).mean()); res["wis"].append((w * r).sum() / w.sum())
        res["tis"].append((w.clamp(max=C) * r).mean()); res["ess"].append(w.sum() ** 2 /
 ↪  (w ** 2).sum())
    out = {k: torch.stack(v) for k, v in res.items()}
    return true, kl, out
print(" shift   KL   true |  ordinary IS bias/std | self-norm bias/std | truncated
 ↪  bias/std | ESS of 256")
for shift in (0.25, 0.5, 1.0, 2.0):
    true, kl, o = estimates(shift)
    row = " ".join(f"{(o[k].mean() - true).item():+.3f}/{o[k].std().item():.3f}" for k
 ↪  in ("ois", "wis", "tis"))
    print(f"{shift:5.2f} {kl:5.2f} {true:.3f} | {row} | {o['ess'].mean().item():6.1f}")
```

As the target moves away from the behaviour policy, the ordinary estimator's standard deviation grows, the self-normalised one stays much smaller, and the truncated one becomes clearly biased. The effective sample size falls: a batch of 256 samples behaves like a much smaller batch. In LLM RL this is why PPO clips the ratio, why asynchronous systems bound staleness (Chapter 44), and why off-policy corrections are applied **per token** with truncation rather than as a product over a whole sequence (a product of thousands of ratios has astronomically large variance).

**Notation on this page:** $r(x, y)$: the reward for response $y$ to prompt $x$ • $N$: number of examples • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mu$: behaviour policy (generated the data) • $a$: an action • $\mathbb{E}$: expected value (average over the distribution in the subscript)

**Why this, not that? Why per-token truncated ratios, not one importance weight for the whole sequence?**

**Reason.** The sequence-level weight is a product of per-token ratios; its log is a sum of thousands of small terms, so its variance grows with length and a few sequences get almost all the weight. Per-token ratios (as in PPO, and in TIS corrections) keep the variance bounded. **Trade-off.** Per-token ratios with truncation are biased: they ignore how earlier tokens changed the distribution of later states. **When the alternative is better.** Short sequences or nearly identical policies (the mismatch is tiny), or sequence-level methods designed to control the variance, such as GSPO's length-normalised sequence ratio (Chapter 28).

## 18.7 Offline RL for language models: weighting, filtering and DPO

**Offline RL** learns from a fixed dataset without new interaction. For LLMs the most useful offline methods are simple, because they are all derived from the same KL-regularised optimum (Chapter 22):

$$\pi^*(y \mid x) = \frac{1}{Z(x)}\, \pi_{\text{ref}}(y \mid x)\, \exp(r(x,y)/\beta).$$

If the data was sampled from $\pi_{\text{ref}}$, then fitting $\pi^*$ by maximum likelihood means **weighting each sample by** $\exp(r/\beta)$. That is **reward-weighted regression** (or **advantage-weighted regression**, AWR, when $r$ is replaced by an advantage):

$$\mathcal{L}_{\text{AWR}}(\theta) = -\,\mathbb{E}_{(x,y)\sim\mathcal{D}}\Big[\, \exp(A(x,y)/\beta)\, \log \pi_\theta(y \mid x)\Big].$$

Three methods in this book are points on this line:

- **Filtered SFT / rejection-sampling fine-tuning** (RAFT, ReST, STaR; Chapters 8, 26) uses weight 1 for samples above a threshold and 0 otherwise: the $\beta \to 0$ limit with a hard cut.
- **AWR** uses smooth weights $\exp(A/\beta)$: small $\beta$ trusts the reward more and concentrates on the best samples; large $\beta$ stays close to plain SFT.
- **DPO** uses pairs instead of absolute rewards and solves the same objective without ever estimating $Z(x)$.

**Worked example.** Four sampled answers to one prompt with rewards $1, 1, 0, 0$ and advantages $+0.5, +0.5, -0.5, -0.5$. With $\beta = 0.5$ the weights are $e^1 = 2.72$ for the correct answers and $e^{-1} = 0.37$ for the wrong ones: the correct answers get 7.4 times the weight, but the wrong ones are still imitated a little. Filtered SFT gives weights 1 and 0: wrong answers are ignored, but the model never learns that they are wrong (weights are never negative). Only on-policy methods with negative advantages (or DPO's rejected term) actively push probability away from errors.

```
import torch
torch.manual_seed(1)
K = 6
ref = torch.softmax(torch.randn(K), 0); rwd = torch.tensor([0.0, 0.2, 1.0, 0.1, 0.8,
 ↪  0.3]); beta = 0.25
```

**Notation on this page:** $\pi_{\text{ref}}$: the frozen reference model • $\pi^*$: the optimal policy • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $r(x,y)$: the reward for response $y$ to prompt $x$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\mathcal{L}$: a loss to minimise • $\mathcal{D}$: a dataset

```
target = ref * torch.exp(rwd / beta); target /= target.sum()          # closed-form
↪ KL-regularised optimum
y = torch.multinomial(ref, 20000, replacement=True)                   # offline data from
↪ the reference
w = torch.exp(rwd[y] / beta)
awr = torch.zeros(K).index_add_(0, y, w); awr /= awr.sum()            # weighted maximum
↪ likelihood
filt = torch.zeros(K).index_add_(0, y, (rwd[y] >= 0.8).float()); filt /= filt.sum()
print("optimum   ", [round(v, 3) for v in target.tolist()])
print("AWR fit   ", [round(v, 3) for v in awr.tolist()])
print("filtered  ", [round(v, 3) for v in filt.tolist()], "(keeps only reward >= 0.8)")
print("expected reward: ref %.3f | optimum %.3f | AWR %.3f | filtered %.3f" % tuple(
    (p * rwd).sum().item() for p in (ref, target, awr, filt)))
```

The weighted fit recovers the KL-regularised optimum from offline samples. Filtering reaches a higher expected reward here but moves much further from the reference (it drops every answer below the threshold), which is the offline analogue of a very small $\beta$.

**Why this, not that? Why weighted or filtered SFT, not online RL, as a first step?**

**Reason.** It is just supervised learning: stable, cheap, easy to debug, and it reuses the SFT pipeline. Rejection-sampling rounds (sample, filter, SFT, repeat) capture much of the gain of RL on verifiable tasks. **Trade-off.** It cannot push probability away from errors (no negative weights), it needs the good behaviour to already appear in samples, and it suffers from distribution shift between rounds. **When the alternative is better.** When the model must learn from its failures, when good samples are rare, or when long-horizon credit assignment matters: online RL with negative advantages (GRPO, PPO) or preference methods that use rejected samples.

## 18.8 Where off-policy data comes from in LLM RL

Even "on-policy" LLM RL is slightly off-policy in four ways. Knowing them lets you name the right correction.

1. **Several updates per batch.** PPO and GRPO with `num_iterations` $> 1$ (or several mini-batches per generation batch) update the policy while still using samples from the old one. The ratio $\pi_\theta / \pi_{\theta_{\text{old}}}$ and clipping handle this (Chapter 17).
2. **Asynchronous generation.** Rollout workers use weights one or more versions old (Chapter 44). Corrections: a staleness limit, truncated importance weights, or decoupled PPO objectives that use the behaviour policy for the ratio and a recent policy as the trust-region anchor.
3. **Engine–trainer mismatch.** The inference engine computes probabilities with different kernels and precision, so even "the same" weights define a slightly different behaviour policy (Chapter 43). Corrections: truncated importance sampling with engine log-probabilities, or masking tokens whose ratio is extreme.
4. **Replay and data reuse.** Re-using good samples from earlier steps (replay buffers for rare successes, or mixing in teacher solutions) is strongly off-policy; the ratio to the current

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights

policy can be tiny. Corrections: treat them as an SFT-style auxiliary loss rather than as policy-gradient samples, or apply clipped importance weights.

**Worked example.** A one-step-off asynchronous run with 8 mini-batch updates per generation batch: the last mini-batch is trained on samples that are effectively 1 version (staleness) plus 7 updates (PPO epochs) away from the current policy. If the per-update KL is about $10^{-3}$ nats per token, the accumulated divergence is still small (about $8 \times 10^{-3}$ per token), but over a 4,000-token response the *sequence-level* log-ratio has a standard deviation that can reach order one. This is another reason ratios are applied per token.

## 18.9 Common mistakes

- **Multiplying per-token ratios into a sequence weight** without truncation or normalisation: the variance explodes with length.
- **Reporting an off-policy estimate without its effective sample size.** A small ESS means the estimate rests on a handful of samples.
- **Calling filtered SFT "RL" without saying what it cannot do**: it never sees negative examples.

**Interview check**

**Q1. Why do trajectory-level importance weights explode?** They are products of many per-step ratios; their variance grows exponentially with length.
**Q2. What does V-trace do?** Corrects off-policy value targets with truncated importance ratios $\min(\bar{c}, \pi/\mu)$, keeping variance bounded.
**Q3. Why is DPO a form of offline RL?** It learns from a fixed dataset with a KL constraint to a reference policy, and it suffers offline RL's distribution-shift problems.
**Q4. What is the KL of best-of-$n$ from the base policy?** $\log n - (n-1)/n$.
**Q5. Why does SAC add entropy?** To keep exploration and robustness; it maximises reward plus entropy (maximum entropy RL).
**Q6. Why can't DDPG be used for token generation?** It needs gradients of $Q$ with respect to continuous actions; tokens are discrete.
**Q7. What controls the variance of ordinary importance sampling?** $\mathbb{E}_\mu[w^2] = \sum_a \pi(a)^2/\mu(a)$, the exponential of the order-2 Rényi divergence; it is dominated by actions the target likes and the behaviour policy rarely samples.
**Q8. What is the effective sample size of a weighted sample?** $(\sum w_i)^2 / \sum w_i^2$: $N$ for equal weights, close to 1 when one sample dominates.
**Q9. How is advantage-weighted regression related to the KL-regularised optimum?** If data comes from the reference policy, weighting log-likelihood by $\exp(A/\beta)$ fits $\pi^* \propto \pi_{\text{ref}} e^{r/\beta}$ by maximum likelihood.
**Q10. What is the key limitation of rejection-sampling fine-tuning?** It only imitates good samples; it never pushes probability away from bad ones, and it needs good samples to appear in the first place.
**Q11. Name four sources of off-policy data in "on-policy" LLM RL.** Multiple updates per batch, asynchronous (stale) generation, inference-engine numerical mismatch, and replayed

**Notation on this page:** $\mu$: behaviour policy (generated the data) • $c_t = \min(\bar{c}, \pi/\mu)$: truncated trace weight • $n$: number of samples (best-of-$n$); $n_a$: times arm $a$ was tried • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $a$: an action • $N$: number of examples • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\pi_{\text{ref}}$: the frozen reference model • $\pi^*$: the optimal policy • $r(x, y)$: the reward for response $y$ to prompt $x$

or external samples.

**Key takeaways**

- Importance sampling corrects off-policy data but products of ratios explode; truncate, clip, normalise or mask.
- V-trace and IMPALA's actor-learner split are the ancestors of asynchronous LLM RL.
- SAC and DDPG are continuous-control methods; SAC's entropy bonus relates to KL-regularised RLHF.
- DPO is offline RL; online methods avoid distribution shift at higher cost.
- Best-of-$n$ is a contextual-bandit baseline with KL $\log n - (n-1)/n$.
- IS variance grows with $\sum \pi^2/\mu$; self-normalisation and truncation trade small bias for large variance cuts; watch the effective sample size.
- Filtered SFT, AWR and DPO are offline routes to the same KL-regularised optimum; only methods with negative signals push away from errors.

**Notation on this page:** $n$: number of samples (best-of-$n$); $n_a$: times arm $a$ was tried • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mu$: behaviour policy (generated the data)

PART IV

# Reinforcement Learning from Human Feedback

CHAPTER 19

# Reward Models

**Why this chapter matters.**
Human preferences cannot be computed by a rule, so RLHF first trains a **reward model** (RM): a network that reads a prompt and a response and outputs a score that predicts how much humans would like it. Every RLHF policy is only as good as its reward model, and every reward model can be exploited. Reward models are also used for best-of-$n$ sampling, data filtering and evaluation. This chapter derives the pairwise loss, shows how RMs are built and trained, and covers generative judges and RM evaluation.

## 19.1 Collecting preference data

People are bad at giving consistent absolute scores ("rate this answer from 1 to 10") but good at **comparisons** ("which of these two is better?"). So preference data is usually collected as pairs:

1. Take a prompt $x$.
2. Sample two (or more) responses, usually from the current model or several models.
3. A labeller picks the better one, giving a **chosen** response $y_w$ (winner) and a **rejected** one $y_l$ (loser). Often they also give a strength ("much better", "slightly better") or mark a tie.

Labelling guidelines matter as much as the data volume: labellers need clear definitions of helpfulness, honesty and harmlessness, and how to trade them off. Agreement between labellers is often only 60 to 75%, which caps how accurate any RM can be.

InstructGPT (*Training language models to follow instructions with human feedback*, Ouyang et al., arXiv 2203.02155) asked labellers to *rank* between 4 and 9 responses per prompt, giving $\binom{K}{2}$ comparisons from each ranking. Because comparisons from one ranking are highly correlated, shuffling them into the dataset made the RM overfit in a single pass; they instead put all comparisons from one prompt into one batch element.

## 19.2 The Bradley-Terry model

The **Bradley-Terry** model (Chapter 2) assumes each response has a hidden score $r(x, y)$, and the probability that $y_w$ is preferred to $y_l$ is

$$P(y_w \succ y_l \mid x) = \frac{\exp r(x, y_w)}{\exp r(x, y_w) + \exp r(x, y_l)} = \sigma(r(x, y_w) - r(x, y_l)),$$

where $\sigma(z) = 1/(1 + e^{-z})$. Only the *difference* in scores matters: adding a constant to all rewards changes nothing.

**Notation on this page:** $n$: number of samples • $x$: the prompt (input) • $y_w, y_l$: chosen (winning) and rejected (losing) responses • $y$: the response (output tokens); $y_t$ is token $t$ • $K$: number of ranked responses • $r(x, y)$, $r_w$, $r_l$: reward-model scores (chosen, rejected) • $\sigma(z)$: the logistic (sigmoid) function $1/(1 + e^{-z})$ • $a \succ b$: $a$ is preferred to $b$ • $z$: logits (raw scores before softmax)

**The pairwise loss.** Fit the RM $r_\phi$ by maximum likelihood over a dataset $\mathcal{D}$ of comparisons:

$$\mathcal{L}_{\text{RM}}(\phi) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\Big[\log \sigma(r_\phi(x,y_w) - r_\phi(x,y_l))\Big].$$

This is binary cross-entropy where the "logit" is the reward difference.

**Gradient.** With $\Delta = r_\phi(x,y_w) - r_\phi(x,y_l)$, and using $\frac{d}{d\Delta}\log\sigma(\Delta) = 1-\sigma(\Delta) = \sigma(-\Delta)$:

$$\nabla_\phi \mathcal{L}_{\text{RM}} = -\sigma(-\Delta)\,(\nabla_\phi r_\phi(x,y_w) - \nabla_\phi r_\phi(x,y_l)).$$

*Intuition.* Push the chosen score up and the rejected score down, weighted by $\sigma(-\Delta)$, the model's current probability of getting the pair *wrong*. Pairs it already ranks correctly with a big margin contribute almost nothing.

*Worked example.* Scores $r_w = 1.2$, $r_l = 0.4$: $\Delta = 0.8$, $P(\text{correct}) = \sigma(0.8) = 0.69$, loss $= -\log 0.69 = 0.37$, gradient weight $\sigma(-0.8) = 0.31$.

## 19.3 Architecture: a value head on a language model

A reward model is usually a pretrained or SFT language model with its vocabulary output layer replaced by a **scalar head**: a linear layer from the hidden size to one number. The prompt and response are concatenated, run through the transformer, and the head is applied to the hidden state of the **last token** (the end-of-sequence token). That number is $r_\phi(x,y)$. In Hugging Face this is `AutoModelForSequenceClassification` with `num_labels=1`, which is what TRL's `RewardTrainer` loads.

```
import torch, torch.nn as nn, torch.nn.functional as F

class TinyRM(nn.Module):
    """Stand-in for an LLM backbone + scalar head; reads the last real token's hidden
     ↪ state."""
    def __init__(self, vocab=50, d=32):
        super().__init__()
        self.emb = nn.Embedding(vocab, d)
        self.rnn = nn.GRU(d, d, batch_first=True)       # pretend transformer
        self.head = nn.Linear(d, 1)
    def forward(self, ids, lengths):
        h, _ = self.rnn(self.emb(ids))
        last = h[torch.arange(ids.size(0)), lengths - 1]  # hidden state of last real
 ↪ token
        return self.head(last).squeeze(-1)

def rm_loss(r_chosen, r_rejected, margin=None):
    diff = r_chosen - r_rejected
    if margin is not None:
        diff = diff - margin
    return -F.logsigmoid(diff).mean()

# Toy preference: responses containing token 7 are preferred.
torch.manual_seed(0)
```

**Notation on this page:** $\phi$: reward-model weights; $r_\phi$ the reward model • $r(x,y)$, $r_w$, $r_l$: reward-model scores (chosen, rejected) • $\mathcal{D}$: a dataset • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $\mathcal{L}$: a loss to minimise • $y_w$, $y_l$: chosen (winning) and rejected (losing) responses • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $d$: hidden size of the model • $\nabla_\theta$: gradient with respect to $\theta$

```
def batch(n=64, T=12):
    chosen = torch.randint(10, 50, (n, T)); chosen[:, 3] = 7
    rejected = torch.randint(10, 50, (n, T))
    lens = torch.randint(6, T + 1, (n,))
    return chosen, rejected, lens
rm = TinyRM(); opt = torch.optim.Adam(rm.parameters(), lr=3e-3)
for step in range(200):
    c, r, lens = batch()
    loss = rm_loss(rm(c, lens), rm(r, lens))
    opt.zero_grad(); loss.backward(); opt.step()
c, r, lens = batch(1000)
with torch.no_grad():
    acc = (rm(c, lens) > rm(r, lens)).float().mean()
print(f"final loss {loss.item():.3f}, held-out pairwise accuracy {acc:.3f}")
```

**Why this, not that? Why pairwise comparisons, not absolute ratings?**

**Why.** Humans agree much more on "A is better than B" than on "A is a 7 out of 10". Comparisons remove each labeller's personal scale and give cleaner signal per label.
**Trade-off.** Each comparison carries at most one bit of information, rewards are only defined up to a constant, and a ranking says nothing about whether *either* response is actually good.
**When the alternative wins.** Absolute ratings (or rubric scores, or binary "acceptable / not acceptable" labels as used by KTO) are better when you need calibrated scores across prompts, when only one response per prompt is available, or when labels come from a checkable criterion rather than taste.

## 19.4 Training details

- **Initialise from the SFT model.** The RM should understand the kind of text it scores. InstructGPT used a 6B RM initialised from a 6B SFT model, trained for a **single epoch** (it overfits after one) at learning rate $9 \times 10^{-6}$ with a cosine schedule.
- **Normalise.** Since rewards are shift-invariant, InstructGPT added a bias so that labeller demonstrations had a mean score of 0 before RL.
- **Margins.** If labellers give a preference strength, subtract a margin $m(y_w, y_l) \geq 0$ inside the sigmoid: $-\log \sigma(r_w - r_l - m)$. This forces a bigger score gap for clearer preferences. *Llama 2* (arXiv 2307.09288) used a margin from a four-level strength scale. In TRL, a `"margin"` column in the dataset turns this on.
- **Centring.** An auxiliary loss $c\,\mathbb{E}[(r_w + r_l)^2]$ keeps rewards centred around zero, from *Helping or Herding? Reward Model Ensembles Mitigate but do not Eliminate Reward Hacking* (Eisenstein et al., arXiv 2312.09244). TRL: `RewardConfig(center_rewards_coefficient=...)`, default `None`, recommended 0.01.
- **Ties.** Either drop them, or treat a tie as a target probability of 0.5.
- **Learning rate.** TRL's `RewardConfig` defaults: `learning_rate=1e-4` (intended for LoRA-style setups; full fine-tuning of large RMs typically uses around $10^{-6}$ to $10^{-5}$), `max_length=1024`.

**Notation on this page:** $y_w, y_l$: chosen (winning) and rejected (losing) responses • $y$: the response (output tokens); $y_t$ is token $t$ • $m$: preference margin (optional) • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $r(x, y)$, $r_w$, $r_l$: reward-model scores (chosen, rejected) • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $c$: centring-loss coefficient

## 19.5 Ensembles and uncertainty

A single RM has blind spots that a policy can find and exploit. **Ensembles** (several RMs trained with different seeds or data) reduce this: use the mean, the minimum, or mean minus standard deviation as the reward. Eisenstein et al. (arXiv 2312.09244) found ensembles reduce reward hacking, especially when members differ in *pretraining* seeds, but do not eliminate it, since all members can share the same errors.

## 19.6 Generative reward models and LLM judges

A **generative reward model** uses the language model's text output instead of a scalar head:

- **LLM-as-a-judge.** Prompt a strong model to compare two answers or score one against a rubric. *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena* (Zheng et al., arXiv 2306.05685) found strong judges agree with humans about as often as humans agree with each other, but also documented **position bias** (preferring the first answer), **verbosity bias** (preferring longer answers) and **self-enhancement bias** (preferring their own outputs).
- **Generative verifiers.** *Generative Verifiers: Reward Modeling as Next-Token Prediction* (Zhang et al., arXiv 2408.15240) trains the verifier to write a chain-of-thought critique and then answer "Yes" or "No"; the probability of "Yes" is the score. This lets the RM "think" before judging, and use more compute for harder judgements by sampling several critiques.

**Why this, not that? Why a scalar (Bradley-Terry) reward model, not an LLM judge?**

**Why.** A scalar RM is one forward pass per response, so it is fast enough to score millions of RL rollouts; it is trained directly on your preference data; and its scores are smooth, which suits RL.
**Trade-off.** It cannot explain its scores, cannot reason through hard cases, and is easier to hack because it learns shallow features (length, confident tone).
**When the alternative wins.** Generative judges win when you have little preference data, when judgements need reasoning (maths proofs, code review, rubric-based grading), when you want explanations, or for evaluation rather than high-volume training. Many recent pipelines use rubric-based LLM judges as RL rewards for open-ended tasks.

## 19.7 Evaluating reward models

- **Pairwise accuracy** on held-out preference data: the share of pairs where $r(y_w) > r(y_l)$. Typical values on human data are 65 to 75%, limited by labeller agreement.
- **Benchmarks.** *RewardBench* (Lambert et al., arXiv 2403.13787) measures accuracy on chosen/rejected pairs across chat, hard chat, safety and reasoning categories. *RewardBench 2* (arXiv 2506.01937) uses harder, mostly best-of-4 formats and reports correlation with downstream use.
- **Downstream checks.** The real test is what happens when you optimise against it: best-of-$n$ win rates as $n$ grows, or RL with a held-out "gold" judge. A small accuracy gain can matter less than robustness to exploitation.

**Notation on this page:** $y_w$, $y_l$: chosen (winning) and rejected (losing) responses • $y$: the response (output tokens); $y_t$ is token $t$ • $r(x, y)$, $r_w$, $r_l$: reward-model scores (chosen, rejected) • $n$: number of samples

- **Bias probes.** Check correlation with length, with formatting (lists, bold text), and with agreement with the user (sycophancy).

## 19.8 Where Bradley-Terry comes from: random utilities

Why a *sigmoid* of the score difference? Suppose each time a labeller judges a response, they perceive its quality as the true score plus random noise: $U_w = r_w + \varepsilon_w$ and $U_l = r_l + \varepsilon_l$, and they pick whichever $U$ is larger. Then

$$P(y_w \succ y_l) = P(\varepsilon_l - \varepsilon_w < r_w - r_l) = F_{\varepsilon_l - \varepsilon_w}(r_w - r_l),$$

the cumulative distribution of the noise difference. Two classic choices:

- **Gumbel noise** (independent, same scale): the difference of two Gumbel variables follows a **logistic** distribution, whose CDF is exactly $\sigma(\cdot)$. This gives **Bradley-Terry**.
- **Gaussian noise**: the difference is Gaussian, so $P = \Phi((r_w - r_l)/(\sqrt{2}s))$, the **Thurstone** model (a probit instead of a logit).

The two curves are almost the same shape after rescaling; BT is used because its log-likelihood is simple, convex in the score difference, and has the clean gradient above. The same derivation explains the **scale** of rewards: multiplying all rewards by $c$ is the same as dividing the noise by $c$, so the reward scale encodes *how consistent* the labellers are. This is why rewards are defined up to a shift, but their scale is meaningful relative to labelling noise.

## 19.9 The ceiling set by label noise

If labellers agree with the majority view only 70% of the time on a pair, then even a perfect reward model, which predicts $P(y_w \succ y_l) = 0.7$, gets that pair "right" (as measured by the observed label) only 70% of the time, and its expected loss is the binary entropy

$$H(0.7) = -0.7 \log 0.7 - 0.3 \log 0.3 \approx 0.611.$$

So a validation loss of 0.6 and accuracy of 70% may already be close to the best achievable on that data, while a training loss far below this means **memorisation**. When someone quotes an RM accuracy, ask what the inter-annotator agreement is.

## 19.10 Shortcut features: the length example

A reward model learns whatever features predict the labels. If longer answers are more often preferred in the data (because good answers tend to include more detail), a model that cannot read quality well will learn to reward **length itself**, which a policy can then exploit by padding. A simulated example with a linear Bradley-Terry RM:

```
import torch, torch.nn.functional as F
torch.manual_seed(0)
```

**Notation on this page:** $\varepsilon$: labeller noise in the random-utility model • $r(x, y)$, $r_w$, $r_l$: reward-model scores (chosen, rejected) • $U$: perceived utility (score + noise) • $a \succ b$: $a$ is preferred to $b$ • $y_w$, $y_l$: chosen (winning) and rejected (losing) responses • $y$: the response (output tokens); $y_t$ is token $t$ • $F$: CDF of the noise difference • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $s$: noise scale (Thurstone) • $c$: centring-loss coefficient • $H$: entropy (uncertainty) of a distribution • *and 1 more: see the Notation table*

```
# Each response has a true quality q and a length L (in hundreds of tokens).
# Labellers prefer higher quality, but in this data better answers also tend to be
 ↪ longer.
n = 4000
q_w, q_l = torch.randn(n), torch.randn(n)
L_w = 2 + 0.8 * q_w + 0.5 * torch.randn(n)
L_l = 2 + 0.8 * q_l + 0.5 * torch.randn(n)
p_pref = torch.sigmoid(2.0 * (q_w - q_l))                # true preference depends on
 ↪ quality only
swap = torch.rand(n) > p_pref                           # sample which one actually "won"
qa, qb = torch.where(swap, q_l, q_w), torch.where(swap, q_w, q_l)
La, Lb = torch.where(swap, L_l, L_w), torch.where(swap, L_w, L_l)

def fit(features_a, features_b, steps=500):
    w = torch.zeros(features_a.size(1), requires_grad=True)
    opt = torch.optim.Adam([w], lr=0.05)
    for _ in range(steps):
        loss = -F.logsigmoid(features_a @ w - features_b @ w).mean()
        opt.zero_grad(); loss.backward(); opt.step()
    return w.detach(), loss.item()

w1, l1 = fit(La[:, None], Lb[:, None])                          # RM that only "sees"
 ↪ length
w2, l2 = fit(torch.stack([qa, La], 1), torch.stack([qb, Lb], 1))    # RM that sees
 ↪ quality and length
print(f"length-only RM: weight on length {w1[0]:.2f}, loss {l1:.3f}")
print(f"quality+length RM: weight on quality {w2[0]:.2f}, on length {w2[1]:.2f}, loss
 ↪ {l2:.3f}")
print(f"Bayes-optimal loss (true probabilities): {F.binary_cross_entropy(p_pref,
 ↪ p_pref).item():.3f}")
```

The RM that can see quality puts almost no weight on length (length carries no extra information once quality is known). The weaker RM puts a large positive weight on length, because length is correlated with the labels. Real RMs sit somewhere between: they can partly read quality and partly fall back on shortcuts. Fixes include **length-balanced data** (pairs where the shorter answer wins), **length as an explicit input or penalty**, and checking the correlation between reward and length on held-out data.

## 19.11 What a scalar reward cannot represent

Bradley-Terry assumes preferences come from **one score per response**, so they must be **transitive**: if A beats B and B beats C, A must beat C. Real populations of labellers can be intransitive (different groups care about different things), like rock-paper-scissors. No scalar reward fits such data. *Nash Learning from Human Feedback* (Munos et al., arXiv 2312.00886) instead learns a **preference model** $P(y \succ y' \mid x)$ directly and seeks a policy that no other policy beats more than half the time (a Nash equilibrium). This line of work leads to general-preference methods covered in Chapter 24.

**Multi-objective reward models** are a practical middle ground: predict several scores (helpful-

**Notation on this page:** $a \succ b$: $a$ is preferred to $b$ • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$

ness, correctness, safety, verbosity) and combine them. *Llama 2* (arXiv 2307.09288) trained separate **helpfulness** and **safety** RMs and used the safety score for prompts flagged as unsafe or when the safety score was low. *Interpretable Preferences via Multi-Objective Reward Modeling and Mixture-of-Experts* (ArmoRM, Wang et al., arXiv 2406.12845) predicts many attribute scores and learns a gating network that weights them per prompt, with an explicit correction for verbosity.

**Why this, not that? Why read the reward from the last token, not average over all tokens?**

**Why.** In a causal (left-to-right) transformer, only the last position has attended to the entire prompt and response, so its hidden state is the only one that "knows" the whole answer. Averaging would mix in early positions that have not seen the end, where errors often appear.
**Trade-off.** The score depends on one position, so the end token must be present and identified correctly (padding side, truncation); a response truncated before its end gets scored at a misleading position.
**When the alternative wins.** Per-token or per-step scores (process reward models, Chapter 29) are better when you need credit assignment inside the response; mean pooling is common for bidirectional encoders, where every position sees everything.

## 19.12 Using TRL's `RewardTrainer`

The dataset has `chosen` and `rejected` columns (optionally a `prompt` column and a `margin` column). Fields from `RewardConfig` in TRL 1.16.0:

```
from datasets import load_dataset
from trl import RewardConfig, RewardTrainer

ds = load_dataset("trl-lib/ultrafeedback_binarized", split="train")
config = RewardConfig(
    output_dir="rm-out",
    max_length=1024,                    # default; longer pairs are truncated
    learning_rate=1e-5,                 # default 1e-4 suits LoRA; full fine-tuning wants
↪   much lower
    num_train_epochs=1,                 # default is 3.0, but RMs overfit after about one
↪   epoch
    per_device_train_batch_size=8,
    center_rewards_coefficient=0.01,    # default None; adds 0.01 * mean((r_c + r_r)^2)
    disable_dropout=True,               # default True
    bf16=True,
)
trainer = RewardTrainer(model="Qwen/Qwen2.5-0.5B-Instruct",   # loaded as a sequence
↪   classifier, num_labels=1
                        args=config, train_dataset=ds)
trainer.train()
```

Note the epoch default: TRL inherits `num_train_epochs=3.0` from the Hugging Face `TrainingArguments`, while InstructGPT found one epoch best. Watch the eval accuracy and stop when the train/eval gap opens.

**Why this, not that? Why initialise the reward model from the SFT model, not from a base model or a different family?**

**Why.** The SFT model already understands the instruction-following format and the kind of responses the policy produces, so the RM starts with useful features and needs less preference data. It also shares the tokenizer and context window with the policy.
**Trade-off.** It inherits the SFT model's blind spots, and an RM the same size as the policy is expensive to run during RL. InstructGPT used a 6B RM for a 175B policy, because 175B RMs were unstable to train and too costly.
**When the alternative wins.** A larger or stronger RM (or a different family) can be more robust to hacking, since the policy cannot easily "see" its weaknesses; Gao et al. (Chapter 21) found larger RMs over-optimise more slowly. Specialised open RMs (e.g. Skywork-Reward, arXiv 2410.18451) are often better starting points than your own SFT model.

## 19.13 Common mistakes

- Reading the score at a padding token instead of the last real token.
- Training several epochs on correlated comparisons and overfitting.
- Comparing reward values across different RMs or after re-training: rewards are only defined up to a shift (and in practice a scale).
- Treating high RM accuracy as proof that RL against it will work.
- Trusting TRL's default 3 epochs for reward-model training without watching eval accuracy.
- Reporting RM accuracy without the inter-annotator agreement that bounds it.
- Ignoring correlations between reward and length or formatting on held-out data.

**Interview check**

**Q1. Write the reward model loss and its gradient.** $-\log \sigma(r_w - r_l)$; gradient $-\sigma(r_l - r_w)(\nabla r_w - \nabla r_l)$.
**Q2. Why are reward models only defined up to a constant?** Bradley-Terry depends only on reward differences.
**Q3. How is a reward model built from an LLM?** Replace the LM head with a scalar linear head and read it at the final token of prompt+response.
**Q4. What is a margin loss?** $-\log \sigma(r_w - r_l - m)$ with $m$ larger for clearer preferences, as in Llama 2.
**Q5. Name three LLM-judge biases.** Position, verbosity and self-enhancement bias.
**Q6. Why train a reward model for only one epoch?** Comparisons are correlated and RMs overfit quickly; InstructGPT found overfitting after one epoch.
**Q7. Do RM ensembles solve reward hacking?** They reduce it, especially with different pretraining seeds, but do not eliminate it.
**Q8. What does RewardBench measure?** Pairwise accuracy of RMs on curated chosen/rejected pairs across chat, hard chat, safety and reasoning.
**Q9. Derive Bradley-Terry from a random-utility model.** Perceived utility = score + Gumbel noise; the difference of two Gumbels is logistic, so $P(y_w \succ y_l) = \sigma(r_w - r_l)$. Gaussian noise gives the Thurstone (probit) model.

**Notation on this page:** $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $r(x,y)$, $r_w$, $r_l$: reward-model scores (chosen, rejected) • $\nabla_\theta$: gradient with respect to $\theta$ • $m$: preference margin (optional) • $a \succ b$: $a$ is preferred to $b$ • $y_w$, $y_l$: chosen (winning) and rejected (losing) responses • $y$: the response (output tokens); $y_t$ is token $t$

**Q10. What does the scale of a BT reward mean?** It is relative to labelling noise: multiplying rewards by $c$ is equivalent to dividing the noise scale by $c$. Shifts are meaningless; scale reflects consistency.
**Q11. If labellers agree 70% of the time, what is the best possible RM loss?** The binary entropy $H(0.7) \approx 0.611$ nats; accuracy is capped near 70%.
**Q12. Why do RMs learn length bias, and how do you fix it?** Length correlates with preference labels, so a model that cannot fully read quality uses it as a shortcut; fix with length-balanced data, explicit length terms, or penalties during RL.
**Q13. What can't a scalar reward model represent?** Intransitive (cyclic) preferences; general preference models and Nash learning handle them.
**Q14. How did Llama 2 combine helpfulness and safety?** Two separate RMs; the safety score was used for unsafe-flagged prompts or when it was low, otherwise helpfulness.
**Q15. TRL's `RewardConfig` default epochs and learning rate?** `num_train_epochs=3.0` (inherited) and `learning_rate=1e-4`; for full fine-tuning use about one epoch and a much lower LR.

**Key takeaways**

- Preference data = (prompt, chosen, rejected); comparisons beat absolute ratings for consistency.
- Bradley-Terry: $P(y_w \succ y_l) = \sigma(r_w - r_l)$; loss $-\log \sigma(r_w - r_l)$, optionally with a margin.
- RM = LLM + scalar head at the last token, initialised from SFT, trained about one epoch.
- Ensembles and centring help; generative judges reason but have position, length and self biases.
- Evaluate with held-out accuracy, RewardBench, and how the RM behaves under optimisation.
- BT = Gumbel random utility; reward scale is relative to label noise.
- Label agreement caps accuracy and sets a floor on loss ($H(p)$).
- RMs learn shortcuts (length); scalar RMs cannot capture intransitive preferences.

**Notation on this page:** $c$: centring-loss coefficient • $H$: entropy (uncertainty) of a distribution • $H(p)$: binary entropy • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $a \succ b$: $a$ is preferred to $b$ • $y_w, y_l$: chosen (winning) and rejected (losing) responses • $y$: the response (output tokens); $y_t$ is token $t$ • $r(x,y), r_w, r_l$: reward-model scores (chosen, rejected) • $p$: a probability distribution (often the target or data)

CHAPTER 20

# The PPO-RLHF Loop

**Why this chapter matters.**
Classic RLHF combines everything so far: an SFT policy, a reward model, a KL penalty to a reference model and PPO with a critic. It is the method behind InstructGPT and the first ChatGPT. Even if your team uses DPO or GRPO, interviewers expect you to walk through this loop: the four models, the per-token reward, how advantages are computed, and what each piece costs.

## 20.1 The four models

| Model | Role | Trained? | Typical init |
|---|---|---|---|
| Policy (actor) $\pi_\theta$ | Generates responses | Yes | SFT model |
| Reference $\pi_{\text{ref}}$ | Anchor for the KL penalty | No (frozen) | SFT model |
| Reward model $r_\phi$ | Scores complete responses | No (frozen) | Trained RM (Chapter 19) |
| Value model (critic) $V_\psi$ | Per-token value estimates | Yes | Reward model (InstructGPT) or SFT |

Here $\psi$ denotes the critic's weights. Four large models in memory, two of them trained with Adam, is what makes PPO-RLHF expensive.

## 20.2 The KL-regularised objective

RLHF maximises reward while staying close to the reference model:

$$\max_\theta \; \mathbb{E}_{x\sim\mathcal{D},\, y\sim\pi_\theta(\cdot|x)}[r_\phi(x,y)] - \beta\,\mathrm{KL}(\pi_\theta(\cdot\mid x)\,\|\,\pi_{\text{ref}}(\cdot\mid x)).$$

The KL term (reverse KL, Chapter 3) does three jobs:

1. **Limits reward hacking.** The reward model is only accurate near the data it was trained on. The KL keeps the policy there.
2. **Keeps fluency and knowledge.** The reference model's language ability is preserved.
3. **Keeps diversity.** Without it, the policy collapses onto a few high-reward responses.

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\pi_{\text{ref}}$: the frozen reference model • $\phi$: reward-model weights • $r_t$: per-token reward; $r_\phi(x,y)$: RM score • $V$: vocabulary size • $\psi$: critic (value model) weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $D_{\text{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\mathcal{D}$: a dataset • *and 2 more: see the Notation table*

This objective has a closed-form optimum, $\pi^*(y \mid x) \propto \pi_{\text{ref}}(y \mid x) \exp(r(x,y)/\beta)$, which is the starting point of DPO (Chapter 22).

**InstructGPT's full objective** (arXiv 2203.02155) also mixed in the pretraining loss to avoid regressions on public NLP tasks ("PPO-ptx"):

$$\mathbb{E}_{(x,y)\sim\pi_\theta}\Big[r_\phi(x,y) - \beta \log \frac{\pi_\theta(y \mid x)}{\pi_{\text{SFT}}(y \mid x)}\Big] + \gamma_{\text{ptx}}\, \mathbb{E}_{x\sim\mathcal{D}_{\text{pretrain}}}[\log \pi_\theta(x)],$$

with $\beta = 0.02$ and $\gamma_{\text{ptx}} = 27.8$ (a coefficient on the pretraining gradients).

## 20.3 The per-token reward

PPO works on the token-level MDP (Chapter 13). The KL penalty is spread over tokens and the reward-model score is added at the end:

$$r_t = -\beta(\log \pi_\theta(y_t \mid x, y_{<t}) - \log \pi_{\text{ref}}(y_t \mid x, y_{<t})) + \mathbb{1}[t = T]\, r_\phi(x,y),$$

where $T$ is the last token of the response. The KL part uses the single-sample $k_1$ estimate: the sum over tokens of the log-ratio is an unbiased estimate of the sequence KL. It is computed with the log-probs from the rollout (old policy), and treated as part of the reward, so no gradient flows through it directly.

*Worked example.* A 4-token response with per-token log-ratios $(0.1, -0.05, 0.2, 0.0)$, $\beta = 0.05$ and RM score 1.5: $r = (-0.005, 0.0025, -0.01, 1.5)$.

## 20.4 One iteration, step by step

1. **Sample prompts** from the RL prompt set.
2. **Generate** responses with the current policy (temperature 1.0, no top-p, so log-probs match the true policy).
3. **Score**: run the reward model on each (prompt, response) to get $r_\phi(x,y)$. Apply the **EOS trick**: if a response hit the length limit without an end-of-sequence token, its reward is unreliable, so replace it with a fixed penalty (a reward of $-1$ in the N+ implementation details study).
4. **Forward passes** for: policy log-probs (these are $\log \pi_{\theta_{\text{old}}}$), reference log-probs, and critic values per token.
5. **Per-token rewards** as above; optionally **whiten** rewards (normalise mean and variance).
6. **GAE**: compute advantages and value targets (usually $\gamma = 1$; InstructGPT applied no discount).
7. **PPO updates**: split into mini-batches; for each, compute the new log-probs, the clipped policy loss, and the critic loss; update both models.
8. Repeat.

```
import torch

def rlhf_rewards(logp_policy, logp_ref, rm_score, mask, beta=0.05):
    """Per-token rewards for PPO-RLHF. logp_*: (B, T) log-probs of the sampled tokens.
```

**Notation on this page:** $\pi_{\text{ref}}$: the frozen reference model • $\pi^*$: the optimal policy • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $r_t$: per-token reward; $r_\phi(x,y)$: RM score • $\theta$: the model's trainable weights • $\phi$: reward-model weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\gamma$: discount (1 in RLHF); $\gamma_{\text{ptx}}$: pretraining-loss weight • *and 7 more: see the Notation table*

```
    mask: (B, T) 1 for response tokens. rm_score: (B,). Returns (B, T) rewards."""
    kl = (logp_policy - logp_ref) * mask                          # k1 per token
    rewards = -beta * kl
    last = mask.sum(1).long() - 1                                 # index of last response
↪   token
    rewards[torch.arange(len(last)), last] += rm_score
    return rewards, kl.sum(1)

def gae_batch(rewards, values, mask, gamma=1.0, lam=0.95):
    B, T = rewards.shape
    adv = torch.zeros_like(rewards); running = torch.zeros(B)
    for t in reversed(range(T)):
        next_v = values[:, t + 1] * mask[:, t + 1] if t + 1 < T else torch.zeros(B)
        delta = rewards[:, t] + gamma * next_v - values[:, t]
        running = (delta + gamma * lam * running) * mask[:, t]
        adv[:, t] = running
    return adv, adv + values

torch.manual_seed(0); torch.set_printoptions(sci_mode=False)
B, T = 2, 5
mask = torch.tensor([[1, 1, 1, 1, 0], [1, 1, 1, 1, 1]], dtype=torch.float)
lp, lr = torch.randn(B, T) * 0.1 - 1, torch.randn(B, T) * 0.1 - 1
rewards, seq_kl = rlhf_rewards(lp, lr, torch.tensor([1.5, -0.5]), mask)
adv, ret = gae_batch(rewards, torch.zeros(B, T), mask)
print("rewards:\n", rewards.round(decimals=3))
print("sequence KL estimate:", seq_kl.round(decimals=3))
print("advantages (lam=0.95, zero critic):\n", adv.round(decimals=3))
```

## 20.5 The KL coefficient and KL controllers

$\beta$ is the most important RLHF hyperparameter. Too small, and the policy hacks the reward model; too large, and it barely moves. *Fine-Tuning Language Models from Human Preferences* (Ziegler et al., arXiv 1909.08593) introduced an **adaptive KL controller** that targets a KL value:

$$e_t = \mathrm{clip}\Big(\frac{\mathrm{KL}_t - \mathrm{KL}_{\text{target}}}{\mathrm{KL}_{\text{target}}}, -0.2, 0.2\Big), \qquad \beta_{t+1} = \beta_t\,(1 + K_\beta\, e_t), \quad K_\beta = 0.1.$$

If the KL is above target, $\beta$ grows; if below, it shrinks. This makes runs with different seeds comparable at the same KL "budget".

**Why this, not that? Why put the KL penalty in the reward, not in the loss?**

**Why.** In the reward, the KL becomes part of the return that GAE and the critic handle, so each token is penalised for its own divergence *and* for the divergence it leads to later. This is the "KL-regularised RL" objective solved exactly by the closed form used in DPO. It is how InstructGPT and most PPO-RLHF code do it.
**Trade-off.** The critic must learn the KL part of the return too, and the KL signal gets mixed with reward noise in the advantages.

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $\mathrm{clip}(u, a, b)$: limit $u$ to the range $[a, b]$ • $t$: time step or token position • $K_\beta$: controller gain • $e_t$: clipped relative KL error (adaptive controller)

**When the alternative wins.** Critic-free methods (GRPO) usually put a KL *loss* term directly in the objective, with its own estimator (k3, Chapter 3), because without a critic there is no per-token return to put it into. Many reasoning-RL recipes drop the KL entirely (DAPO, Chapter 28).

## 20.6 Whitening and length effects

**Advantage whitening** (normalising advantages to mean 0, std 1 per batch) is standard. **Reward whitening** is optional and was found to change results noticeably in the N+ study.

**Length.** RLHF policies tend to get **longer** over training, because reward models often prefer longer answers (Chapter 21). With a per-token KL penalty, longer responses also accumulate more KL, which partly pushes back. Monitor mean response length alongside reward.

## 20.7 The InstructGPT recipe

From *Training language models to follow instructions with human feedback* (arXiv 2203.02155):

1. **SFT.** About 13k training prompts with labeller demonstrations. 16 epochs, cosine LR decay; they picked the final SFT model by RM score on a validation set, even though validation loss overfit after one epoch.
2. **Reward model.** About 33k training prompts; labellers ranked $K = 4$ to 9 responses; a **6B** RM, one epoch at LR $9 \times 10^{-6}$; rewards normalised so demonstrations score 0 on average.
3. **PPO.** About 31k prompts (API only), 256k episodes, batch 512, mini-batch 64 (8 mini-batches, one inner epoch), clip ratio 0.2, rollout temperature 1, **no discount** in GAE, EMA of weights (decay 0.992), $\beta = 0.02$, value function initialised from the 6B RM. **PPO-ptx** added pretraining gradients with coefficient 27.8.

**Result:** outputs from the **1.3B** InstructGPT model were preferred by labellers to outputs from the **175B** GPT-3, despite having over 100 times fewer parameters.

## 20.8 The N+ implementation details

*The N+ Implementation Details of RLHF with PPO: A Case Study on TL;DR Summarization* (Huang et al., arXiv 2403.17031) reproduced OpenAI's RLHF summarisation work with Pythia models and listed details that matter, including:

- **Disable dropout** in the policy (and SFT and RM): with dropout, the log-probs used in the PPO ratio are not reproducible, and the KL estimate becomes unreliable.
- **Reward normalisation** based on SFT demonstrations (mean zero).
- **The EOS trick**: penalise responses without an end token with a reward of $-1$.
- **Reward whitening** (optional), **value head initialisation**, and careful handling of padding and left-padding during generation.

## 20.9 Memory and compute budget

For a policy of $N$ parameters in bf16 with AdamW (about 16 bytes per trained parameter, Chapter 10):

- Policy: about $16N$ bytes. Critic of the same size: another $16N$.
- Reference and reward model: about $2N$ each (inference only).
- Plus activations for training and the KV cache for generation.

So about $36N$ bytes before activations, more than twice SFT. For a 7B policy that is about 250 GB, which needs several GPUs with ZeRO or FSDP sharding (Chapter 42). Generation usually dominates wall-clock time. These costs drove the move to DPO (no RM, critic or generation) and to GRPO (no critic).

**Why this, not that? Why PPO with a reward model, not DPO on the same preference data?**

**Why.** PPO is online: it scores fresh samples from the current policy, so it fixes the policy's actual mistakes and generalises beyond the preference dataset. A reward model can also score prompts that have no human comparisons. Careful comparisons (arXiv 2404.10719) found well-tuned PPO beating DPO.
**Trade-off.** Four models, generation in the loop, many sensitive hyperparameters ($\beta$, clip, GAE, learning rates for two models), and the risk of reward hacking.
**When the alternative wins.** DPO wins when compute and engineering time are limited, when you want a stable, reproducible pipeline, or when the preference data already comes from a policy close to yours.

## 20.10 Why the per-token penalty gives the sequence KL

The objective penalises the KL between whole-response distributions, but PPO's reward is per token. They match because of the **chain rule of KL** for autoregressive models. With $\pi(y \mid x) = \prod_t \pi(y_t \mid x, y_{<t})$,

$$\mathrm{KL}(\pi_\theta(\cdot \mid x) \| \pi_{\text{ref}}(\cdot \mid x)) = \mathbb{E}_{y\sim\pi_\theta}\Big[\log \frac{\pi_\theta(y \mid x)}{\pi_{\text{ref}}(y \mid x)}\Big] = \mathbb{E}_{y\sim\pi_\theta}\Big[\sum_{t=1}^{T} \log \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\text{ref}}(y_t \mid x, y_{<t})}\Big].$$

So the sum of per-token $k_1$ terms along a sampled response is an **unbiased single-sample estimate of the sequence KL**. Adding $-\beta\times$ (per-token log-ratio) to each token's reward, and the RM score at the end, makes the undiscounted return of each response exactly

$$\sum_t r_t = r_\phi(x, y) - \beta \log \frac{\pi_\theta(y \mid x)}{\pi_{\text{ref}}(y \mid x)},$$

whose expectation is the KL-regularised objective. With $\gamma = 1$, the return-to-go from token $t$ contains the KL of all **later** tokens: an early token that leads the response into unusual territory is charged for the divergence it causes downstream. A per-token KL *loss* (as in GRPO) charges each token only for itself.

**Notation on this page:** $N$: number of examples • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\pi_\theta$: the policy (the model), with weights $\theta$ • $y_{<t}$: the tokens before position $t$ • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $t$: time step or token position • $\pi_{\text{ref}}$: the frozen reference model • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • *and 5 more: see the Notation table*

Individual $k_1$ terms can be **negative** (when the sampled token is more likely under the reference than under the policy), so the per-token "KL reward" sometimes pays the policy. Only the expectation is guaranteed non-negative. Some implementations clamp or use $k_3$ for logging; veRL exposes the choice as `algorithm.kl_penalty` (`kl/k1`, `abs`, `mse/k2`, `low_var_kl/k3`, and `full`).

## 20.11 A complete toy PPO-RLHF loop

The pieces above, put together on a problem small enough to run in seconds. The "policy" is a table of logits for each of 6 positions over 5 tokens. A frozen "reward model" gives the share of tokens equal to 3. The reference is the starting policy. A critic table predicts the value at each position. Each iteration does rollout, scoring, KL-in-reward, GAE, advantage whitening and two PPO epochs with clipping. Because the problem is this simple, we can also compute the exact KL-regularised optimum $\pi^* \propto \pi_{\text{ref}} \exp(r/\beta)$ and check that PPO finds it.

```
import torch.nn.functional as F

def ppo_rlhf(beta, iters=150, T=6, V=5, B=64, seed=0):
    """Toy PPO-RLHF: a position-wise policy over V tokens, a frozen 'reward model' that
    likes token 3, a frozen reference, a learned critic, KL-in-reward, GAE and
↪   clipping."""
    torch.manual_seed(seed)
    ref_logits = torch.randn(T, V) * 0.5
    pol = ref_logits.clone().requires_grad_(True)            # policy starts at the
↪   reference
    critic = torch.zeros(T, requires_grad=True)              # V(state) per position
    opt = torch.optim.Adam([pol, critic], lr=0.05)
    mask = torch.ones(B, T)
    for it in range(iters):
        with torch.no_grad():                                # 1-2. rollout and old log-probs
            dist = torch.distributions.Categorical(logits=pol.expand(B, T, V))
            y = dist.sample()
            logp_old = dist.log_prob(y)
            logp_ref = torch.distributions.Categorical(logits=ref_logits.expand(B, T,
↪   V)).log_prob(y)
            score = (y == 3).float().mean(1)                 # 3. reward model score in
↪   [0, 1]
            rewards, seq_kl = rlhf_rewards(logp_old, logp_ref, score, mask, beta)
            values = critic.expand(B, T)
            adv, ret = gae_batch(rewards, values, mask)      # 4. GAE with gamma = 1
            adv = (adv - adv.mean()) / (adv.std() + 1e-8)    # advantage whitening
        for epoch in range(2):                               # 5. PPO epochs over
↪       mini-batches
            for mb in torch.randperm(B).split(16):
                logp = torch.distributions.Categorical(logits=pol.expand(len(mb), T,
↪   V)).log_prob(y[mb])
                ratio = (logp - logp_old[mb]).exp()
                pg = -torch.min(ratio * adv[mb], ratio.clamp(0.8, 1.2) * adv[mb]).mean()
                vf = 0.5 * ((critic.expand(len(mb), T) - ret[mb]) ** 2).mean()
                opt.zero_grad(); (pg + vf).backward(); opt.step()
```

**Notation on this page:** $k$: an index or count (see text) • $\pi_{\text{ref}}$: the frozen reference model • $\pi^*$: the optimal policy • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $r_t$: per-token reward; $r_\phi(x, y)$: RM score

```
    return score.mean().item(), seq_kl.mean().item()

for beta in (0.0, 0.05, 0.2, 1.0):
    s, kl = ppo_rlhf(beta)
    print(f"beta={beta:4.2f}: final RM score {s:.2f}, KL to reference {kl:.2f}")

# The closed-form optimum pi* = pi_ref * exp(r / beta) / Z, position by position (reward
 ↪ 1/T per token 3).
torch.manual_seed(0)
ref_logits = torch.randn(6, 5) * 0.5
for beta in (0.05, 0.2, 1.0):
    opt_logits = ref_logits.clone(); opt_logits[:, 3] += (1 / 6) / beta
    p, q = torch.softmax(opt_logits, -1), torch.softmax(ref_logits, -1)
    print(f"optimum beta={beta:4.2f}: RM score {p[:, 3].mean():.2f}, KL {(p * (p /
     ↪ q).log()).sum():.2f}")
```

With $\beta = 0$ the policy goes all the way to the reward model's favourite output (score 1.0) and the KL keeps growing: with a real, imperfect RM this is where hacking happens. As $\beta$ grows, the final score and KL shrink, and they match the closed-form optimum closely. This is the trade-off curve every RLHF run lives on: $\beta$ **chooses a point on the reward-versus-KL frontier.**

## 20.12 Training the critic well

The critic decides the quality of the advantages, and in RLHF it starts in a poor state:

- **Initialisation.** InstructGPT initialised the value model from the reward model, so it already "knows" what good responses look like. But the RM predicts a score at the *last* token only; at earlier positions its head outputs are meaningless as values.
- **Value pretraining.** *What's Behind PPO's Collapse in Long-CoT? Value Optimization Holds the Secret* (VC-PPO, arXiv 2503.01491) showed that a critic initialised from the RM has a bias towards early tokens that hurts long responses, and fixed it by **pretraining the value model** on rollouts from the initial policy (with $\lambda = 1$, Monte Carlo targets) before PPO starts, plus **decoupled GAE**: $\lambda_{\text{critic}} = 1.0$ for unbiased value targets and $\lambda_{\text{actor}} = 0.95$ for lower-variance advantages.
- **Explained variance.** Log $1 - \mathrm{Var}(\hat{R} - V)/\mathrm{Var}(\hat{R})$. Near 1 means the critic predicts returns well; near 0 or negative means its advantages are mostly noise.
- **Separate learning rate.** The critic is often given a higher learning rate or a warm-up period during which the policy is not updated.

**Why this, not that? Why initialise the critic from the reward model, not from the SFT policy?**

**Why.** The RM's features already encode response quality, so value predictions start closer to the true returns and advantages are useful sooner. It also frees the critic from having to learn the reward model from scratch through noisy returns.
**Trade-off.** The RM was trained to score complete responses at the last token; its outputs at earlier positions are biased, which can mislead long responses until the critic adapts (the VC-PPO finding). It also ties the critic's size and tokenizer to the RM's.

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\lambda$: GAE parameter (0 to 1), trades bias for variance • $V$: vocabulary size • $\hat{R}$: return / value target

**When the alternative wins.** Initialise from the policy (or pretrain the critic) when the RM is a different family or size, when responses are long (reasoning), or when the reward is rule-based and there is no RM at all.

## 20.13 Monitoring a PPO-RLHF run

| Metric | What it tells you | Warning sign |
|---|---|---|
| Mean RM score | progress on the proxy | rising fast while samples get worse (hacking) |
| KL to reference (sequence, $k_1$ mean) | how far the policy moved | runaway growth; negative mean (a bug) |
| Total objective (score $-\beta\cdot$KL) | what PPO actually optimises | falling |
| Response length | length exploitation | steady growth without quality gains |
| Policy entropy | exploration left | collapse to near 0 |
| Clip fraction, approx. KL (old vs new) | step size | high or growing |
| Value loss, explained variance | critic quality | explained variance near 0 |
| Win rate vs SFT on held-out prompts (judge or humans) | the real goal | flat or falling while RM score rises |

Always read samples. A classic early warning is the policy discovering a phrase or format the RM loves (e.g. long apologies, excessive bullet lists) and using it everywhere.

## 20.14 PPO-RLHF in today's frameworks

TRL 1.16 no longer ships a `PPOTrainer` (neither at the top level nor in `trl.experimental`); its online RL path is `GRPOTrainer`, which can also run RLOO-style and REINFORCE-style variants. Full PPO with a critic is mainly done in **veRL** and **OpenRLHF**. In veRL (main branch, checked October 2026) the relevant defaults are: `algorithm.adv_estimator=gae`, `algorithm.gamma=1.0`, `algorithm.lam=1.0`, `algorithm.use_kl_in_reward=false` with `algorithm.kl_ctrl.type=fixed` and `kl_coef=0.001` (an `adaptive` type with `target_kl` and `horizon` also exists), and on the actor `clip_ratio=0.2`, `clip_ratio_c=3.0` (dual clip), `loss_agg_mode=token-mean`, `entropy_coeff=0`, `use_kl_loss=false`, `ppo_epochs=1`; the critic has `cliprange_value=0.5`. So classic RLHF ("KL in reward") is something you have to switch on (`use_kl_in_reward=true`), and the reasoning-RL style (KL loss or no KL) is closer to the defaults.

**Systems layout.** Generation is the bottleneck. Frameworks either **colocate** all models on the same GPUs and switch between training and inference modes (veRL's hybrid engine), or place them on **separate** GPU pools (OpenRLHF with Ray, vLLM for generation). After each update,

**Notation on this page:** $k$: an index or count (see text) • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay)

the new policy weights must be synced to the inference engine, and the trainer must recompute log-probs because the inference engine's numbers differ slightly (Chapter 28).

**Why this, not that? Why a fixed small β in modern recipes, not the adaptive KL controller?**

**Why.** A fixed $\beta$ is simpler to reason about and reproduce, and with strong rewards (verifiers) or short runs the KL rarely needs active control. Adaptive control can also hide problems by quietly raising $\beta$.
**Trade-off.** With a fixed $\beta$, the final KL depends on the learning rate, run length and reward scale, so runs are harder to compare, and hacking can start silently if the KL drifts.
**When the alternative wins.** Adaptive control (Ziegler et al.) is useful with learned reward models in long runs where you want a fixed "KL budget", and for fair comparisons between methods at matched KL.

## 20.15 Common mistakes

- Clamping negative per-token KL terms to zero without realising it biases the KL estimate upwards.
- Starting PPO before the critic is reasonable (check explained variance), especially for long responses.
- Assuming a framework's defaults implement classic RLHF (veRL defaults to no KL in the reward).

**Interview check**

**Q1. Name the four models in PPO-RLHF and which are trained.** Policy (trained), reference (frozen), reward model (frozen), value model (trained).
**Q2. Write the per-token reward.** $r_t = -\beta(\log \pi_\theta - \log \pi_{\text{ref}})(y_t) + \mathbb{1}[t = T]\, r_\phi(x, y)$.
**Q3. What does the KL penalty prevent?** Reward hacking, loss of fluency/knowledge, and collapse of diversity.
**Q4. How does an adaptive KL controller work?** It raises $\beta$ when measured KL is above a target and lowers it when below, with a clipped proportional update.
**Q5. What is PPO-ptx?** Adding the pretraining log-likelihood to the RL objective to avoid regressions on standard NLP tasks.
**Q6. What is the EOS trick?** Give truncated responses without an end token a fixed penalty (e.g. $-1$), since the RM score is not meaningful for them.
**Q7. Why disable dropout in RLHF?** Dropout makes log-probs random, breaking the PPO ratio and the KL estimate.
**Q8. Roughly how much memory does PPO-RLHF need relative to SFT?** About twice or more: two trained models (policy, critic) plus two frozen ones, plus generation.
**Q9. What was InstructGPT's headline result?** 1.3B InstructGPT outputs were preferred to 175B GPT-3 outputs.
**Q10. Why is the sum of per-token log-ratios an unbiased estimate of the sequence KL?** By the chain rule: $\log \pi(y)/\pi_{\text{ref}}(y) = \sum_t$ log-ratio per token, and its expectation under $\pi_\theta$ is the sequence KL.

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\phi$: reward-model weights • $\mathbb{1}[\cdot]$: 1 if the condition is true, else 0 • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $r_t$: per-token reward; $r_\phi(x, y)$: RM score • $t$: time step or token position • $T$: last token of the response

**Q11. Can a per-token KL reward be positive? Why?** Yes: a single $k_1$ term is negative when the sampled token is more likely under the reference; only the expectation is non-negative.
**Q12. How does KL-in-reward differ from a KL loss in credit assignment?** In the reward with $\gamma = 1$, each token's return includes the KL of all later tokens, so early tokens are charged for downstream divergence; a per-token loss charges each token only for itself.
**Q13. What is explained variance and why log it?** $1 - \mathrm{Var}(\hat{R} - V)/\mathrm{Var}(\hat{R})$; it shows whether the critic predicts returns. Near zero means noisy advantages.
**Q14. What did VC-PPO change about the critic?** Value pretraining before PPO and decoupled GAE ($\lambda_{\text{critic}} = 1, \lambda_{\text{actor}} = 0.95$) to fix the critic's bias on long responses.
**Q15. How does $\beta$ shape the final policy, in one sentence?** It picks a point on the reward-vs-KL frontier; the exact optimum is $\pi_{\text{ref}} \exp(r/\beta)/Z$.
**Q16. Where is PPO-RLHF run in practice today?** veRL and OpenRLHF (TRL 1.16 has no PPOTrainer); with colocated or separate generation and training pools.

**Key takeaways**

- Four models: policy, reference, reward, value.
- Objective: reward minus $\beta$ KL to the reference; per-token reward = $-\beta \cdot$log-ratio, plus RM score at the last token.
- Loop: generate, score (EOS trick), forward passes, GAE, PPO mini-batch updates.
- $\beta$ is crucial; adaptive controllers target a KL value.
- InstructGPT: 13k SFT / 33k RM / 31k PPO prompts, 6B RM, $\beta = 0.02$, PPO-ptx; 1.3B beat 175B GPT-3.
- Per-token $k_1$ sums to an unbiased sequence-KL estimate; KL-in-reward charges early tokens for later divergence.
- $\beta$ selects a point on the reward-KL frontier; PPO reaches the closed-form optimum in the toy.
- Critic quality (init, pretraining, explained variance) decides advantage quality.

**Notation on this page:** $k$: an index or count (see text) • $\gamma$: discount (1 in RLHF); $\gamma_{\text{ptx}}$: pretraining-loss weight • $V$: vocabulary size • $\hat{R}$: return / value target • $\lambda$: GAE parameter (0 to 1), trades bias for variance • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\pi_{\text{ref}}$: the frozen reference model • $r_t$: per-token reward; $r_\phi(x, y)$: RM score

CHAPTER 21

# Reward Hacking and Overoptimisation

> **Why this chapter matters.**
> Every reward is a proxy for what we actually want. Optimise a proxy hard enough and the policy finds ways to score highly without doing what we meant: this is **reward hacking**. It appears in RLHF (long, flattering answers), in reasoning RL (gaming the answer checker), and in agent RL (editing the tests instead of fixing the code). Interviewers ask how you would detect and prevent it.

## 21.1 Goodhart's law

"When a measure becomes a target, it ceases to be a good measure." A reward model is trained on a limited set of comparisons. Within that data, its score correlates with quality. When a policy is optimised against it, the policy moves to places where the RM has never seen data, and where the correlation breaks. The **proxy reward** keeps rising while the **true quality** (what humans would say) rises, then falls.

## 21.2 Scaling laws for overoptimisation

*Scaling Laws for Reward Model Overoptimization* (Gao et al., arXiv 2210.10760) studied this with a synthetic setup: a large "gold" RM plays the role of humans, and smaller "proxy" RMs are trained on labels from it. Policies are optimised against the proxy, and the gold score is measured. Writing $d = \sqrt{\mathrm{KL}(\pi \| \pi_{\text{init}})}$ for the distance from the initial policy, the gold reward follows

$$R_{\text{BoN}}(d) = d\,(\alpha_{\text{BoN}} - \beta_{\text{BoN}}\, d), \qquad R_{\text{RL}}(d) = d\,(\alpha_{\text{RL}} - \beta_{\text{RL}} \log d),$$

for best-of-$n$ and RL respectively, with $R(0) = 0$ and coefficients $\alpha$, $\beta$ fitted per setup. Both rise, peak and then fall. Key findings:

- The coefficients scale smoothly with RM size: **larger reward models** are hacked more slowly.
- More RM training data also helps.
- RL moves further in KL than best-of-$n$ for the same gain, so KL is not a perfect common currency between methods.

```
import math
# Shape of Gao et al.'s RL curve: gold reward rises, peaks, then falls as the policy
↪  moves away.
alpha, beta = 1.0, 0.5
for kl in (0.5, 2, 8, 20, 50, 100):
```

**Notation on this page:** $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $d = \sqrt{\mathrm{KL}}$: distance from the initial policy • $\alpha, \beta$: fitted constants (Gao et al.); elsewhere the KL coefficient • $\alpha, \beta$: fitted constants in the over-optimisation laws • $R$: gold reward score • $n$: number of samples in best-of-$n$

```
    d = math.sqrt(kl)
    print(f"KL={kl:5.1f} nats  d={d:5.2f}  gold reward={d * (alpha - beta *
    ↪  math.log(d)):+.2f}")
print("peak at d = exp(alpha/beta - 1) =", round(math.exp(alpha / beta - 1), 2))
```

(The peak follows from setting $\frac{d}{dd}[d(\alpha - \beta \log d)] = \alpha - \beta \log d - \beta = 0$.)

## 21.3 Common forms of hacking

**Length hacking.** Reward models often prefer longer answers, and RL exploits this. *A Long Way to Go: Investigating Length Correlations in RLHF* (Singhal et al., arXiv 2310.03716) found that much of the reward gain from RLHF in their settings could be explained by length increases, and that optimising for length alone reproduced much of the improvement.

**Sycophancy.** Agreeing with the user, flattering them, or changing a correct answer when the user pushes back. *Towards Understanding Sycophancy in Language Models* (Sharma et al., arXiv 2310.13548) showed that RLHF-trained assistants are consistently sycophantic and that human preference data itself rewards answers that match the user's views.

**Format and style hacking.** Bullet lists, bold text, emojis, confident tone, or hedging phrases that judges like regardless of content.

**Verifier hacking (reasoning RL).** With rule-based rewards: outputting several answers hoping one matches, exploiting a lenient parser, printing the expected format without correct work. With unit tests: special-casing the test inputs, editing or deleting tests, or exiting early with a success code.

**Environment hacking (agents).** Using tools in unintended ways: reading answer files, modifying the grader, or exploiting bugs in the sandbox (Chapter 35).

**Why this, not that? Why rule-based (verifiable) rewards for reasoning, not a neural reward model?**

**Why.** A correct checker cannot be fooled by style or length, so it is far harder to hack, and it gives a clean 0/1 signal. DeepSeek-R1 avoided neural reward models for reasoning precisely because they found them prone to reward hacking during large-scale RL (arXiv 2501.12948).
**Trade-off.** Only works where answers can be checked (maths, code, formats); rule-based checkers have their own loopholes (lenient parsers, weak tests); and the signal is sparse.
**When the alternative wins.** For open-ended tasks (writing, helpfulness, safety) there is no checker; a reward model or rubric-based judge is the only option, with KL, ensembles and monitoring to contain hacking.

## 21.4 Mitigations

**1. KL regularisation.** The penalty $\beta\,\mathrm{KL}(\pi_\theta \| \pi_{\text{ref}})$ limits how far the policy moves, i.e. limits $d$ in Gao's curves. Choose $\beta$ (or a KL target) so training stops near the peak.

**2. Early stopping on a held-out judge.** Track a separate, stronger judge or human evaluation, and keep the checkpoint where it peaks.

**Notation on this page:** $\alpha, \beta$: fitted constants (Gao et al.); elsewhere the KL coefficient • $\alpha, \beta$: fitted constants in the over-optimisation laws • $d = \sqrt{\mathrm{KL}}$: distance from the initial policy • $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$

**3. Better reward models.** Larger RMs, more and more diverse data, and regular refreshing of the RM with comparisons of the *current* policy's outputs (iterated RLHF, as in Llama 2).

**4. Ensembles and uncertainty penalties.** Reward = ensemble mean minus a multiple of the ensemble's standard deviation, or the minimum across members (arXiv 2312.09244).

**5. Length control.** Explicit length penalties; length-normalised rewards; **disentangled reward models** that learn a separate length head and use only the length-independent part, as in *ODIN: Disentangled Reward Mitigates Hacking in RLHF* (Chen et al., arXiv 2402.07319); length-controlled evaluation.

**6. Reward shaping and clipping.** Cap or squash rewards so extreme scores do not dominate; normalise rewards per prompt.

**7. Robust verifiers.** For reasoning: strict answer parsing, one final answer only, format penalties. For code: hidden tests, read-only test files, sandboxing, and checking for test tampering.

**8. Monitoring.** Watch length, repetition, the share of responses with lists, refusal rate, KL, and samples by eye. A reward curve that rises faster than expected is a warning sign, not good news.

**Why this, not that? Why keep a KL penalty to the reference, not trust the reward fully?**

**Why.** The reward model is accurate only near its training distribution. The KL penalty is the simplest, most general limit on how far into unreliable territory the policy can go, and it also preserves fluency and diversity.
**Trade-off.** It also limits *good* change: the policy cannot move far from the reference even where the reward is reliable. A fixed $\beta$ may be too strong early and too weak late.
**When the alternative wins.** With trustworthy rewards (verifiable maths and code answers), many recipes use a tiny or zero KL (R1-Zero used 0.001; DAPO uses none), because the reward cannot be hacked by drifting, and a KL would only slow down learning of long reasoning.

## 21.5 Goodhart in a simulation: light vs heavy-tailed errors

Model a proxy reward as gold quality plus an error, $\hat{r} = g + e$, and select the best of $n$ samples by the proxy. Whether optimisation helps or hurts depends on the **shape** of the error distribution:

```
import math, torch
torch.manual_seed(0)

# Best-of-n against a proxy reward = gold quality + error. Light-tailed vs heavy-tailed
↪  error.
def best_of_n(n, heavy, trials=4000):
    gold = torch.randn(trials, n)
    err = torch.distributions.StudentT(df=1.5).sample((trials, n)) * 0.5 if heavy else
↪  0.5 * torch.randn(trials, n)
    proxy = gold + err
    pick = proxy.argmax(1)
```

**Notation on this page:** $\alpha, \beta$: fitted constants (Gao et al.); elsewhere the KL coefficient • $r(x, y)$: the reward for response $y$ to prompt $x$ • $g$: gold quality • $e$: proxy error • $n$: number of samples in best-of-$n$

```
    return gold[torch.arange(trials), pick].mean().item(), proxy[torch.arange(trials),
     ↪ pick].mean().item()

for n in (1, 4, 16, 64, 256, 1024):
    g_l, _ = best_of_n(n, heavy=False)
    g_h, p_h = best_of_n(n, heavy=True)
    kl = math.log(n) - (n - 1) / n
    print(f"n={n:5d} (KL {kl:4.2f} nats): gold, light-tailed error {g_l:+.2f} |
     ↪ heavy-tailed error {g_h:+.2f} (proxy {p_h:+.1f})")
```

With **light-tailed** (Gaussian) error, the selected sample's gold quality keeps rising with $n$, only more slowly than the proxy. With **heavy-tailed** error, gold quality rises at first, **peaks** (around $n = 16$ here) and then falls back towards zero, while the proxy score explodes: at large $n$ the winner is simply the sample with the most extreme *error*. This is the shape of Gao et al.'s over-optimisation curves, and the KL column uses the best-of-$n$ formula $\log n - (n-1)/n$. *Catastrophic Goodhart: regularizing RLHF with KL divergence does not mitigate heavy-tailed reward misspecification* (Kwa et al., arXiv 2407.14503) proved the strong version: if the reward error is heavy-tailed, there are policies with tiny KL to the reference that get arbitrarily high proxy reward and no gain in true utility. So a KL penalty alone cannot be a complete defence; you also need rewards whose errors are bounded (verifiers, ensembles, clipping, rubric judges with caps).

**Why this, not that? Why cap or normalise rewards (clipping, ensembles, rank-based rewards), not use the raw RM score?**

**Why.** The largest raw scores are where the RM is most likely wrong (extremes of the error distribution). Capping or ranking removes the incentive to chase those extremes, and ensembles shrink idiosyncratic errors.
**Trade-off.** Caps and ranks throw away real information at the top end (a genuinely excellent answer is scored like a good one), and ensembles cost more compute; shared errors across members remain.
**When the alternative wins.** Raw scores are fine for small amounts of optimisation (low $n$, short RL runs, strong KL), and for verifiable rewards whose errors are rare and bounded.

## 21.6 Hacking in reasoning and agent training

With verifiable rewards the reward function is code, and code has bugs. Patterns seen in 2025–2026 RL runs:

- **Test special-casing.** Coding agents edit or delete unit tests, hard-code expected outputs, or exit early with a success code.
- **Checker exploits.** Answers formatted to confuse the parser (several boxed answers, an answer that matches by substring), or judges persuaded by confident text.
- **Environment tampering.** In agentic environments with shell access, modifying the grader or reading the answer file.

*Monitoring Reasoning Models for Misbehavior and the Risks of Promoting Obfuscation* (Baker et al., arXiv 2503.11926) found that a weaker model (GPT-4o) reading a frontier reasoning model's

**Notation on this page:** $n$: number of samples in best-of-$n$

**chain of thought** could detect reward hacking in agentic coding tasks far better than by watching actions alone. But when they put the CoT monitor **into the reward**, a little optimisation produced more capable and more aligned agents, while too much led to **obfuscated** hacking: the model kept hacking but stopped revealing it in its reasoning. The lesson: use CoT monitors for detection and evaluation, and be careful about optimising directly against them.

**Defences that work in practice:** sandboxing and read-only tests; hidden test sets; checking the diff for edits to tests; multiple independent verifiers; reading samples; tracking the gap between the training reward and a held-out evaluation with a different checker.

**Interview check**

**Q1. What is reward hacking?** The policy increases the proxy reward in ways that do not improve (or harm) the true objective.
**Q2. Describe Gao et al.'s overoptimisation result.** Gold reward vs $d = \sqrt{\mathrm{KL}}$ rises then falls: $d(\alpha - \beta d)$ for best-of-$n$ and $d(\alpha - \beta \log d)$ for RL; larger RMs overoptimise more slowly.
**Q3. Name four forms of hacking in RLHF.** Length, sycophancy, formatting/style, and exploiting RM blind spots (e.g. confident tone).
**Q4. How do you detect hacking during training?** Held-out stronger judge or human eval, length and format statistics, KL, reading samples, and checking for proxy-gold divergence.
**Q5. How do ensembles help?** Members disagree off-distribution; using mean-minus-std or min penalises uncertain regions.
**Q6. Why did DeepSeek-R1 avoid neural reward models for reasoning?** They were prone to reward hacking in large-scale RL; rule-based accuracy and format rewards were more robust.
**Q7. How can a code-RL agent hack unit-test rewards?** Special-case tests, edit or delete tests, exit early with success, or read expected outputs.
**Q8. Why can gold quality fall while best-of-$n$ proxy score keeps rising?** With heavy-tailed proxy errors, the top proxy score at large $n$ is driven by the most extreme error, not by quality.
**Q9. Does a KL penalty prevent reward hacking?** Not fully: with heavy-tailed reward errors, small-KL policies can still get huge proxy reward with no true gain (Catastrophic Goodhart).
**Q10. What did Baker et al. find about chain-of-thought monitors?** They detect hacking well, but optimising strongly against them causes obfuscated hacking that the CoT no longer reveals.
**Q11. Name three reward-hacking patterns in coding RL and a defence for each.** Editing tests (read-only tests, diff checks), hard-coding outputs (hidden tests), early exit with success codes (verify outputs, not exit status).

**Key takeaways**

- Goodhart: optimising a proxy eventually hurts the true objective.
- Gao et al.: gold reward vs $\sqrt{\mathrm{KL}}$ rises then falls; bigger RMs help.
- Hacks: length, sycophancy, format, verifier and environment exploits.
- Mitigate with KL, early stopping on a held-out judge, better/ensembled RMs, length control, robust verifiers and monitoring.
- Light-tailed errors: optimisation slows; heavy-tailed: gold peaks then falls.

**Notation on this page:** $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $d = \sqrt{\mathrm{KL}}$: distance from the initial policy • $\alpha, \beta$: fitted constants (Gao et al.); elsewhere the KL coefficient • $\alpha, \beta$: fitted constants in the over-optimisation laws • $n$: number of samples in best-of-$n$

- KL alone cannot stop heavy-tailed misspecification; bound the reward errors too.
- Monitor chains of thought, but avoid optimising hard against the monitor.

PART V

# Preference Optimisation without a Separate Reward Model

CHAPTER 22

# Direct Preference Optimisation

**Why this chapter matters.**
**DPO** (Direct Preference Optimization) replaced PPO-RLHF in many open-model pipelines because it trains on preference pairs with a simple classification-style loss: no reward model, no critic, no generation during training. Its derivation is the most-asked maths question in post-training interviews. You should be able to go from the KL-regularised objective to the DPO loss in about ten lines, explain the gradient, and describe its failure modes.

## 22.1 Step 1: the optimal policy of KL-regularised RL

Start from the RLHF objective of Chapter 20, for a fixed prompt $x$:

$$\max_{\pi} \; \mathbb{E}_{y\sim\pi(\cdot|x)}[r(x,y)] - \beta\,\mathrm{KL}(\pi(\cdot\mid x)\,\|\,\pi_{\text{ref}}(\cdot\mid x)).$$

**Claim.** The maximiser is

$$\pi^*(y\mid x) = \frac{1}{Z(x)}\,\pi_{\text{ref}}(y\mid x)\,\exp\Big(\frac{1}{\beta}r(x,y)\Big), \qquad Z(x) = \sum_y \pi_{\text{ref}}(y\mid x)\exp\Big(\frac{1}{\beta}r(x,y)\Big).$$

$Z(x)$ is the **partition function**: the normalising constant that makes $\pi^*$ sum to 1. It is a sum over all possible responses, so it cannot be computed in practice.

**Proof.** Write the objective as an expectation under $\pi$ and divide by $-\beta$ (turning max into min):

$$\min_{\pi} \; \mathbb{E}_{y\sim\pi}\Big[\log\frac{\pi(y\mid x)}{\pi_{\text{ref}}(y\mid x)} - \frac{1}{\beta}r(x,y)\Big] = \min_{\pi} \; \mathbb{E}_{y\sim\pi}\Big[\log\frac{\pi(y\mid x)}{\pi_{\text{ref}}(y\mid x)\exp(r(x,y)/\beta)}\Big].$$

Multiply and divide inside the log by $Z(x)$:

$$= \min_{\pi} \; \mathbb{E}_{y\sim\pi}\Big[\log\frac{\pi(y\mid x)}{\pi^*(y\mid x)}\Big] - \log Z(x) = \min_{\pi} \; \mathrm{KL}(\pi\,\|\,\pi^*) - \log Z(x).$$

$Z(x)$ does not depend on $\pi$, and the KL is minimised (at zero) exactly when $\pi = \pi^*$. □

*Intuition.* The best policy is the reference policy **reweighted** by $\exp(r/\beta)$: responses with higher reward get exponentially more probability, and $\beta$ sets how sharply. Large $\beta$: stay near the reference. Small $\beta$: concentrate on the highest-reward responses.

## 22.2 Step 2: solve for the reward

Take logs of the closed form and rearrange:

$$r(x,y) = \beta\log\frac{\pi^*(y\mid x)}{\pi_{\text{ref}}(y\mid x)} + \beta\log Z(x).$$

**Notation on this page:** $x$: the prompt (input) • $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $D_{\text{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $y$: the response (output tokens); $y_t$ is token $t$ • $r(x,y)$: reward; $\hat{r}_\theta$: implicit reward $\beta\log(\pi_\theta/\pi_{\text{ref}})$ • $\pi^*$: the optimal policy • $Z(x)$: partition function (normaliser over all responses)

Any reward function can be written in terms of its own optimal policy. This reparameterisation is the key step.

## 22.3 Step 3: plug into Bradley-Terry

The Bradley-Terry model (Chapter 19) depends only on reward *differences*, and $\beta \log Z(x)$ is the same for both responses to the same prompt, so it cancels:

$$P(y_w \succ y_l \mid x) = \sigma(r(x, y_w) - r(x, y_l)) = \sigma\Big(\beta \log \frac{\pi^*(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta \log \frac{\pi^*(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)}\Big).$$

The intractable $Z(x)$ is gone. Preferences are now expressed directly in terms of the policy.

## 22.4 Step 4: the DPO loss

Replace $\pi^*$ with the trainable policy $\pi_\theta$ and maximise the likelihood of the observed preferences:

$$\mathcal{L}_{\text{DPO}}(\theta) = -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}}\Big[\log \sigma\Big(\beta \log \frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta \log \frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)}\Big)\Big].$$

This is the reward-model loss of Chapter 19 with the reward replaced by the **implicit reward**

$$\hat{r}_\theta(x, y) = \beta \log \frac{\pi_\theta(y \mid x)}{\pi_{\text{ref}}(y \mid x)}.$$

Hence the paper's title, *Direct Preference Optimization: Your Language Model is Secretly a Reward Model* (Rafailov et al., arXiv 2305.18290): training the policy with this loss is the same as fitting a reward model whose optimal policy is $\pi_\theta$.

Each $\log \pi_\theta(y \mid x)$ is a **sum** of token log-probs over the response (prompt tokens excluded), from a single forward pass, exactly as in SFT.

## 22.5 Step 5: the gradient

Let $u = \hat{r}_\theta(x, y_w) - \hat{r}_\theta(x, y_l)$. Using $\frac{d}{du}[-\log \sigma(u)] = -\sigma(-u)$:

$$\nabla_\theta \mathcal{L}_{\text{DPO}} = -\beta\, \mathbb{E}\Big[\underbrace{\sigma(\hat{r}_\theta(x, y_l) - \hat{r}_\theta(x, y_w))}_{\uparrow \text{ weight: how wrong the implicit RM is}} \Big(\underbrace{\nabla_\theta \log \pi_\theta(y_w \mid x)}_{\uparrow \text{ raise chosen}} - \underbrace{\nabla_\theta \log \pi_\theta(y_l \mid x)}_{\uparrow \text{ lower rejected}}\Big)\Big].$$

*Intuition.* Increase the chosen response's likelihood and decrease the rejected one's, weighted by how much the implicit reward model currently gets the pair wrong. Pairs that are already well separated get little weight. The DPO paper showed that without this weighting (plain "unlikelihood" on rejected responses) models degenerate.

*Worked example.* $\beta = 0.1$. Chosen: $\log \pi_\theta = -40$, $\log \pi_{\text{ref}} = -42$, so $\hat{r}_w = 0.2$. Rejected: $\log \pi_\theta = -35$, $\log \pi_{\text{ref}} = -34$, so $\hat{r}_l = -0.1$. Margin $u = 0.3$, loss $= -\log \sigma(0.3) = 0.554$, weight $= \sigma(-0.3) = 0.43$.

## 22.6 Code: the DPO loss

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $x$: the prompt (input) • $Z(x)$: partition function (normaliser over all responses) • $\pi_{\text{ref}}$: the frozen reference model • $\pi^*$: the optimal policy • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $a \succ b$: $a$ is preferred to $b$ • $y_w, y_l$: chosen (winning) and rejected (losing) responses • $y$: the response (output tokens); $y_t$ is token $t$ • $r(x, y)$: reward; $\hat{r}_\theta$: implicit reward $\beta \log(\pi_\theta / \pi_{\text{ref}})$ • *and 6 more: see the Notation table*

```
import torch, torch.nn.functional as F

def seq_logp(logits, labels, mask):
    """Sum of log-probs of `labels` under `logits`, over positions where mask = 1.
    logits: (B, T, V) already aligned so logits[:, t] predicts labels[:, t]."""
    logp = torch.log_softmax(logits.float(), dim=-1).gather(-1, labels[..., None]).squeeze(-1)
    return (logp * mask).sum(-1)

def dpo_loss(pc, pr, rc, rr, beta=0.1):
    """pc/pr: policy log-prob of chosen/rejected (B,); rc/rr: reference log-probs (B,)."""
    chosen_reward = beta * (pc - rc)
    rejected_reward = beta * (pr - rr)
    margin = chosen_reward - rejected_reward
    loss = -F.logsigmoid(margin).mean()
    return loss, chosen_reward.detach(), rejected_reward.detach()

# The worked example:
loss, cr, rj = dpo_loss(torch.tensor([-40.]), torch.tensor([-35.]),
    torch.tensor([-42.]), torch.tensor([-34.]))
print(f"loss {loss:.3f}  implicit rewards: chosen {cr.item():.2f}, rejected {rj.item():.2f}")

# A toy run: a "policy" of free logits over 5 tokens, 3-token responses.
torch.manual_seed(0)
V, T = 5, 3
ref_logits = torch.randn(T, V)
theta = ref_logits.clone().requires_grad_(True)
yw, yl = torch.tensor([[0, 1, 2]]), torch.tensor([[3, 3, 4]])
mask = torch.ones(1, T)
opt = torch.optim.SGD([theta], lr=0.5)
with torch.no_grad():
    rc, rr = seq_logp(ref_logits[None], yw, mask), seq_logp(ref_logits[None], yl, mask)
for step in range(30):
    pc, pr = seq_logp(theta[None], yw, mask), seq_logp(theta[None], yl, mask)
    loss, cr, rj = dpo_loss(pc, pr, rc, rr)
    opt.zero_grad(); loss.backward(); opt.step()
print(f"after training: loss {loss.item():.3f}, chosen logp change {(pc - rc).item():+.2f}, "
      f"rejected logp change {(pr - rr).item():+.2f}")
```

## 22.7 $\beta$ and the reference model

$\beta$ plays the same role as the KL coefficient in RLHF. The DPO paper used $\beta = 0.1$ for most experiments (0.5 for TL;DR summarisation), batch size 64, RMSprop at learning rate $10^{-6}$ with 150 warmup steps. TRL's `DPOConfig` defaults are `beta=0.1` and `learning_rate=1e-6`. Smaller $\beta$ lets the policy move further from the reference; very small values tend to overfit and hack the implicit reward.

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay)

**The reference model** is usually the SFT model the policy starts from. Options:

- Keep a frozen copy (2x model memory).
- **Precompute** reference log-probs for the whole dataset once, then drop the reference model (`precompute_ref_log_probs=True` in TRL).
- With LoRA, the reference is the base model with the adapter disabled (Chapter 10).
- **Update the reference** periodically (TR-DPO): TRL's `sync_ref_model=True` mixes the policy into the reference every `ref_model_sync_steps=512` steps with weight `ref_model_mixup_alpha=0.6`.

**Data must be close to the reference.** DPO assumes the preference pairs are plausible under $\pi_{\text{ref}}$. If chosen and rejected responses come from a very different model, first SFT the reference on the chosen responses (the DPO paper's recommendation when no SFT model is available), or the implicit rewards start far from meaningful values.

**Why this, not that? Why DPO, not PPO-RLHF?**

**Why.** DPO needs only the policy and a reference (or precomputed log-probs): no reward model, no critic, no generation in the loop. It is a stable supervised-style loss with few hyperparameters, so it is cheap, fast and reproducible.
**Trade-off.** It is **offline**: it never sees its own new outputs, so it cannot correct mistakes outside the dataset, and it suffers distribution shift. It can also exploit the dataset in odd ways (lower the probability of everything, prefer length).
**When the alternative wins.** Online RL (PPO, GRPO, online DPO) wins when you have a reward model or verifier, enough compute, and need the best possible quality, especially for reasoning, where the model must explore its own solutions.

## 22.8 What DPO does to likelihoods

A surprising, well-documented effect: during DPO, the log-probability of the **chosen** responses often **decreases** too. The loss only cares about the *gap*, and it is often easier to push the rejected response down a lot and the chosen one down a little. Where does the probability go? To other responses, sometimes ones that are worse than both. This is linked to:

- **Shared prefixes.** Chosen and rejected responses often share most tokens; pushing the rejected down also pushes the shared tokens down.
- **Overfitting to the pairs.** Training many epochs drives the margin up without improving generations.

Fixes include adding an SFT (negative log-likelihood) term on the chosen responses (RPO-style regularisation, Chapter 23), using a larger $\beta$, fewer epochs, or DPO variants with a likelihood-preserving term (Chapter 23).

**Notation on this page:** $\pi_{\text{ref}}$: the frozen reference model • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay)

## 22.9 Length exploitation

Since $\log \pi_\theta(y \mid x)$ is a *sum* over tokens, longer responses have larger-magnitude log-ratios, and if chosen responses tend to be longer, DPO learns "longer is better". *Disentangling Length from Quality in Direct Preference Optimization* (Park et al., arXiv 2403.19159) documented this and proposed a length regulariser. SimPO (Chapter 23) uses the length-normalised (average) log-prob instead; TRL's `ld_alpha` implements LD-DPO's down-weighting of the "verbose" part of longer responses.

## 22.10 What to monitor

TRL and most DPO code log:

- `rewards/chosen`, `rewards/rejected`: mean implicit rewards. Rejected should fall; watch whether chosen also falls a lot.
- `rewards/margins`: chosen minus rejected; should rise.
- `rewards/accuracies`: share of pairs with chosen reward > rejected. Close to 1.0 on the training set early on often means the task is too easy or overfitting.
- `logps/chosen`, `logps/rejected`: raw log-probs.
- And, crucially, **generation quality on held-out prompts** (win rate vs the SFT model by a judge), plus mean response length.

## 22.11 From pairs to rankings: the Plackett-Luce form

Sometimes annotators rank $K$ responses instead of comparing two. The **Plackett-Luce** model generalises Bradley-Terry: the probability of a full ranking $\tau$ (best first) is the chance of picking the top response from all $K$, then the next from the remaining $K-1$, and so on:

$$P(\tau \mid y_1, \ldots, y_K, x) = \prod_{k=1}^{K} \frac{\exp\left(r(x, y_{\tau(k)})\right)}{\sum_{j=k}^{K} \exp\left(r(x, y_{\tau(j)})\right)}.$$

Every factor is a softmax over the responses that are still unranked. The term $\beta \log Z(x)$ appears in every reward, so it cancels from every softmax, exactly as in the pairwise case. Substituting the implicit reward gives the ranking version of DPO, which the DPO paper derives in its appendix. With $K = 2$ it reduces to the ordinary DPO loss. In practice most teams instead split rankings into pairs (all $\binom{K}{2}$ pairs, as InstructGPT did for its reward model, or "best vs worst").

## 22.12 Why DPO overfits: the deterministic-preference argument

*A General Theoretical Paradigm to Understand Learning from Human Preferences* (Azar et al., arXiv 2310.12036) points out a weakness that follows directly from the derivation.

Suppose that, in the data, $y_w$ is **always** preferred to $y_l$ (true preference probability 1, common when each pair is labelled once). The Bradley-Terry likelihood $\sigma(u)$ keeps rising as the margin

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $K$: number of ranked responses • $\tau$: a ranking • $r(x, y)$: reward; $\hat{r}_\theta$: implicit reward $\beta \log(\pi_\theta / \pi_{\text{ref}})$ • $k$: an index or count (see text) • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $Z(x)$: partition function (normaliser over all responses) • $y_w, y_l$: chosen (winning) and rejected (losing) responses • *and 2 more: see the Notation table*

$u$ grows, so the maximum-likelihood margin is $u \to \infty$. In DPO terms,

$$\beta\Big(\log\frac{\pi_\theta(y_w)}{\pi_{\text{ref}}(y_w)} - \log\frac{\pi_\theta(y_l)}{\pi_{\text{ref}}(y_l)}\Big) \to \infty \quad\Longrightarrow\quad \frac{\pi_\theta(y_l)}{\pi_\theta(y_w)} \to 0,$$

**whatever the value of $\beta$.** $\beta$ only rescales the margin; it does not stop it from diverging. The KL regulariser that $\beta$ was supposed to provide holds only for the *exact* optimum over *all* responses; with a finite dataset nothing stops the policy from pushing $\pi_\theta(y_l)$ towards zero and moving far from the reference. In practice early stopping and the finite learning rate hide this, but it explains why DPO overfits with many epochs.

**IPO** (Identity Preference Optimisation) fixes it by regressing the log-ratio margin to a finite target instead of maximising a sigmoid:

$$\mathcal{L}_{\text{IPO}} = \Big(h_\theta(x, y_w, y_l) - \frac{1}{2\beta}\Big)^2, \qquad h_\theta = \log\frac{\pi_\theta(y_w)}{\pi_{\text{ref}}(y_w)} - \log\frac{\pi_\theta(y_l)}{\pi_{\text{ref}}(y_l)}.$$

Now $\beta$ really controls how far the policy moves: the margin settles at $1/(2\beta)$.

```
import torch, torch.nn.functional as F
torch.manual_seed(0)

# Three possible responses; reference is uniform. Data: response 0 always beats response
↪ 1.
ref = torch.log(torch.ones(3) / 3)
def run(kind, beta=0.1, steps=2000, lr=0.1):
    th = ref.clone().requires_grad_(True)
    opt = torch.optim.SGD([th], lr=lr)
    for _ in range(steps):
        lp = torch.log_softmax(th, -1)
        h = (lp[0] - ref[0]) - (lp[1] - ref[1])            # log-ratio margin
        loss = -F.logsigmoid(beta * h) if kind == "dpo" else (h - 1 / (2 * beta)) ** 2
        opt.zero_grad(); loss.backward(); opt.step()
    p = torch.softmax(th, -1).detach()
    kl = (p * (p.log() - ref)).sum()
    return h.item(), p, kl.item()

for kind in ("dpo", "ipo"):
    for steps in (500, 5000):
        h, p, kl = run(kind, steps=steps)
        print(f"{kind} {steps:5d} steps: margin {h:6.2f}, probs {[round(v, 3) for v in
        ↪ p.tolist()]}, KL {kl:.2f}")
```

The DPO margin keeps growing with more steps (the rejected response's probability heads to zero), while the IPO margin stops at $1/(2\beta) = 5$. Note also response 2, which never appeared in the data: its probability is set purely as a side effect of the updates (here it shrinks from 0.33 as DPO concentrates everything on response 0). In a real model with billions of possible responses, nothing in the loss controls where the freed mass goes.

**Notation on this page:** $u$: implicit-reward margin $\hat{r}_w - \hat{r}_l$ • $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $y_w, y_l$: chosen (winning) and rejected (losing) responses • $y$: the response (output tokens); $y_t$ is token $t$ • $\mathcal{L}$: a loss to minimise • $x$: the prompt (input) • $h_\theta$: log-ratio margin (IPO)

**Why this, not that? Why the sigmoid (Bradley-Terry) loss, not IPO's squared loss?**

**Why.** The sigmoid loss is the maximum-likelihood estimate under Bradley-Terry, the same loss as reward-model training, and it focuses its gradient on pairs the model still gets wrong (the $\sigma(-u)$ weight). It is the best-tested choice and usually works well with one or two epochs.
**Trade-off.** It never "finishes": with deterministic labels the margin is pushed to infinity, so it overfits with many epochs or tiny datasets, and $\beta$ does not really bound the drift.
**When the alternative wins.** IPO (TRL `loss_type="ipo"`) is safer for small or noisy datasets, many epochs, or when you need $\beta$ to act as a true leash. In practice TRL's IPO averages log-probs over tokens, so its $\beta$ is not comparable to DPO's.

## 22.13 The token-level view: DPO as a Q-function

DPO is derived for a "bandit": the whole response is one action. But the policy generates token by token. *From r to Q*: Your Language Model is Secretly a Q-Function* (Rafailov et al., arXiv 2404.12358) shows that in the token-level MDP (state = prompt plus tokens so far, action = next token), DPO can be derived as a form of inverse Q-learning. Write the implicit reward as a sum over tokens:

$$\hat{r}_\theta(x, y) = \sum_{t=1}^{|y|} \beta \log \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\text{ref}}(y_t \mid x, y_{<t})}.$$

Each term acts like a per-token advantage. Practical consequences:

- **Credit assignment.** After DPO, the per-token log-ratios tend to be large exactly at the tokens where chosen and rejected responses differ in quality (for example, the token where an error is introduced). You can plot them to see what DPO learned.
- **Search.** Likelihood-based search (beam search) on a DPO policy is a form of search on the implicit reward; the paper found beam search gave a meaningful improvement over sampling from the DPO policy.
- **Why the chosen likelihood drops.** Their analysis also explains the falling chosen likelihood: under the MDP view, DPO training starting from the SFT policy is expected to lower the implicit reward of the chosen responses.

## 22.14 Is DPO really better than PPO?

*Is DPO Superior to PPO for LLM Alignment? A Comprehensive Study* (Xu et al., arXiv 2404.10719) found, theoretically and in experiments, that DPO can find solutions that exploit **out-of-distribution** responses: because the loss only constrains the responses in the dataset, the policy can put high probability on responses that neither the data nor the reference covers. Iterating DPO with fresh on-policy data and a reward model reduced the problem. They then identified the factors that make PPO work well for LLMs (advantage normalisation, large batch sizes, and updating the reference model with an exponential moving average) and reported that well-tuned PPO beat DPO and other alignment methods on every testbed they tried, from dialogue to competitive code generation.

**Notation on this page:** $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $u$: implicit-reward margin $\hat{r}_w - \hat{r}_l$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $y_{<t}$: the tokens before position $t$ • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $r(x, y)$: reward; $\hat{r}_\theta$: implicit reward $\beta \log(\pi_\theta/\pi_{\text{ref}})$ • $t$: time step or token position

The interview answer: DPO is cheaper, simpler and great for preference tuning with a fixed dataset; well-tuned online RL is usually stronger when you can afford it and have a good reward signal; on-policy or iterative DPO (Chapter 24) sits in between.

## 22.15 Using TRL's `DPOTrainer`

The dataset needs `prompt`, `chosen` and `rejected` columns (conversational or plain text). The fields below are from `DPOConfig` in TRL 1.16.0; `max_length` (default 1024) is the total length of prompt plus completion, and `truncation_mode` accepts only `"keep_start"`. There is no separate `max_prompt_length` field in this version.

```
from datasets import load_dataset
from peft import LoraConfig
from trl import DPOConfig, DPOTrainer

ds = load_dataset("trl-lib/ultrafeedback_binarized", split="train")
config = DPOConfig(
    output_dir="dpo-out",
    beta=0.1,                       # KL strength (implicit reward scale)
    loss_type=["sigmoid"],          # standard DPO; ["sigmoid", "sft"] adds an NLL term
 ↪  (RPO-style)
    learning_rate=5e-7,             # full fine-tuning; LoRA usually needs ~10x more
    per_device_train_batch_size=4,
    gradient_accumulation_steps=16, # effective batch 64 per device
    num_train_epochs=1,
    max_length=2048,                # prompt + completion tokens
    precompute_ref_log_probs=False, # True: compute reference log-probs once, then drop
 ↪  the reference
    bf16=True,
)
trainer = DPOTrainer(
    model="Qwen/Qwen2.5-1.5B-Instruct",   # ideally your SFT model
    ref_model=None,                       # None: a frozen copy is made (or adapter
 ↪  disabled with PEFT)
    args=config,
    train_dataset=ds,
    peft_config=LoraConfig(r=16, lora_alpha=32, target_modules="all-linear"),
)
trainer.train()
```

## 22.16 Choosing the hyperparameters

| Setting | Typical range | Notes |
|---|---|---|
| $\beta$ | 0.01 – 0.5 (0.1 default) | Lower = more movement and more risk of hacking/overfitting |

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay)

| Setting | Typical range | Notes |
|---|---|---|
| Learning rate | $5 \times 10^{-7} - 10^{-6}$ full; $5 \times 10^{-6} - 5 \times 10^{-5}$ LoRA | Much lower than SFT; too high destroys chosen likelihood |
| Epochs | 1 – 3 | Margins keep growing after the model stops improving |
| Effective batch size | 32 – 128 pairs | Larger batches smooth the noisy pairwise signal |
| Max length | cover 95%+ of pairs | Truncated pairs give misleading log-prob differences |

**Sweep $\beta$ and learning rate together**: they interact, because the effective step on the margin scales with both.

**Why this, not that? Why precompute reference log-probs, not keep the reference model loaded?**

**Why.** The reference is frozen, so its log-probs for each (prompt, response) pair never change. Computing them once frees a whole model's worth of GPU memory and halves forward passes during training.
**Trade-off.** It needs an extra pass over the dataset before training, the results must be stored, and it rules out changing the reference during training (TR-DPO's `sync_ref_model`) or data augmentation that changes the token sequences.
**When the alternative wins.** Keep the live reference when the dataset is streamed or re-generated (online/iterative DPO), when you use `sync_ref_model`, or with LoRA, where the reference is free (disable the adapter).

## 22.17 A worked example of the gradient weight

The weight $\sigma(\hat{r}_l - \hat{r}_w)$ decides how much each pair matters. With $\beta = 0.1$:

| Margin $u$ (in reward units) | Loss $-\log \sigma(u)$ | Weight $\sigma(-u)$ |
|---|---|---|
| $-2$ (badly wrong) | 2.13 | 0.88 |
| 0 (undecided, e.g. at the start) | 0.69 | 0.50 |
| 0.3 (the example above) | 0.55 | 0.43 |
| 2 (well separated) | 0.13 | 0.12 |
| 5 | 0.007 | 0.007 |

At initialisation $\pi_\theta = \pi_{\text{ref}}$, so every margin is 0, the loss is $\log 2 = 0.693$ and every pair gets weight 0.5. **If your first logged DPO loss is not 0.693, something is wrong** (the reference differs from the starting policy, dropout is on, or prompt tokens are included inconsistently). This is a great sanity check to mention in interviews.

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $r(x, y)$: reward; $\hat{r}_\theta$: implicit reward $\beta \log(\pi_\theta / \pi_{\text{ref}})$ • $u$: implicit-reward margin $\hat{r}_w - \hat{r}_l$ • $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights

## 22.18 Common mistakes

- Including prompt tokens in the sequence log-prob.
- Averaging log-probs over tokens when using the standard DPO loss (that is a different method: SimPO-style).
- Leaving dropout on (TRL disables it by default with `disable_dropout=True`).
- Training on pairs that come from a model very different from the reference without first doing SFT.
- Judging success by the training reward accuracy.
- Starting the policy from a different checkpoint than the reference (first loss not $\log 2$).
- Using an SFT-level learning rate ($10^{-5}$) for full-parameter DPO.
- Running many epochs and celebrating rising margins (deterministic-preference overfitting).
- Comparing $\beta$ values across DPO and IPO (or SimPO) as if they meant the same thing.

**Interview check**

**Q1. Derive the optimal policy of KL-regularised RL.** Rewrite the objective as $\min_\pi \mathrm{KL}(\pi \| \pi_{\text{ref}} e^{r/\beta}/Z) - \log Z$; minimum at $\pi^* = \pi_{\text{ref}} e^{r/\beta}/Z$.
**Q2. Why does $Z(x)$ disappear in DPO?** Bradley-Terry uses the reward difference for the same prompt, and $\beta \log Z(x)$ is shared by both responses.
**Q3. Write the DPO loss.** $-\log \sigma(\beta \log \frac{\pi_\theta(y_w)}{\pi_{\text{ref}}(y_w)} - \beta \log \frac{\pi_\theta(y_l)}{\pi_{\text{ref}}(y_l)})$.
**Q4. What is the implicit reward?** $\beta \log(\pi_\theta(y|x)/\pi_{\text{ref}}(y|x))$.
**Q5. Explain the DPO gradient.** Raise chosen and lower rejected log-likelihood, weighted by $\sigma(\hat{r}_l - \hat{r}_w)$, how wrong the implicit reward currently is.
**Q6. Why might the chosen response's likelihood go down during DPO?** The loss depends only on the margin; reducing both, with rejected falling faster, satisfies it. Shared tokens and overfitting make it worse.
**Q7. What does $\beta$ control?** How far the policy can move from the reference (KL strength); small $\beta$ = more deviation.
**Q8. How can you avoid keeping a reference model in memory?** Precompute reference log-probs, or use LoRA and disable the adapter.
**Q9. Is DPO on-policy or off-policy?** Offline/off-policy: it trains on a fixed dataset of pairs.
**Q10. What should the DPO loss be at step 0, and why?** $\log 2 \approx 0.693$: the policy equals the reference, so every implicit reward and margin is 0 and $-\log \sigma(0) = \log 2$.
**Q11. Why can DPO overfit even with a KL-regularised derivation?** With deterministic preferences the Bradley-Terry optimum has infinite margin regardless of $\beta$; the KL leash holds only at the exact optimum over all responses, not on a finite dataset (Azar et al.). IPO targets a finite margin $1/(2\beta)$.
**Q12. How does DPO handle rankings of more than two responses?** Through the Plackett-Luce model, where $Z$ cancels from every softmax factor; in practice rankings are often split into pairs.
**Q13. What is the token-level interpretation of DPO?** In the token MDP, DPO is a form of inverse Q-learning; per-token log-ratios $\beta \log(\pi_\theta/\pi_{\text{ref}})$ act like token advantages, giving credit assignment (Rafailov et al., From r to Q*).

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\pi_{\text{ref}}$: the frozen reference model • $D_{\text{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $r(x, y)$: reward; $\hat{r}_\theta$: implicit reward $\beta \log(\pi_\theta/\pi_{\text{ref}})$ • $Z(x)$: partition function (normaliser over all responses) • $\pi^*$: the optimal policy • $\pi_\theta$: the policy (the model), with weights $\theta$ • $x$: the prompt (input) • $\theta$: the model's trainable weights • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $y_w$, $y_l$: chosen (winning) and rejected (losing) responses • *and 1 more: see the Notation table*

**Q14. Give a theoretical weakness of DPO versus PPO.** DPO only constrains responses in the dataset, so it can put mass on out-of-distribution responses; PPO's on-policy samples are scored by the reward model (Xu et al.).
**Q15. Which TRL setting adds an SFT term to DPO in TRL 1.16?** `loss_type=["sigmoid", "sft"]` with `loss_weights=[1.0, alpha]`; there is no `rpo_alpha` field.
**Q16. Where does probability mass go when both chosen and rejected likelihoods fall?** To other responses outside the pair, including unseen ones, which may be better or worse; monitor generations, not only margins.

**Key takeaways**

- KL-regularised RL has optimum $\pi^* \propto \pi_{\text{ref}} \exp(r/\beta)$.
- Reparameterise $r = \beta \log(\pi^*/\pi_{\text{ref}}) + \beta \log Z$; $Z$ cancels in Bradley-Terry.
- DPO loss $= -\log \sigma(\beta[\text{log-ratio chosen} - \text{log-ratio rejected}])$; summed token log-probs.
- Gradient: up chosen, down rejected, weighted by how wrong the implicit RM is.
- Pitfalls: chosen likelihood drops, length exploitation, distribution mismatch; monitor generations, not just margins.
- First-step loss must be $\log 2$; the gradient weight is $\sigma(-u)$.
- With deterministic labels DPO's margin diverges regardless of $\beta$; IPO targets $1/(2\beta)$.
- Well-tuned PPO can beat DPO; DPO can exploit out-of-distribution responses.

**Notation on this page:** $\pi_{\text{ref}}$: the frozen reference model • $\pi^*$: the optimal policy • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $r(x, y)$: reward; $\hat{r}_\theta$: implicit reward $\beta \log(\pi_\theta/\pi_{\text{ref}})$ • $Z(x)$: partition function (normaliser over all responses) • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $u$: implicit-reward margin $\hat{r}_w - \hat{r}_l$

CHAPTER 23

# DPO Variants

**Why this chapter matters.**
Dozens of DPO variants exist. Each one fixes a specific weakness: overfitting to deterministic preferences (IPO), noisy labels (cDPO, robust DPO), the need for paired data (KTO), the need for a reference model (ORPO, SimPO), or falling likelihoods (RPO, APO). Interviewers rarely expect every formula, but they do expect you to know what problem each one solves and to write the main ones. TRL exposes many as `loss_type` options of `DPOConfig`, which makes comparisons easy.

Throughout, write the per-response log-ratio and the DPO "margin" as

$$h_w = \log \frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)}, \quad h_l = \log \frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)}, \quad u = h_w - h_l.$$

Standard DPO is $-\log \sigma(\beta u)$.

## 23.1 IPO: fixing overfitting to certain preferences

*A General Theoretical Paradigm to Understand Learning from Human Preferences* (Azar et al., arXiv 2310.12036) points out a problem. If the data says $y_w$ beats $y_l$ every time (a deterministic preference), the Bradley-Terry probability is 1, so the ideal reward gap is infinite. DPO then pushes $u \to \infty$ and the KL regularisation stops working: the policy can drift arbitrarily far. With finite data, many preferences *look* deterministic.

IPO replaces the log-sigmoid with a squared loss towards a **finite target margin**:

$$\mathcal{L}_{\text{IPO}} = \left(u - \frac{1}{2\tau}\right)^2,$$

where $\tau$ is the regularisation strength (it plays the role of $\beta$). The margin is pushed to $1/(2\tau)$ and no further. In TRL (`loss_type="ipo"`), `beta` is $\tau$, and the log-probs are **averaged** over completion tokens before taking the difference (TRL's code comments say this matches the IPO authors' own runs).

## 23.2 Noisy labels: cDPO and robust DPO

Human preference labels are noisy: perhaps 20 to 30% of pairs would be flipped by another labeller. Suppose each label is flipped with probability $\epsilon$.

**Conservative DPO (cDPO)** uses label smoothing: treat each pair as "chosen wins" with probability $1 - \epsilon$:

$$\mathcal{L}_{\text{cDPO}} = -(1-\epsilon)\log \sigma(\beta u) - \epsilon \log \sigma(-\beta u).$$

**Notation on this page:** $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $y_w, y_l$: chosen (winning) and rejected (losing) responses • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $h$: log-ratio margin • $u$: margin between implicit rewards • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\tau$: a trajectory (states, actions, rewards of one episode) • *and 2 more: see the Notation table*

Its optimum has a finite margin, which also limits overfitting.

**Robust DPO**, from *Provably Robust DPO: Aligning Language Models with Noisy Feedback* (Chowdhury et al., arXiv 2403.00409), builds an *unbiased* estimate of the clean loss:

$$\mathcal{L}_{\text{rDPO}} = \frac{(1-\epsilon)(-\log\sigma(\beta u)) - \epsilon(-\log\sigma(-\beta u))}{1-2\epsilon}.$$

In TRL 1.16.0 this is `loss_type="robust"` with `label_smoothing` $= \epsilon \in [0, 0.5)$. Note that the plain `"sigmoid"` loss in this version ignores `label_smoothing`; the smoothed form is used by `"robust"`, `"exo_pair"` and `"aot"`.

## 23.3 KTO: no pairs needed

*KTO: Model Alignment as Prospect Theoretic Optimization* (Ethayarajh et al., arXiv 2402.01306) works with **unpaired** binary feedback: each example is one response labelled "desirable" or "undesirable" (thumbs up / down), which is much easier to collect than pairs. It borrows from **prospect theory** (Kahneman and Tversky): people judge outcomes relative to a reference point, and losses hurt more than equal gains please.

With $r_\theta(x,y) = \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}$ and a reference point $z_0$,

$$v(x,y) = \begin{cases} \lambda_D\,\sigma(\beta(r_\theta(x,y) - z_0)) & \text{if } y \text{ is desirable} \\ \lambda_U\,\sigma(\beta(z_0 - r_\theta(x,y))) & \text{if } y \text{ is undesirable} \end{cases}, \qquad \mathcal{L}_{\text{KTO}} = \mathbb{E}[\lambda_y - v(x,y)].$$

$z_0$ is an estimate of $\text{KL}(\pi_\theta \| \pi_{\text{ref}})$. In practice it is estimated within a micro-batch from *mismatched* pairs $(x_i, y_{i+1})$, clamped at zero, and no gradient flows through it. $\lambda_D$ and $\lambda_U$ weight desirable and undesirable examples; the paper recommends choosing them so that $\lambda_D n_D / (\lambda_U n_U) \in [1, 1.5]$, where $n_D$, $n_U$ count the examples of each kind (with 1 desirable per 10 undesirable, set $\lambda_U = 1$, $\lambda_D \in [10, 15]$). TRL: `KTOConfig(beta=0.1, desirable_weight=1.0, undesirable_weight=1.0)`.

**Why this, not that? Why KTO, not DPO, when you have thumbs-up / thumbs-down data?**

**Why.** KTO uses unpaired binary labels directly, so you can train on product feedback without constructing pairs, and it handles imbalanced data with $\lambda_D$, $\lambda_U$. The paper reports it matching or beating DPO at the 1B to 30B scale.
**Trade-off.** Binary labels carry less information than comparisons, the KL reference point is a rough estimate, and there are extra hyperparameters.
**When the alternative wins.** When you already have high-quality paired comparisons of responses to the same prompt, DPO (or a variant) uses that relative signal more directly.

## 23.4 ORPO: no reference model, one stage

*ORPO: Monolithic Preference Optimization without Reference Model* (Hong et al., arXiv 2403.07691) merges SFT and preference learning into one stage and drops the reference model. With the

**Notation on this page:** $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\varepsilon$: assumed label-flip rate (cDPO / robust DPO) • $\mathcal{L}$: a loss to minimise • $u$: margin between implicit rewards • $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $r(x,y)$: the reward for response $y$ to prompt $x$ • *and 5 more: see the Notation table*

length-averaged probability $P_\theta(y \mid x) = \exp\left(\frac{1}{|y|}\sum_t \log \pi_\theta(y_t \mid x, y_{<t})\right)$ and the **odds** $\text{odds}_\theta(y \mid x) = \frac{P_\theta(y|x)}{1-P_\theta(y|x)}$:

$$\mathcal{L}_{\text{ORPO}} = \mathcal{L}_{\text{SFT}}(y_w) + \lambda\, \mathcal{L}_{\text{OR}}, \qquad \mathcal{L}_{\text{OR}} = -\log \sigma\Big(\log \frac{\text{odds}_\theta(y_w \mid x)}{\text{odds}_\theta(y_l \mid x)}\Big).$$

The SFT term teaches the format; the odds-ratio term pushes the rejected style down. TRL has it in `trl.experimental.orpo`, where `ORPOConfig(beta=0.1)` is the paper's $\lambda$.

## 23.5 SimPO: length-normalised, no reference

*SimPO: Simple Preference Optimization with a Reference-Free Reward* (Meng et al., arXiv 2405.14734) uses the **average** log-probability as the implicit reward, so it matches how generation is scored and does not reward length, and adds a target margin $\gamma$:

$$\mathcal{L}_{\text{SimPO}} = -\log \sigma\Big(\frac{\beta}{|y_w|}\log \pi_\theta(y_w \mid x) - \frac{\beta}{|y_l|}\log \pi_\theta(y_l \mid x) - \gamma\Big).$$

No reference model is needed, saving memory and compute. The paper recommends $\beta$ between 2.0 and 2.5 and $\gamma$ between 0.5 and 1.5 (larger values of $\beta$ than DPO, because average log-probs are much smaller in magnitude than summed ones). In TRL it is `trl.experimental.cpo.CPOConfig(loss_type="simpo", cpo_alpha=0.0, simpo_gamma=...)`.

**Why this, not that? Why a reference-free method (SimPO, ORPO), not DPO with a reference model?**

**Why.** No second model in memory or precomputation pass; the length-normalised reward of SimPO directly fights length exploitation; ORPO even removes the separate SFT stage.
**Trade-off.** Without a reference, nothing explicitly anchors the policy to its starting point, so it can drift, forget, or degrade fluency, especially with long training or a small $\beta$. Hyperparameters $(\beta, \gamma)$ are more sensitive.
**When the alternative wins.** DPO with a reference is safer when you care about preserving the SFT model's behaviour, when training many epochs, or when the preference data is far from the model's own outputs.

## 23.6 RPO, APO and other likelihood fixes

**NLL-regularised DPO (RPO).** Add a negative log-likelihood term on the chosen response: $\mathcal{L} = \mathcal{L}_{\text{DPO}} + \alpha\, \mathcal{L}_{\text{NLL}}(y_w)$. *Iterative Reasoning Preference Optimization* (Pang et al., arXiv 2404.19733) found this term essential for reasoning tasks, where DPO alone lowered the chosen likelihood and hurt accuracy. In TRL 1.16.0, there is no `rpo_alpha`; combine losses instead: `loss_type=["sigmoid", "sft"]` with `loss_weights=[1.0, alpha]`.

**APO.** *Anchored Preference Optimization and Contrastive Revisions* (D'Oosterlinck et al., arXiv 2408.06266) makes the direction explicit. `apo_zero` raises chosen and lowers rejected likelihoods (use when chosen responses are better than your model's outputs); `apo_down` lowers both but the rejected more (use when even chosen responses are worse than your model).

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $y_{<t}$: the tokens before position $t$ • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $t$: time step or token position • $P_\theta$: exp of the average token log-prob (ORPO) • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $\lambda$: ORPO weight; $\lambda_D, \lambda_U$: KTO desirable/undesirable weights • $\mathcal{L}$: a loss to minimise • $y_w$, $y_l$: chosen (winning) and rejected (losing) responses • *and 3 more: see the Notation table*

**Hinge (SLiC).** `loss_type="hinge"`: $\max(0, 1-\beta u)$, from *SLiC-HF: Sequence Likelihood Calibration with Human Feedback* (Zhao et al., arXiv 2305.10425).

**BCO.** *Binary Classifier Optimization for Large Language Model Alignment* (Jung et al., arXiv 2404.04656) trains the implicit reward as a binary classifier; `bco_pair` in DPO, and an unpaired version exists too.

**WPO.** *WPO: Enhancing RLHF with Weighted Preference Optimization* (Zhou et al., arXiv 2406.11827) reweights off-policy pairs by how likely the current policy is to produce them, making offline data behave more like on-policy data. TRL: `use_weighting=True`.

## 23.7 One comparison table

| Method | Data | Reference? | Loss core | Fixes | TRL |
|---|---|---|---|---|---|
| DPO | pairs | yes | $-\log \sigma(\beta u)$ | (baseline) | `sigmoid` |
| IPO | pairs | yes | $(u - \frac{1}{2\tau})^2$ | overfitting to certain prefs | `ipo` |
| cDPO | pairs | yes | smoothed log-sigmoid | label noise | (via `robust` / `exo_pair`) |
| Robust DPO | pairs | yes | unbiased noisy-label loss | label noise | `robust` |
| Hinge/SLiC | pairs | yes | $\max(0, 1-\beta u)$ | simplicity | `hinge` |
| KTO | unpaired | yes | prospect-theory value | no pairs | `KTOTrainer` |
| ORPO | pairs | no | SFT + odds ratio | one stage, no ref | experimental |
| SimPO | pairs | no | avg log-prob margin $-\gamma$ | length, no ref | experimental CPO |
| RPO | pairs | yes | DPO + NLL | falling likelihood | `["sigmoid", "sft"]` |
| APO | pairs | yes | anchored terms | likelihood direction | `apo_zero` / `apo_down` |
| WPO | pairs | yes | weighted DPO | off-policy data | `use_weighting` |

## 23.8 Code: several losses side by side

```
import torch, torch.nn.functional as F

def losses(hw, hl, beta=0.1, tau=0.1, eps=0.1):
    u = hw - hl
```

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $u$: margin between implicit rewards • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $\tau$: a trajectory (states, actions, rewards of one episode) • $\gamma$: SimPO target margin

```
    return {
        "dpo":    -F.logsigmoid(beta * u),
        "ipo":    (u - 1 / (2 * tau)) ** 2,
        "cdpo":   -(1 - eps) * F.logsigmoid(beta * u) - eps * F.logsigmoid(-beta * u),
        "robust": ((1 - eps) * -F.logsigmoid(beta * u) - eps * -F.logsigmoid(-beta * u))
         ↪  / (1 - 2 * eps),
        "hinge":  torch.relu(1 - beta * u),
    }

# How each loss treats an already-large margin: does it keep pushing?
for u in (0.0, 5.0, 20.0, 60.0):
    hw = torch.tensor(u, requires_grad=True)
    row = []
    for name, l in losses(hw, torch.tensor(0.0)).items():
        g, = torch.autograd.grad(l, hw)
        row.append(f"{name}:{-g.item():+.3f}")
    print(f"margin {u:5.1f} -> push on margin  " + "  ".join(row))
```

Read the output as "how hard does each loss still push the margin up?". DPO's push shrinks but never reaches zero. IPO pushes the margin towards $1/(2\tau) = 5$ and *pulls it back* beyond that. cDPO's push turns negative past its finite optimum. Hinge stops completely once $\beta u \geq 1$. Robust DPO keeps pushing: it is an unbiased estimate of the *clean* DPO loss, so it corrects the noise without capping the margin.

## 23.9 One picture: every pairwise loss is a function of the margin

Most of these methods compute the same **margin** $u$ between chosen and rejected implicit rewards and differ only in the function applied to it (and in how the rewards are defined: summed vs averaged log-probs, with or without a reference). What really distinguishes them is the **push** $-\partial\ell/\partial u$: how hard each pair is pushed at a given margin.

```
import torch, torch.nn.functional as F

# Each pairwise loss is a function of the margin u (reward units). Compare value and
 ↪  gradient weight -dl/du.
beta = 0.1
losses = {
    "sigmoid (DPO)": lambda u: -F.logsigmoid(u),
    "hinge (SLiC)":  lambda u: torch.relu(1 - u),
    "IPO":           lambda u: (u - 0.5) ** 2,      # (h - 1/(2 beta))^2 with u = beta*h,
     ↪  rescaled by beta^2
    "SimPO (gamma=0.5)": lambda u: -F.logsigmoid(u - 0.5),
}
print(f"{'margin u':>18}" + "".join(f"{u:>9.1f}" for u in (-2, 0, 0.5, 2, 5)))
for name, f in losses.items():
    vals, grads = [], []
    for u0 in (-2, 0, 0.5, 2, 5):
        u = torch.tensor(float(u0), requires_grad=True)
```

**Notation on this page:** $\tau$: a trajectory (states, actions, rewards of one episode) • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $u$: margin between implicit rewards

```
        l = f(u); l.backward()
        vals.append(l.item()); grads.append(-u.grad.item())
    print(f"{name:>18}" + "".join(f"{v:9.3f}" for v in vals) + "   loss")
    print(f"{'':>18}" + "".join(f"{g:9.3f}" for g in grads) + "   push (-dl/du)")
```

Read the "push" rows:

- **Sigmoid (DPO)** pushes every pair, a lot when wrong and a little when right, but **never zero**: margins grow forever.
- **Hinge** pushes with constant force until the margin reaches 1, then **stops**: a "good enough" rule.
- **IPO** pushes towards the target and **pulls back** when the margin overshoots (negative push at $u = 2$ and 5): a true regression to a target.
- **SimPO's** margin $\gamma$ shifts the sigmoid right, so pairs keep a strong push until they are separated by at least $\gamma$.

This "margin function" view is how *Generalized Preference Optimization* (Tang et al., arXiv 2402.05749) unifies offline preference losses: choosing the convex function is choosing how the method regularises.

## 23.10 KTO's view: prospect theory and HALOs

*KTO: Model Alignment as Prospect Theoretic Optimization* (Ethayarajh et al., arXiv 2402.01306) borrows from **prospect theory** (Tversky and Kahneman): people judge outcomes relative to a **reference point**, and they are more sensitive to losses than to gains of the same size (**loss aversion**). The paper defines **human-aware losses (HALOs)**: losses that apply a value function to the implicit reward measured relative to a reference point. It shows that DPO and PPO-Clip are HALOs and finds that, among existing methods, HALOs tended to work better.

KTO's value for a desirable output is $\lambda_D\,\sigma(\beta(\hat{r}_\theta(x,y) - z_0))$ and for an undesirable one $\lambda_U\,\sigma(\beta(z_0 - \hat{r}_\theta(x,y)))$, with $z_0$ an estimate of the KL between the policy and the reference (the reference point, computed from mismatched prompt/response pairs and not back-propagated). Because the sigmoid saturates, outputs far from the reference point get little gradient, which the authors argue makes KTO robust to noisy and imbalanced data. Reported findings: SFT+KTO was competitive with SFT+DPO from 1B to 30B; KTO matched DPO while using **up to 90% fewer desirable examples**; and with a good enough base model, KTO could **skip SFT**, while DPO without SFT tended to ramble.

## 23.11 ORPO's odds ratio, worked out

ORPO uses the **odds** of a response, $\text{odds}_\theta(y \mid x) = \frac{P_\theta(y|x)}{1-P_\theta(y|x)}$, with $P_\theta$ the exponentiated *average* token log-probability, and adds a log-odds-ratio term to the SFT loss on the chosen response:

$$\mathcal{L}_{\text{ORPO}} = \mathcal{L}_{\text{NLL}}(y_w) + \lambda\Big[-\log\sigma\Big(\log\frac{\text{odds}_\theta(y_w \mid x)}{\text{odds}_\theta(y_l \mid x)}\Big)\Big].$$

**Notation on this page:** $u$: margin between implicit rewards • $\gamma$: SimPO target margin • $\theta$: the model's trainable weights • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\lambda$: ORPO weight; $\lambda_D, \lambda_U$: KTO desirable/undesirable weights • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $r(x,y)$: the reward for response $y$ to prompt $x$ • $z_0$: KTO reference point (KL estimate) • $P_\theta$: exp of the average token log-prob (ORPO) • *and 2 more: see the Notation table*

*Worked example.* Average log-probs $-0.5$ (chosen) and $-0.9$ (rejected) give $P_w = 0.607$, $P_l = 0.407$; log-odds 0.434 and $-0.377$; log odds ratio 0.811; odds-ratio loss $-\log \sigma(0.811) = 0.368$.

```
def log_odds(avg_logp):
    return avg_logp - torch.log1p(-avg_logp.exp())          # log(P / (1 - P)) with P =
     ↪ exp(avg log-prob)
lw, ll = torch.tensor(-0.5), torch.tensor(-0.9)             # average log-prob of chosen
 ↪ / rejected
ratio_term = -F.logsigmoid(log_odds(lw) - log_odds(ll))
print(f"ORPO: P_w={lw.exp():.3f}, P_l={ll.exp():.3f}, log odds ratio {(log_odds(lw) -
 ↪ log_odds(ll)).item():.3f}, "
      f"L_OR={ratio_term.item():.3f}")
```

The ORPO paper (arXiv 2403.07691) argues the odds ratio gives a *mild* penalty to the rejected style, which is enough during SFT, whereas a plain probability ratio pushes rejected responses down too hard. In TRL, ORPO's $\lambda$ is the `beta` field of `trl.experimental.orpo.ORPOConfig` (default 0.1).

**Why this, not that? Why average log-probs (SimPO, ORPO), not summed log-probs (DPO)?**

**Why.** The summed log-prob of a long response is large in magnitude simply because it has more tokens, so margins and gradients scale with length and the method can learn "longer is better". The average log-prob measures how likely each token is, which matches how decoding scores text and removes the length lever.
**Trade-off.** Averaging changes the theory: the implicit reward is no longer the one whose optimal policy solves the KL-regularised objective, so the clean link to RLHF is lost; it can also under-reward genuinely necessary length.
**When the alternative wins.** Summed log-probs (DPO) keep the exact derivation and the token-level credit-assignment interpretation, and are fine when chosen and rejected lengths are balanced or when length is controlled separately (LD-DPO's `ld_alpha`).

## 23.12 Choosing a method in practice

1. **Paired data, a good SFT model, moderate data size:** DPO (sigmoid), $\beta \approx 0.1$, one or two epochs. The default.
2. **Few or noisy labels, many epochs:** IPO or robust DPO.
3. **Only thumbs-up/down, or imbalanced data:** KTO.
4. **Memory-limited, single stage from a base model:** ORPO; reference-free and length-normalised: SimPO.
5. **Reasoning/maths preference pairs:** DPO + NLL term (RPO-style, `loss_type=["sigmoid","sft"]`), because chosen likelihood must not fall.
6. **Any of the above, and you can afford generation:** make it **online/iterative** (Chapter 24); fresh pairs matter more than the loss shape.

**Notation on this page:** $P_\theta$: exp of the average token log-prob (ORPO) • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $\lambda$: ORPO weight; $\lambda_D, \lambda_U$: KTO desirable/undesirable weights • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay)

**Interview check**

**Q1. What problem does IPO solve?** With deterministic preferences DPO drives the margin to infinity and ignores the KL regularisation; IPO's squared loss targets a finite margin $1/(2\tau)$.
**Q2. How does cDPO handle label noise?** Label smoothing: target probability $1-\epsilon$ for "chosen wins", giving a finite optimal margin.
**Q3. What data does KTO need and what is $z_0$?** Unpaired desirable/undesirable responses; $z_0$ is a batch estimate of $\mathrm{KL}(\pi_\theta \| \pi_{\text{ref}})$ from mismatched pairs, clamped at 0, with no gradient.
**Q4. How does SimPO differ from DPO?** No reference model; length-averaged log-prob as reward; a target margin $\gamma$; larger $\beta$.
**Q5. What is ORPO's loss?** SFT loss on chosen plus $\lambda$ times $-\log\sigma$(log odds ratio of chosen vs rejected).
**Q6. Why add an NLL term to DPO?** To stop the chosen likelihood from falling; important for reasoning tasks (Iterative RPO).
**Q7. How do you get RPO behaviour in TRL 1.16?** `loss_type=["sigmoid", "sft"]` with `loss_weights`.
**Q8. When would you use `apo_down`?** When even the chosen responses are worse than your model's typical outputs.
**Q9. Compare how DPO, hinge and IPO losses treat a pair with a large positive margin.** DPO still pushes a little (never zero); hinge stops once the margin is above 1; IPO pulls the margin back towards its target.
**Q10. What is a HALO?** A human-aware loss: a value function applied to the implicit reward relative to a reference point (prospect theory); DPO and PPO-Clip are examples, KTO is designed as one.
**Q11. What is the reference point $z_0$ in KTO and how is it computed?** An estimate of the policy-reference KL, from mismatched prompt/response pairs in the batch, clamped at zero and not back-propagated.
**Q12. Write the ORPO loss.** $\mathcal{L}_{\text{NLL}}(y_w) + \lambda[-\log\sigma(\log\text{odds}(y_w) - \log\text{odds}(y_l))]$, with odds $= P/(1-P)$ from the average log-prob.
**Q13. Why does SimPO need a larger $\beta$ than DPO?** Its rewards are average log-probs, much smaller in magnitude than summed ones, so a larger scale (2.0–2.5) is needed for a useful margin.
**Q14. Which variant would you choose for imbalanced thumbs-up/down feedback?** KTO, with $\lambda_D, \lambda_U$ set so $\lambda_D n_D/(\lambda_U n_U)$ is in $[1, 1.5]$.

**Key takeaways**

- IPO: squared loss to a finite margin. cDPO/robust DPO: noisy labels.
- KTO: unpaired binary feedback with prospect-theory weighting.
- ORPO and SimPO: no reference model; SimPO length-normalises and adds a margin $\gamma$.
- RPO/APO: control likelihood direction; WPO reweights off-policy pairs.
- In TRL, most are `DPOConfig.loss_type` values; KTO has its own trainer; ORPO and SimPO (via CPO) are experimental.

**Notation on this page:** $\tau$: a trajectory (states, actions, rewards of one episode) • $\varepsilon$: assumed label-flip rate (cDPO / robust DPO) • $z_0$: KTO reference point (KL estimate) • $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $D_{\text{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $\gamma$: SimPO target margin • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\lambda$: ORPO weight; $\lambda_D, \lambda_U$: KTO desirable/undesirable weights • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • *and 5 more: see the Notation table*

- All pairwise losses are $f(\text{margin})$; the "push" $-f'(u)$ decides overfitting and robustness.
- KTO = prospect-theory value with reference point $z_0$; works with unpaired, imbalanced data.
- ORPO = NLL + log-odds-ratio penalty, no reference; SimPO = average log-probs + margin.

CHAPTER 24

# Online and Iterative Preference Learning

**Why this chapter matters.**
Offline DPO trains on responses from some other model, collected once. Most strong post-training pipelines instead run several **rounds**: generate with the current model, label, train, repeat. This "iterative" or "online" preference learning closes much of the gap between DPO and full RL at a fraction of the engineering cost. Llama 3, for example, used rounds of rejection sampling, SFT and DPO.

## 24.1 Offline vs online data

**Offline (off-policy) data**: preference pairs were generated by a different model (or an older version of yours). The policy is trained to prefer $y_w$ over $y_l$, but neither may look like what it would actually write. Learning is indirect, and the policy can increase the probability of responses that appear nowhere in the data, where nothing constrains them.

**Online (on-policy) data**: the pairs are sampled from the *current* policy and labelled (by humans, a reward model or a judge). Each update addresses the policy's actual behaviour.

*Direct Language Model Alignment from Online AI Feedback* (Guo et al., arXiv 2402.04792) ran DPO, IPO and SLiC in an online loop: at each step, sample two responses from the current policy, have an LLM annotator pick the better one, and update. Online versions beat their offline counterparts clearly in their human evaluations.

**Why this, not that? Why fresh on-policy pairs, not a larger offline dataset?**

**Why.** On-policy pairs compare things the model would actually say, so every update fixes a real behaviour, and the implicit reward is only queried where the policy has mass. This removes most of the distribution-shift problems of offline DPO.
**Trade-off.** You need a labeller in the loop (reward model, judge or humans), generation compute, and a more complex pipeline; label quality now limits you directly, and a judge can be exploited like any reward model.
**When the alternative wins.** Offline data wins when labelling is expensive and only possible once (human experts), when you have a large high-quality public dataset, or for a quick first alignment pass.

## 24.2 Iterative DPO

A simple and widely used recipe:

**Notation on this page:** $y_w$, $y_l$: chosen (winning) and rejected (losing) responses • $y$: the response (output tokens); $y_t$ is token $t$

1. Start with policy $\pi_0$ (the SFT model).
2. For round $k = 0, 1, 2, \ldots$:
   a. Sample $n$ responses per prompt from $\pi_k$ (for example $n = 4$ to 8).
   b. Score them with a reward model or judge.
   c. Build pairs: best vs worst (or best vs a random other).
   d. Train $\pi_{k+1}$ with DPO on these pairs, using $\pi_k$ (or the SFT model) as the reference.

*RLHF Workflow: From Reward Modeling to Online RLHF* (Dong et al., arXiv 2405.07863) describes an open recipe of this kind. *The Llama 3 Herd of Models* (arXiv 2407.21783) describes several rounds of post-training, each with rejection sampling, SFT and DPO on fresh preference data from the latest models. TRL's `trl.experimental.online_dpo` trainer implements the fully online version: it generates two completions per prompt during training and ranks them with reward functions.

```
import torch, torch.nn.functional as F
torch.manual_seed(0)

# Toy "policy": logits over 6 canned responses. True quality is hidden; a noisy judge
↪ labels pairs.
quality = torch.tensor([0.0, 0.2, 0.4, 0.6, 0.8, 1.0])
def judge(i, j):                    # returns True if i is judged better than j (noisy)
    return torch.rand(()) < torch.sigmoid(5 * (quality[i] - quality[j]))

logits = torch.zeros(6)
for rnd in range(4):
    ref = logits.clone()
    theta = logits.clone().requires_grad_(True)
    opt = torch.optim.SGD([theta], lr=1.0)
    probs = F.softmax(ref, -1)
    for _ in range(200):            # on-policy pairs from the current round's model
        i, j = torch.multinomial(probs, 2, replacement=False).tolist()
        w, l = (i, j) if judge(i, j) else (j, i)
        lp, lr = F.log_softmax(theta, -1), F.log_softmax(ref, -1)
        loss = -F.logsigmoid(0.5 * ((lp[w] - lr[w]) - (lp[l] - lr[l])))
        opt.zero_grad(); loss.backward(); opt.step()
    logits = theta.detach()
    p = F.softmax(logits, -1)
    print(f"round {rnd}: expected quality {(p * quality).sum():.3f}, P(best)
    ↪ {p[-1]:.2f}")
```

Each round uses the previous round's model as both the sampler and the reference, so the policy can keep improving instead of being anchored to the SFT model forever.

## 24.3 Self-rewarding models

*Self-Rewarding Language Models* (Yuan et al., arXiv 2401.10020) uses the model as its *own* judge. In each iteration, the model generates candidate responses, scores them with an LLM-as-a-judge prompt (a 5-point rubric), and is trained with DPO on the resulting pairs. Both instruction follow-

ing and judging ability improved over three iterations of Llama 2 70B. The obvious risk: errors and biases in self-judgement get reinforced, so gains eventually plateau.

## 24.4 Nash and game-theoretic methods

Bradley-Terry assumes preferences come from a scalar reward, which implies they are **transitive** (if A beats B and B beats C, A beats C). Real human preferences can be **intransitive** (cyclic) and depend on the comparison. Game-theoretic methods drop the reward and work with a general preference function $\mathcal{P}(y \succ y' \mid x)$:

- *Nash Learning from Human Feedback* (Munos et al., arXiv 2312.00886) looks for a policy that no other policy beats more than half the time: a **Nash equilibrium** of the two-player game where each player picks a response and the preferred one wins.
- *Self-Play Preference Optimization for Language Model Alignment* (Wu et al., arXiv 2405.00675; SPPO) approximates the Nash equilibrium with iterative self-play updates; TRL's `sppo_hard` DPO loss is a variant.
- *Direct Nash Optimization* (Rosset et al., arXiv 2404.03715) gives a batched on-policy algorithm with a regression-style objective.

In interviews, the key point is the motivation: **reward models cannot represent intransitive preferences; preference models and Nash-style objectives can.**

## 24.5 Offline vs online in a simulation

A policy over 20 possible responses with known true rewards; the best response is rare under the reference. **Offline** DPO builds best-vs-worst pairs from samples of the *reference* only; **online** DPO builds them from samples of the *current* policy each step. Everything else (loss, $\beta$, number of pairs) is the same.

```
import torch, torch.nn.functional as F
torch.manual_seed(0)

V = 20
true_r = torch.randn(V)
best = true_r.argmax().item()
ref_logits = torch.randn(V)
ref_logits[best] = -2.0                                  # the best response is rare
 ↪  under the reference

def dpo_step(theta, yw, yl, beta=0.1, lr=0.5):
    lp = torch.log_softmax(theta, -1); lr_ = torch.log_softmax(ref_logits, -1)
    margin = beta * ((lp[yw] - lr_[yw]) - (lp[yl] - lr_[yl]))
    loss = -F.logsigmoid(margin).mean()
    g, = torch.autograd.grad(loss, theta)
    return (theta - lr * g).detach().requires_grad_(True)

def run(online, iters=300, n=8, B=16):
```

**Notation on this page:** $a \succ b$: $a$ is preferred to $b$ • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay)

```
    theta = ref_logits.clone().requires_grad_(True)
    for _ in range(iters):
        src = theta.detach() if online else ref_logits          # online: sample from
↪   the current policy
        y = torch.multinomial(torch.softmax(src, -1).expand(B, V), n, replacement=True)
        r = true_r[y]
        yw = y[torch.arange(B), r.argmax(1)]; yl = y[torch.arange(B), r.argmin(1)]   #
↪   best vs worst of n
        theta = dpo_step(theta, yw, yl)
    p = torch.softmax(theta.detach(), -1)
    return (p * true_r).sum().item(), p[best].item()

print(f"reference policy: expected reward {(torch.softmax(ref_logits, -1) *
↪   true_r).sum():.2f}, max possible {true_r.max():.2f}")
for online in (False, True):
    er, pb = run(online)
    print(f"{'online ' if online else 'offline'} DPO: expected reward {er:.2f}, P(best
     ↪   response) {pb:.2f}")
```

Online DPO ends with clearly higher expected reward: as the policy improves, its best-vs-worst pairs compare *better* responses, so the signal keeps sharpening where the policy actually is. Offline DPO keeps learning from the reference's pairs, which say little about the region the policy has moved into. But note the second number: **neither** method discovers the rare best response. Online methods only compare what they sample; once the policy concentrates on good-but-not-best responses, the best one is almost never sampled again. Exploration (temperature, diverse prompts, more samples per prompt, seeding with expert data) is the other half of online learning.

## 24.6 Online DPO in TRL

TRL 1.16 keeps online DPO in `trl.experimental.online_dpo`. Each step it samples two completions per prompt from the current policy, scores them with a reward model or reward functions (or a judge), and applies the DPO loss to the resulting pair. Fields and defaults checked in the installed version: `learning_rate=5e-7`, `max_new_tokens=64` (raise it for real tasks), `temperature=0.9`, `missing_eos_penalty=None` (set a value to penalise completions that do not finish), `loss_type="sigmoid"`; the trainer takes `reward_funcs`.

```
from trl.experimental.online_dpo import OnlineDPOConfig, OnlineDPOTrainer

config = OnlineDPOConfig(output_dir="odpo", learning_rate=5e-7, max_new_tokens=512,
                         temperature=0.9, missing_eos_penalty=1.0, beta=0.1)
trainer = OnlineDPOTrainer(model=model_name, reward_funcs=reward_model_or_functions,
                           args=config, train_dataset=prompts)   # prompts only: pairs
↪   are generated
trainer.train()
```

**Why this, not that? Why best-vs-worst of n samples, not two random samples, to build online pairs?**

**Why.** The extreme pair has the largest reward gap, so the preference is clearest (least label noise) and the DPO gradient is most informative. With $n = 4$ to 8 samples it also behaves like a soft version of best-of-$n$ training.
**Trade-off.** Generation cost grows with $n$; extreme pairs can be *too* easy (the worst sample is often obviously broken, so the model learns little about fine distinctions); and an imperfect reward model is most wrong at the extremes.
**When the alternative wins.** Random or adjacent-rank pairs teach finer distinctions and are cheaper; using *all* $\binom{n}{2}$ pairs, or a group-based RL method like GRPO, uses every sample instead of two.

**Interview check**

**Q1. Why does online DPO usually beat offline DPO?** On-policy pairs target the model's actual outputs and avoid distribution shift; the implicit reward is only evaluated where the policy has probability.
**Q2. Describe iterative DPO.** Each round: sample from the current model, label with an RM/judge, build best-vs-worst pairs, train with DPO, then use the new model for the next round.
**Q3. What is a self-rewarding model and its main risk?** The model judges its own outputs to create DPO pairs; it can reinforce its own biases and errors.
**Q4. Why consider Nash-style methods?** Human preferences can be intransitive; a scalar reward (Bradley-Terry) cannot represent that, a general preference model can.
**Q5. What does the reference model become in iterative DPO?** Usually the previous round's policy, so the model is not anchored to SFT forever.
**Q6. In the simulation, why does online DPO beat offline DPO?** Its pairs come from the current policy, so the comparisons stay informative in the region the policy has moved to; offline pairs describe the reference's region.
**Q7. Why might neither find a rare best response?** Online methods only learn from what they sample; once the policy concentrates elsewhere, the rare best response is never sampled. Exploration is needed.
**Q8. How does TRL's online DPO build its pairs?** Two completions per prompt from the current policy, ranked by reward functions/RM or a judge; then the DPO loss.
**Q9. What does `missing_eos_penalty` do?** Subtracts a fixed penalty from the score of completions that never emit the end token, discouraging rambling and truncation.

**Key takeaways**

- Offline pairs suffer distribution shift; online pairs from the current policy fix the model's real mistakes.
- Iterative DPO: sample, label, pair, DPO, repeat (Llama 3, RLHF Workflow).
- Self-rewarding models use the policy as its own judge.
- NLHF, SPPO and DNO treat alignment as a game to handle intransitive preferences.
- Online pairs follow the policy; offline pairs go stale as soon as the policy moves.
- Online learning still needs exploration: you only improve among what you sample.

**Notation on this page:** $n$: samples per prompt

- TRL online DPO: experimental module, LR 5e-7, temperature 0.9, set `max_new_tokens` and `missing_eos_penalty`.

CHAPTER 25

# RLAIF, Constitutional AI and Rubrics

**Why this chapter matters.**
Human labels are slow, expensive and inconsistent. Most modern preference data is produced, at least partly, by AI models following written principles or rubrics. This chapter covers RLAIF, Anthropic's Constitutional AI, rubric-based rewards for non-verifiable tasks, judge biases, and spec-based safety training such as deliberative alignment.

## 25.1 AI feedback instead of human feedback

**RLAIF** (reinforcement learning from AI feedback) replaces human preference labels with labels from an LLM. *RLAIF vs. RLHF: Scaling Reinforcement Learning from Human Feedback with AI Feedback* (Lee et al., arXiv 2309.00267) found RLAIF achieved comparable results to RLHF on summarisation and dialogue tasks, and introduced **direct RLAIF**: skip the reward model and use the LLM judge's score directly as the RL reward.

Good practice for AI labelling:

- Write a clear **preamble** describing what "better" means.
- Ask the judge to **reason first** (chain of thought) before choosing.
- **Swap the order** of the two responses and average, to cancel position bias.
- Use the judge's **probabilities** (e.g. of the tokens "1" and "2") as soft labels instead of a hard choice.
- **Calibrate** against a small set of human labels and measure agreement.

## 25.2 Constitutional AI

*Constitutional AI: Harmlessness from AI Feedback* (Bai et al., arXiv 2212.08073) trains a harmless but helpful assistant with human labels only for helpfulness. Harmlessness comes from a **constitution**: a short list of natural-language principles ("choose the response that is least harmful", "avoid being preachy", and so on). Two stages:

**1. Supervised stage (critique and revision).**

a. Prompt a helpful-only model with red-team prompts designed to get harmful answers.
b. Ask the model to **critique** its own response according to a randomly chosen principle.
c. Ask it to **revise** the response to address the critique. Repeat a few times.
d. Fine-tune on the revised responses (SL-CAI).

**2. RL stage (RLAIF).**

a. Sample pairs of responses from the SL-CAI model.
b. Ask a model which response better follows a sampled principle (multiple-choice format, with chain of thought), producing AI preference labels for harmlessness.
c. Train a preference model on these AI labels mixed with human helpfulness labels.
d. Run RL against this preference model (RL-CAI).

The result was a model that was both less harmful and less evasive than RLHF models trained on human harmlessness labels: it explains its objections instead of refusing with no reason.

**Why this, not that? Why written principles and AI labels, not more human harmlessness labels?**

**Why.** Principles are explicit, auditable and easy to change; AI labels scale to millions of comparisons cheaply and consistently; and human labellers are not exposed to large volumes of harmful content.
**Trade-off.** The labels are only as good as the labelling model and the principles; biases in the judge carry into the policy; and a model judging its own family's outputs can share blind spots.
**When the alternative wins.** Human labels win for new kinds of risk the principles do not cover, for calibrating and auditing the AI labeller, and for subjective quality judgements where AI judges disagree with people.

## 25.3 Rubrics and LLM judges as rewards

For tasks without a checkable answer (medical advice, writing, research reports), a growing approach is to write a **rubric** per prompt or per task: a list of specific criteria with weights ("mentions the drug interaction", "cites a source", "does not exceed 200 words"). An LLM judge scores each criterion and the weighted sum is the reward. *Rubrics as Rewards: Reinforcement Learning Beyond Verifiable Domains* (Gunjal et al., arXiv 2507.17746) used such rubric rewards with GRPO-style training and reported gains over judging with a single overall (Likert-style) score in medical and science domains.

```
# A rubric reward: weighted criteria, each judged pass/fail (here by simple rules; in
↪ practice an LLM judge).
rubric = [
    (3.0, "states the final dose", lambda r: "mg" in r),
    (2.0, "warns about the interaction", lambda r: "interaction" in r.lower()),
    (1.0, "is concise (< 60 words)", lambda r: len(r.split()) < 60),
    (-2.0, "invents a citation", lambda r: "et al." in r),
]
def rubric_reward(response):
    total = sum(w for w, _, check in rubric if w > 0)
    score = sum(w for w, _, check in rubric if check(response))
    return score / total

a = "Take 500 mg twice daily. Note the interaction with warfarin."
b = "Smith et al. say take 500 mg."
print(round(rubric_reward(a), 2), round(rubric_reward(b), 2))
```

Rubric rewards have the advantages of verifiable rewards (explicit, per-criterion, harder to fool with style) and can be applied to open-ended tasks. Their weaknesses: writing good rubrics is labour-intensive (often AI-assisted), and the judge can still be gamed (e.g. by mentioning every keyword).

## 25.4 Judge biases

From *Judging LLM-as-a-Judge* (arXiv 2306.05685) and later work:

| Bias | What happens | Mitigation |
|---|---|---|
| Position | Prefers the first (or second) answer | Evaluate both orders; count only consistent wins |
| Verbosity | Prefers longer answers | Length-controlled comparisons; instruct to ignore length |
| Self-enhancement | Prefers outputs of its own model family | Use a different judge family; panels of judges |
| Style over substance | Rewards formatting, confident tone | Rubrics with content criteria; reference answers |
| Limited capability | Cannot check hard maths or code | Give reference solutions; use tools or execution |

## 25.5 Deliberative alignment and spec-based training

*Deliberative Alignment: Reasoning Enables Safer Language Models* (Guan et al., arXiv 2412.16339) teaches reasoning models to **recall and reason over the written safety specification** in their chain of thought before answering. In outline:

1. Generate training examples where the model's reasoning explicitly cites the relevant parts of the spec, by prompting a model *with* the spec, then train the model on them *without* the spec in the prompt (so it must internalise the spec). Filter with a judge that has the spec.
2. Run RL with a reward model (judge) that sees the spec and scores compliance.

The approach reports better refusal of truly harmful requests together with fewer over-refusals of benign ones, and it generalises to new jailbreak styles, because the model reasons about *why* something is or is not allowed instead of pattern-matching on surface features. The broader trend: **write the desired behaviour down as a specification, and use AI judges and reasoning to train towards it.**

## 25.6 From judge outputs to training labels: the maths of soft labels

An LLM judge comparing responses A and B can output more than a choice: the probabilities it assigns to the tokens "A" and "B". Normalising them gives $p = P_{\text{judge}}(A \succ B)$. Two simple steps

turn this into a good training label.

**1. Remove position bias.** Ask twice, once with A first and once with B first, and average:

$$\bar{p} = \tfrac{1}{2}\big(P(\text{"first"} \mid A, B) + P(\text{"second"} \mid B, A)\big).$$

If the judge always prefers the first slot regardless of content, the two terms cancel to 0.5, which is the honest answer: "this judge cannot tell".

**2. Train the reward model on the soft label.** The Bradley–Terry loss with a soft target is binary cross-entropy:

$$\mathcal{L}(\Delta) = -\bar{p}\log\sigma(\Delta) - (1-\bar{p})\log\sigma(-\Delta), \qquad \Delta = r_\phi(A) - r_\phi(B).$$

Setting the derivative $\sigma(\Delta) - \bar{p}$ to zero gives the optimum $\Delta^* = \log\frac{\bar{p}}{1-\bar{p}}$, the judge's log-odds. With a hard label ($\bar{p} = 1$) the optimum is $\Delta^* = \infty$: the reward model is pushed to make the gap ever larger, even for pairs where the judge was barely sure. Soft labels keep reward differences proportional to the judge's confidence.

**Worked example.** The judge gives "A" probability 0.80 when A is first, and "B" probability 0.40 when B is first (so A gets 0.60 in that order). Then $\bar{p} = (0.80 + 0.60)/2 = 0.70$ and the target reward gap is $\ln(0.7/0.3) = 0.85$. A hard label would have said "A wins" and pushed the gap without limit; the swap shows that part of the 0.80 was position bias.

```
import torch, math
torch.manual_seed(0)
def debiased_prob(p_first_wins_AB, p_first_wins_BA):
    """Probability A beats B from two orderings: A-first and B-first."""
    return 0.5 * (p_first_wins_AB + (1 - p_first_wins_BA))
p = debiased_prob(0.80, 0.40)
print("debiased p(A>B) = %.2f, target margin = %.3f" % (p, math.log(p / (1 - p))))
# Train a reward gap on soft vs hard labels for pairs of varying judge confidence
true_p = torch.tensor([0.55, 0.7, 0.9, 0.99])
for name, target in [("soft", true_p), ("hard", torch.ones(4))]:
    gap = torch.zeros(4, requires_grad=True); opt = torch.optim.SGD([gap], lr=1.0)
    for _ in range(500):
        loss = torch.nn.functional.binary_cross_entropy_with_logits(gap, target)
        opt.zero_grad(); loss.backward(); opt.step()
    print(f"{name} labels -> learned gaps {[round(g, 2) for g in gap.tolist()]}")
print("judge log-odds         ", [round(math.log(q / (1 - q)), 2) for q in
 ↪  true_p.tolist()])
```

The soft-label gaps move to the judge's log-odds (the 0.99 pair is still approaching 4.6); the hard-label gaps keep growing with training time and are the same for a 55% pair and a 99% pair. In RL this matters: the policy is rewarded as strongly for winning a coin-flip comparison as for a clear win.

**Why this, not that? Why soft labels from judge probabilities, not hard choices?**

**Reason.** Soft labels carry the judge's confidence; the reward model learns gaps proportional to it, ties stay ties, and label noise on uncertain pairs is down-weighted automatically. **Trade-**

**Notation on this page:** $B$: batch size • $p$: a probability distribution (often the target or data) • $\phi$: weights of a second network (reward model or critic) • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $\mathcal{L}$: a loss to minimise • $r(x, y)$: the reward for response $y$ to prompt $x$

**off.** They need access to token probabilities (not every API returns them), and a badly calibrated judge passes its miscalibration into the reward scale. **When the alternative is better.** When the judge reasons at length before answering, its final-token probability is often near 0 or 1 and not calibrated; then majority votes over several sampled judgements, or human-calibrated hard labels, are more reliable.

## 25.7 Measuring a judge before you trust it

Before using a judge for labels or rewards, measure it against a few hundred human labels on the same distribution.

**Agreement and Cohen's kappa.** Raw agreement $p_o$ (fraction of items where judge and human agree) is inflated when one answer usually wins. **Cohen's kappa** corrects for agreement expected by chance:

$$\kappa = \frac{p_o - p_e}{1 - p_e}, \qquad p_e = \sum_c P_{\text{judge}}(c)\, P_{\text{human}}(c),$$

where $c$ ranges over labels ("A", "B", "tie"). $\kappa = 0$ means chance-level agreement, $\kappa = 1$ perfect agreement. Human–human agreement on preference data is often only around 0.6–0.8 raw agreement, which caps what any judge can reach.

**Position consistency.** The fraction of pairs on which the judge picks the same response in both orders. A judge that is consistent on only 70% of pairs gives an arbitrary answer on the rest.

**Worked example.** On 200 pairs, judge and human agree on 150 ($p_o = 0.75$). The judge says "A" on 60% of pairs, the human on 55%. Then $p_e = 0.6 \times 0.55 + 0.4 \times 0.45 = 0.51$ and $\kappa = (0.75 - 0.51)/(1 - 0.51) = 0.49$: "moderate" agreement, much less impressive than 75% sounds.

```
def cohen_kappa(a, b):
    labels = sorted(set(a) | set(b)); n = len(a)
    po = sum(x == y for x, y in zip(a, b)) / n
    pe = sum((a.count(c) / n) * (b.count(c) / n) for c in labels)
    return po, (po - pe) / (1 - pe)
human = ["A"] * 110 + ["B"] * 90
judge = ["A"] * 90 + ["B"] * 20 + ["A"] * 30 + ["B"] * 60      # 150 agreements; judge
 ↪  says A 60%
po, k = cohen_kappa(judge, human)
print(f"raw agreement {po:.2f}, Cohen's kappa {k:.2f}")
ab = ["A"] * 170 + ["B"] * 30                                  # judge's pick with A shown
 ↪  first
ba = ["A"] * 120 + ["B"] * 80                                    # judge's pick (true
 ↪  identity) with B shown first
print("position consistency %.2f" % (sum(x == y for x, y in zip(ab, ba)) / 200))
```

## 25.8 Rubric rewards inside an RL loop

A rubric reward is a function $R(x, y) = \sum_k w_k\, c_k(x, y) / \sum_{k:w_k>0} w_k$, where each $c_k \in \{0, 1\}$ (or $[0, 1]$) is one criterion judged by an LLM. Practical design rules, each answering a failure seen in

**Notation on this page:** $p$: a probability distribution (often the target or data) • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $k$: an index or count (see text) • $R$: return or total reward

practice:

- **Make criteria checkable from the text.** "Mentions that ibuprofen can raise blood pressure" is checkable; "is medically excellent" is not.
- **Include negative criteria** with negative weights ("recommends a prescription dose without a caveat"), so that adding content is not free.
- **Ask the judge to quote evidence** for each criterion it marks as met. Keyword stuffing then fails, because the quote must make sense in context.
- **Cap length** or add a conciseness criterion; otherwise listing every possible point maximises coverage.
- **Judge each criterion separately** (or in small groups) for long rubrics; one giant prompt makes the judge skim.
- **Keep the judge fixed** during a run and version it; a changing judge makes curves impossible to compare.

**Hacking check.** Every few hundred steps, take the 20 highest-reward responses and read them against the rubric. If they contain criterion phrases without substance, tighten the evidence rule or add negative criteria.

**Why this, not that? Why per-criterion rubric scores, not one overall 1-to-10 score?**

**Reason.** An overall score mixes many qualities, is poorly calibrated across prompts and is easy to sway with style; per-criterion checks are explicit, auditable and closer to verifiable rewards, so the gradient points at specific behaviour. **Trade-off.** Rubrics must be written (often per prompt, often AI-drafted and human-checked), they cost several judge calls per response, and they reward only what they list. **When the alternative is better.** Pure style or creative tasks where quality is holistic, or quick experiments where pairwise preference against a reference response is cheaper and good enough.

## 25.9 Self-rewarding loops

A model can also judge *its own* samples. *Self-Rewarding Language Models* (Yuan et al., arXiv 2401.10020) trains one model both to answer and to score answers (LLM-as-a-judge prompting with a 5-point rubric). Each iteration: generate several answers per prompt, score them with the same model, build preference pairs from the highest and lowest scores, and train with DPO. Over three iterations the paper reports gains in both instruction following and judging ability. *Meta-Rewarding Language Models* (arXiv 2407.19594) adds a third role, a **meta-judge** that compares the model's judgements, to keep the judging skill improving too.

The appeal is a loop that needs no external labels. The risks follow from everything above: the judge's biases (length, style, self-preference) are optimised directly and reinforced each round, and nothing anchors the scores to people. Practical safeguards are a fixed external evaluation set with human or strong-model labels, length control in pair construction (for example, pairing responses of similar length), and stopping when external evaluations stop improving even if self-scores rise.

**Worked example.** In round 1 the model's self-scores average 3.6 and an external judge's win-rate

against the starting model is 58%. In round 3 self-scores average 4.4, but the external win-rate falls to 55% and responses are 40% longer. The loop is now optimising its own length bias: stop, or add length-matched pairing and retrain from round 2.

## 25.10 Common mistakes

- **Using a single ordering** of the two responses: position bias becomes label noise.
- **Reporting raw agreement only**: compute Cohen's kappa and position consistency.
- **Letting the policy and the judge be the same model** without checks: self-preference and shared blind spots.
- **Rubrics without negative criteria or evidence requirements**: keyword stuffing and padding win.

**Interview check**

**Q1. What is RLAIF?** RL where preference labels or rewards come from an AI model instead of humans; can be via an RM trained on AI labels or "direct" with the judge's score.
**Q2. Describe Constitutional AI's two stages.** SL: critique and revise harmful responses using principles, then SFT on revisions. RL: AI preference labels by principles, train a preference model, RL against it.
**Q3. How do you reduce position bias in an LLM judge?** Evaluate both orders and average, or count only consistent judgements.
**Q4. What are rubric rewards and when are they useful?** Weighted, specific criteria scored by a judge; for non-verifiable tasks where a single overall score is too vague.
**Q5. What is deliberative alignment?** Training a reasoning model to recall and reason about the safety spec in its chain of thought, using spec-aware data generation and a spec-aware judge for RL.
**Q6. Why might a model judging its own outputs be risky?** Self-enhancement bias and shared blind spots; errors are reinforced.
**Q7. What reward gap does a Bradley–Terry model learn from a soft label $\bar{p}$?** The judge's log-odds, $\log \frac{\bar{p}}{1-\bar{p}}$; with hard labels the optimum gap is infinite.
**Q8. How do you debias a judge for position?** Ask in both orders and average the probability that A wins; count a hard win only when both orders agree.
**Q9. What is Cohen's kappa and why use it for judges?** Agreement corrected for chance, $(p_o - p_e)/(1 - p_e)$; raw agreement is inflated when one label dominates.
**Q10. How do you stop a rubric reward from being gamed by keyword stuffing?** Require quoted evidence for each criterion, add negative criteria and a length limit, and read top-reward samples regularly.
**Q11. What is the main risk of self-rewarding training, and how do you detect it?** The model's own judging biases (length, style) are amplified each round; detect it with a fixed external evaluation that does not use the model as judge, and by tracking response length.

**Notation on this page:** $p$: a probability distribution (often the target or data)

**Key takeaways**

- RLAIF matches RLHF on several tasks at much lower cost; good judge prompting matters.
- Constitutional AI: critique-and-revise SFT, then RL on AI preference labels from written principles.
- Rubric rewards extend RL to non-verifiable domains.
- Judges have position, verbosity, self and style biases; design around them.
- Deliberative alignment trains models to reason over a written spec.
- Debias with both orders; train on soft labels so reward gaps equal the judge's log-odds.
- Validate judges with kappa and position consistency against human labels before using them.
- Rubric rewards: checkable criteria, negative weights, evidence quotes, length control.

PART VI

# Reasoning through Reinforcement Learning

CHAPTER 26

# RL with Verifiable Rewards

**Why this chapter matters.**
Since DeepSeek-R1 (January 2025), the most important use of RL in post-training is **RL with verifiable rewards** (RLVR): reward the model when a program can check that its answer is right. RLVR produced the long chain-of-thought "reasoning" models. This chapter explains what makes a reward verifiable, how answers are extracted and checked, how to measure success with pass@$k$, and what the evidence says about whether RL teaches new skills or only sharpens existing ones.

## 26.1 What a verifiable reward is

A **verifiable reward** is computed by a deterministic program, not a learned model:

- **Maths**: extract the final answer and compare it with the reference answer, allowing equivalent forms ($\frac{1}{2}$, 0.5, `\frac12`).
- **Code**: run the program against unit tests in a sandbox; reward = all pass (or the fraction passed).
- **Formats and constraints**: valid JSON, a required number of words, the reasoning inside `<think>` tags, a specific language.
- **Games and puzzles**: Sudoku validity, a chess engine's evaluation, a solved puzzle.
- **Tool and agent tasks**: the final state of an environment matches the goal (a file exists, a test suite passes, a database has the right row).

The reward is usually **binary** (1 if correct, 0 if not), sometimes with a small format component. DeepSeek-R1-Zero used an accuracy reward plus a format reward for putting reasoning between `<think>` and `</think>` (arXiv 2501.12948).

**Why this, not that? Why binary correctness rewards, not partial credit?**

**Why.** Binary rewards are hard to game, need no tuning, and match what we care about: the final answer. Partial-credit schemes (rewarding plausible steps, closeness of a number) invite the policy to optimise the partial signal instead of the answer.
**Trade-off.** The signal is sparse: a prompt the model always fails, or always solves, gives no learning signal under group-relative methods. Long responses get one bit for thousands of tokens.
**When the alternative wins.** Partial credit helps for code (fraction of tests passed) when full passes are rare, for multi-part answers, and early in training on hard tasks. Process rewards (Chapter 29) give denser signals but are easier to hack.

**Notation on this page:** $k$: number of attempts in pass@$k$

## 26.2 Answer extraction and checking

The weakest link in RLVR is often the checker. Typical pipeline for maths:

1. **Instruct** the model to put the final answer in a fixed place, for example `\boxed{...}`.
2. **Extract**: find the *last* `\boxed{...}` (handling nested braces), or the text after "Answer:".
3. **Normalise**: strip spaces, units, `\left`/`\right`, trailing full stops; convert `\frac{a}{b}`, `\dfrac`, decimals and percentages.
4. **Compare**: exact string match after normalisation, then numeric comparison within a tolerance, then **symbolic** equivalence (for example with SymPy) for expressions.

Hugging Face's `math-verify` library implements robust parsing and equivalence for LaTeX and plain-text answers, and is used by many open RL recipes; TRL ships reward functions built on it in `trl.rewards` (`accuracy_reward`, plus `think_format_reward` for the `<think>` format and `get_soft_overlong_punishment` for DAPO-style length shaping).

```python
import re
from fractions import Fraction

def last_boxed(text):
    """Return the content of the last \\boxed{...}, handling nested braces."""
    start = text.rfind("\\boxed{")
    if start < 0:
        return None
    i, depth, out = start + len("\\boxed{"), 1, []
    while i < len(text) and depth:
        c = text[i]
        depth += (c == "{") - (c == "}")
        if depth:
            out.append(c)
        i += 1
    return "".join(out) if depth == 0 else None

def to_number(s):
    s = s.replace(" ", "").replace("\\dfrac", "\\frac").rstrip(".")
    m = re.fullmatch(r"\\frac\{(-?\d+)\}\{(-?\d+)\}", s)
    if m:
        return Fraction(int(m[1]), int(m[2]))
    try:
        return Fraction(s)
    except ValueError:
        return None

def math_reward(completion, reference):
    ans = last_boxed(completion)
    if ans is None:
        return 0.0                                  # no parsable answer: no reward
    a, b = to_number(ans), to_number(reference)
    if a is not None and b is not None:
        return float(a == b)
```

```
    return float(ans.strip() == reference.strip())

print(math_reward("... so x = \\boxed{\\frac{1}{2}}.", "0.5"))     # 1.0
print(math_reward("I think \\boxed{3} or maybe \\boxed{4}", "3"))   # 0.0: last box wins
print(math_reward("The answer is 0.5", "0.5"))                      # 0.0: not boxed
```

Checker design choices affect what the model learns:

- **Take only the last answer** (or require exactly one), or the model learns to list several answers.
- **Strict format** teaches formatting quickly; too strict, and correct answers get zero reward (false negatives), which adds noise.
- **False positives** (a lenient checker accepting wrong answers) are worse: the model learns to exploit them.
- **Datasets** should have answers that are checkable. DAPO built DAPO-Math-17K with integer answers precisely to make checking reliable (arXiv 2503.14476).

For **code**, run tests in an isolated sandbox with time and memory limits, no network, and read-only test files (Chapter 35).

## 26.3 Sparse rewards and pass@$k$

**pass@$k$** is the probability that at least one of $k$ samples is correct. Estimating it by literally drawing $k$ samples is noisy. *Evaluating Large Language Models Trained on Code* (Chen et al., arXiv 2107.03374) gives an **unbiased estimator**: draw $n \geq k$ samples, count $c$ correct, and compute

$$\widehat{\text{pass@}k} = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}},$$

the probability that a random subset of $k$ of the $n$ samples contains at least one correct one.

```
import numpy as np

def pass_at_k(n, c, k):
    """Unbiased pass@k from n samples with c correct (numerically stable product
    ↪ form)."""
    if n - c < k:
        return 1.0
    return 1.0 - np.prod(1.0 - k / np.arange(n - c + 1, n + 1))

for k in (1, 8, 64):
    print(f"n=64, c=4: pass@{k} = {pass_at_k(64, 4, k):.3f}")
```

Related metrics: **avg@$n$** (mean accuracy over $n$ samples, a lower-variance pass@1 estimate; DAPO reports avg@32) and **maj@$k$** (accuracy of the majority-vote answer among $k$ samples).

For training, what matters is the per-prompt **success rate** $p$. With group-based methods, a prompt gives signal only if its group contains both successes and failures. With $G$ samples, the chance that all are the same is $p^G + (1-p)^G$. Prompts with $p$ near 0 or 1 are wasted compute, which is

**Notation on this page:** $k$: number of attempts in pass@$k$ • $n$: samples drawn • $c$: number of correct samples • $p$: probability of solving the prompt • $G$: group size (samples per prompt)

why recipes filter or resample them (dynamic sampling, Chapter 28; difficulty filtering, Chapter 31).

## 26.4 Does RL add new abilities or sharpen existing ones?

This was the most debated RLVR question of 2025. The evidence:

**"Sharpening" view.** *Does Reinforcement Learning Really Incentivize Reasoning Capacity in LLMs Beyond the Base Model?* (Yue et al., arXiv 2504.13837) evaluated pass@$k$ at large $k$ across model families, algorithms and tasks. RLVR models win at small $k$ (pass@1), but **base models achieve higher pass@$k$ when $k$ is large**. Coverage and perplexity analyses suggested the reasoning paths of RL models were already present in the base model; six popular RLVR algorithms performed similarly and far from the base model's ceiling. In contrast, **distillation** from a stronger teacher did introduce new reasoning patterns.

**"Expansion" view.** *ProRL: Prolonged Reinforcement Learning Expands Reasoning Boundaries in Large Language Models* (Liu et al., arXiv 2505.24864) trained much longer, with KL control, **reference policy resetting** and a diverse task suite, and found RL models beating base models across pass@$k$, including tasks where the base model fails at any $k$. Gains were largest where the base model was weak. *Reinforcement Learning with Verifiable Rewards Implicitly Incentivizes Correct Reasoning in Base LLMs* (Wen et al., arXiv 2506.14245) argued that answer-only pass@$k$ is misleading because base models can reach correct answers through flawed reasoning; with a **CoT-pass@$k$** metric that also requires correct reasoning, RLVR extends the boundary.

**Warnings about evidence.** *Spurious Rewards: Rethinking Training Signals in RLVR* (Shao et al., arXiv 2506.10947) found that GRPO with **random** rewards raised Qwen2.5-Math-7B's MATH-500 score by 21.4 points, close to the 29.1-point gain from true rewards. They traced this to a bias from GRPO's clipping that amplifies behaviours already common in pretraining (here, "code reasoning", which rose from 65% to over 90%). The same spurious rewards did not help Llama 3 or OLMo 2. *Reinforcement Learning for Reasoning in Large Language Models with One Training Example* (Wang et al., arXiv 2504.20571) raised Qwen2.5-Math-1.5B's MATH-500 accuracy from 36.0% to 73.6% with RLVR on a **single** example.

**A balanced summary for interviews:**

1. At typical training lengths, much of RLVR's gain is **eliciting and sharpening** behaviours the base model can already produce: better pass@1, less diversity.
2. Longer, well-regularised training on diverse tasks can **expand** what the model solves, especially where the base model is weak but not hopeless.
3. Results on a single model family (especially Qwen2.5-Math) can mislead; validate across families and with careful metrics.
4. Distillation from a stronger model is still the most reliable way to give a *small* model new reasoning patterns.

**Notation on this page:** $k$: number of attempts in pass@$k$

**Why this, not that? Why RL on a strong base model, not RL from a weak one?**

**Why.** RL can only reinforce what it samples. A strong base model already samples correct reasoning some of the time, so there is signal from the first step; a weak one gets almost all zeros.
**Trade-off.** Gains then partly reflect the base model's latent ability, and the base model's habits (including quirks such as code-style reasoning) get amplified.
**When the alternative wins.** For small models, distil first from a strong reasoner (SFT on its traces) and only then run RL; DeepSeek-R1's distilled small models beat RL-from-scratch on the same small bases (arXiv 2501.12948).

## 26.5 A map of verifiers

"Verifiable" covers a range of checkers with different error profiles. Choose by the task, and know which error each one makes.

| Verifier | Typical tasks | False positives (wrong marked right) | False negatives (right marked wrong) |
|---|---|---|---|
| Exact string match | Multiple choice, short labels | Rare | Common: formatting, units, synonyms |
| Numeric match with tolerance | Arithmetic word problems | Rare (beware tolerance too loose) | Fractions vs decimals, "1,000", units |
| Symbolic equivalence (e.g. SymPy) | Algebra, closed-form answers | Rare; possible with simplification bugs | Unusual notation the parser cannot read |
| Unit tests in a sandbox | Code | Weak or leaked tests; tests the model can edit | Flaky tests, timeouts, environment differences |
| Formal proof checker (e.g. Lean) | Theorem proving | Almost none (if the statement is right) | None for the proof, but autoformalisation of the *statement* can be wrong |
| Constraint checks | Instruction following ("exactly 3 bullets") | Satisfying the letter, not the intent | Ambiguous constraints |
| Model-based verifier | Free-form answers, proofs in prose | Persuasive wrong answers | Correct answers in unexpected forms |

A symbolic checker reduces the false negatives of string matching:

```
import sympy as sp
from sympy.parsing.latex import parse_latex
def sym_equal(pred, ref):
```

```
    """True if two answers are mathematically equal; parse plain or simple LaTeX, fall
    ↪ back to False."""
    def parse(s):
        s = s.strip().strip("$")
        try:
            return sp.sympify(s.replace("^", "**"), rational=True)
        except (sp.SympifyError, SyntaxError, TypeError):
            pass
        try:
            return parse_latex(s)
        except Exception:
            return None
    a, b = parse(pred), parse(ref)
    if a is None or b is None: return False
    try:
        return bool(sp.simplify(a - b) == 0)
    except TypeError:
        return False
cases = [("1/2", "0.5"), ("2*(x+1)", "2*x+2"), ("sqrt(8)", "2*sqrt(2)"), ("x^2-1",
↪ "(x-1)*(x+1)"),
         ("3", "4"), ("pi/2", "1.5708")]
for p, r in cases:
    print(f"{p:>12s} vs {r:<12s} string-equal {p == r!s:5}  symbolic-equal {sym_equal(p,
    ↪ r)}")
```

The last line is a deliberate edge case: $\pi/2$ is not *exactly* 1.5708, so a symbolic checker says no, while a numeric checker with tolerance $10^{-4}$ would say yes. Decide per dataset which one the reference answers expect, and test your checker on a few hundred real model outputs before training: count disagreements between the checker and a careful human (or a strong model) and fix the biggest classes.

LaTeX parsing in SymPy needs an optional parser package; when it is missing, the function above falls back to plain-expression parsing, which is enough for these cases.

## 26.6 What verifier errors do to learning

Let a response be truly correct with probability $p$ (under the current policy, for a given prompt). A verifier with false-positive rate $\alpha$ and false-negative rate $\delta$ gives reward 1 with probability

$$\mathbb{E}[\tilde{R}] = (1-\delta)\,p + \alpha\,(1-p) = \alpha + (1-\alpha-\delta)\,p.$$

Two consequences:

1. **Symmetric noise shrinks the signal but keeps its direction.** The expected reward is still increasing in $p$, with slope $1-\alpha-\delta$. Policy gradients point the same way, only weaker and noisier: 10% false negatives and 5% false positives keep 85% of the signal. Learning still works; it is slower.
2. **Noise that depends on the response is what hurts.** If false positives are concentrated on a particular kind of output (an answer format the parser misreads as correct, code that

**Notation on this page:** $p$: probability of solving the prompt • $\alpha$: a weighting coefficient (defined in the text) • $\delta_t$: TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $R$: return or total reward

deletes the tests), then that kind of output has a higher expected reward than correct answers, and the policy will move towards it. This is reward hacking, and it can happen with a small overall $\alpha$.

**Worked example.** A maths checker accepts any answer containing the right number anywhere in the last line. For a hard problem the policy is correct 20% of the time; a "hedging" style that lists three candidate numbers is correct-by-checker 45% of the time. The hedging style has a higher expected reward than honest answers, so RL will learn it, even though the checker has almost no false positives on honest answers. The fix is in the verifier (exactly one boxed answer; Project 6), not in the RL algorithm.

```
import torch
torch.manual_seed(0)
def train(alpha, delta, hack_fp=0.0, steps=300, lr=0.5, G=16):
    """Policy over 3 styles: 0 = careless (p=0.2), 1 = careful (p=0.6), 2 = hedge
    ↪ (p=0.2, but fools the checker)."""
    logits = torch.zeros(3, requires_grad=True); p_true = torch.tensor([0.2, 0.6, 0.2])
    opt = torch.optim.SGD([logits], lr=lr)
    for _ in range(steps):
        pi = torch.softmax(logits, 0); a = torch.multinomial(pi, G, replacement=True)
        correct = torch.rand(G) < p_true[a]
        fp = alpha + hack_fp * (a == 2).float()                    # hedging adds
↪ false positives
        r = torch.where(correct, (torch.rand(G) > delta).float(), (torch.rand(G) <
↪ fp).float())
        adv = r - r.mean()
        loss = -(adv * torch.log_softmax(logits, 0)[a]).mean()
        opt.zero_grad(); loss.backward(); opt.step()
    return [round(v, 2) for v in torch.softmax(logits, 0).tolist()]
print("clean verifier                ->", train(0.0, 0.0))
print("symmetric noise (a=.05, d=.10) ->", train(0.05, 0.10))
print("hackable: hedging fools it 60% ->", train(0.0, 0.0, hack_fp=0.6))
```

With a clean or symmetrically noisy verifier the policy moves to the careful style. When one style fools the checker, the policy moves to it, because its expected reward ($0.2 + 0.8 \times 0.6 = 0.68$) beats careful work (0.6).

**Why this, not that? Why spend effort on the verifier, not on the RL algorithm, when results are bad?**

**Reason.** The algorithm maximises whatever the verifier rewards. Response-dependent verifier errors define a different optimum, and no amount of clipping or normalisation changes the optimum; it only changes how fast you reach it. **Trade-off.** Verifier work is unglamorous and task-specific (parsers, sandboxes, test suites), and over-strict checkers create false negatives that slow learning. **When the alternative is better.** When the verifier is already audited (low, symmetric error measured on real outputs) and the curves show optimisation problems (entropy collapse, zero-variance groups, instability), the algorithm is the right place to work.

## 26.7 Code rewards in practice

For code, the reward is computed by running tests. Choices that matter:

- **All-pass vs fraction passed.** All-pass (1 only if every test passes) matches how code is judged and avoids rewarding programs that handle only easy cases; fraction-passed gives denser signal for hard problems but rewards partial hacks (special-casing visible tests). Many recipes use all-pass with enough tests per problem.
- **Hidden tests.** Show the model a few examples in the prompt, verify on hidden tests; otherwise the model can hard-code the visible outputs.
- **Test integrity.** Run tests in a fresh sandbox where the solution cannot modify or read them; agentic code RL (SWE-style tasks) has seen models edit tests or exit early with a success code (Chapter 35).
- **Timeouts and resource limits.** A per-test time limit (seconds) and memory limit, with timeouts counted as failures; otherwise infinite loops stall the reward service.
- **Determinism.** Seed randomness in tests; flaky tests are false negatives that look like noise.

## 26.8 Common mistakes

- **Not measuring the verifier.** Run it on a few hundred real policy outputs and compare with careful human or strong-model judgement before training.
- **Treating all verifier noise as harmless.** Symmetric noise slows learning; response-dependent false positives redirect it.
- **Visible-only tests for code**: the model learns to hard-code outputs.

**Interview check**

**Q1. What makes a reward "verifiable"?** It is computed by a deterministic checker (answer match, unit tests, format rules) rather than a learned model.
**Q2. Write the unbiased pass@$k$ estimator.** $1 - \binom{n-c}{k} / \binom{n}{k}$ from $n$ samples with $c$ correct.
**Q3. Why take only the last boxed answer?** Otherwise the model learns to output several answers, hoping one matches.
**Q4. Which is worse in a checker, false negatives or false positives?** False positives: the model learns to exploit them. False negatives add noise.
**Q5. What did Yue et al. (2504.13837) find?** RLVR improves pass@1 but base models have higher pass@$k$ at large $k$; reasoning is bounded by the base model at current training scales.
**Q6. What counter-evidence exists?** ProRL (long training, reference resets, diverse tasks) and CoT-pass@$k$ analyses show boundary expansion.
**Q7. What do spurious-reward results teach?** Gains on one model family can come from amplifying pretraining behaviours via clipping bias, not from the reward; test across families.
**Q8. Why do prompts with success rate near 0 or 1 waste compute?** With group baselines, all-equal groups have zero advantage and give no gradient.
**Q9. How does a verifier with false-positive rate $\alpha$ and false-negative rate $\delta$ change the expected reward?** $\mathbb{E}[\tilde{R}] = \alpha + (1 - \alpha - \delta)p$: the slope in the true success rate shrinks by

**Notation on this page:** $k$: number of attempts in pass@$k$ • $n$: samples drawn • $c$: number of correct samples • $\alpha$: a weighting coefficient (defined in the text) • $\delta_t$: TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $p$: probability of solving the prompt • $R$: return or total reward

$1 - \alpha - \delta$, but the direction is kept if errors do not depend on the response.
**Q10. Why is response-dependent noise worse than random noise?** It changes which behaviour has the highest expected reward, so RL moves toward the exploit.
**Q11. All-pass or fraction-of-tests reward for code?** All-pass for alignment with real correctness and resistance to partial hacks; fraction-passed for denser signal on hard problems, with more hacking risk.
**Q12. When would you use a symbolic checker rather than string match?** Whenever answers can be written in several equivalent forms (fractions, radicals, factorised expressions); string match then gives many false negatives.

**Key takeaways**

- RLVR rewards checkable outcomes: maths answers, unit tests, formats, environment states.
- The checker (extraction, normalisation, equivalence, sandboxing) is a critical, hackable component.
- Use unbiased pass@$k$, avg@$n$ and maj@$k$; keep prompts with mixed success.
- Evidence: RLVR mostly sharpens at moderate scale, can expand with prolonged diverse training; beware single-family results.
- Pick verifiers by error profile; audit them on real outputs.
- Symmetric verifier noise scales the signal by $1 - \alpha - \delta$; response-dependent false positives create reward hacks.
- Code rewards: hidden tests, sandboxed and protected, with time limits and deterministic tests.

CHAPTER 27

# GRPO

**Why this chapter matters.**
**Group Relative Policy Optimization** (GRPO) is the algorithm that made large-scale reasoning RL cheap and popular: DeepSeekMath introduced it, and DeepSeek-R1 used it. It is PPO without a critic: the baseline for each response is the average reward of several responses to the same prompt. It is the default algorithm in TRL's `GRPOTrainer`, veRL and OpenRLHF, and the base of almost every 2025 to 2026 variant. You should be able to write its objective, derive it from PPO, and implement a step.

## 27.1 From PPO to GRPO

PPO for LLMs needs a critic $V_\psi$ the size of the policy (Chapter 20). For reasoning tasks with one reward at the end of a long response, the critic is hard to train and its per-token values are not very accurate (VinePPO, *VinePPO: Refining Credit Assignment in RL Training of LLMs*, Kazemnejad et al., arXiv 2410.01679, found PPO's value networks "barely outperform a random baseline" when ranking alternative steps). What do we actually need the critic for? A baseline: an estimate of how good the prompt's typical response is.

GRPO gets that baseline by **sampling**. For each prompt $q$, sample a **group** of $G$ responses $\{o_1, \dots, o_G\}$ from the old policy, score them with rewards $r_1, \dots, r_G$, and use the group's mean as the baseline, normalised by its standard deviation:

$$\hat{A}_{i,t} = \frac{r_i - \text{mean}(r_1, \dots, r_G)}{\text{std}(r_1, \dots, r_G)} \quad \text{for every token } t \text{ of response } i.$$

Every token in response $i$ gets the same advantage (the sequence-level bandit view of Chapter 13). With binary rewards and success rate $p$ in the group, correct responses get $\hat{A} = (1-p)/\sqrt{p(1-p)} = \sqrt{(1-p)/p}$ and incorrect ones $-\sqrt{p/(1-p)}$ (using the population standard deviation). So a correct answer to a *hard* prompt (small $p$) gets a large positive advantage.

## 27.2 The GRPO objective

From *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models* (Shao et al., arXiv 2402.03300), GRPO maximises

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q,\,\{o_i\}\sim\pi_{\theta_{\text{old}}}}\Bigg[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\Big\{\min\left(\rho_{i,t}\hat{A}_{i,t},\ \text{clip}(\rho_{i,t}, 1-\epsilon, 1+\epsilon)\hat{A}_{i,t}\right) - \beta\,\mathbb{D}_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}]_{i,t}\Big\}\Bigg],$$

**Notation on this page:** $V$: vocabulary size • $q$: a question (prompt) • $G$: group size (responses per prompt) • $o_i$: the $i$-th sampled response; $|o_i|$ its length • $r_i$: reward of response $i$ • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $t$: time step or token position • $p$: a probability distribution (often the target or data) • $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • *and 8 more: see the Notation table*

with the token-level importance ratio $\rho_{i,t} = \pi_\theta(o_{i,t} \mid q, o_{i,<t}) / \pi_{\theta_{\text{old}}}(o_{i,t} \mid q, o_{i,<t})$ and the per-token KL estimate

$$\mathbb{D}_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}]_{i,t} = \frac{\pi_{\text{ref}}(o_{i,t} \mid \cdot)}{\pi_\theta(o_{i,t} \mid \cdot)} - \log \frac{\pi_{\text{ref}}(o_{i,t} \mid \cdot)}{\pi_\theta(o_{i,t} \mid \cdot)} - 1,$$

which is Schulman's $k_3$ estimator (Chapter 3): always non-negative, unbiased as a *value* when the token is sampled from $\pi_\theta$.

Reading it piece by piece:

1. **The clipped surrogate** is PPO's (Chapter 17), per token.
2. **The KL** is a *loss term*, not part of the reward (PPO-RLHF puts it in the reward). This keeps the advantage clean: it depends only on the task reward.
3. **Normalisation**: average over tokens within each response ($1/|o_i|$), then over responses ($1/G$). This "sequence mean of token means" is the choice later papers criticise (next section, and Chapter 28).

DeepSeekMath's settings: learning rate $10^{-6}$, $\beta = 0.04$, $G = 64$ outputs per question, maximum length 1,024, batch size 1,024, and a single policy update per exploration stage. DeepSeek-R1-Zero later used $\beta = 0.001$, $G = 16$ and much longer outputs (Chapter 30).

**Why this, not that? Why a group-mean baseline, not a learned critic?**

**Why.** No value model to train or store (about half the memory and compute of PPO's trainable models), no critic initialisation bias, and an unbiased-in-expectation Monte Carlo baseline that matches the comparative nature of rewards. It works naturally with binary verifiable rewards.
**Trade-off.** Needs $G$ samples per prompt (expensive generation), gives every token in a response the same advantage, gives zero signal when all $G$ rewards are equal, and including response $i$ in its own baseline adds a small bias (RLOO fixes this).
**When the alternative wins.** With a well-trained critic (VC-PPO, VAPO, Chapter 16), PPO can be more sample-efficient on long reasoning; with intermediate rewards in multi-turn tasks, per-step values help credit assignment.

## 27.3 The KL term in practice

In a GRPO loss, the KL term is differentiated, so what matters is its **gradient**, not its value (Chapter 3). Key facts, checked against *On a few pitfalls in KL divergence gradient estimation for RL* (Tang and Munos, arXiv 2506.09477) and *DeepSeek-V3.2* (arXiv 2512.02556):

- With samples from $\pi_{\theta_{\text{old}}} \neq \pi_\theta$ (any update after the first), the $k_3$ value is no longer unbiased for $\text{KL}(\pi_\theta \| \pi_{\text{ref}})$. DeepSeek-V3.2 multiplies it by the importance ratio $\pi_\theta / \pi_{\theta_{\text{old}}}$; TRL implements this as `use_bias_correction_kl=True`.
- The gradient of the $k_3$ *loss* is the gradient of the *forward* KL $\text{KL}(\pi_{\text{ref}} \| \pi_\theta)$, not the reverse KL the objective names; per-token KL losses also miss the dependence of future tokens on earlier ones.
- In practice many reasoning recipes set $\beta = 0$ (DAPO; TRL's default `beta=0.0`), or keep it tiny (R1-Zero 0.001), because verifiable rewards are hard to hack and the KL slows learning

**Notation on this page:** $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\rho$: importance ratio $\pi_\theta / \pi_{\theta_{\text{old}}}$ for a token • $t$: time step or token position • $q$: a question (prompt) • $o_i$: the $i$-th sampled response; $|o_i|$ its length • $\pi_{\text{ref}}$: the frozen reference model • $D_{\text{KL}}(p \| q)$: KL divergence of $q$ from $p$ • $k$: an index or count (see text) • $G$: group size (responses per prompt) • *and 1 more: see the Notation table*

of long reasoning. DeepSeek-V3.2 notes that weaker or no KL can help in maths.

## 27.4 Loss aggregation

Given per-token losses $\ell_{i,t}$ with mask $m_{i,t}$, three common reductions:

| Name | Formula | Used by | Effect |
|---|---|---|---|
| Sequence-mean of token-means | $\frac{1}{G}\sum_i \frac{1}{\lvert o_i\rvert}\sum_t \ell_{i,t}$ | original GRPO (`loss_type="grpo"`) | each response weighs the same; tokens in long responses weigh less |
| Token-mean | $\frac{1}{\sum_i \lvert o_i\rvert}\sum_i \sum_t \ell_{i,t}$ | DAPO (`"dapo"`, TRL default) | each token weighs the same |
| Constant normaliser | $\frac{1}{G\cdot L_{\max}}\sum_i \sum_t \ell_{i,t}$ | Dr. GRPO (`"dr_grpo"`) | no length-dependent weighting at all |

Why it matters: with the original GRPO normalisation, a wrong answer's per-token penalty is divided by its length, so **long wrong answers are penalised less per token than short wrong ones**. Over training this encourages wrong answers to grow longer. Dr. GRPO (*Understanding R1-Zero-Like Training: A Critical Perspective*, Liu et al., arXiv 2503.20783) identified this length bias, along with a difficulty bias from dividing by the group standard deviation. Chapter 28 covers the fixes.

**Why this, not that? Why normalise advantages by the group standard deviation (original GRPO), not just subtract the mean?**

**Why.** It puts every prompt's advantages on the same scale, so prompts with tiny reward differences (e.g. reward-model scores 0.71 vs 0.72) still produce a useful gradient, and it acts like an adaptive learning rate per prompt.
**Trade-off.** For binary rewards it up-weights prompts that are almost always solved or almost always failed (small std), which Dr. GRPO calls a question-level difficulty bias; noise in near-uniform groups gets amplified.
**When the alternative wins.** Dr. GRPO drops the std and uses only the mean; batch-level normalisation (REINFORCE++, ScaleRL; TRL `scale_rewards="batch"`) keeps the scaling benefit without per-prompt bias; TRL also offers `scale_rewards="none"`.

## 27.5 Code: a minimal GRPO step

A tiny task makes the mechanics visible. The "language model" is a small GRU over 12 tokens (digits 0 to 9, an unused token, and an end token). The prompt is a digit $a$; the reward is 1 if the response is a list of digits summing to $10+a$ followed by the end token. Everything else (sampling a group, group-normalised advantages, the clipped ratio, the $k_3$ KL, masking after the end token, sequence-mean-of-token-mean aggregation, $\mu = 2$ updates per batch) is exactly GRPO.

**Notation on this page:** $t$: time step or token position • $\ell_{i,t}$: loss on token $t$ of response $i$ • $m_{i,t}$: loss mask • $G$: group size (responses per prompt) • $o_i$: the $i$-th sampled response; $\lvert o_i\rvert$ its length • $L_{\max}$: maximum completion length • $k$: an index or count (see text) • $\mu$: a mean

```
import torch, torch.nn as nn, torch.nn.functional as F, copy
torch.manual_seed(0)
V, EOS, MAXLEN = 12, 11, 6          # tokens 0-9 are digits, 10 unused, 11 = end of
 ↪  sequence

class TinyLM(nn.Module):
    def __init__(self, d=64):
        super().__init__()
        self.emb = nn.Embedding(V, d); self.rnn = nn.GRU(d, d, batch_first=True);
         ↪  self.out = nn.Linear(d, V)
    def forward(self, ids):                         # (B, T) -> logits (B, T, V)
        h, _ = self.rnn(self.emb(ids)); return self.out(h)

def reward_fn(prompt, completion):
    """Task: given digit a, answer with digits that sum to 10 + a, then EOS. Reward 1 if
     ↪  exact."""
    if EOS not in completion: return 0.0
    digits = completion[:completion.index(EOS)]
    return float(len(digits) > 0 and sum(digits) == 10 + prompt)

@torch.no_grad()
def generate(model, prompts, G):
    ids = prompts.repeat_interleave(G)[:, None]  # (B*G, 1)
    for _ in range(MAXLEN):
        logits = model(ids)[:, -1]
        ids = torch.cat([ids, torch.multinomial(F.softmax(logits, -1), 1)], 1)
    comp = ids[:, 1:]
    # mask: 1 up to and including the first EOS
    is_eos = (comp == EOS).int()
    after = (is_eos.cumsum(1) - is_eos) > 0
    return ids, (~after).float()

def token_logps(model, ids):
    logits = model(ids[:, :-1])
    return torch.log_softmax(logits, -1).gather(-1, ids[:, 1:, None]).squeeze(-1)

def grpo_step(policy, ref, opt, prompts, G=8, eps=0.2, beta=0.04, mu=2):
    ids, mask = generate(policy, prompts, G)
    rewards = torch.tensor([reward_fn(p, c.tolist()) for p, c in
                            zip(prompts.repeat_interleave(G).tolist(), ids[:, 1:])])
    r = rewards.view(-1, G)                                          # (B, G)
    adv = ((r - r.mean(1, keepdim=True)) / (r.std(1, keepdim=True) + 1e-4)).view(-1, 1)
    with torch.no_grad():
        old_logp, ref_logp = token_logps(policy, ids), token_logps(ref, ids)
    for _ in range(mu):                                              # mu updates per batch
        logp = token_logps(policy, ids)
        ratio = (logp - old_logp).exp()
        pg = torch.min(ratio * adv, ratio.clamp(1 - eps, 1 + eps) * adv)
        log_r = ref_logp - logp
        kl = log_r.exp() - log_r - 1                                 # k3 estimator, per token
        per_token = -(pg - beta * kl)
```

```
        loss = ((per_token * mask).sum(1) / mask.sum(1)).mean()   # GRPO: mean over
↪  tokens, then sequences
        opt.zero_grad(); loss.backward(); opt.step()
    return rewards.mean().item(), ((kl * mask).sum(1) / mask.sum(1)).mean().item()

policy = TinyLM()
ref = copy.deepcopy(policy).requires_grad_(False)
opt = torch.optim.Adam(policy.parameters(), lr=3e-3)
for step in range(151):
    prompts = torch.randint(0, 10, (16,))
    mean_r, kl = grpo_step(policy, ref, opt, prompts)
    if step % 25 == 0:
        print(f"step {step:3d}: mean reward {mean_r:.2f}  KL to ref {kl:.3f}")
```

This runs in a few seconds on a CPU. Reward climbs from near 0 to over 0.9 while the KL to the reference grows: the policy moves exactly as far as it needs to. Try `beta=0.0` (faster, larger KL), `G=2` (noisier), or replacing the std normalisation with plain mean subtraction.

## 27.6 Using TRL's `GRPOTrainer`

TRL 1.16.0 parameter names and defaults (checked against the installed version):

```
from trl import GRPOConfig, GRPOTrainer
from trl.rewards import accuracy_reward

config = GRPOConfig(
    output_dir="grpo-out",
    learning_rate=1e-6,                # TRL default
    num_generations=8,                 # G, the group size (default 8)
    max_completion_length=512,         # default 512
    temperature=1.0,                   # sampling temperature (default 1.0)
    beta=0.0,                          # KL coefficient; default 0.0 (no reference model
↪  loaded)
    epsilon=0.2,                       # lower clip; epsilon_high=None means symmetric
    loss_type="dapo",                  # default; also "grpo", "dr_grpo", "bnpo",
↪  "cispo", "sapo", ...
    scale_rewards="group",            # std normalisation: "group" (default), "batch" or
↪  "none"
    num_iterations=1,                  # mu: updates per generation batch (default 1)
    mask_truncated_completions=False,  # True = drop truncated outputs from the loss
↪  (DAPO's overlong filtering)
    use_vllm=True,                     # fast generation (needs vLLM)
)
trainer = GRPOTrainer(model="Qwen/Qwen2.5-0.5B-Instruct", reward_funcs=accuracy_reward,
                      args=config, train_dataset=dataset)   # dataset with "prompt" and
↪  "solution" columns
trainer.train()
```

Two things to notice. TRL's defaults are not the original GRPO: the default `loss_type` is `"dapo"`

(token-mean aggregation) and the default `beta` is 0. And `num_iterations=1` with one gradient step per batch means the ratio is 1 and the clip never activates unless you set `steps_per_generation` or `num_iterations` higher.

## 27.7 Derivation: what gradient does GRPO follow?

Take the simplest case: one gradient step per batch (ratio $\rho = 1$, so the clip is inactive), $\beta = 0$, mean-only advantages $\hat{A}_i = r_i - \bar{r}$ with $\bar{r} = \frac{1}{G}\sum_j r_j$, and write $g_i = \nabla_\theta \log \pi_\theta(o_i \mid q)$ for the score of the whole response (the sum of its token scores). Then the GRPO gradient for one prompt is proportional to $\frac{1}{G}\sum_i \hat{A}_i g_i$ (ignoring the $1/|o_i|$ weights for now). Take its expectation over the group:

$$\mathbb{E}[(r_i - \bar{r})\, g_i] = \Big(1 - \frac{1}{G}\Big)\mathbb{E}[r_i g_i] - \frac{1}{G}\sum_{j\neq i} \mathbb{E}[r_j]\,\underbrace{\mathbb{E}[g_i]}_{\uparrow\, =0} = \frac{G-1}{G}\,\nabla_\theta\, \mathbb{E}_{o\sim\pi_\theta}[r(o)].$$

Two facts were used: the samples are independent, so $r_j$ (for $j \neq i$) is independent of $g_i$; and the score has zero mean, $\mathbb{E}[g_i] = \sum_o \pi(o)\nabla \log \pi(o) = \nabla \sum_o \pi(o) = 0$ (Chapter 15).

So **mean-only GRPO is an unbiased policy gradient scaled by** $(G-1)/G$. Including response $i$ in its own baseline only shrinks the gradient; it does not tilt its direction. RLOO's leave-one-out baseline removes the factor exactly, and in fact $r_i - \bar{r} = \frac{G-1}{G}\big(r_i - \frac{1}{G-1}\sum_{j\neq i} r_j\big)$ (Chapter 28). Dividing by the group standard deviation is different: the std depends on $r_i$ itself and differs between prompts, so the result is no longer a scaled version of the true gradient. This is the bias Dr. GRPO removes. The $1/|o_i|$ weights add a second, length-dependent bias.

We can check the identity numerically on a three-response "prompt" with known rewards:

```
import torch
torch.manual_seed(0)

# A 3-response "prompt": responses 0, 1, 2 with rewards 1, 0, 0.5 (deterministic).
theta = torch.tensor([0.2, 0.5, -0.3], requires_grad=True)
rew = torch.tensor([1.0, 0.0, 0.5])
p = torch.softmax(theta, -1)
true_grad = torch.autograd.grad((p * rew).sum(), theta)[0]     # exact gradient of E[r]

def grpo_grad(G, n_groups=20000, use_std=False):
    p = torch.softmax(theta, -1).detach()
    o = torch.multinomial(p, G * n_groups, replacement=True).view(n_groups, G)
    r = rew[o]
    adv = r - r.mean(1, keepdim=True)
    if use_std:
        adv = adv / (r.std(1, keepdim=True) + 1e-4)
    score = torch.nn.functional.one_hot(o, 3).float() - p       # d log softmax / d theta
    return (adv[..., None] * score).mean((0, 1))

print("true gradient          ", [round(v, 4) for v in true_grad.tolist()])
for G in (2, 4, 16):
    g = grpo_grad(G)
    print(f"G={G:2d} mean-only GRPO   ", [round(v, 4) for v in g.tolist()],
```

**Notation on this page:** $\rho$: importance ratio $\pi_\theta/\pi_{\theta_{\text{old}}}$ for a token • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $r_i$: reward of response $i$ • $G$: group size (responses per prompt) • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\nabla_\theta$: gradient with respect to $\theta$ • $q$: a question (prompt) • $g_i$: score $\nabla \log \pi_\theta(o_i)$ of a whole response • $o_i$: the $i$-th sampled response; $|o_i|$ its length • *and 1 more: see the Notation table*

```
          f" ratio to true {(g @ true_grad / true_grad.norm()**2).item():.3f}  (G-1)/G =
           ↪  {(G-1)/G:.3f}")
print("G= 4 with std (TRL-like)", [round(v, 4) for v in grpo_grad(4,
 ↪  use_std=True).tolist()])
```

The measured ratio matches $(G-1)/G$. With the std division the gradient is larger and its direction changes slightly; the third component moves from about 6% to 10% of the first.

## 27.8 A worked example of group advantages

Take $G = 4$ binary rewards. TRL divides by the **sample** standard deviation (Bessel's correction, $n-1$ in the denominator) plus $10^{-4}$, so its numbers differ from the population-std formula used earlier:

| Rewards | Mean | Sample std | TRL advantages (group std) | Mean-only (Dr. GRPO) |
|---|---|---|---|---|
| 1, 0, 0, 0 | 0.25 | 0.50 | +1.5; others −0.5 | +0.75; others −0.25 |
| 1, 1, 0, 0 | 0.50 | 0.577 | ±0.87 | ±0.50 |
| 1, 1, 1, 0 | 0.75 | 0.50 | +0.5; last −1.5 | +0.25; last −0.75 |
| 1, 1, 1, 1 | 1.00 | 0 | all 0 (no signal) | all 0 |

Read the first row: the single correct answer to a hard prompt gets a big push (+1.5), and each wrong answer a small one. In the third row the single *wrong* answer to an easy prompt gets a big negative push. The sum of advantages in each group is always zero: GRPO only redistributes probability among the group's responses.

## 27.9 The gradient coefficient view (DeepSeekMath's unified paradigm)

DeepSeekMath (arXiv 2402.03300, section 5.2.1) puts SFT, rejection-sampling fine-tuning, DPO, PPO and GRPO in one frame. Every method's gradient has the form

$$\nabla_\theta \mathcal{J} = \mathbb{E}_{(q,o)\sim\mathcal{D}}\Big[\frac{1}{|o|}\sum_t \mathrm{GC}(q,o,t)\,\nabla_\theta \log \pi_\theta(o_t \mid q, o_{<t})\Big],$$

and methods differ in three choices: the **data source** $\mathcal{D}$, the **reward function**, and the **gradient coefficient** GC.

| Method | Data source | GC |
|---|---|---|
| SFT | human/expert responses | 1 |
| RFT (rejection-sampling FT) | samples from the SFT model, filtered | 1 if correct, else 0 |

**Notation on this page:** $G$: group size (responses per prompt) • $n$: number of samples • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\nabla_\theta$: gradient with respect to $\theta$ • $J(\theta)$: objective to maximise (expected reward) • $\mathcal{D}$: a dataset • $t$: time step or token position • $q$: a question (prompt) • $o_i$: the $i$-th sampled response; $|o_i|$ its length

| Method | Data source | GC |
|---|---|---|
| Online RFT | samples from the current policy, filtered | 1 if correct, else 0 |
| DPO | pairs sampled from the SFT model | $+\beta\sigma(\cdot)$ on chosen, $-\beta\sigma(\cdot)$ on rejected |
| PPO / GRPO | samples from the current policy | the advantage (can be negative; graded by reward) |

Two lessons the paper drew: **online** data (from the current policy) beats offline data, and **graded, signed** coefficients (which also push down wrong answers, in proportion to how bad they are) beat 0/1 filtering. Their analysis also found that RL improved Maj@K but **not Pass@K**: it made the output distribution more robust by boosting correct answers already in the top-$K$, rather than adding new abilities. This is an early version of the sharpening debate in Chapter 31.

**Why this, not that? Why GRPO (signed advantages), not rejection-sampling fine-tuning (keep only correct samples)?**

**Why.** RFT's coefficient is 1 for correct samples and 0 for wrong ones, so it never pushes probability *away* from mistakes and treats an easy and a hard correct answer the same. GRPO's signed, prompt-relative advantage does both, and the clipped ratio lets it reuse samples safely.
**Trade-off.** GRPO needs reward differences inside each group and more machinery (old log-probs, clipping, optionally a reference model); RFT is plain SFT on filtered data, trivially stable and easy to scale offline.
**When the alternative wins.** RFT (also called STaR-style or expert iteration) is a strong, cheap baseline, a good warm start before RL, and the standard way to build distillation data (R1's 600k rejection-sampled set, Chapter 30).

## 27.10 Process supervision and iterative GRPO

DeepSeekMath also defined GRPO with **process rewards**. A process reward model scores each step $j$ of each output $o_i$, giving rewards $r_i^{\text{index}(j)}$ placed at the step's last token. All step rewards in the group are normalised together, $\tilde{r} = (r - \text{mean}(\mathbf{R}))/\text{std}(\mathbf{R})$, and each token's advantage is the **sum of the normalised rewards of the steps that end at or after it**:

$$\hat{A}_{i,t} = \sum_{\text{index}(j)\geq t} \tilde{r}_i^{\text{index}(j)}.$$

So tokens early in a solution are credited with everything that follows, and a bad final step lowers only the tokens before it by its own amount. In their experiments process supervision beat outcome supervision. They also used **iterative RL**: after each round, train the reward model further on new samples from the policy (with a replay of 10% historical data), set the reference model to the current policy, and continue.

**Notation on this page:** $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $K_i$: number of steps in response $i$ • $o_i$: the $i$-th sampled response; $|o_i|$ its length • $r_i$: reward of response $i$ • $R$: return or total reward • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $t$: time step or token position

## 27.11 Reward functions in TRL

A TRL reward function is any Python callable that takes the batch and returns one float per completion (or `None` to say "not applicable"):

```
import re

def format_reward(completions, **kwargs):
    """1.0 if the completion has <think>...</think> then an answer, else 0.0."""
    texts = [c[0]["content"] if isinstance(c, list) else c for c in completions]   #
 ↪  conversational or plain
    return [1.0 if re.match(r"(?s)^<think>.*?</think>\s*\S", t) else 0.0 for t in texts]

def correctness_reward(completions, solution, **kwargs):
    """Extra dataset columns (here 'solution') arrive as keyword arguments, one entry
     ↪  per completion."""
    texts = [c[0]["content"] if isinstance(c, list) else c for c in completions]
    return [1.0 if t.strip().endswith(s) else 0.0 for t, s in zip(texts, solution)]

trainer = GRPOTrainer(model=model_name, reward_funcs=[correctness_reward,
 ↪  format_reward],
                      args=GRPOConfig(reward_weights=[1.0, 0.2], output_dir="out"),
 ↪  train_dataset=ds)
```

What TRL 1.16.0 passes, from the source of `_calculate_rewards`: `prompts`, `completions`, `completion_ids`, every extra dataset column (repeated to match the generations), `trainer_state` (for schedules that depend on the step), and helpers `log_extra` and `log_metric` for logging. With several functions, `multi_objective_aggregation="sum_then_normalize"` (default) sums the weighted rewards and then normalises per group; `"normalize_then_sum"` normalises each reward per group first (so a reward with a large scale cannot dominate), then sums and normalises over the batch.

**Why this, not that? Why G = 8 to 16 samples per prompt, not G = 64 as in DeepSeekMath?**

**Why.** Generation dominates cost, and cost grows linearly with $G$. With binary rewards, most of the benefit comes from having at least one correct and one wrong answer in the group; $G = 8$ to 16 already gives a usable baseline, and spending the same budget on more *prompts* gives more diverse gradient signal.
**Trade-off.** Small groups give noisy baselines and more all-same groups on very easy or very hard prompts (see $P(\text{mixed})$ in Chapter 31), wasting a larger share of samples.
**When the alternative wins.** Larger $G$ helps on hard prompts with low pass rates (more chance of one success), with graded rewards where finer ranking matters, and when prompts are scarce.

## 27.12 Compute and memory: GRPO vs PPO

**Notation on this page:** $G$: group size (responses per prompt)

| | PPO-RLHF | GRPO |
|---|---|---|
| Trainable models | policy + critic | policy only |
| Frozen models | reference, reward model | reference (only if $\beta > 0$); reward model or rule |
| Samples per prompt | usually 1 | $G$ (8 to 64) |
| Advantage | GAE from critic, per token | group-relative, per sequence |
| Main cost | training two big models | generation of $G$ samples |

In TRL, `beta=0.0` means no reference model is even loaded; with LoRA and `beta > 0`, the reference is the adapter-disabled base. In large runs the generation engine (vLLM or SGLang) uses most of the GPU time, which is why asynchronous and off-policy pipelines (Chapter 28) matter.

## 27.13 Common mistakes

- Sampling with top-p/top-k or a different temperature than the trainer uses for log-probs. TRL applies the sampling temperature when computing log-probs; custom code must too.
- Forgetting to mask tokens after the end token, or masking the end token itself (then the model never learns to stop).
- Using too small a group (e.g. $G = 2$ with binary rewards gives signal only when exactly one is right).
- Keeping prompts that are always solved or always failed: zero advantage, wasted generation.
- Reading the KL value as a progress metric; read rewards, response length, entropy, and held-out accuracy.
- Assuming TRL's advantages use the population std: it uses the sample std (Bessel) plus $10^{-4}$.
- Returning rewards in a different order from the completions, or one reward per prompt instead of per completion.
- Expecting the KL term to do anything with TRL's default `beta=0.0`.

**Interview check**

**Q1. How does GRPO compute advantages?** $\hat{A}_i = (r_i - \text{mean}(r))/\text{std}(r)$ within the group of $G$ responses to the same prompt; the same value for every token of response $i$.
**Q2. Write the GRPO objective.** Per-token PPO clipped surrogate minus $\beta$ times the $k_3$ KL to the reference, averaged over tokens per response and over the group.
**Q3. How is GRPO's KL different from PPO-RLHF's?** GRPO adds the KL as a loss term with the $k_3$ estimator; PPO-RLHF puts a $k_1$ KL penalty into the per-token reward.
**Q4. With binary rewards and success rate $p$, what advantage does a correct response get?** $\sqrt{(1-p)/p}$ (population std); large for hard prompts.
**Q5. What is the length bias of the original GRPO loss?** Dividing by $|o_i|$ makes long wrong answers less penalised per token, so wrong answers drift longer.
**Q6. What are TRL's GRPO defaults for `loss_type` and `beta`?** `"dapo"` and `0.0`.

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $G$: group size (responses per prompt) • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $r_i$: reward of response $i$ • $k$: an index or count (see text) • $p$: a probability distribution (often the target or data) • $o_i$: the $i$-th sampled response; $|o_i|$ its length

**Q7. When does the clip in GRPO actually matter?** Only when a generation batch is used for more than one gradient step (or rollouts are stale); with one step the ratio is 1.
**Q8. Why is the $k_3$ KL with a ratio correction used in DeepSeek-V3.2?** After the first update, samples come from $\pi_{\theta_{\text{old}}}$; multiplying by $\pi_\theta/\pi_{\theta_{\text{old}}}$ makes the estimate unbiased and its gradient the correct reverse-KL gradient.
**Q9. Show that mean-only GRPO advantages give an unbiased (scaled) gradient.** $\mathbb{E}[(r_i - \bar{r})g_i] = (1 - 1/G)\mathbb{E}[r_i g_i]$ because $r_{j\neq i}$ is independent of $g_i$ and $\mathbb{E}[g_i] = 0$; the direction is the true policy gradient, scaled by $(G-1)/G$.
**Q10. Why does dividing by the group std introduce bias?** The std depends on the sampled rewards (including $r_i$) and varies by prompt, so the estimator is no longer a fixed multiple of the true gradient; it up-weights near-uniform groups.
**Q11. Compute TRL's advantages for rewards (1, 0, 0, 0).** Mean 0.25, sample std 0.5: $+1.5$ for the correct answer, $-0.5$ for each wrong one.
**Q12. What is DeepSeekMath's "gradient coefficient" view?** All methods are $\mathbb{E}[\text{GC} \cdot \nabla \log \pi]$, differing in data source (offline vs online), reward function and GC (1 for SFT, 0/1 for RFT, signed advantage for PPO/GRPO).
**Q13. How does GRPO use process rewards?** Normalise all step rewards in the group, then give each token the sum of normalised rewards of steps ending at or after it.
**Q14. How do you pass extra information (like the reference answer) to a TRL reward function?** Put it in a dataset column; TRL passes each extra column as a keyword argument, one entry per completion.
**Q15. What did DeepSeekMath find about Pass@K vs Maj@K after RL?** RL improved Maj@K but not Pass@K: it sharpened the distribution towards correct answers already reachable.

**Key takeaways**

- GRPO = PPO-clip + group-mean (std-normalised) baseline + $k_3$ KL loss; no critic.
- DeepSeekMath: LR $10^{-6}$, $\beta = 0.04$, $G = 64$; R1-Zero: $\beta = 0.001$, $G = 16$.
- Loss aggregation creates length effects: GRPO (seq-mean-token-mean) vs DAPO (token-mean) vs Dr. GRPO (constant).
- TRL defaults: `loss_type="dapo"`, `beta=0.0`, `num_generations=8`, `learning_rate=1e-6`.
- Mean-only group baseline = true gradient times $(G-1)/G$; std division and $1/|o_i|$ add bias.
- TRL: sample std + $10^{-4}$; reward functions get dataset columns as kwargs.
- Unified view: GC = 1 (SFT), 0/1 (RFT), signed advantage (GRPO); online and signed wins.

**Notation on this page:** $k$: an index or count (see text) • $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\theta$: the model's trainable weights • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $r_i$: reward of response $i$ • $G$: group size (responses per prompt) • $g_i$: score $\nabla \log \pi_\theta(o_i)$ of a whole response • $\nabla_\theta$: gradient with respect to $\theta$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $o_i$: the $i$-th sampled response; $|o_i|$ its length

CHAPTER 28

# GRPO Variants and Critic-Free Methods

**Why this chapter matters.**
Between 2025 and 2026, dozens of papers modified GRPO. Most change one of five things: **how the loss is averaged** (length bias), **how advantages are normalised** (difficulty bias), **how the importance ratio is clipped or weighted** (exploration and stability), **which samples are used** (dynamic sampling, filtering, masking), and **how to stay correct when rollouts are off-policy** (asynchronous training, train/inference mismatch, MoE routing). If you understand these five axes, every new variant is easy to place. TRL 1.16.0 exposes most of them as `loss_type` values and flags of `GRPOConfig`, so they are also easy to try.

## 28.1 A common frame

Write every method in this chapter as minimising a per-token loss, aggregated over tokens:

$$\mathcal{L}(\theta) = \frac{1}{Z}\sum_{i=1}^{G}\sum_{t=1}^{|o_i|} m_{i,t}\,\ell_{i,t}(\theta), \qquad \ell_{i,t} = -\,w_{i,t}(\theta)\,\hat{A}_i \;[\times \log \pi_\theta(o_{i,t}) \text{ for REINFORCE-style forms}],$$

where $m_{i,t}$ is a mask (which tokens count), $w_{i,t}$ is an importance weight (possibly clipped), $\hat{A}_i$ is the advantage, and $Z$ is the normaliser. The methods differ in:

| Axis | Choices |
|---|---|
| Normaliser $Z$ | per-sequence length (GRPO), total tokens (DAPO), constant (Dr. GRPO), … |
| Advantage $\hat{A}_i$ | group mean and std (GRPO), group mean only (Dr. GRPO), leave-one-out (RLOO), group mean + global std (REINFORCE++ w/ baseline) |
| Weight $w$ | clipped token ratio (PPO/GRPO), asymmetric clip (DAPO), sequence ratio (GSPO), clipped stop-gradient weight (CISPO), soft gate (SAPO) |
| Mask $m$ | truncated responses, high-KL negative sequences (V3.2), low-entropy tokens (80/20), train/inference mismatch masks |
| Samples | dynamic sampling, difficulty filtering, overlong filtering |

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $A, \hat{A}$: advantage (how much better an action is than average); hat = estimate • $\mathcal{L}$: a loss to minimise • $t$: time step or token position • $G$: group size • $\ell_{i,t}$: per-token loss • $m_{i,t}$: loss mask • $o_i$: the $i$-th sampled response; $|o_i|$ its length

## 28.2 Dr. GRPO: removing two biases

*Understanding R1-Zero-Like Training: A Critical Perspective* (Liu et al., arXiv 2503.20783) found two biases in GRPO's objective.

**Bias 1: response-length normalisation.** GRPO divides each response's token sum by $|o_i|$. Consider the gradient contribution of one response with advantage $\hat{A}_i$:

$$\nabla_\theta \frac{1}{|o_i|} \sum_t \rho_{i,t} \hat{A}_i \;\approx\; \frac{\hat{A}_i}{|o_i|} \sum_t \nabla_\theta \log \pi_\theta(o_{i,t}).$$

Each token's push is scaled by $1/|o_i|$. For **correct** answers ($\hat{A} > 0$), short responses get a larger per-token push, so the model is nudged towards brevity among correct answers. For **incorrect** answers ($\hat{A} < 0$), long responses get a *smaller* per-token penalty, so being long and wrong is penalised less. Over training, incorrect responses grow longer, which looks like "the model is thinking more" but is partly an artefact of the loss.

**Bias 2: standard-deviation normalisation.** Dividing by $\mathrm{std}(r)$ gives larger weight to prompts whose rewards vary little: prompts the model almost always solves or almost always fails. With binary rewards, $\mathrm{std} = \sqrt{p(1-p)}$, which is small near $p = 0$ or 1. Dr. GRPO calls this a **question-level difficulty bias**.

**The fix ("GRPO Done Right").** Remove both: use $\hat{A}_i = r_i - \mathrm{mean}(r)$, and normalise the token sum by a **constant** (they use the maximum generation length, `MAX_TOKENS`) instead of $|o_i|$. The paper notes that the same length-normalisation issue exists in many open-source PPO implementations. Their minimalist R1-Zero recipe reached 43.3% on AIME 2024 with a 7B base model (Oat-Zero-7B). In TRL: `loss_type="dr_grpo"` (normaliser = batch size times `max_completion_length`) with `scale_rewards="none"`.

```
import torch

def per_token_weight(lengths, adv, kind):
    """Gradient weight each token of response i receives (ignoring the ratio, which is 1
     ↪ on-policy)."""
    L = torch.tensor(lengths, dtype=torch.float)
    if kind == "grpo":    w = adv / L / len(L)            # 1/|o_i|, then mean over G
    elif kind == "dapo":  w = adv / L.sum() * torch.ones_like(L)
    elif kind == "dr":    w = adv / (len(L) * 1000.0) * torch.ones_like(L)   # constant:
     ↪ G * MAX_TOKENS
    return w

lengths = [100, 400, 100, 400]                # two short and two long responses
r = torch.tensor([1.0, 1.0, 0.0, 0.0])        # short/long correct, short/long wrong
adv = r - r.mean()
for kind in ("grpo", "dapo", "dr"):
    w = per_token_weight(lengths, adv, kind)
    total = w * torch.tensor(lengths)
    print(f"{kind:5s} per-token weight {[f'{x:+.5f}' for x in w.tolist()]}  "
          f"total push per response {[f'{x:+.3f}' for x in total.tolist()]}")
```

**Notation on this page:** $o_i$: the $i$-th sampled response; $|o_i|$ its length • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\nabla_\theta$: gradient with respect to $\theta$ • $\rho$: importance ratio $\pi_\theta/\pi_{\theta_{\text{old}}}$ for a token • $t$: time step or token position • $r_i$: reward of response $i$; $r_{i,t}$ token ratio • $p$: a probability distribution (often the target or data)

Read the "total push" column: under GRPO every response gets the same total push regardless of length, so each token of a long wrong answer is penalised four times less than a token of a short wrong one. Under DAPO and Dr. GRPO, every token is weighted equally, so a long wrong answer receives four times the total penalty of a short one.

**Why this, not that? Why a constant or token-level normaliser (Dr. GRPO, DAPO), not GRPO's per-response length?**

**Why.** Per-response length normalisation makes the per-token gradient depend on length, which biases the policy towards long wrong answers and short right ones. Equal per-token weighting removes this artefact and gives cleaner, more token-efficient training.
**Trade-off.** With token-level weighting, long responses dominate the gradient; if long responses are noisy (rambling, repetition), their noise dominates too. A fixed constant normaliser (Dr. GRPO) makes the gradient scale depend on how much of the length budget is used.
**When the alternative wins.** If responses have very different lengths for reasons unrelated to quality (mixed tasks: short-answer QA with long proofs), per-sequence normalisation stops long tasks from swamping short ones. ScaleRL (arXiv 2510.13786) found prompt-level averaging worked best in its recipe.

## 28.3 DAPO: four practical techniques

*DAPO: An Open-Source LLM Reinforcement Learning System at Scale* (Yu et al., arXiv 2503.14476) reached 50 points on AIME 2024 with Qwen2.5-32B as the base model, using 50% of the training steps of DeepSeek-R1-Zero-Qwen-32B. It is built on the veRL framework and has no KL term. Its four techniques:

**1. Clip-Higher.** Decouple the clip range: $\text{clip}(\rho, 1-\epsilon_{\text{low}}, 1+\epsilon_{\text{high}})$ with $\epsilon_{\text{low}} = 0.2$, $\epsilon_{\text{high}} = 0.28$.

*Why it helps.* The upper clip limits how much a token's probability can grow in one update. For a token with old probability 0.9, the cap $1.2 \times 0.9 = 1.08$ is irrelevant (probability cannot exceed 1). For a rare token with old probability 0.01, the cap allows only 0.012. So the symmetric clip mostly restricts **low-probability "exploration" tokens** from growing, which speeds up **entropy collapse**. Raising $\epsilon_{\text{high}}$ gives rare-but-good tokens more room. The lower clip stays at 0.2 so probabilities are not crushed to zero.

**2. Dynamic sampling.** If all $G$ responses to a prompt have the same reward (all correct or all wrong), every advantage is zero and the prompt contributes nothing, but it still counts in the batch, shrinking the effective batch. DAPO **over-samples** prompts and keeps only groups with $0 < \#\text{correct} < G$, filling the batch with informative groups.

**3. Token-level policy-gradient loss.** Normalise by the total number of tokens in the batch, $\frac{1}{\sum_i |o_i|} \sum_i \sum_t$, so every token has equal weight (the "dapo" row above). This fixes GRPO's length bias, and long, low-quality responses (gibberish, repetition) are penalised in proportion to their length.

**4. Overlong reward shaping.** Responses cut off at the length limit get punished even if their reasoning was correct, which adds noise. DAPO uses **overlong filtering** (mask truncated samples

**Notation on this page:** $\epsilon$: clip range for the probability ratio • $\rho$: importance ratio $\pi_\theta / \pi_{\theta_{\text{old}}}$ for a token • $\text{clip}(u, a, b)$: limit $u$ to the range $[a, b]$ • $G$: group size • $t$: time step or token position • $o_i$: the $i$-th sampled response; $|o_i|$ its length

out of the loss) and a **soft overlong punishment**: with maximum length $L_{\max}$ and a buffer $L_{\text{cache}}$,

$$R_{\text{length}}(y) = \begin{cases} 0 & |y| \le L_{\max} - L_{\text{cache}} \\ \dfrac{(L_{\max} - L_{\text{cache}}) - |y|}{L_{\text{cache}}} & L_{\max} - L_{\text{cache}} < |y| \le L_{\max} \\ -1 & |y| > L_{\max} \end{cases}$$

added to the correctness reward. DAPO used $L_{\max} = 20{,}480$ and $L_{\text{cache}} = 4{,}096$ (so the penalty ramps from 0 at 16,384 tokens to $-1$ at 20,480).

**Other DAPO settings.** Learning rate $10^{-6}$ (constant, 20 warmup steps), prompt batch 512 with 16 responses each, mini-batch 512 (so 16 gradient updates per rollout batch), no KL, the DAPO-Math-17K dataset with integer answers, evaluation at temperature 1.0 and top-p 0.7 with avg@32.

```
import torch

def soft_overlong(length, L_max=20480, L_cache=4096):
    if length <= L_max - L_cache:
        return 0.0
    if length <= L_max:
        return ((L_max - L_cache) - length) / L_cache
    return -1.0

def dynamic_sampling(rewards, G):
    """Keep only prompt groups with mixed outcomes. rewards: (num_prompts * G,) of
     ↪ 0/1."""
    r = rewards.view(-1, G)
    keep = (r.sum(1) > 0) & (r.sum(1) < G)
    return keep

print([round(soft_overlong(n), 2) for n in (10000, 16384, 18432, 20480, 25000)])
r = torch.tensor([1, 1, 1, 1,  0, 1, 0, 0,  0, 0, 0, 0,  1, 0, 1, 1.])
print("groups kept:", dynamic_sampling(r, 4).tolist())
```

TRL equivalents: `epsilon=0.2`, `epsilon_high=0.28`, `loss_type="dapo"` (the default), `mask_truncated_completions=True` for overlong filtering, and `trl.rewards.get_soft_overlong_punishment(max_completion_len, soft_punish_cache)` as an extra reward function. Dynamic sampling has no single flag in TRL; you can approximate it by filtering the dataset by difficulty (Chapter 31).

**Why this, not that? Why clip-higher (asymmetric clipping), not a symmetric clip or an entropy bonus?**

**Why.** The symmetric clip mainly blocks rare tokens from growing, which is exactly how entropy collapses. Raising only the upper bound targets that mechanism directly, without adding a global entropy bonus that can push the model into random text across a 150k-token vocabulary.
**Trade-off.** Larger upward steps make training less conservative; too high an $\epsilon_{\text{high}}$ can destabilise training, and the right value depends on model and task.
**When the alternative wins.** For small or very stable runs, the symmetric 0.2 clip is fine. If

**Notation on this page:** $L_{\max}$: maximum length; $L_{\text{cache}}$: soft-penalty window • $R$: return or total reward • $\epsilon$: clip range for the probability ratio

entropy still collapses, targeted methods (Clip-Cov/KL-Cov from the entropy-mechanism paper, high-entropy-token training) or an adaptive entropy coefficient may be better.

**Why this, not that? Why dynamic sampling, not just a bigger batch?**

**Why.** Zero-advantage groups cost generation but give no gradient. Filtering them keeps the number of *useful* samples per step constant as the model improves and more prompts become "always solved".
**Trade-off.** You must generate extra rollouts to fill the batch (more wall-clock per step, and variable cost), and you discard data that might matter later (e.g. as a KL anchor).
**When the alternative wins.** Pre-filtering the dataset by difficulty once (offline) is cheaper; and with a dense or continuous reward (reward models), all-equal groups are rare, so dynamic sampling adds little.

## 28.4 RLOO: leave-one-out baselines

*Back to Basics: Revisiting REINFORCE Style Optimization for Learning from Human Feedback in LLMs* (Ahmadian et al., arXiv 2402.14740) argues that PPO's machinery is unnecessary for LLM fine-tuning: the starting policy is already strong, and the high-variance problems PPO was designed for in robotics are less severe. They propose **REINFORCE Leave-One-Out** (RLOO): sample $k$ responses and use, for each response, the mean of the *other* $k-1$ rewards as its baseline:

$$\hat{A}_i = r_i - \frac{1}{k-1}\sum_{j\neq i} r_j.$$

Because the baseline does not include $r_i$, it is independent of response $i$, so the gradient estimate is **unbiased** (Chapter 15). A neat identity shows how this relates to the group mean $\bar{r}$:

$$r_i - \frac{1}{k-1}\sum_{j\neq i} r_j = r_i - \frac{k\bar{r} - r_i}{k-1} = \frac{k}{k-1}(r_i - \bar{r}).$$

So RLOO's advantage is the mean-centred advantage scaled by $k/(k-1)$: identical in direction to Dr. GRPO's advantage, just larger. In RLHF settings the paper found RLOO beat PPO, DPO and RAFT on their benchmarks while using less memory.

TRL 1.16.0 has a dedicated `RLOOTrainer`/`RLOOConfig` (defaults: `num_generations=2`, `beta=0.05` with the KL added to the reward, `normalize_advantages=False`, `epsilon=0.2`).

```
import torch
r = torch.tensor([1.0, 0.0, 0.0, 1.0, 1.0])
k = len(r)
loo = r - (r.sum() - r) / (k - 1)
print(loo, (k / (k - 1)) * (r - r.mean()))     # identical
```

**Notation on this page:** $k$: an index or count (see text) • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $r_i$: reward of response $i$; $r_{i,t}$ token ratio

## 28.5 REINFORCE++ and ReMax

**REINFORCE++.** *REINFORCE++: Stabilizing Critic-Free Policy Optimization with Global Advantage Normalization* (Hu et al., arXiv 2501.03262) argues that normalising advantages within small prompt groups is biased and overfits, and instead normalises **across the whole global batch**. Two variants:

- **REINFORCE++** ($k \geq 1$ samples per prompt): reward with a KL penalty folded in, return computed per token, then advantages normalised by the batch mean and standard deviation. Suited to general RLHF where you sample one response per prompt.
- **REINFORCE++ with baseline** ($k > 1$): subtract the **group mean** (a local baseline), then divide by the **global batch** standard deviation; the KL is applied as a loss (the paper favours the $k_2$ estimator). Suited to reasoning tasks.

TRL's `scale_rewards="batch"` gives the "group mean, batch std" form inside `GRPOTrainer`.

**ReMax.** *ReMax: A Simple, Effective, and Efficient Reinforcement Learning Method for Aligning Large Language Models* (Li et al., arXiv 2310.10505) uses the reward of the **greedy** response as the baseline: $\hat{A} = r(x, y_{\text{sampled}}) - r(x, y_{\text{greedy}})$. One extra greedy generation per prompt replaces the critic. Its weakness: when the greedy answer is right, every sampled correct answer gets advantage 0.

**Why this, not that? Why batch-level advantage normalisation (REINFORCE++), not per-group normalisation (GRPO)?**

**Why.** The batch standard deviation is estimated from many more samples, so it is stable, and it does not up-weight near-solved or near-impossible prompts the way per-group std does. Advantages from different prompts stay on a comparable scale.
**Trade-off.** Prompts with small reward differences (e.g. reward-model scores) get tiny advantages and contribute little; the gradient scale depends on the batch mix.
**When the alternative wins.** Per-group std helps when reward scales differ wildly across prompts (different reward models or tasks in one batch), where a global scale would let one task dominate.

## 28.6 GSPO: sequence-level importance ratios

*Group Sequence Policy Optimization* (Zheng et al., Qwen team, arXiv 2507.18071) argues that GRPO's **token-level** ratio is ill-posed: the reward is given to the whole sequence, but the importance correction is applied per token, using a single sample per token position, which is a very noisy estimate of the true distribution shift. With long responses, this noise accumulates and can cause **model collapse**, which they observed especially in **mixture-of-experts** (MoE) models, where a token's routed experts can change between the old and new policy, making token ratios swing wildly.

GSPO uses a **length-normalised sequence ratio** and clips the whole sequence:

$$s_i(\theta) = \Big(\frac{\pi_\theta(y_i \mid x)}{\pi_{\theta_{\text{old}}}(y_i \mid x)}\Big)^{1/|y_i|} = \exp\Big(\frac{1}{|y_i|}\sum_t \log \frac{\pi_\theta(y_{i,t} \mid x, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t} \mid x, y_{i,<t})}\Big),$$

**Notation on this page:** $k$: an index or count (see text) • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $r_i$: reward of response $i$; $r_{i,t}$ token ratio • $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $t$: time step or token position

$$\mathcal{J}_{\text{GSPO}}(\theta) = \mathbb{E}\Big[\frac{1}{G}\sum_{i=1}^{G} \min\left(s_i(\theta)\hat{A}_i,\ \text{clip}(s_i(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_i\right)\Big].$$

$s_i$ is the **geometric mean** of the token ratios. The $1/|y_i|$ exponent keeps it on a comparable scale for short and long responses (the raw sequence ratio of a 10,000-token response would explode). Because $s_i$ is much closer to 1 than individual token ratios, the clip range is tiny: the paper used $3 \times 10^{-4}$ (left) and $4 \times 10^{-4}$ (right), compared with 0.2 and 0.27 in its GRPO baseline. Despite clipping a far larger fraction of tokens than GRPO, GSPO trained more efficiently.

**Gradient.** Differentiating, every token of response $i$ gets the same weight $s_i(\theta)\hat{A}_i/|y_i|$ times its $\nabla \log \pi_\theta$: all tokens in a sequence are treated equally, unlike GRPO, where each token has its own ratio. The paper also gives **GSPO-token**, which allows per-token advantages while keeping the sequence-level ratio value (via a stop-gradient trick).

**MoE.** Previously the Qwen team needed **Routing Replay** (caching the experts the old policy used and forcing the new policy to use them when computing ratios) to train MoE models stably with GRPO. GSPO removes this need because the sequence likelihood is robust to individual routing changes. GSPO was used to train the Qwen3 models.

TRL: `importance_sampling_level="sequence"`, with small `epsilon`/`epsilon_high` values (e.g. `3e-4` and `4e-4`).

**Why this, not that? Why a sequence-level ratio (GSPO), not per-token ratios (GRPO)?**

**Why.** The reward is per sequence, so the off-policy correction should be too. A geometric-mean ratio is low-variance, robust to single-token noise and MoE routing changes, and makes clipping decisions per response, which stabilised large MoE training.
**Trade-off.** All tokens in a response share one weight, so a few badly off-policy tokens can cause the whole response to be clipped (losing its signal), and token-level credit is lost. The clip range needs re-tuning (four orders of magnitude smaller).
**When the alternative wins.** For dense models with on-policy or near-on-policy data, token-level ratios work well and keep finer control. SAPO (below) tries to keep sequence coherence while down-weighting only the offending tokens.

## 28.7 CISPO: clip the weight, keep every token

*MiniMax-M1: Scaling Test-Time Compute Efficiently with Lightning Attention* (MiniMax, arXiv 2506.13585) observed that in PPO/GRPO, tokens whose ratio leaves the clip range get **zero gradient**. In their runs, the clipped tokens were often rare, high-value "fork" tokens such as *However*, *Recheck*, *Wait*, *Aha*, which start reflective reasoning. Low probability under the old policy means they easily exceed $1+\epsilon$ after one update and are then dropped from further updates, so reflective behaviour cannot grow.

**CISPO** (Clipped IS-weight Policy Optimization) clips the **importance weight** but never drops a token's gradient:

$$\mathcal{J}_{\text{CISPO}}(\theta) = \mathbb{E}\Big[\frac{1}{\sum_i |o_i|}\sum_{i=1}^{G}\sum_{t=1}^{|o_i|} \text{sg}(\hat{r}_{i,t}(\theta))\,\hat{A}_{i,t}\,\log \pi_\theta(o_{i,t} \mid q, o_{i,<t})\Big],$$

**Notation on this page:** $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\epsilon$: clip range for the probability ratio • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $J(\theta)$: objective to maximise (expected reward) • $\text{clip}(u, a, b)$: limit $u$ to the range $[a, b]$ • $G$: group size • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\nabla_\theta$: gradient with respect to $\theta$ • $\text{sg}(\cdot)$: stop-gradient (treated as a constant in backprop) • $r_i$: reward of response $i$; $r_{i,t}$ token ratio • *and 3 more: see the Notation table*

$$\hat{r}_{i,t} = \text{clip}(r_{i,t}, 1 - \epsilon^{IS}_{\text{low}}, 1 + \epsilon^{IS}_{\text{high}}),$$

where $r_{i,t}$ is the token ratio and sg is stop-gradient. This is a REINFORCE-style objective ($\hat{A} \log \pi$) with a **bounded, constant** importance weight: the weight controls the step size, and the gradient always flows through $\log \pi$. Without the clip it reduces to the standard off-policy policy gradient. MiniMax set $\epsilon^{IS}_{\text{low}}$ large (effectively no lower bound) and tuned only the upper bound. They also used DAPO's dynamic sampling and length penalty, no KL, and reported a **2x speed-up over DAPO** in a controlled comparison on Qwen2.5-32B.

Other MiniMax-M1 RL lessons: a **precision mismatch** between the training and inference engines' LM head made training and inference probabilities diverge and blocked reward growth; computing the **LM head in fp32** fixed it. The full RL run used 512 H800 GPUs for three weeks, at a rental cost of about $534,700.

*The Art of Scaling Reinforcement Learning Compute for LLMs* (Khatri et al., arXiv 2510.13786, "ScaleRL") ran more than 400,000 GPU-hours of ablations, fitted **sigmoidal** compute-performance curves, and found that loss aggregation, normalisation, curriculum and off-policy algorithm mainly change **compute efficiency**, while some choices also change the **asymptote**. Its recommended recipe (ScaleRL) uses CISPO as the loss and fp32 logits. TRL: `loss_type="cispo"` with `epsilon_high` interpreted as the maximum weight $\epsilon_{\text{max}}$ (TRL's documentation recommends 5.0, following ScaleRL); TRL's implementation clamps the ratio from above at `epsilon_high` and detaches it.

```
import torch

def clipped_ppo_grad(logp, logp_old, adv, eps_low=0.2, eps_high=0.28):
    logp = logp.clone().requires_grad_(True)
    ratio = (logp - logp_old).exp()
    loss = -torch.min(ratio * adv, ratio.clamp(1 - eps_low, 1 + eps_high) * adv).sum()
    loss.backward(); return logp.grad

def cispo_grad(logp, logp_old, adv, eps_max=5.0):
    logp = logp.clone().requires_grad_(True)
    w = (logp - logp_old).exp().clamp(max=eps_max).detach()
    loss = -(w * adv * logp).sum()
    loss.backward(); return logp.grad

# Three tokens after one update: a common token, a rare "Wait" token that grew a lot, a
 ↪  normal one.
logp_old = torch.log(torch.tensor([0.90, 0.02, 0.30]))
logp_new = torch.log(torch.tensor([0.92, 0.05, 0.31]))    # "Wait": ratio 2.5
adv = torch.tensor([1.0, 1.0, 1.0])
print("PPO/DAPO grad:", clipped_ppo_grad(logp_new, logp_old, adv))
print("CISPO grad:   ", cispo_grad(logp_new, logp_old, adv))
```

The rare token's loss gradient is exactly zero under the clipped surrogate, so its probability stops growing. Under CISPO it is $-2.5$ (the capped weight times the advantage), so gradient descent keeps raising its probability, with a bounded step. The other two tokens are treated the same way by both methods.

**Notation on this page:** $\epsilon$: clip range for the probability ratio • $\text{clip}(u, a, b)$: limit $u$ to the range $[a, b]$ • $r_i$: reward of response $i$; $r_{i,t}$ token ratio • $t$: time step or token position • $\text{sg}(\cdot)$: stop-gradient (treated as a constant in backprop) • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate

**Why this, not that? Why clip the importance weight (CISPO), not the objective (PPO)?**

**Why.** PPO's clip removes the gradient of any token that has moved too far, and those are often the rare, important reasoning tokens. CISPO bounds the *size* of each token's contribution but always keeps it, so every sampled token helps learning; this improved efficiency in MiniMax-M1 and in ScaleRL's large ablations.
**Trade-off.** No trust region in the PPO sense: with many updates per batch, a token's probability can keep moving in the same direction; the clipped weight biases the gradient slightly; the weight cap needs tuning.
**When the alternative wins.** With few updates per batch (nearly on-policy), PPO-style clipping rarely activates and the difference is small; where conservative updates matter (fragile models, large step sizes), PPO's hard trust region is safer.

## 28.8 SAPO and other soft gates

*Soft Adaptive Policy Optimization* (Gao et al., Qwen team, arXiv 2511.20347) replaces hard clipping with a smooth, temperature-controlled **gate**. In TRL's implementation, the per-token surrogate is

$$\ell_{i,t} = -\frac{4}{\tau}\,\sigma(\tau\,(\rho_{i,t}-1))\,\hat{A}_i,$$

with temperature $\tau$ (TRL defaults: `sapo_temperature_pos=1.0` for positive advantages, `sapo_temperature_neg=1.05` for others). Its derivative with respect to $\rho$ is $4\,\sigma(1-\sigma)\hat{A}$, which equals $\hat{A}$ at $\rho = 1$ (like the unclipped objective on-policy) and decays smoothly as $\rho$ moves away from 1, instead of dropping to zero at a hard edge. The paper describes it as sequence-coherent and token-adaptive: like GSPO it behaves well at the sequence level, but when a sequence has a few highly off-policy tokens, only those tokens are down-weighted.

Other recent options exposed by TRL 1.16.0 (listed so you recognise them):

- **INTELLECT-2's two-sided clipping** (`delta`, arXiv 2505.07291): for negative advantages, the unclipped term $\rho\hat{A}$ can be arbitrarily large when $\rho \gg 1$; capping $\rho$ at $\delta > 1 + \epsilon$ bounds it (similar to dual-clip PPO, Chapter 17).
- **LUSPO** (`"luspo"`, arXiv 2602.05261): a length-unbiased sequence-level objective.
- **VESPO** (`"vespo"`, arXiv 2602.10693): a variational sequence-level soft weighting for stable off-policy training.
- **BNPO** (`"bnpo"`): token-level averaging within the local batch.

## 28.9 Masking: which tokens and sequences to train on

**Truncated responses** (`mask_truncated_completions`): remove from the loss (DAPO's overlong filtering).

**High-entropy tokens only.** *Beyond the 80/20 Rule: High-Entropy Minority Tokens Drive Effective Reinforcement Learning for LLM Reasoning* (Wang et al., arXiv 2506.01939) found that only a small fraction of tokens in a chain of thought have high entropy, and these act as **forks** that decide the reasoning path. RLVR mainly changes these tokens' entropies. Restricting the policy gradient

**Notation on this page:** $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $\rho$: importance ratio $\pi_\theta/\pi_{\theta_{\text{old}}}$ for a token • $\tau$: SAPO gate temperature • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $t$: time step or token position • $\ell_{i,t}$: per-token loss • $\epsilon$: clip range for the probability ratio • $\delta_t$: TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$

to the top 20% highest-entropy tokens matched full training on Qwen3-8B and clearly beat it on Qwen3-14B and Qwen3-32B (for example +11.04 on AIME'25 for 32B); training on only the 80% lowest-entropy tokens hurt badly. TRL: `top_entropy_quantile=0.2` (default 1.0 = all tokens).

**Off-policy sequence masking** (DeepSeek-V3.2, arXiv 2512.02556): when the data is off-policy, mask sequences that have **negative advantage** and a **high KL** between the sampling policy and the current policy (above a threshold $\delta$). The reasoning: learning to avoid behaviour the current policy no longer produces is noisy and destabilising, while positive examples remain useful. TRL: `off_policy_mask_threshold` (e.g. 0.5).

## 28.10 Off-policy data, asynchronous RL and train/inference mismatch

Three sources make "on-policy" RL quietly off-policy:

1. **Multiple updates per batch** ($\mu > 1$ or several mini-batches): handled by the clipped ratio $\pi_\theta / \pi_{\theta_{\text{old}}}$.
2. **Asynchronous training**: generation runs one or more steps behind training, so rollouts come from an older policy. Needs the ratio to the *actual* sampling policy, plus limits on staleness.
3. **Train/inference mismatch**: rollouts are generated by an inference engine (vLLM, SGLang) with different kernels, precision, batch-dependent numerics, or quantisation. So even with identical weights, $\pi_{\text{infer}}(y_t) \neq \pi_{\text{train}}(y_t)$. Over long responses, small per-token differences add up.

**Truncated importance sampling (TIS).** A widely used fix, popularised in 2025 by a blog post from Yao and colleagues ("Your Efficient RL Framework Secretly Brings You Off-Policy RL Training"): multiply each token's loss by a **capped** correction $\min(\pi_{\text{train}}(y_t)/\pi_{\text{infer}}(y_t), C)$, where $\pi_{\text{train}}$ is the trainer's probability under the rollout weights and $\pi_{\text{infer}}$ is the probability reported by the inference engine. **Masked IS** (MIS) instead drops tokens or sequences whose ratio is out of range. TRL applies such a correction by default when using vLLM: `vllm_importance_sampling_correction=True`, `vllm_importance_sampling_mode="sequence_mask"` (sequence-level ratios, masking sequences outside the bounds), `vllm_importance_sampling_clip_max=3.0`; token-level and truncation modes are also available.

**Engineering fixes** reduce the mismatch at the source:

- **fp32 LM head** (MiniMax-M1), deterministic or batch-invariant kernels.
- *Defeating the Training-Inference Mismatch via FP16* (Qi et al., arXiv 2510.26788): bf16's few mantissa bits cause large rounding differences between engines; switching both training and inference to **fp16** (with loss scaling) largely removed the mismatch and gave more stable, faster RL in their experiments.
- **MoE routing**: replay the experts chosen at inference when computing training log-probs (**Routing Replay**, also used in DeepSeek-V3.2 as "Keep Routing"), or use sequence-level methods (GSPO) that tolerate routing changes.
- **Sampling masks**: if rollouts use top-p/top-k, the sampled token comes from a *truncated* distribution. DeepSeek-V3.2's **Keep Sampling Mask** stores the truncation mask and applies it to $\pi_\theta$ when computing ratios, so both sides use the same action space.

**Notation on this page:** $\delta_t$: TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$ • $\mu$: a mean • $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $t$: time step or token position

**Why this, not that? Why correct the train/inference mismatch with importance weights, not just make the engines match?**

**Why.** Exact numerical agreement between a fast inference engine and a training framework is very hard (different kernels, batch sizes, parallelism, precision). An importance-weight correction (truncated or masked) works with any engine and turns the problem into a known off-policy correction.
**Trade-off.** Truncation adds bias; masking throws away data; and large mismatches mean the correction is a band-aid on a real numerical problem, which can still destabilise training.
**When the alternative wins.** Where possible, fix the source too: fp32 LM heads, fp16 instead of bf16, routing replay for MoE, and consistent sampling masks. The best systems do both.

## 28.11 One comparison table

| Method | Advantage | Ratio / weight | Normaliser | Key idea | TRL |
|---|---|---|---|---|---|
| GRPO | group mean/std | token, clip $\pm\epsilon$ | per response, then group | critic-free PPO | `loss_type="grpo"` |
| Dr. GRPO | group mean | token clip | constant ($G \cdot L_{\max}$) | remove length and std biases | `"dr_grpo"`, `scale_rewards="none"` |
| DAPO | group mean/std | token, clip $[1-0.2, 1+0.28]$ | all tokens | clip-higher, dynamic sampling, overlong shaping | `"dapo"` (default) |
| RLOO | leave-one-out | none / PPO clip | per sequence | unbiased baseline | `RLOOTrainer` |
| REINFORCE++ | global batch norm (+ group mean) | PPO clip | tokens | global normalisation | `scale_rewards="batch"` |
| ReMax | minus greedy reward | none | | greedy baseline | not in TRL |
| GSPO | group mean/std | sequence geometric mean, clip $3\text{–}4 \times 10^{-4}$ | per sequence | sequence-level IS, MoE stability | `importance_sampling_level="sequence"` |

**Notation on this page:** $\epsilon$: clip range for the probability ratio • $G$: group size • $L_{\max}$: maximum length; $L_{\text{cache}}$: soft-penalty window

| Method | Advantage | Ratio / weight | Normaliser | Key idea | TRL |
|---|---|---|---|---|---|
| CISPO | group | clipped stop-grad weight $\times \log \pi$ | all tokens | keep every token's gradient | `"cispo"` |
| SAPO | group | soft sigmoid gate | per sequence | smooth trust region | `"sapo"` |

## 28.12 Common mistakes

- Comparing methods with different normalisers at the same learning rate: the effective step size differs.
- Using GSPO with GRPO's clip range (0.2): the sequence ratio almost never reaches it, so nothing is clipped.
- Forgetting that CISPO's weight is detached: computing the gradient through the weight turns it back into a ratio objective.
- Treating TIS/MIS as optional when using vLLM with bf16: mismatch effects grow with response length.
- Reporting gains from a single seed or model family (Chapter 26).

**Interview check**

**Q1. What two biases did Dr. GRPO identify?** Response-length normalisation (long wrong answers penalised less per token) and std normalisation (difficulty bias towards near-solved and near-failed prompts).
**Q2. List DAPO's four techniques.** Clip-higher ($\epsilon_{\text{high}} = 0.28$), dynamic sampling, token-level loss, overlong reward shaping/filtering.
**Q3. Why does clip-higher help entropy?** The upper clip mostly limits the growth of low-probability tokens; raising it lets rare exploratory tokens grow, slowing entropy collapse.
**Q4. Show that RLOO's advantage is a scaled mean-centred advantage.** $r_i - \frac{1}{k-1}\sum_{j\neq i} r_j = \frac{k}{k-1}(r_i - \bar{r})$.
**Q5. What is GSPO's importance ratio and why is its clip range tiny?** The geometric mean of token ratios, $(\pi_\theta(y)/\pi_{\text{old}}(y))^{1/|y|}$; it stays very close to 1, so clip ranges around $3\text{–}4 \times 10^{-4}$ are used.
**Q6. How does CISPO differ from PPO-clip?** It uses $\text{sg}(\text{clip}(\rho))\hat{A}\log\pi$: the weight is bounded but every token keeps a gradient.
**Q7. What is the train/inference mismatch and two fixes?** Rollout-engine and trainer probabilities differ for the same weights; fix with truncated/masked IS, fp32 LM head, fp16, routing replay, consistent sampling masks.
**Q8. What does the 80/20 paper recommend?** Train only on the top ~20% highest-entropy (forking) tokens; TRL `top_entropy_quantile=0.2`.
**Q9. What does DeepSeek-V3.2's off-policy sequence masking do?** Masks negative-

**Notation on this page:** $\epsilon$: clip range for the probability ratio • $r_i$: reward of response $i$; $r_{i,t}$ token ratio • $k$: an index or count (see text) • $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\rho$: importance ratio $\pi_\theta/\pi_{\theta_{\text{old}}}$ for a token • $A, \hat{A}$: advantage (how much better an action is than average); hat = estimate • $\text{sg}(\cdot)$: stop-gradient (treated as a constant in backprop) • $\text{clip}(u, a, b)$: limit $u$ to the range $[a, b]$

advantage sequences whose KL between sampling and current policy exceeds a threshold. **Q10. What did ScaleRL find?** RL performance follows sigmoidal curves in compute; many choices change efficiency, some change the asymptote; its recipe uses CISPO and fp32 logits.

**Key takeaways**

- Five axes: normaliser, advantage, weight/clip, mask, sample selection; plus off-policy corrections.
- Dr. GRPO removes length and std biases; DAPO adds clip-higher, dynamic sampling, token-level loss and overlong shaping.
- RLOO = unbiased leave-one-out baseline; REINFORCE++ = global normalisation; ReMax = greedy baseline.
- GSPO = sequence-level geometric-mean ratio (MoE stable); CISPO = clipped detached weight, all tokens learn; SAPO = soft gate.
- Train/inference mismatch: correct with TIS/MIS and fix at the source (fp32 head, fp16, routing replay, sampling masks).

CHAPTER 29

# Process vs Outcome Rewards and Verifiers

**Why this chapter matters.**
An outcome reward says only whether the final answer is right. A long chain of thought can contain correct steps and a single slip, or a lucky guess after nonsense. **Process reward models** (PRMs) score each step. Verifiers of both kinds are used at training time (as rewards) and at test time (to pick or guide answers). Interviewers ask about the trade-offs, how step labels are made, and why DeepSeek-R1 chose *not* to use a PRM.

## 29.1 Outcome vs process supervision

**Outcome reward model (ORM).** Reads the full solution and predicts the probability that the final answer is correct. Trained on (problem, solution, correct?) labels, which are cheap when reference answers exist.

**Process reward model (PRM).** Reads a solution split into steps and predicts, for each step $k$, whether it is correct (or whether the solution is still on track). The solution score is usually the **product** or **minimum** of step scores:

$$\text{score}(y) = \prod_{k=1}^{K} p(\text{step } k \text{ correct} \mid x, s_{1:k}) \quad \text{or} \quad \min_k p(\text{step } k \text{ correct} \mid x, s_{1:k}),$$

where $s_{1:k}$ are the first $k$ steps and $K$ is the number of steps.

| | ORM | PRM |
|---|---|---|
| Labels | one per solution | one per step |
| Label cost | low (answer check) | high (human or many rollouts) |
| Signal | sparse, final | dense, localised |
| Catches | wrong answers | wrong reasoning, including in right answers |
| Hackability | low with exact checks; higher for learned ORMs | higher (learned, step boundaries fuzzy) |

## 29.2 The evidence

*Training Verifiers to Solve Math Word Problems* (Cobbe et al., arXiv 2110.14168) introduced GSM8K (8.5k grade-school maths problems) and **verifiers**: generate many candidate solutions, score

**Notation on this page:** $k$: step index • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $s_k$: step $k$ of a solution; $s_{1:k}$ the first $k$ steps • $p$: a probability distribution (often the target or data) • $K$: number of steps

them with a trained verifier, and pick the best. Verification beat fine-tuning alone and scaled better with more data.

*Solving math word problems with process- and outcome-based feedback* (Uesato et al., arXiv 2211.14275) compared both on GSM8K. Pure outcome supervision gave **similar final-answer error rates** with less labelling, but **reasoning errors** (wrong steps in solutions with correct answers) needed process supervision or a learned reward model that imitates it; their best system cut reasoning errors among correct-answer solutions from 14.0% to 3.4%.

*Let's Verify Step by Step* (Lightman et al., arXiv 2305.20050) trained on harder problems (MATH) with human step labels: each step marked positive, negative or neutral. They found process supervision **significantly outperforms** outcome supervision for best-of-$N$ selection; their PRM-selected answers solved 78% of a representative subset of the MATH test set. They released **PRM800K**, the 800,000 step-level labels, and showed that **active learning** (labelling the most convincing wrong-answer solutions) made labelling much more efficient.

## 29.3 Making step labels without humans: Monte Carlo rollouts

Human step labels are expensive. *Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations* (Wang et al., arXiv 2312.08935) labels steps automatically: from each partial solution $s_{1:k}$, sample $N$ completions with a "completer" model and check their final answers.

- **Hard estimate**: step $k$ is good if **any** completion reaches the correct answer.
- **Soft estimate**: step $k$'s label is the **fraction** of completions that are correct, an estimate of $V(s_{1:k})$, the value of the partial solution.

This is a Monte Carlo value estimate (Chapter 14): a step is good if it leaves you in a state from which the problem can still be solved. Math-Shepherd used the PRM both for **re-ranking** and for **step-by-step PPO** (rewards at each step), improving Mistral-7B from 77.9% to 84.1% on GSM8K and from 28.6% to 33.0% on MATH, and further to 89.1% and 43.5% with verification at test time.

```
import random
random.seed(0)

# Toy: a "solution" is a list of steps; each step is correct with prob 0.9, and one
 ↪  wrong step dooms the answer.
def complete(prefix_ok, remaining, p=0.9):
    """Simulate finishing a partial solution: succeeds only if prefix is ok and every
     ↪  remaining step is ok."""
    return prefix_ok and all(random.random() < p for _ in range(remaining))

def mc_step_labels(steps_ok, N=32, p=0.9):
    """Soft Math-Shepherd labels: fraction of N rollouts from each prefix that reach the
     ↪  right answer."""
    labels, prefix_ok = [], True
    for k, ok in enumerate(steps_ok):
        prefix_ok = prefix_ok and ok
        remaining = len(steps_ok) - k - 1
```

**Notation on this page:** $N$: completions per step or candidates per problem • $s_k$: step $k$ of a solution; $s_{1:k}$ the first $k$ steps • $k$: step index • $V(s)$: state value (expected future return from $s$) • $V$: vocabulary size

```
        labels.append(sum(complete(prefix_ok, remaining, p) for _ in range(N)) / N)
    return labels

print("all steps right :", mc_step_labels([True, True, True, True, True]))
print("error at step 3 :", mc_step_labels([True, True, False, True, True]))
```

The labels rise towards 1 as a correct solution nears the end (fewer chances left to slip), and drop to 0 at and after the wrong step: the PRM learns to localise the error.

**Why this, not that? Why Monte Carlo step labels (Math-Shepherd), not human step labels (PRM800K)?**

**Why.** They need no human annotators, scale to millions of steps, and have a clean meaning (the success probability from this point), which is exactly a value function.
**Trade-off.** Labels depend on the completer model: a weak completer marks good steps as bad; a step can be "recoverable" yet wrong; and generating $N$ rollouts per step is expensive. Noisy labels make the PRM noisy.
**When the alternative wins.** Human labels win for subtle errors that later steps can hide (a wrong step that still leads to the right number), for evaluation sets, and for domains without checkable final answers.

## 29.4 Generative verifiers and "thinking" PRMs

A scalar PRM has to judge a step in one forward pass. **Generative** verifiers instead write a critique first:

- *Generative Verifiers: Reward Modeling as Next-Token Prediction* (Zhang et al., arXiv 2408.15240): train the verifier to produce a verification chain of thought, then "Yes"/"No"; score = $P$(Yes); sample several critiques and average (majority voting over verifications).
- *Process Reward Models That Think* (Khalifa et al., arXiv 2504.16828; ThinkPRM): a long-chain-of-thought verifier that checks each step in text, fine-tuned on about 1% of PRM800K's labels, outperforming discriminative PRMs trained on the full set and LLM-as-a-judge baselines on ProcessBench, MATH-500 and AIME'24, and transferring better out of domain.

The trend mirrors policies: verifiers that **reason** before judging scale better with test-time compute.

## 29.5 Using verifiers: training time vs test time

**Test time.**

- **Best-of-$N$**: sample $N$ solutions, return the highest-scoring one.
- **Weighted majority voting**: group solutions by final answer and sum their verifier scores; choose the answer with the largest total. Usually better than plain majority voting or plain best-of-$N$.
- **Step-level search**: beam search or tree search where a PRM scores partial solutions and prunes bad branches early (Chapter 30). *Scaling LLM Test-Time Compute Optimally can be*

**Notation on this page:** $N$: completions per step or candidates per problem

*More Effective than Scaling Model Parameters* (Snell et al., arXiv 2408.03314) found that the best way to spend test-time compute (search against a PRM vs revising answers sequentially) depends on prompt difficulty.

**Training time.**

- **Outcome rewards** for RL (RLVR): the dominant approach for maths and code today.
- **Process rewards** for RL: add step-level rewards (Math-Shepherd's step PPO; DeepSeek-Math also tested process supervision with GRPO, giving each token the normalised reward sum of the steps after it).
- **Data filtering**: use verifiers to filter SFT data (rejection sampling).

**Why DeepSeek-R1 did not use a PRM.** The R1 paper (arXiv 2501.12948) lists three problems: it is hard to define a "step" in general reasoning; it is hard to judge whether an intermediate step is correct (model-based annotation was unsatisfactory, human annotation does not scale); and a model-based PRM inevitably invites **reward hacking**, while retraining it complicates the pipeline. PRMs remained useful for re-ranking and guided search, but in their large-scale RL the benefits did not justify the overhead.

**Why this, not that? Why outcome rewards for large-scale RL, not process rewards?**

**Why.** A rule-based outcome check is nearly unhackable and needs no extra model; it lets the policy discover its own reasoning style instead of matching a PRM's idea of a "good step"; and it scales with no labelling.
**Trade-off.** Sparse credit assignment: a long solution with one slip gets the same zero as nonsense; correct answers reached by flawed reasoning get full reward.
**When the alternative wins.** PRMs help at test time (search, re-ranking), when outcome checks are unavailable, for very long multi-step tasks where sparse rewards stall learning, and to discourage lucky-guess reasoning (education, high-stakes proofs).

## 29.6 Training a PRM

A discriminative PRM is usually a language model with a classification head read at the **last token of each step** (often a special step separator or a newline). With step labels $c_k \in \{0, 1\}$ (hard) or $c_k \in [0, 1]$ (soft Monte Carlo estimates), the loss is binary cross-entropy summed over steps:

$$\mathcal{L}_{\text{PRM}} = -\sum_{k=1}^{K} \Big[ c_k \log p_\phi(s_{1:k}) + (1 - c_k) \log \left(1 - p_\phi(s_{1:k})\right) \Big],$$

where $p_\phi(s_{1:k})$ is the predicted probability that step $k$ is correct (or that the solution is still on track after it). PRM800K labels are three-way (positive, negative, neutral), so a training setup must decide how to map neutral steps to targets. A common implementation trick is to train the LM head to predict special "+"/"−" tokens at the step separators, reusing the language-model head instead of adding a new one.

**Which label definition?** "Step is correct" (human, PRM800K) and "the solution can still succeed from here" (Monte Carlo, Math-Shepherd) are different targets. *The Lessons of Developing Process*

**Notation on this page:** $k$: step index • $c_k$: step label (hard 0/1 or soft MC estimate) • $\phi$: PRM weights; $p_\phi$ predicted step correctness • $\mathcal{L}$: a loss to minimise • $s_k$: step $k$ of a solution; $s_{1:k}$ the first $k$ steps • $p$: a probability distribution (often the target or data) • $K$: number of steps

*Reward Models in Mathematical Reasoning* (Zhang et al., arXiv 2501.07301) found that Monte Carlo labelling usually gave **worse** PRMs than LLM-as-a-judge or human labels, because completions can recover from a wrong step (so the step looks fine) or fail after a correct one. They used **consensus filtering**: keep a step label only when Monte Carlo estimation and an LLM judge agree. They also warned that best-of-$N$ evaluation of PRMs is biased: policies produce correct answers with flawed reasoning, PRMs that tolerate these score well on best-of-$N$, and many PRMs put their minimum score on the final answer step, behaving more like ORMs.

**Evaluating step-level judgement directly.** *ProcessBench: Identifying Process Errors in Mathematical Reasoning* (Zheng et al., arXiv 2412.06559) has 3,400 competition-level solutions with the **earliest wrong step** marked by experts; a model must find that step or say all steps are correct. Existing PRMs generalised poorly beyond GSM8K/MATH-level problems and were beaten by prompted general models acting as critics.

## 29.7 Selecting answers with verifiers: a simulation

The simulation below draws 16 candidate solutions of 8 steps each. Each step is wrong with probability 0.08, a wrong step ruins the final answer 90% of the time (so 10% of flawed solutions are "lucky"), wrong answers scatter over three values, and both verifiers are noisy. It compares selection rules on final-answer accuracy, and for best-of-$N$ also on whether the chosen solution has **fully correct reasoning**.

```
import torch
from collections import defaultdict
torch.manual_seed(0)

K, N, trials = 8, 16, 3000         # steps per solution, candidates per problem, problems

def make_candidates():
    """Each step is wrong with prob 0.08. Any wrong step usually ruins the answer
     ↪  (90%)."""
    step_ok = torch.rand(N, K) > 0.08
    all_ok = step_ok.all(1)
    lucky = (~all_ok) & (torch.rand(N) < 0.1)          # flawed reasoning but right answer
    correct = all_ok | lucky
    answers = torch.where(correct, torch.zeros(N, dtype=torch.long), torch.randint(1, 4,
 ↪  (N,)))
    # Noisy verifiers: a PRM scores each step, an ORM scores the whole solution.
    step_p = torch.sigmoid(torch.where(step_ok, 2.0, -1.0) + 1.2 * torch.randn(N, K))
    orm_p = torch.sigmoid(torch.where(correct, 1.0, -1.0) + 1.5 * torch.randn(N))
    return answers, correct, all_ok, step_p, orm_p

def weighted_vote(answers, w):
    tot = defaultdict(float)
    for a, s in zip(answers.tolist(), w.tolist()): tot[a] += s
    return max(tot, key=tot.get)
```

```
methods = {"pass@1 (random)": 0, "majority vote": 0, "ORM best-of-N": 0, "PRM-min
↪ best-of-N": 0,
          "PRM-product best-of-N": 0, "PRM-min weighted vote": 0}
sound = {"ORM best-of-N": 0, "PRM-min best-of-N": 0}
for _ in range(trials):
    ans, cor, ok, sp, op = make_candidates()
    methods["pass@1 (random)"] += cor[0].item()
    methods["majority vote"] += weighted_vote(ans, torch.ones(N)) == 0
    i = op.argmax(); methods["ORM best-of-N"] += cor[i].item(); sound["ORM best-of-N"]
↪ += ok[i].item()
    j = sp.min(1).values.argmax(); methods["PRM-min best-of-N"] += cor[j].item();
↪ sound["PRM-min best-of-N"] += ok[j].item()
    methods["PRM-product best-of-N"] += cor[sp.prod(1).argmax()].item()
    methods["PRM-min weighted vote"] += weighted_vote(ans, sp.min(1).values) == 0
for m, v in methods.items():
    extra = f"  (fully correct reasoning: {sound[m] / trials:.2f})" if m in sound else ""
    print(f"{m:>22}: accuracy {v / trials:.2f}{extra}")
```

Three lessons, all of which match the literature: majority voting is a strong baseline when wrong answers scatter; combining a verifier with voting (weighted vote) is best; and a PRM picks solutions with **sound reasoning** much more often than an ORM, even when final-answer accuracy is similar (the Uesato et al. finding). Change the error model (for example, make wrong answers agree on one value) and voting degrades quickly while verifiers do not.

**Why this, not that? Why aggregate step scores with the minimum, not the product?**

**Why.** A solution is only as good as its worst step. The minimum is insensitive to the number of steps, while the product shrinks with every step, so it unfairly prefers short solutions and depends on calibration of every step score.
**Trade-off.** The minimum ignores everything except one step, so it is sensitive to a single noisy low score and cannot tell "one doubtful step" from "many doubtful steps".
**When the alternative wins.** The product is the right choice when step scores are well-calibrated probabilities of independent errors (it is then the probability that all steps are correct), and when solution lengths are similar. Some systems use the last step's score or a learned aggregator.

## 29.8 Common mistakes

- Treating a PRM trained on one model's solutions as reliable on a different model's style (distribution shift).
- Using step boundaries (newlines) that do not match the policy's actual formatting.
- Taking the product of many step probabilities without calibration: long solutions are unfairly penalised.
- Letting the policy optimise a learned PRM for long without monitoring: classic reward hacking (e.g. many short "safe" steps).

**Interview check**

**Q1. ORM vs PRM?** ORM scores the final outcome of a solution; PRM scores each step. PRMs give denser, localised signal but cost more labels and are easier to hack.
**Q2. What did Let's Verify Step by Step show?** Process supervision beat outcome supervision for best-of-$N$ on MATH (78% of a test subset solved); released PRM800K; active learning helps.
**Q3. How does Math-Shepherd label steps automatically?** Roll out $N$ completions from each step; hard label = any reaches the right answer, soft label = fraction that do.
**Q4. How do you combine step scores into a solution score?** Product or minimum of step correctness probabilities (min is common and robust).
**Q5. What is weighted majority voting?** Sum verifier scores per distinct final answer and choose the answer with the highest total.
**Q6. Why did DeepSeek-R1 avoid PRMs?** Hard to define steps, hard to label step correctness, reward hacking and pipeline complexity.
**Q7. What is a generative verifier?** A verifier that writes a critique (chain of thought) before outputting a judgement; its probability of "Yes" is the score.
**Q8. What did Uesato et al. find?** Outcome supervision gives similar final-answer error rates with fewer labels, but reasoning errors need process-style supervision.
**Q9. Write the PRM training loss.** Binary cross-entropy at each step's last token: $-\sum_k [c_k \log p_\phi(s_{1:k}) + (1 - c_k) \log(1 - p_\phi(s_{1:k}))]$ with hard or soft labels $c_k$.
**Q10. Why can Monte Carlo step labels be wrong?** Completions can recover after a wrong step (label too high) or fail after a correct one (label too low); quality depends on the completer. Consensus with an LLM judge helps.
**Q11. What does ProcessBench measure?** Whether a model can find the earliest erroneous step in a solution (3,400 expert-labelled cases); many PRMs generalise poorly to harder problems.
**Q12. Why is best-of-N a biased way to evaluate PRMs?** Correct answers with flawed reasoning count as successes, so PRMs that tolerate bad steps (behaving like ORMs) look good.
**Q13. When does majority voting beat verifier-based selection?** When wrong answers scatter and the verifier is noisy; it fails when errors are systematic (a common wrong answer).

**Key takeaways**

- ORMs judge outcomes; PRMs judge steps; solution score = product or min of steps.
- Step labels: humans (PRM800K) or Monte Carlo rollouts (Math-Shepherd, a value estimate).
- Generative/thinking verifiers reason before judging and scale with compute.
- Verifiers power best-of-$N$, weighted voting and search at test time; outcome rewards dominate large-scale RL training.
- PRM loss = per-step BCE at step boundaries; labels from humans, judges or MC (consensus filtering helps).
- Evaluate PRMs on step-error detection (ProcessBench), not only best-of-$N$.
- Weighted voting usually beats either voting or best-of-$N$ alone; PRMs select sounder

**Notation on this page:** $N$: completions per step or candidates per problem • $\phi$: PRM weights; $p_\phi$ predicted step correctness • $s_k$: step $k$ of a solution; $s_{1:k}$ the first $k$ steps • $p$: a probability distribution (often the target or data) • $k$: step index • $c_k$: step label (hard 0/1 or soft MC estimate)

reasoning.

CHAPTER 30

# Test-Time Compute and the DeepSeek-R1 Recipe

**Why this chapter matters.**
"Reasoning models" (OpenAI o1 and successors, DeepSeek-R1, Qwen3, Kimi and others) trade more computation **at inference** for better answers: they think longer, sample more, or search. DeepSeek-R1 showed in the open how RL produces this behaviour. Interviewers ask about majority voting, best-of-$N$, tree search, thinking budgets, and they very often ask you to walk through the R1 pipeline step by step with its numbers. This chapter gives you all of it, checked against the papers.

## 30.1 Parallel test-time compute: sample many, pick one

**Best-of-$N$.** Sample $N$ responses, score them with a verifier or reward model, return the best. Quality grows slowly with $N$ (Chapter 18) and depends entirely on the scorer; with a learned scorer it is subject to reward hacking at large $N$ (Chapter 21).

**Majority voting (self-consistency).** *Self-Consistency Improves Chain of Thought Reasoning in Language Models* (Wang et al., arXiv 2203.11171): sample several reasoning paths at non-zero temperature, extract each final answer, and return the most common one. No verifier is needed. It improved chain-of-thought prompting by large margins (for example +17.9% on GSM8K). It works because errors tend to scatter across many different wrong answers, while correct reasoning paths converge on the same answer.

**Weighted voting.** Sum verifier scores per distinct answer and return the answer with the highest total (Chapter 29).

```
import random
from collections import Counter
random.seed(0)

def sample_answer(p_correct=0.4, n_wrong=6):
    """A model that is right 40% of the time; wrong answers scatter over 6 values."""
    return "42" if random.random() < p_correct else str(random.randint(0, n_wrong - 1))

def majority(k):
    votes = Counter(sample_answer() for _ in range(k))
    return votes.most_common(1)[0][0]

for k in (1, 5, 15, 45):
```

**Notation on this page:** $N(s), N(s,a)$: visit counts in MCTS

```
    acc = sum(majority(k) == "42" for _ in range(2000)) / 2000
    print(f"maj@{k:2d}: accuracy {acc:.2f}")
```

A model that is right only 40% of the time becomes nearly always right with enough votes, because its wrong answers do not agree with each other. If the wrong answers *concentrated* on one value (a common misconception), voting would not help, and it would make things worse once the misconception had more mass than the truth.

**Why this, not that? Why majority voting, not best-of-N with a reward model?**

**Why.** Voting needs no extra model, cannot be reward-hacked, and works well whenever answers can be compared exactly (numbers, multiple choice, short strings).
**Trade-off.** It needs extractable, comparable answers (not essays or code with many valid forms) and fails when the model's errors are systematic. It cannot pick a rare correct answer out of a crowd of identical wrong ones.
**When the alternative wins.** Best-of-$N$ or weighted voting with a good verifier wins for open-ended outputs, for code (run the tests), and when the correct answer is a minority among samples but recognisable by a verifier.

## 30.2 Search: tree search and step-level beam search

**Beam search over steps.** Generate the solution step by step. At each step, expand each partial solution with several candidate next steps, score them with a PRM, and keep the top $B$. This prunes bad paths early.

**Monte Carlo tree search (MCTS).** Treat partial solutions as tree nodes. Repeat: **select** a path down the tree using a rule that balances value and exploration, **expand** a leaf with new steps, **evaluate** it (by rollouts or a value model), and **back up** the result along the path. AlphaZero-style selection uses the PUCT rule:

$$a^* = \arg\max_a \left[ Q(s,a) + c_{\text{puct}}\, \pi(a \mid s)\, \frac{\sqrt{N(s)}}{1 + N(s,a)} \right],$$

where $Q(s,a)$ is the average value of child $a$, $\pi(a \mid s)$ is the policy's prior, $N$ counts visits, and $c_{\text{puct}}$ sets how much to explore.

*rStar-Math: Small LLMs Can Master Math Reasoning with Self-Evolved Deep Thinking* (Guan et al., arXiv 2501.04519) used MCTS with a small policy model and a **process preference model**, plus code-verified steps, over four rounds of self-evolution on 747k problems, raising Qwen2.5-Math-7B from 58.8% to 90.0% on MATH.

**Why search is less popular for LLM reasoning at scale.** DeepSeek-R1's authors tried MCTS and reported two problems: the token search space is exponentially larger than chess (limiting node expansion leads to local optima), and training a fine-grained value model to guide search is very hard, so iterative self-improvement stalled (arXiv 2501.12948). Long chain-of-thought trained with RL does a kind of **search inside one sample**: the model proposes, checks, backtracks ("Wait, ...") and tries again, without an external tree.

**Notation on this page:** $N(s), N(s,a)$: visit counts in MCTS • $B$: batch size • $\pi_\theta$: the policy (the model), with weights $\theta$ • $Q(s,a)$: average value of child $a$ (MCTS) • $s$: a partial solution (tree node) • $a$: a child node (next step) in the search tree • $c_{\text{puct}}$: exploration constant

## 30.3 Sequential test-time compute: thinking longer

A reasoning model writes a long hidden or visible **chain of thought** (often inside `<think>...</think>`) before the answer. Accuracy tends to rise with the length of thinking, up to a point.

**Thinking budgets.** Products let the user or system choose the effort level (low/medium/high, or a token budget). Hybrid models (for example the Qwen3 family) support "thinking" and "non-thinking" modes in one model.

**Budget forcing.** *s1: Simple test-time scaling* (Muennighoff et al., arXiv 2501.19393) fine-tuned Qwen2.5-32B-Instruct on just 1,000 curated reasoning traces (s1K) and controlled thinking at inference by (a) forcing the end of thinking when a budget is reached, or (b) suppressing the end-of-thinking token and appending "Wait" to make the model keep going. This often made the model re-check and fix its answer, raising AIME24 from 50% to 57%.

**Compute-optimal allocation.** Snell et al. (arXiv 2408.03314) found the best way to use extra inference compute depends on difficulty: for easy problems, sequential revision of one answer works well; for harder ones, parallel sampling and search against a verifier help more. Allocating compute per prompt adaptively beat a fixed best-of-$N$ baseline by a large factor in efficiency.

**Why this, not that? Why train long chain-of-thought with RL, not use external search at inference?**

**Why.** A single long sample is simple to serve (no orchestration, no value model), the model learns *when* to backtrack and verify, and the behaviour transfers across domains. RL with outcome rewards produced this directly in R1-Zero.
**Trade-off.** Long outputs are slow and expensive per query, models over-think easy problems, and you cannot easily parallelise one chain of thought.
**When the alternative wins.** External search or parallel sampling wins when you have a strong verifier (code tests, formal proofs), when latency allows parallel hardware use, or when you need guarantees (search until a proof checks).

## 30.4 DeepSeek-R1-Zero: RL straight from the base model

*DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning* (DeepSeek-AI, arXiv 2501.12948; the revised version is the reference here). **R1-Zero** applies RL directly to **DeepSeek-V3-Base**, with **no SFT first**.

**Template.** A simple instruction: the assistant first thinks, then answers, with reasoning in `<think>...</think>` and the answer in `<answer>...</answer>`; nothing about *how* to reason.

**Rewards (rule-based).** An **accuracy reward** (answer check for maths, compiler and test cases for code) and a **format reward** (thinking inside the tags), weighted equally. No neural reward model, because they found neural RMs prone to reward hacking at scale.

**Algorithm and settings.** GRPO with:

| Setting | Value |
|---|---|
| Learning rate | $3 \times 10^{-6}$ |

**Notation on this page:** $N(s), N(s,a)$: visit counts in MCTS

| Setting | Value |
|---|---|
| KL coefficient $\beta$ | 0.001 |
| Rollout temperature | 1.0 |
| Samples per question $G$ | 16 |
| Max length | 32,768 tokens, raised to 65,536 after step 8.2k |
| Training steps | 10,400 (about 1.6 epochs) |
| Questions per step | 32 (batch of 512 responses) |
| Reference model | replaced by the current policy every 400 steps |
| Rollout / update | 8,192 outputs per rollout, split into 16 mini-batches, 1 inner epoch |

**Results and behaviour.** Average pass@1 on AIME 2024 rose from **15.6% to 77.9%**, and to 86.7% with self-consistency voting. **Response length grew steadily** during training without any length reward: the model learned to spend more tokens thinking. It developed **reflection** and **verification**, including the famous "**aha moment**", where an intermediate checkpoint wrote "Wait, wait. Wait. That's an aha moment I can flag here." and re-evaluated its solution. Jumps in performance and length appeared when the length limit was raised at step 8.2k.

**Problems.** Poor readability and **language mixing** (switching between English and Chinese mid-reasoning), because the base model is multilingual and nothing rewarded consistency.

## 30.5 DeepSeek-R1: the four-stage pipeline

R1 fixes readability while keeping the reasoning:

1. **Cold start.** Collect **thousands** of long chain-of-thought examples in a conversational, human-friendly style and SFT the base model on them. This gives a readable starting format and faster RL convergence.
2. **Reasoning-focused RL.** GRPO with rule-based rewards as in R1-Zero, plus a **language-consistency reward**: the share of words in the chain of thought that are in the target language,
$$R_{\text{language}} = \frac{\text{Num}(\text{Words}_{\text{target}})}{\text{Num}(\text{Words})}.$$
Their ablation showed it slightly lowers benchmark performance but improves readability, which users prefer. The paper also notes that **the clip ratio matters**: too low truncates gradients for many tokens and hurts performance; too high destabilises training.
3. **Rejection sampling and SFT.** From the RL checkpoint, sample many responses per prompt and keep only correct ones (filtering out mixed languages, long paragraphs and code blocks in the chain of thought): about **600k reasoning samples**. Add about **200k non-reasoning samples** (writing, factual QA, self-cognition, translation) reusing parts of DeepSeek-V3's SFT data, with a short generated chain of thought for some queries and none for trivial ones like "hello". SFT the model on this ~800k set.

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $G$: samples per question (16 for R1-Zero) • $R_{\text{language}}$: language-consistency reward

4. **Second RL stage (all scenarios).** RL for both reasoning (rule-based rewards) and general helpfulness and harmlessness (reward models trained on preference data), over diverse prompts.

**Result.** DeepSeek-R1 scored 79.8% pass@1 on AIME 2024 and 97.3% on MATH-500 in the paper's comparison table, on par with OpenAI-o1-1217 at release.

## 30.6 Distilling R1 into small models

Using the same ~800k samples, DeepSeek fine-tuned (SFT only, no RL) several open base models for 2 to 3 epochs, with a cosine schedule decaying to one-tenth of the initial learning rate, context 32,768 and batch size 64. Initial learning rates: Qwen-1.5B (from Qwen2.5-Math-1.5B) $1 \times 10^{-4}$; Qwen-7B (Qwen2.5-Math-7B) $8 \times 10^{-5}$; Qwen-14B $7 \times 10^{-5}$; Qwen-32B $6 \times 10^{-5}$; Llama-8B (Llama-3.1-8B) $5 \times 10^{-5}$; Llama-70B (Llama-3.3-70B-Instruct) $2 \times 10^{-5}$. Note how the learning rate falls as the model grows.

The key finding: **distillation beat RL for small models.** Running large-scale RL directly on Qwen2.5-32B-Base for over 10k steps (learning rate $2 \times 10^{-6}$, KL 0.001, 16 samples) gave Qwen2.5-32B-Zero at 47.0% AIME 2024 pass@1, on par with QwQ-32B-Preview (50.0%), while the distilled R1-Distill-Qwen-32B reached 72.6% and was significantly better on every benchmark in the comparison. Reasoning patterns discovered by a large model transfer to small ones more cheaply than small models can discover them by RL.

**Why this, not that? Why a cold-start SFT before RL (R1), not pure RL from the base model (R1-Zero)?**

**Why.** A few thousand readable examples fix the output format and language from the start, speed up RL convergence, and make the final model usable; pure RL produced language mixing and hard-to-read reasoning.
**Trade-off.** The cold-start data shapes the reasoning style, possibly limiting what RL discovers, and it needs curation (human or model-generated with filtering).
**When the alternative wins.** Pure "zero" RL is the cleaner scientific experiment for studying what RL alone can elicit, and it is the cheapest start when you have no curated reasoning data; many open reproductions began there.

## 30.7 Open reproductions and what they taught us

- **Open-Reasoner-Zero** (Hu et al., arXiv 2503.24290): vanilla **PPO with GAE** ($\lambda = 1, \gamma = 1$), rule-based rewards, **no KL**, on Qwen2.5-32B base; beat DeepSeek-R1-Zero-Qwen-32B on AIME 2024, MATH-500 and GPQA Diamond with one-tenth of the training steps. The learned critic helped by devaluing repetitive patterns.
- **DAPO** (arXiv 2503.14476): GRPO plus four fixes, 50 on AIME 2024 with Qwen2.5-32B in half of R1-Zero-Qwen-32B's steps (Chapter 28).
- **Dr. GRPO / Oat-Zero** (arXiv 2503.20783): removing GRPO's biases; 43.3% AIME 2024 with a 7B base; also found that DeepSeek-V3-Base already shows "aha moments" and that Qwen2.5 bases reason well even without a prompt template, suggesting pretraining effects.

**Notation on this page:** $\lambda$: GAE parameter (0 to 1), trades bias for variance • $\gamma$: discount factor (0 to 1)

- **SimpleRL-Zoo** (Zeng et al., arXiv 2503.18892): zero-RL on 10 base models (Llama 3, Mistral, DeepSeek-Math, Qwen2.5 from 0.5B to 32B). Format rewards and query difficulty needed adjusting per model; response-length growth did not always come with reflective behaviour; Qwen2.5 bases already show strong instruction-following and self-reflection, so they are not representative.
- **Kimi k1.5** (arXiv 2501.12599): long-context RL (scaling the context window was key) with a simple framework **without MCTS, value functions or PRMs**, plus "long2short" methods to transfer long-CoT gains into shorter answers.
- **Skywork-OR1** (arXiv 2505.22312): RL on top of R1-Distill models, with a detailed study showing that avoiding **premature entropy collapse** is critical.
- **VAPO** (arXiv 2504.05118): value-based PPO with fixes reached 60.4 on AIME 2024 with Qwen 32B (Chapter 16).

**Common lessons.** Rule-based rewards are enough; the KL can be tiny or zero; length grows naturally; base model choice dominates (Qwen2.5 is unusually "RL-ready"); entropy and clipping need care; distillation is the efficient path for small models.

## 30.8 Common mistakes

- Claiming R1 used a PRM or MCTS: it explicitly did not (both are listed as unsuccessful attempts).
- Saying R1-Zero had "no SFT at all" and R1 had "no SFT": R1 used a cold start and a large rejection-sampling SFT stage.
- Saying the distilled models were trained with RL: they were SFT only.
- Treating response-length growth as proof of better reasoning (Dr. GRPO's length bias, SimpleRL-Zoo's observations).

**Interview check**

**Q1. What is self-consistency and why does it work?** Sample many reasoning paths and take the majority answer; errors scatter across different wrong answers while correct paths agree.

**Q2. Walk through DeepSeek-R1's pipeline.** Cold-start SFT on thousands of readable long CoTs; reasoning RL (GRPO, rule rewards, language-consistency reward); rejection sampling (~600k reasoning + ~200k non-reasoning) and SFT; second RL stage with rule rewards and reward models for general helpfulness/harmlessness.

**Q3. What were R1-Zero's main hyperparameters?** GRPO, LR $3 \times 10^{-6}$, KL 0.001, temperature 1, 16 samples, max length 32,768 then 65,536 after step 8.2k, 10,400 steps, 32 questions per step, reference refreshed every 400 steps.

**Q4. What rewards did R1-Zero use and why not a neural RM?** Rule-based accuracy and format rewards; neural RMs were prone to reward hacking at scale.

**Q5. What was the "aha moment"?** An intermediate R1-Zero checkpoint spontaneously stopped, flagged its mistake ("Wait, wait…") and re-evaluated its solution: emergent reflection from outcome-only RL.

**Q6. Why did R1 add a language-consistency reward?** R1-Zero mixed languages in its

reasoning; the reward (share of target-language words) improves readability at a small accuracy cost.

**Q7. Why did DeepSeek's MCTS attempt fail?** Exponential token search space and the difficulty of training a fine-grained value model to guide search.

**Q8. How were the R1-Distill models trained, and what did they show?** SFT only on ~800k R1 samples for 2–3 epochs; distillation beat large-scale RL on the same base (72.6% vs 47.0% AIME 2024 for 32B).

**Q9. What is budget forcing?** s1's inference trick: stop thinking at a budget, or append "Wait" to extend thinking, controlling test-time compute.

**Q10. Name three open R1-style reproductions and one lesson from each.** Open-Reasoner-Zero (plain PPO, $\lambda = \gamma = 1$, no KL suffices); DAPO (clip-higher, dynamic sampling, token loss, overlong shaping); SimpleRL-Zoo (base-model family matters; length growth is not always reflection).

**Key takeaways**

- Parallel test-time compute: best-of-$N$, majority and weighted voting; sequential: longer thinking, budget forcing; search: beam/MCTS with PRMs.
- R1-Zero: GRPO on V3-Base, rule rewards, KL 0.001, $G = 16$, 10,400 steps; AIME 15.6% → 77.9%; emergent reflection and length growth; language mixing.
- R1: cold start → reasoning RL (+ language consistency) → rejection sampling SFT (~800k) → all-scenario RL; 79.8% AIME 2024.
- Distillation from R1 beat RL for small models.
- Open reproductions agree: simple rule rewards, tiny/no KL, careful clipping and entropy, base model matters.

**Notation on this page:** $\gamma$: discount factor (0 to 1) • $\lambda$: GAE parameter (0 to 1), trades bias for variance • $N(s), N(s,a)$: visit counts in MCTS • $G$: samples per question (16 for R1-Zero)

CHAPTER 31

# Practical RL Engineering for Reasoning Models

> **Why this chapter matters.**
> Papers report final numbers. Jobs need you to make a run *work*: choose the data, watch the right curves, notice when something is wrong, and fix it. Interviewers love "your GRPO run's reward is flat / entropy collapsed / lengths exploded — what do you check?" This chapter is the debugging manual: data and difficulty, entropy, length, the metrics to log, evaluation with pass@k, and the long-run tricks used in 2025–2026.

## 31.1 Data: difficulty is everything

In group-based methods (GRPO, RLOO, DAPO), a prompt whose $G$ samples are **all right** or **all wrong** has zero advantage for every sample, so it contributes **no gradient** (Chapter 27). If a model solves a prompt with probability $p$, the chance a group of $G$ is "mixed" is

$$P(\text{mixed}) = 1 - p^G - (1-p)^G.$$

For $G = 8$: $p = 0.5$ gives 0.992, $p = 0.95$ gives 0.337, $p = 0.01$ gives 0.077. Very easy and very hard prompts waste compute.

**Practical recipe.**

1. **Offline difficulty filtering.** Before training, sample $k$ answers per prompt with the starting model and estimate its pass rate. Drop prompts with pass rate 0 or 1 (or keep a band such as 0.1–0.9).
2. **Online filtering (dynamic sampling).** During training, drop groups that are all-correct or all-wrong and resample until the batch is full (DAPO, arXiv 2503.14476).
3. **Curriculum.** As the model improves, prompts become too easy; re-estimate pass rates and refresh the pool, or schedule from easier to harder data.
4. **Clean the answers.** Wrong reference answers teach the model to be wrong; ambiguous answers (multiple valid forms) need a robust checker (`math_verify` in `trl.rewards.accuracy_reward`).
5. **Decontaminate.** Remove training problems that overlap with evaluation sets (AIME, MATH-500), or your numbers mean nothing.

```
import numpy as np
rng = np.random.default_rng(0)

def p_mixed(p, G):
    return 1 - p**G - (1 - p)**G
```

**Notation on this page:** $G$: group size • $p$: probability the model solves a prompt • $k$: attempts in pass@$k$

```
for p in (0.01, 0.1, 0.5, 0.9, 0.99):
    print(f"pass rate {p:4.2f}: P(group gives gradient) = {p_mixed(p, 8):.3f}")

# Offline filter: estimate pass rate with k=16 samples, keep the useful band.
true_p = rng.beta(0.5, 0.5, size=1000)            # many very easy and very hard prompts
est = rng.binomial(16, true_p) / 16
keep = (est > 0) & (est < 1)
print(f"kept {keep.sum()} of 1000 prompts;",
      f"mean P(mixed) before {p_mixed(true_p, 8).mean():.2f}, after
      ↪ {p_mixed(true_p[keep], 8).mean():.2f}")
```

**Why this, not that? Why filter by difficulty offline, not rely only on dynamic sampling?**

**Why.** Offline filtering is cheap (done once) and removes prompts that will never help, so online generation is not wasted on them every epoch.
**Trade-off.** Pass rates drift as the model learns, so offline estimates go stale; filtering out "impossible" prompts can also remove the hardest problems the model might eventually learn.
**When the alternative wins.** Dynamic sampling adapts to the current policy automatically and is better for long runs; in practice teams use both, plus periodic re-estimation (curriculum).

## 31.2 Entropy: the exploration budget

**Policy entropy** at a token is $H = -\sum_v \pi(v \mid \text{context}) \log \pi(v \mid \text{context})$; we log its average over completion tokens. In RLVR without intervention, entropy usually **drops sharply early** and performance then **saturates**.

*The Entropy Mechanism of Reinforcement Learning for Reasoning Language Models* (Cui et al., arXiv 2505.22617) fitted an empirical law between entropy $H$ and downstream performance $R$:

$$R = -a\, e^{H} + b,$$

so performance is "bought" with entropy, and when entropy reaches 0 the ceiling is $R = -a + b$. They derived that the entropy change is driven by the **covariance** between a token's log-probability and its advantage (for softmax policies under policy-gradient updates): confident tokens with positive advantage get more confident, which lowers entropy. Their fixes, **Clip-Cov** and **KL-Cov**, restrict updates on the small set of tokens with the highest covariance.

*Beyond the 80/20 Rule: High-Entropy Minority Tokens Drive Effective Reinforcement Learning for LLM Reasoning* (Wang et al., arXiv 2506.01939) found that only a small fraction of tokens have high entropy, acting as **forks** in the reasoning; training only on the top 20% highest-entropy tokens matched full updates on Qwen3-8B and beat them on Qwen3-32B (+11.04 on AIME'25). TRL exposes this as `GRPOConfig(top_entropy_quantile=...)`.

**Tools to manage entropy.**

- **Clip-higher** (DAPO): a larger $\varepsilon_{\text{high}}$ lets low-probability tokens rise, slowing collapse.

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $R$: downstream performance • $a, b$: fitted constants in $R = -ae^H + b$ • $b$: fitted constant • $\epsilon$: clip range for the probability ratio

- **Entropy bonus**: $-\alpha H$ in the loss; simple but can blow up entropy if $\alpha$ is too large; adaptive versions tune $\alpha$ to a target.
- **Higher sampling temperature** during rollouts (with care: it changes the behaviour policy).
- **Covariance-based clipping** (Clip-Cov/KL-Cov) or **forking-token-only** updates.
- **Data diversity**: narrow data collapses entropy faster.

```
import torch
torch.manual_seed(0)

# Why entropy falls: positive-advantage updates on already-likely tokens sharpen the
↪ distribution.
logits = torch.tensor([2.0, 1.0, 0.5, 0.0], requires_grad=True)
def entropy(l):
    p = torch.softmax(l, -1); return -(p * p.log()).sum()

opt = torch.optim.SGD([logits], lr=0.5)
for step in range(6):
    p = torch.softmax(logits, -1)
    a = torch.multinomial(p, 1).item()
    adv = 1.0 if a == 0 else -0.2          # the top token is usually right
    loss = -adv * torch.log_softmax(logits, -1)[a]
    opt.zero_grad(); loss.backward(); opt.step()
    print(f"step {step}: sampled {a}, entropy {entropy(logits).item():.3f}")
```

**Why this, not that? Why clip-higher or token selection, not a plain entropy bonus?**

**Why.** An entropy bonus pushes up entropy on *all* tokens, including ones that should be deterministic (syntax, copying), and it is sensitive to its coefficient; targeted methods only free up the tokens that matter for exploration.
**Trade-off.** Targeted methods add hyperparameters ($\varepsilon_{\text{high}}$, quantile, covariance thresholds) and are newer, with less evidence across model families.
**When the alternative wins.** A small, adaptive entropy bonus is fine for short runs or when you cannot change the loss internals; it is also the classic baseline to report.

## 31.3 Length: the other runaway variable

Mean response length tells you a lot:

- **Steady growth with rising accuracy**: healthy (R1-Zero pattern).
- **Growth driven by wrong answers**: often GRPO's per-sequence length normalisation (Dr. GRPO, arXiv 2503.20783); use token-level or constant normalisation.
- **Many responses hitting the maximum length**: truncated samples get zero reward and noisy gradients; use overlong filtering or soft overlong punishment (DAPO), raise the limit in stages (R1-Zero raised 32,768 → 65,536), or add a length penalty.
- **Collapse to very short answers**: often a format reward that is too easy to get, or a KL that is too strong.

Also log the **clip ratio** (share of tokens where the ratio was clipped). Very high means updates

are too large or the data is stale (off-policy); near zero with no progress means the learning rate may be too small.

## 31.4 The metrics to log

| Metric | Healthy | Warning sign |
|---|---|---|
| Mean reward (train) | rising, then slowly | flat from the start (data too hard/easy, broken reward) |
| Reward std within groups | clearly $> 0$ | near 0 (no learning signal) |
| Share of mixed groups | high | low (filter data) |
| Entropy | slow decline | crash to near 0 in the first steps; or rising without bound |
| Response length | stable or steady growth | explosion, or many hitting the max |
| Clip ratio | a few % | large and growing |
| KL to reference | slow growth | sudden jumps |
| Train/infer log-prob gap | small | growing (vLLM vs trainer mismatch, Chapter 28) |
| Held-out accuracy, pass@1 and pass@k | rising | train reward up but eval flat (hacking or overfitting) |

**Always read samples.** Every few hundred steps, print several completions per prompt. Many failures (reward hacks, format exploits, language mixing, repetition) are obvious to a human and invisible in averages.

## 31.5 Evaluating reasoning: pass@k and its trap

**pass@k** is the chance that at least one of $k$ samples is correct. With $n \geq k$ samples of which $c$ are correct, the unbiased estimator from *Evaluating Large Language Models Trained on Code* (Chen et al., arXiv 2107.03374) is

$$\text{pass@}k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}.$$

Report **pass@1 averaged over several samples** (DeepSeek-R1 used temperature 0.6 sampling rather than greedy decoding, because greedy decoding of long reasoning models repeats and varies a lot).

```
from math import comb

def pass_at_k(n, c, k):
    """Unbiased pass@k from n samples with c correct (Chen et al., 2021)."""
    if n - c < k:
        return 1.0
```

**Notation on this page:** $k$: attempts in pass@$k$ • $n$: samples per problem • $c$: number of correct samples

```
    return 1.0 - comb(n - c, k) / comb(n, k)

for k in (1, 4, 16, 64):
    print(f"n=64, c=8 -> pass@{k:2d} = {pass_at_k(64, 8, k):.3f}")
```

**The trap.** *Does Reinforcement Learning Really Incentivize Reasoning Capacity in LLMs Beyond the Base Model?* (Yue et al., arXiv 2504.13837) found that RLVR models win at small $k$, but at large $k$ the **base model often catches up or wins**: RL mostly sharpens the distribution towards solutions the base model could already find. *ProRL: Prolonged Reinforcement Learning Expands Reasoning Boundaries in Large Language Models* (Liu et al., arXiv 2505.24864) argued the opposite for **long** training on diverse tasks: with KL control and periodic **reference-policy resets**, RL models beat the base model across pass@k, even on tasks where the base model fails at any number of attempts. In interviews, mention both: RL sharpens, and with enough diverse, prolonged training it can also expand.

**Why this, not that? Why report pass@k curves, not only pass@1?**

**Why.** pass@1 shows how often the model is right; pass@k at large $k$ shows what it *can* find. Together they show whether RL sharpened or expanded the model, and whether diversity was lost.
**Trade-off.** Large-$k$ evaluation costs many samples and is only meaningful with a reliable checker (otherwise guessing inflates it).
**When the alternative wins.** For product decisions, pass@1 (or majority-vote accuracy at a fixed budget) is what users experience; pass@k is a research diagnostic.

## 31.6 Long runs: resets, staging and stability

- **Reference resets.** Replace the reference model (and optionally the optimiser state) with the current policy every few hundred steps, so the KL penalty does not pull back to a stale model (R1-Zero every 400 steps; ProRL periodic resets).
- **Staged context length.** Train shorter first, then raise the maximum length (R1-Zero; many open recipes).
- **Precision.** Mismatch between rollout engine and trainer grows with long sequences; use importance-sampling correction (TIS, MIS) or match numerics (*Defeating the Training-Inference Mismatch via FP16*, arXiv 2510.26788).
- **Checkpoints and rollbacks.** Save often; when entropy or KL spike, roll back and lower the learning rate or tighten clipping.
- **Asynchronous pipelines.** Generation is the bottleneck; async and off-policy pipelines (Chapter 28) speed up training but require staleness limits and ratio corrections.

## 31.7 A debugging walk-through

*"My GRPO reward is flat at 0.05 after 300 steps."*

1. Check the reward function on known good and bad answers (unit tests). Is it parsing the

**Notation on this page:** $k$: attempts in pass@$k$

answer tag the model actually uses?
2. Check the share of mixed groups. If most groups are all-wrong, the data is too hard: filter, use an SFT warm start, or increase $G$.
3. Read samples: is the model producing the required format? Is it truncated at the maximum length?
4. Check the learning rate (typical $10^{-6}$ to $3 \times 10^{-6}$ full-parameter; higher for LoRA) and that gradients are non-zero.
5. Check the chat template and that the loss mask covers only completion tokens.

*"Reward rose quickly then accuracy on the eval set fell."* Read samples for hacks (format exploits, answer guessing, judge manipulation); compare train and eval reward functions; check KL and length.

## 31.8 Common mistakes

- Not unit-testing the reward function.
- Evaluating with greedy decoding on long-CoT models, or with one sample per problem.
- Ignoring the share of zero-advantage groups.
- Changing many things at once in a failing run.
- Forgetting decontamination.

**Interview check**

**Q1. Why filter prompts by difficulty in GRPO?** All-correct or all-wrong groups have zero advantage and give no gradient; $P(\text{mixed}) = 1 - p^G - (1-p)^G$ is small for very easy or hard prompts.
**Q2. What is entropy collapse and how do you fight it?** Policy entropy drops early and performance saturates; fight with clip-higher, targeted token updates (Clip-Cov/KL-Cov, top-entropy tokens), careful entropy bonuses, diverse data.
**Q3. What drives entropy down in policy-gradient RL?** Covariance between log-probability and advantage: likely tokens with positive advantage get more likely (Cui et al.).
**Q4. Write the unbiased pass@k estimator.** $1 - \binom{n-c}{k} / \binom{n}{k}$.
**Q5. Does RLVR expand reasoning?** At large $k$, base models often match RL models (Yue et al.), suggesting sharpening; ProRL shows prolonged, diverse RL with resets can expand the boundary.
**Q6. Which metrics do you log in a GRPO run?** Reward and its within-group std, share of mixed groups, entropy, lengths and truncation rate, clip ratio, KL, train/infer log-prob gap, held-out pass@1/pass@k, plus samples.
**Q7. Response length explodes — causes?** Length-normalisation bias, truncation with noisy rewards, a reward that favours more steps, or genuine learning; check whether growth is in correct or wrong answers.
**Q8. Why reset the reference model?** So the KL penalty anchors to a recent policy rather than holding back progress; used by R1-Zero and ProRL.

**Notation on this page:** $G$: group size • $p$: probability the model solves a prompt • $k$: attempts in pass@$k$ • $n$: samples per problem • $c$: number of correct samples

**Key takeaways**

- Difficulty decides learning signal; filter offline and online, refresh the pool.
- Entropy is the exploration budget; manage it with targeted methods.
- Watch length, clip ratio, KL and train/infer mismatch; always read samples.
- Evaluate with sampled pass@1 and pass@k; know the sharpening vs expanding debate.

PART VII

# RL Environments and Agents

CHAPTER 32

# Environment Basics

**Why this chapter matters.**
Every RL system has two halves: a **policy** that chooses actions and an **environment** that answers. Post-training teams now spend as much time building environments as tuning losses. Interviewers ask you to design one ("build an environment for SQL repair"), and the design questions all come back to the ideas here: what is an observation, what ends an episode, where the reward comes from, and how to run many copies at once. This chapter teaches the standard Gymnasium interface from scratch, then shows how the same ideas carry over to LLM environments in the next chapters.

## 32.1 What an environment is

An **environment** is a program that holds a state of the world, accepts an action, and returns what happened. In the language of Chapter 13 it implements the **transition function** $P(s_{t+1} \mid s_t, a_t)$ and the **reward function** $r(s_t, a_t)$. The agent never sees the state directly unless the environment chooses to show it. What the agent sees is an **observation**: a (possibly partial) view of the state.

Three words to keep apart:

- **State** $s_t$: everything the environment needs to continue the simulation. For a chess game, the board, whose turn it is, castling rights and the move history.
- **Observation** $o_t$: what the agent is shown. In chess the observation can equal the state. In poker it cannot: you do not see the other player's cards. When $o_t \neq s_t$ the problem is a **partially observable MDP** (**POMDP**).
- **Action** $a_t$: what the agent sends back. A number, a vector, or, for an LLM, a whole string of text.

For LLMs the state is usually the full conversation so far plus any hidden world state (a file system, a database, a test suite). The observation is the text the model sees: the conversation rendered with the chat template. The action is the next assistant message.

## 32.2 The Gymnasium API

**Gymnasium** is the maintained successor of OpenAI Gym, now run by the Farama Foundation. The book was checked against Gymnasium 1.4.0 (installed 20 Oct 2026). Its API is the lingua franca of RL: almost every environment library, including several LLM ones, copies its two core methods.

- `reset(seed=None, options=None) -> (observation, info)` starts a new **episode** (one

**Notation on this page:** $s$: a state • $a$: an action • $t$: time step or token position • $r(x, y)$: the reward for response $y$ to prompt $x$

run from start to end). `seed` makes the episode reproducible. `info` is a dictionary of extra diagnostics that the agent should not train on.
- `step(action) -> (observation, reward, terminated, truncated, info)` applies one action and returns five things.

Every environment also declares two **spaces**:

- `action_space`: the set of legal actions, for example `Discrete(2)` (actions 0 and 1) or `Box(low, high, shape)` (real vectors).
- `observation_space`: the set of possible observations.

Spaces let generic code check its inputs and build the right network head without knowing the task.

```
import gymnasium as gym
env = gym.make("CartPole-v1")
obs, info = env.reset(seed=0)
print("observation:", obs.round(3), "space:", env.observation_space.shape,
 ↪  env.action_space)
print("time limit:", env.spec.max_episode_steps)
total, steps = 0.0, 0
while True:
    obs, reward, terminated, truncated, info = env.step(env.action_space.sample())
    total += reward; steps += 1
    if terminated or truncated:
        break
print(f"random policy: {steps} steps, return {total}, terminated={terminated},
 ↪  truncated={truncated}")
print(env)          # the wrapper stack that gym.make built
```

The last line prints the **wrapper stack**. A **wrapper** is an object that holds an environment and changes one aspect of it while passing everything else through. `gym.make("CartPole-v1")` gives `TimeLimit<OrderEnforcing<PassiveEnvChecker<CartPoleEnv>>>`: a time limit of 500 steps, a check that you call `reset` before `step`, and a checker that warns about API misuse.

## 32.3 Terminated vs truncated

Older Gym returned a single `done` flag. Gymnasium splits it in two, and the split matters for learning.

- **Terminated** means the episode reached a **terminal state** of the MDP: the pole fell, the game was won, the answer was submitted. Nothing comes after; the value of a terminal state is zero by definition.
- **Truncated** means the episode was stopped for an *outside* reason: a time limit, a token budget, a crash. The world did not end. If we had let it continue, more reward could have come.

Why this matters: the target for a value function (Chapter 14) is

$$y_t = r_t + \gamma\,(1 - d_t)\,V(s_{t+1}),$$

where $d_t$ should be 1 *only for termination*. If you treat truncation as termination you teach the critic that states near the time limit are worth nothing, which is false, and the critic becomes biased exactly where episodes are cut.

**Worked example: the cost of confusing the flags**
Take a task that pays $r = 1$ per step forever, $\gamma = 0.99$, with a time limit of 100 steps. The true value of every state is $1/(1-\gamma) = 100$. With the correct target, the last step bootstraps from $V(s_{101}) \approx 100$, so the critic learns 100 everywhere. With the wrong target, the last step's target is just 1, the step before it $1 + 0.99 = 1.99$, and so on: the critic learns $V(s_t) = \sum_{k=0}^{100-t} 0.99^k$, which falls from about 63 at the start to 1 at the end. The policy is then told that late states are bad, for no reason.

```
import numpy as np
gamma, T = 0.99, 100
def fitted_values(treat_truncation_as_terminal, sweeps=2000):
    V = np.zeros(T + 2)                       # V[t] for t = 1..T, plus a "next" slot
    for _ in range(sweeps):                   # repeated Bellman backups (policy evaluation)
        for t in range(T, 0, -1):
            last = (t == T)
            boot = 0.0 if (last and treat_truncation_as_terminal) else V[t + 1] if not
 ↪  last else V[t]
            V[t] = 1.0 + gamma * boot
    return V[1:T + 1]
good, bad = fitted_values(False), fitted_values(True)
print("correct:   V(start) %.1f   V(end) %.1f" % (good[0], good[-1]))
print("confused:  V(start) %.1f   V(end) %.1f" % (bad[0], bad[-1]))
```

In the "correct" branch the final step bootstraps from the value of a similar state (here the state itself, since all states are alike). That is what a real implementation does: it keeps the true next observation (Gymnasium puts it in the return value of `step` even when `truncated=True`) and bootstraps from $V$ of it.

**Why this, not that? Why two flags, not one done flag?**

**Reason.** Termination and truncation need different maths: zero bootstrap for the first, normal bootstrap for the second. One flag cannot carry both. **Trade-off.** Two flags mean more code paths and a common bug where people OR them together everywhere. **When one flag is fine.** In LLM RL with $\gamma = 1$ and no critic (GRPO, RLOO), returns are just the final reward and nothing is bootstrapped, so the difference shows up somewhere else: what to do with the *reward* of a truncated response (Chapter 28's overlong filtering and `mask_truncated_completions`). The concept still matters; only the place it bites moves.

For LLMs, the same split appears as "the model emitted end-of-sequence" (terminated) versus "the response hit `max_tokens`" (truncated). Mixing them up is the root cause of several length biases discussed in Part VI.

**Notation on this page:** $\gamma$: discount factor (0 to 1) • $V(s)$: state value (expected future return from $s$) • $y$: the response (output tokens); $y_t$ is token $t$ • $r(x, y)$: the reward for response $y$ to prompt $x$ • $s$: a state • $t$: time step or token position • $V$: vocabulary size • $d$: hidden size of the model • $k$: an index or count (see text)

## 32.4 Writing a custom environment

To write an environment you subclass `gymnasium.Env`, declare the spaces in `__init__`, and implement `reset` and `step`. The rules:

1. Call `super().reset(seed=seed)` at the start of `reset`. It seeds `self.np_random`, the environment's own random generator. Use that generator, never the global `np.random`, so episodes are reproducible.
2. Return observations that lie inside `observation_space`, with the right dtype.
3. Return `reward` as a float and the two flags as bools.
4. Keep everything needed to continue the episode inside the object, so that many copies can run side by side.

Here is a small environment shaped like an LLM task: guess a hidden number from 0 to 15 with "higher/lower" feedback. It has a terminal state (correct guess), a time limit (truncation) and partial observability (the target is hidden).

```
import gymnasium as gym
from gymnasium import spaces
import numpy as np

class GuessEnv(gym.Env):
    """Guess a hidden integer in [0, 15]. Feedback: -1 lower, 0 none yet, +1 higher."""
    metadata = {"render_modes": []}
    def __init__(self, n=16, max_guesses=6):
        self.n, self.max_guesses = n, max_guesses
        self.action_space = spaces.Discrete(n)
        # observation: last guess (or n if none) and last feedback in {-1,0,1} shifted to
        ↪ {0,1,2}
        self.observation_space = spaces.MultiDiscrete([n + 1, 3])
    def reset(self, seed=None, options=None):
        super().reset(seed=seed)
        self.target = int(self.np_random.integers(self.n))
        self.guesses = 0
        return np.array([self.n, 1]), {}
    def step(self, action):
        self.guesses += 1
        a = int(action)
        feedback = 0 if a == self.target else (1 if self.target > a else -1)
        terminated = (a == self.target)
        truncated = (not terminated) and self.guesses >= self.max_guesses
        reward = 1.0 if terminated else 0.0
        return np.array([a, feedback + 1]), reward, terminated, truncated, {"target":
        ↪ self.target}

from gymnasium.utils.env_checker import check_env
env = GuessEnv()
check_env(env)                      # raises if the API contract is broken
def binary_search_episode(env, seed):
```

```
    obs, _ = env.reset(seed=seed); lo, hi = 0, env.n - 1
    while True:
        guess = (lo + hi) // 2
        obs, r, term, trunc, info = env.step(guess)
        if term or trunc: return r, env.guesses
        if obs[1] == 2: lo = guess + 1
        else: hi = guess - 1
results = [binary_search_episode(env, s) for s in range(200)]
print("binary search: success %.2f, mean guesses %.2f" % (np.mean([r for r, _ in
 ↪  results]), np.mean([g for _, g in results])))
rng = np.random.default_rng(0)
succ = 0
for s in range(200):
    env.reset(seed=s)
    for _ in range(env.max_guesses):
        _, r, term, trunc, _ = env.step(int(rng.integers(env.n)))
        if term or trunc: break
    succ += r
print("random guessing: success %.2f" % (succ / 200))
```

`check_env` runs a battery of tests: spaces are valid, `reset` honours the seed, outputs have the right types. Run it on every new environment. Binary search wins every time, in about 3.5 guesses on average and never more than 5 (one more than $\log_2 16 = 4$, because of how the midpoint rounds); random guessing succeeds only about a quarter of the time. The gap between a random policy and the best policy is what RL has to close; if the random policy already succeeds 99% of the time, or 0% of the time, the environment gives little learning signal. Chapter 36 turns this into a design rule.

## 32.5 Wrappers: changing one thing at a time

Wrappers let you change an environment without editing it. Gymnasium 1.4 ships many; the important ones:

| Wrapper | What it does | When you use it |
|---|---|---|
| `TimeLimit(env, max_episode_steps)` | Sets `truncated=True` after a fixed number of steps | Every episodic task; added by `gym.make` |
| `RecordEpisodeStatistics` | Adds episode return and length to `info` at the end | Logging |
| `NormalizeObservation` / `NormalizeReward` | Running mean/std normalisation | Continuous control with PPO |
| `ClipReward`, `TransformReward` | Changes the reward | Reward shaping experiments |
| `FrameStackObservation` | Stacks the last $k$ observations | Partial observability (Atari) |
| `TimeAwareObservation` | Appends the time step to the observation | Lets the agent know a deadline is near |

**Notation on this page:** $k$: an index or count (see text)

| Wrapper | What it does | When you use it |
|---|---|---|
| `Autoreset` | Starts a new episode automatically after the old one ends | Vector environments |

`TimeAwareObservation` deserves a note. If the agent cannot see how much time is left, a time limit makes the problem non-Markov: the same observation can be followed by a cut or by more steps. Adding the step counter restores the Markov property. LLM environments do the same thing when they tell the model "you have 3 tool calls left".

Writing your own wrapper is easy: subclass `gym.Wrapper` (or `ObservationWrapper`, `RewardWrapper`, `ActionWrapper`) and override one method.

```
class StepPenalty(gym.RewardWrapper):
    """Subtract a small cost per step to encourage short episodes."""
    def __init__(self, env, cost=0.05):
        super().__init__(env); self.cost = cost
    def reward(self, reward):
        return reward - self.cost

wrapped = StepPenalty(GuessEnv(), cost=0.05)
obs, _ = wrapped.reset(seed=1)
_, r, term, trunc, info = wrapped.step(7)
print("reward after one guess with step penalty:", r, "target was", info["target"])
```

## 32.6 Vector environments

RL is hungry for samples, and one environment instance gives one sample per step. A **vector environment** runs $N$ copies and steps them together: `step(actions)` takes a batch of $N$ actions and returns batched observations, rewards and flags. Gymnasium offers `SyncVectorEnv` (copies in one process, stepped in a loop) and `AsyncVectorEnv` (one subprocess per copy), created with `gym.make_vec(id, num_envs, vectorization_mode="sync" | "async")`.

The tricky part is what happens when one copy finishes while others continue. Gymnasium 1.x calls this the **autoreset mode** and offers three: `NextStep` (the default), `SameStep` and `Disabled`. In `NextStep` mode, the step that ends an episode returns the final observation as usual; the *next* call to `step` ignores the action for that copy, resets it and returns the first observation of the new episode. Your rollout code must skip that "reset step" when it builds training data, or it will create a fake transition from the last state of one episode to the first state of the next.

```
venv = gym.make_vec("CartPole-v1", num_envs=4, vectorization_mode="sync")
print("autoreset mode:", venv.metadata["autoreset_mode"])
obs, info = venv.reset(seed=0)
prev_done = np.zeros(4, dtype=bool)
valid_transitions, skipped = 0, 0
for _ in range(300):
```

**Notation on this page:** $N$: number of examples

```
    obs, rew, term, trunc, info = venv.step(venv.action_space.sample())
    # In NextStep mode, a copy that finished last step was just reset: this transition
     ↪  is not real.
    valid_transitions += int((~prev_done).sum()); skipped += int(prev_done.sum())
    prev_done = term | trunc
print(f"real transitions: {valid_transitions}, reset steps skipped: {skipped}")
```

**Why this, not that? Why vectorise environments, not just run a bigger batch of episodes one after another?**

**Reason.** The policy network is fastest when it scores many observations in one batch. Vector environments keep the batch full at every step. **Trade-off.** Episode boundaries no longer line up, so you need autoreset handling and per-copy bookkeeping. **When the alternative is better.** When episodes are long and expensive and each step is dominated by the environment, not the network (a browser, a compiler), you want *asynchronous* workers that do not wait for each other, not lock-step vectors. LLM agent systems are in this regime (Chapter 34).

## 32.7 Seeding and reproducibility

Reproducibility has three layers. The environment's own generator is seeded by `reset(seed=...)`, and only the first `reset` needs a seed; later resets continue the same random stream. The policy's sampling has its own seed (`torch.manual_seed`, or `seed` in vLLM's `SamplingParams`). And parallel or GPU code adds non-determinism that seeds alone do not remove. For LLM RL you rarely get bit-exact reruns; aim for *statistical* reproducibility: the same mean curve over three seeds.

**Seeding every reset with the same number**

Calling `env.reset(seed=0)` at the start of every episode replays the same episode forever. The agent overfits one instance and your evaluation looks perfect. Seed once, or seed with a different number per episode.

## 32.8 From Gymnasium to LLM environments

The Gymnasium contract maps onto LLM tasks with small changes.

| Gymnasium idea | LLM version |
|---|---|
| `reset()` returns the first observation | Sample a task; build the prompt (system message, user question, tool descriptions) |
| Action = integer or vector | Action = a whole assistant message (hundreds of tokens), or one token if you view each token as an action |

| Gymnasium idea | LLM version |
|---|---|
| `step(action)` | Parse the message; if it calls a tool, run the tool and return the output as the next observation; if it gives a final answer, score it |
| `terminated` | The model answered, or gave up, or the task's goal state was reached |
| `truncated` | Token budget, turn limit or wall-clock limit hit |
| `reward` | Usually 0 until the end, then a verifier score (Chapter 26) |
| `info` | Diagnostics: test results, which rubric item failed, tool errors |
| Vector environment | Many rollouts in flight on an inference server (vLLM), each with its own environment instance |

Two differences are deep, not cosmetic. First, the action space is enormous: every possible string. Nobody enumerates it; the policy *generates* an action and the environment *parses* it, so parsing becomes part of the environment's job and a source of bugs. Second, there are two time scales. At the token level an episode has thousands of steps and the "environment" is just "append the token". At the turn level an episode has a handful of steps and the environment does real work. RL algorithms for LLMs work at the token level for the loss but at the turn level for the environment (Chapter 34).

**Why this, not that? Why keep the Gymnasium shape for LLM environments, not a custom interface per task?**

**Reason.** A shared `reset/step` contract lets one trainer use many environments and lets environments be shared, tested and benchmarked. OpenEnv (Meta/PyTorch) and several other projects adopt exactly this shape for LLM agents; TRL's `environment_factory` keeps `reset` and turns the other methods into tools. **Trade-off.** Text tasks fit `step` awkwardly: a single-turn maths problem has one step, and squeezing it through a Gym-style loop adds ceremony. **When the alternative is better.** For single-turn RLVR, a plain reward function `reward(prompt, completion, **columns) -> float` (Chapter 26) is simpler and is what TRL, veRL and OpenRLHF all accept.

## 32.9 Common mistakes

1. **ORing the flags.** Using `done = terminated or truncated` in the value target. Use `terminated` only for zero bootstrap.
2. **Global random state.** Using `np.random` or `random` inside `step` instead of `self.np_random`. Seeds stop working and parallel copies become correlated.
3. **Leaking the answer in `info` or the observation.** A field such as `info["target"]` is fine for logs but must never reach the policy's input. In LLM environments the analogous leak is

putting the reference answer in the prompt or in a tool's output.

4. **Training on autoreset transitions.** In `NextStep` mode the step after an episode ends is a reset, not a transition.
5. **Shared mutable state between copies.** A class-level list or a shared temp directory makes vector copies interfere. LLM sandboxes have the same bug at a larger scale (Chapter 35).
6. **No `check_env`.** Run the checker, and also write a scripted "oracle" policy (like binary search above) to prove the task can be solved and the reward can reach its maximum.

**Interview check**

**Q1. What does `step` return in Gymnasium, and why two done flags?** `(observation, reward, terminated, truncated, info)`. `terminated` marks a true terminal state (value 0); `truncated` marks an outside cut such as a time limit, where the critic must still bootstrap from $V(s_{t+1})$.

**Q2. Your critic's values drop sharply near the time limit. Likely bug?** Truncation treated as termination in the TD target, so the last steps bootstrap from zero.

**Q3. What is a POMDP and give an LLM example.** A process where the agent sees an observation, not the full state. An LLM agent editing a repository sees only the files it has opened; the rest of the repository is hidden state.

**Q4. What is a wrapper? Name three.** An object that changes one aspect of an environment and passes the rest through: `TimeLimit`, `RecordEpisodeStatistics`, `NormalizeReward` (also `TimeAwareObservation`, `FrameStackObservation`).

**Q5. Why add the remaining time to the observation?** With a hidden deadline the same observation can lead to different futures, so the problem is not Markov; showing the time restores it.

**Q6. What is autoreset, and what bug does it cause?** In vector environments a finished copy is reset automatically. In Gymnasium's `NextStep` mode the next step for that copy is a reset; treating it as a transition links two unrelated episodes.

**Q7. How do you check that a new environment is correct before training?** `check_env` for API conformance; an oracle policy that reaches maximum reward; a random policy to measure the baseline; unit tests for edge cases (invalid actions, time limit); confirm no answer leakage.

**Q8. Map Gymnasium concepts to a multi-turn tool-use LLM task.** `reset` builds the prompt from a sampled task; the action is an assistant message; `step` parses it, runs any tool call and returns the tool output; `terminated` when the model answers; `truncated` at a token or turn budget; reward from a verifier at the end.

**Q9. When should a "time-out" in an LLM rollout get reward 0, and when should it be masked?** If running out of budget is a real failure you want to discourage (the task requires an answer within the budget), give it the failure reward. If the cut is arbitrary and the answer might have been right, mask it from the loss (DAPO-style overlong filtering) so you do not punish reasoning that was merely long.

**Notation on this page:** $V(s)$: state value (expected future return from $s$) • $s$: a state • $t$: time step or token position • $V$: vocabulary size

**Key takeaways**

- An environment implements `reset` and `step`; observations, actions and rewards are typed by spaces.
- Terminated means the MDP ended (no bootstrap); truncated means it was cut (bootstrap).
- Seed with the environment's own generator; vectorise for throughput; handle autoreset.
- LLM environments keep the same shape: prompt = first observation, message = action, tool output = next observation, verifier = reward.

CHAPTER 33

# LLM Environments

**Why this chapter matters.**
For a language model, "the environment" is the code around generation: it builds the prompt, reads the model's text, runs tools, and turns the outcome into a reward. Most failures in LLM RL come from this code, not from the loss: a parser that misses correct answers, a rubric that pays for formatting, a tokenisation mismatch that makes the trainer score text the model never produced. This chapter shows how text environments are built, what the main libraries look like in October 2026, and the bugs to look for.

## 33.1 Text in, text out

An LLM environment has the same job as a Gymnasium environment (Chapter 32), but its observations and actions are text. The minimum pieces:

1. **A task source.** A dataset of prompts with whatever is needed to judge an answer: a reference answer, unit tests, a target database state. One row is one **task**.
2. **A prompt builder.** Turns a task into messages: a system message with rules and tool descriptions, then the user message. The chat template (Chapter 8) then turns the messages into tokens.
3. **A parser.** Reads the model's text and extracts what matters: the final answer, a tool call, a code block.
4. **A reward function** (or **rubric**: several weighted reward functions). Compares the parsed output with the task's reference and returns a number.
5. **Optionally, tools and state.** For multi-turn tasks, functions the model can call and the world they act on.

The **rollout** is the loop that ties these together: build prompt, generate, parse, (maybe) call tools and generate again, score. The rollout produces a **trajectory**: the prompt tokens, the generated tokens, the log-probabilities the sampler used, a mask saying which tokens the model produced, and the reward.

```
import re

def build_prompt(task):
    return [
        {"role": "system", "content": "Solve the problem. Think step by step, then give
↪   the final answer as \\boxed{...}."},
        {"role": "user", "content": task["question"]},
    ]
```

```
def parse_boxed(text):
    """Return the content of the LAST \\boxed{...}, handling nested braces, or None."""
    start = text.rfind("\\boxed{")
    if start < 0:
        return None
    i, depth, out = start + len("\\boxed{"), 1, []
    while i < len(text):
        c = text[i]
        if c == "{": depth += 1
        elif c == "}":
            depth -= 1
            if depth == 0: return "".join(out).strip()
        out.append(c); i += 1
    return None                                  # unbalanced: treat as no answer

def normalise(ans):
    ans = ans.replace(" ", "").replace("\\!", "").rstrip(".")
    ans = re.sub(r"^\\text\{(.*)\}$", r"\1", ans)
    try:
        return str(float(ans.replace(",", "")))   # "1,000" -> "1000.0", "0.50" -> "0.5"
    except ValueError:
        return ans

def correctness(task, completion):
    pred = parse_boxed(completion)
    return 1.0 if pred is not None and normalise(pred) == normalise(task["answer"]) else
     ↪ 0.0

task = {"question": "What is 1/2 + 1/4?", "answer": "0.75"}
for c in ["... so the sum is \\boxed{0.75}.",
          "... \\boxed{\\frac{3}{4}}",               # right answer, form the normaliser
           ↪ cannot read
          "First guess \\boxed{1}, but correcting: \\boxed{0.750}",
          "The answer is 0.75"]:                    # right answer, wrong format
    print(f"{correctness(task, c):.0f}  <- {c!r}")
```

Look at the second and fourth cases: the model is right, the reward says 0. The parser and normaliser decide what "correct" means, and every gap becomes a **false negative** (right answer, zero reward). False negatives slow learning and push the model toward whatever surface form the checker happens to read. The fix is better checking: symbolic equivalence for maths (libraries such as `math-verify` parse LaTeX and compare with SymPy), execution for code, and a measured false-negative rate on a hand-labelled sample. Chapter 26 discusses verifier design in depth; here the point is that *the parser is part of the environment*, and you must test it like production code.

## 33.2 Single-turn and multi-turn rollouts

**Single-turn** environments generate one response per prompt and score it. GSM8K-style maths, code with unit tests, instruction following with checkable constraints: all single-turn. The "envi-

ronment" can be just a reward function, and every major library accepts one:

- TRL 1.16: `reward_funcs` callables receive `prompts`, `completions`, `completion_ids`, every extra dataset column by name, and `trainer_state` (Chapter 27). They may return `None` for samples they do not apply to.
- veRL 0.9.0: `reward.custom_reward_function.path` and `.name` (default name `compute_score`) point to a Python function scored per sample, keyed by the dataset's `data_source` column (`data.reward_fn_key`).
- OpenRLHF 0.11.2: `--reward.remote_url` can be an HTTP reward server or a `.py` file that defines `reward_func(queries, prompts, labels)`.

**Multi-turn** environments interleave model turns with environment turns. The model writes, the environment answers (a tool result, a user reply, a game move), the model writes again, until a stop condition. Three kinds:

- **Tool use**: the environment executes function calls (search, Python, SQL) and returns results.
- **Interactive tasks**: games, puzzles, web navigation, where the environment has state that changes with each action.
- **Simulated users**: another model plays the user, as in customer-support benchmarks.

The trajectory of a multi-turn rollout is one long token sequence with two kinds of tokens: those the policy generated and those the environment inserted. Only the first kind are actions. The loss must be masked so that environment tokens get no gradient (Chapter 34 explains why in detail).

```
def calculator(expression: str) -> str:
    """A tiny, safe calculator tool: digits, + - * / ( ) and spaces only."""
    if not re.fullmatch(r"[0-9+\-*/(). ]+", expression):
        return "error: unsupported characters"
    try:
        return str(eval(expression, {"__builtins__": {}}, {}))
    except Exception as e:
        return f"error: {type(e).__name__}"

class ScriptedPolicy:
    """Stands in for an LLM: first asks the calculator, then answers with the result."""
    def __call__(self, messages):
        if messages[-1]["role"] == "tool":
            return f"The result is \\boxed{{{messages[-1]['content']}}}"
        return '<tool_call>{"name": "calculator", "arguments": {"expression": "37 *
        ↪  91"}}</tool_call>'

def parse_tool_call(text):
    import json
    m = re.search(r"<tool_call>(.*?)</tool_call>", text, re.S)
    if not m: return None
    try:
        call = json.loads(m.group(1))
```

```
        return call["name"], call["arguments"]
    except (ValueError, KeyError):
        return ("__invalid__", {})

def rollout(task, policy, tools, max_turns=4):
    messages = build_prompt(task)
    segments = []                                   # (text, produced_by_model)
    for turn in range(max_turns):
        reply = policy(messages)
        messages.append({"role": "assistant", "content": reply});
 ↪  segments.append((reply, True))
        call = parse_tool_call(reply)
        if call is None:                            # no tool call: final answer, episode
         ↪  terminates
            return messages, segments, correctness(task, reply), "terminated"
        name, args = call
        result = tools[name](**args) if name in tools else f"error: unknown tool {name}"
        messages.append({"role": "tool", "content": result}); segments.append((result,
 ↪  False))
    return messages, segments, 0.0, "truncated"      # turn budget used up

msgs, segs, reward, how = rollout({"question": "37*91?", "answer": "3367"},
 ↪  ScriptedPolicy(), {"calculator": calculator})
for text, by_model in segs:
    print(("MODEL " if by_model else "ENV   ") + text)
print("reward", reward, "|", how)
```

The `segments` list is the heart of a multi-turn trajectory: it records, piece by piece, who wrote each span of text. The trainer turns it into a **loss mask** at the token level.

**Why this, not that? Why give the environment a turn budget, not let the model talk until it stops?**

**Reason.** Without a limit, a policy that loops (calls the same tool forever) makes rollouts unbounded. A budget bounds cost and makes the batch finish. **Trade-off.** Budgets create truncation: some rollouts that would have succeeded are cut. You must decide whether a cut rollout gets the failure reward or is masked (Chapter 32). **When the alternative is better.** For evaluation of long-horizon agents (software engineering), use generous budgets and measure cost separately; tight budgets at evaluation measure the budget, not the skill.

## 33.3 Tokens in, tokens out

Here is a bug that has cost many teams weeks. The sampler (vLLM) produces **token IDs**. A naive environment decodes them to text, appends the tool result, re-applies the chat template and **re-tokenises** the whole conversation for the next turn and for training. But decoding and re-encoding is not an identity: a model can produce a non-canonical tokenisation, for example `"Hello" + " wor" + "ld"`, which the tokenizer would never produce from the string "Hello world".

```
import os
os.environ.setdefault("HF_HUB_OFFLINE", "1")
from transformers import AutoTokenizer
tok = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B-Instruct")
for pieces in [["Hello", " wor", "ld"], ["The", " function", " returns", " Non", "e"]]:
    sampled = sum((tok.encode(p, add_special_tokens=False) for p in pieces), [])
    text = tok.decode(sampled)
    reencoded = tok.encode(text, add_special_tokens=False)
    print(f"{text!r:30} sampled {sampled}  re-encoded {reencoded}  same: {sampled ==
    ↪  reencoded}")
```

If the trainer computes log-probabilities on the re-encoded IDs, it scores a sequence the model did not sample. The importance ratio $\pi_\theta/\pi_{\text{old}}$ is then computed on the wrong tokens, and on-policy RL silently becomes off-policy. The fix is a principle called **tokens in, tokens out**: keep the exact token IDs the sampler produced, append environment tokens by tokenising *only the new text*, and never re-tokenise the model's own output. Modern frameworks do this (veRL's agent loop and TRL's tool loop work on token IDs and carry a mask of model tokens; TRL calls it `tool_mask` and accepts `env_mask` from a custom `rollout_func`), but if you write your own rollout, you must too.

Chat templates add a second trap. Some templates render earlier turns differently from the latest turn; for example, reasoning-model templates often drop the `<think>` part of earlier assistant turns (Chapter 8). Then the training sequence and the sequence the model saw during generation differ. Again, the fix is to build the training sequence from the tokens actually fed to the sampler at each turn.

**Why this, not that? Why keep token IDs, not just store text and re-tokenise later?**

**Reason.** RL needs the log-probability of exactly what was sampled. Text loses information about how it was tokenised. **Trade-off.** Token-level bookkeeping is more code, ties the environment to one tokenizer, and makes logs harder to read (store the text too, for humans). **When the alternative is better.** For SFT on fixed text (no sampling), and for rejection-sampling fine-tuning, re-tokenising canonical text is fine and even preferable: the model learns the canonical tokenisation.

## 33.4 Parsers, rubrics and reward functions as code

A **rubric** is a weighted set of reward functions. A typical reasoning rubric:

$$R(x,y) = w_{\text{correct}}\,\mathbb{1}[\text{answer correct}] + w_{\text{format}}\,\mathbb{1}[\text{format valid}] - w_{\text{len}}\cdot\text{overlong}(y).$$

Rules that keep rubrics honest:

- **Correctness dominates.** Make $w_{\text{correct}}$ much larger than the rest, so no amount of format reward can make a wrong answer look good. Otherwise the policy learns to format beautifully and give up on the hard part.
- **Format rewards should become free.** A model learns the format within tens of steps; after that the format term is a constant and adds only noise to advantages. Many recipes drop it

**Notation on this page:** $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{1}[\cdot]$: 1 if the condition is true, else 0 • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $R$: return or total reward

entirely and give a wrong-format answer zero correctness instead (DeepSeek-R1 used both terms with equal weight and a rule-based format check; Chapter 30).

- **Each function returns one thing.** Separate functions are easier to log, test and reweight than one function with hidden branches.
- **Log every component separately.** If total reward rises, you need to know which part rose.

```
def format_ok(completion):
    """Exactly one \\boxed{...} with a balanced body, and nothing after it but
     ↪  punctuation."""
    if completion.count("\\boxed{") != 1: return 0.0
    body = parse_boxed(completion)
    if body is None: return 0.0
    tail = completion[completion.rfind("\\boxed{") + len("\\boxed{") + len(body) + 1:]
    return 1.0 if re.fullmatch(r"[\s.!]*", tail) else 0.0

def rubric(task, completion, weights={"correct": 1.0, "format": 0.1}):
    parts = {"correct": correctness(task, completion), "format": format_ok(completion)}
    return sum(weights[k] * v for k, v in parts.items()), parts

# TRL-style reward functions: keyword arguments, one float (or None) per completion.
def correctness_reward(prompts, completions, answer, **kwargs):
    return [correctness({"answer": a}, c) for c, a in zip(completions, answer)]

def format_reward(prompts, completions, **kwargs):
    return [format_ok(c) for c in completions]

batch = {"prompts": ["q1", "q2", "q3"],
         "completions": ["\\boxed{4}", "so \\boxed{5}. Also \\boxed{4}", "\\boxed{4}
          ↪  trailing words"],
         "answer": ["4", "4", "4"]}
print("correct:", correctness_reward(**batch))
print("format: ", format_reward(**batch))
print("rubric for case 2:", rubric({"answer": "4"}, batch["completions"][1]))

def gated_reward(task, completion):
    """Correctness only counts when the format is valid: hedging scores 0."""
    return correctness(task, completion) * format_ok(completion)
print("gated:", [gated_reward({"answer": a}, c) for c, a in zip(batch["completions"],
 ↪  batch["answer"])])
```

Case 2 shows why the rubric needs both parts: it states two answers, and "last box wins" marks it correct. Without a format rule, a policy can learn to **hedge**: write several answers and hope the last one is right. Notice that the *additive* rubric barely helps: the hedge scores 1.0 against 1.1 for a clean answer, a tiny gap that the noise of RL will not respect. The robust fix is a **gate**: correctness counts only when the format is valid, so the hedge scores 0. Gates are stronger than penalties whenever a behaviour is an exploit rather than a matter of degree. (The first example in this chapter has the same hole: its third case hedges and still gets full marks.) Whenever you write a parser, ask "what is the cheapest text that gets full marks?" and test it.

TRL lets you weight reward functions with `reward_weights` (one float per function) and choose

how multiple rewards are combined before advantages with `multi_objective_aggregation` (`sum_then_normalize`, the default, or `normalize_then_sum`). The second option normalises each reward across the group first, so a reward with a large scale cannot drown the others.

## 33.5 LLM environment libraries in 2026

Environments are now packaged and shared like models. The libraries below were inspected from their released packages on 20 Oct 2026; APIs in this area change quickly, so check the version you install.

**TRL `environment_factory`** (TRL 1.16, marked experimental). You pass `GRPOTrainer(..., environment_factory=MyEnv)`, where `MyEnv()` creates one instance per rollout. The class must have a `reset(**kwargs)` method that returns `None` or a string (appended to the last user message). Its other public methods become **tools** the model can call; they need type hints and Google-style docstrings, from which the tool schema is built. If the class defines `get_reward()` (no arguments, returns a float), the environment owns the reward and `reward_funcs` becomes optional. A dictionary of factories lets one run mix environments, selected by an `environment` column in the dataset. Related: `tools=[...]` passes plain functions, `max_tool_calling_iterations` caps turns, and `rollout_func` replaces generation entirely.

```
class CounterEnv:
    """A toy TRL-style environment: the model must make the counter reach a target, then
    ↪ stop."""
    def reset(self, target: int = 3, **kwargs) -> str:
        self.value, self.target, self.calls = 0, target, 0
        return f"The counter starts at 0. Make it exactly {target} using the tools."
    def increment(self, amount: int) -> str:
        """Add an amount to the counter.

        Args:
            amount: How much to add (1 to 5).

        Returns:
            The new counter value, or an error message.
        """
        self.calls += 1
        if not 1 <= amount <= 5:
            return "error: amount must be between 1 and 5"
        self.value += amount
        return str(self.value)
    def get_reward(self) -> float:
        return 1.0 if self.value == self.target else 0.0

env = CounterEnv()
print(env.reset(target=7))
print(env.increment(5), env.increment(9), env.increment(2), "-> reward",
 ↪ env.get_reward(), "after", env.calls, "calls")
```

```
# Requires a tool-capable chat model and (in practice) a GPU. TRL 1.16.
from trl import GRPOConfig, GRPOTrainer
config = GRPOConfig(output_dir="counter-grpo", num_generations=8,
 ↪  max_completion_length=512,
                    max_tool_calling_iterations=6, max_steps=200, learning_rate=1e-6)
trainer = GRPOTrainer(model="Qwen/Qwen3-1.7B", args=config,
                      environment_factory=CounterEnv)      # no train_dataset: reset()
 ↪  supplies the task
trainer.train()
```

When no `train_dataset` is given, the environment's `reset` provides the task, and you must set `max_steps`, because the trainer cannot infer the length of training.

**OpenEnv** (`openenv-core` 0.3.0 on PyPI, from Meta's PyTorch team). An environment subclasses `openenv.core.env_server.interfaces.Environment` and implements Gymnasium-style `reset()`, `step(action)` and a `state` property, with typed pydantic `Action`, `Observation` (which carries `done` and `reward`) and `State` (with `episode_id` and `step_count`) classes. Environments run as HTTP servers, usually inside a Docker container, and trainers talk to them through a client. The CLI scaffolds, builds, validates and publishes environments (`openenv init`, `build`, `validate`, `push`, `serve`). The design goal is isolation and reuse: the same container serves any trainer.

**verifiers** (0.4.0, Prime Intellect). The original library introduced the vocabulary many people now use: environments, parsers and rubrics. Version 0.4 moved to a new stack under `verifiers.v1` with separate concepts for the **task set** (data plus reward), the **harness** (the agent loop that drives the model), the **toolset**, the **runtime** (where code runs: subprocess, Docker, Podman, Apptainer or a hosted sandbox) and **judges** (rubric, reference and decision judges for LLM-graded rewards). It is integrated with Prime Intellect's `prime-rl` trainer and an online **Environments Hub** where people publish environments.

**veRL and OpenRLHF** have their own multi-turn hooks. veRL 0.9.0 has `actor_rollout_ref.rollout.multi_turn` settings (off by default; `max_assistant_turns`, `max_tool_response_length=256`, `tool_response_truncate_side=middle`, tool call `format=hermes`) and an **agent loop** abstraction (`rollout.agent.default_agent_loop=single_turn_agent`, `num_workers=8`). OpenRLHF 0.11.2 loads a Python file given by `--train.agent_func_path`; it must define a class `AgentExecutor`, usually built on `MultiTurnAgentExecutor` with an `AgentInstanceBase` subclass whose async `step` returns a dictionary with `rewards`, `environment_feedback`, `done` and optional `scores`, `extra_logs` and `sampling_params`.

**Why this, not that? Why package an environment as a service, not as a function inside the trainer?**

**Reason.** Real environments run untrusted code, need their own dependencies, and can crash. A separate process (or container) isolates them, lets them scale independently of GPUs, and lets many trainers share them. **Trade-off.** Network calls add latency and failure modes (timeouts, retries, partial results). Debugging spans two processes. **When the alternative is better.** Pure, fast, trusted reward functions (string match, a regex, a small maths checker) belong in-process: a service would add only overhead.

## 33.6 Designing the observation

What you show the model is a design decision with large effects:

- **Tool descriptions.** The model can only call tools whose schemas are in the prompt. Short, precise docstrings help; long ones waste context in every turn.
- **Error messages.** When a tool fails, return a short, informative error ("error: amount must be between 1 and 5"), not a stack trace. Models learn to recover from errors they can understand.
- **Truncating long outputs.** A tool can return megabytes (a log file, a web page). Truncate, ideally keeping the start and end (veRL's `tool_response_truncate_side=middle` cuts from the middle), and say that you truncated.
- **Budgets.** Tell the model how many turns or tokens remain. As with `TimeAwareObservation`, it makes the problem closer to Markov.
- **No leaks.** The reference answer, the test file contents (if hidden tests decide the reward) and the grader's code must not be reachable through any tool.

## 33.7 Common mistakes

1. **Untested parsers.** Measure false-negative and false-positive rates of the reward on 100 hand-labelled samples before training.
2. **Re-tokenising model output.** Keep sampled token IDs; tokenise only environment text.
3. **Format reward that outweighs correctness.** The model will collect the easy reward and stop trying.
4. **Hedging accepted.** "Last answer wins" or "any answer matches" parsers reward listing many answers.
5. **Training on tool output.** Environment tokens must be masked out of the loss.
6. **Shared state across rollouts.** One environment instance per rollout; TRL's factory does this for you, hand-written loops often do not.
7. **Silent tool failures.** A tool that returns an empty string on error looks like "no result" to the model and to you. Return explicit errors and count them (`tool_failure` rates are a key metric).

**Interview check**

**Q1. What are the components of an LLM environment?** A task source, a prompt builder, a parser, a reward function or rubric, and optionally tools with state. The rollout loop ties them together and produces token IDs, sampler log-probs, a model-token mask and a reward.
**Q2. Why can a correct answer get zero reward, and how do you measure it?** Parser or normaliser failures (different format, equivalent form, unbalanced braces). Hand-label a sample and compute the verifier's false-negative rate.
**Q3. What is the "tokens in, tokens out" principle?** Keep the exact token IDs the sampler produced and never re-tokenise model text; otherwise the trainer computes log-probs of a different sequence, breaking the importance ratio.
**Q4. Give an example of retokenisation mismatch.** A model samples "Hello" + " wor" +

"ld"; decoding gives "Hello world", which re-encodes to "Hello" + " world". Same text, different tokens.

**Q5. How does TRL 1.16 support environments?** `environment_factory` creates one instance per rollout; `reset` returns text appended to the prompt; public methods become tools; optional `get_reward` supplies the reward; tool-result tokens are masked from the loss. It is marked experimental.

**Q6. Why should correctness dominate a rubric?** So that no combination of auxiliary rewards can make a wrong answer score higher than a right one; otherwise the policy optimises the cheap parts.

**Q7. A model learns to write several boxed answers. What happened and how do you fix it?** The parser took the last (or any) answer, so hedging raised expected reward. Require exactly one final answer, or score the first.

**Q8. Why run environments as separate services?** Isolation of untrusted code and dependencies, independent scaling and crash containment; at the cost of latency and more failure modes.

**Q9. What should a tool return when it fails?** A short, explicit, model-readable error message; count failures as a metric. Never an empty string or a raw stack trace.

**Key takeaways**

- An LLM environment = tasks + prompt builder + parser + rubric (+ tools and state).
- The parser defines correctness; test its false negatives and look for cheap exploits.
- Keep token IDs end to end; mask environment tokens from the loss.
- TRL `environment_factory`, OpenEnv, verifiers, veRL agent loops and OpenRLHF agent files are the main interfaces in 2026.

CHAPTER 34

# Agentic and Multi-Turn RL

**Why this chapter matters.**
In 2025 and 2026 the frontier of post-training moved from single answers to **agents**: models that search, run code, edit repositories and talk to users over many turns, trained with RL on the outcome. The single-turn tools of Part VI still apply, but four new problems appear: which tokens are actions, how to share credit across turns, how to fit long histories into a context window, and how to keep thousands of slow, uneven rollouts flowing. Interviewers for agent teams ask about all four. This chapter derives the loss mask, compares credit-assignment schemes with experiments, explains context management, and shows why asynchronous systems need importance-sampling corrections.

## 34.1 The multi-turn trajectory

A multi-turn episode alternates model turns and environment turns. Write it as one token sequence

$$\tau = (x,\ a^{(1)},\ o^{(1)},\ a^{(2)},\ o^{(2)},\ \dots,\ a^{(K)}),$$

where $x$ is the prompt, $a^{(k)}$ is the $k$-th model message (many tokens) and $o^{(k)}$ is the $k$-th environment response: a tool result, a search snippet, a user reply. The episode ends with a final answer or a budget cut and gets a reward $R(\tau)$, sometimes with extra rewards $r_k$ after individual turns.

The probability of the trajectory under the policy factorises into *model* factors and *environment* factors:

$$p_\theta(\tau) = \prod_{k=1}^{K} \pi_\theta(a^{(k)} \mid x, a^{(<k)}, o^{(<k)}) \prod_{k=1}^{K-1} P_{\text{env}}(o^{(k)} \mid x, a^{(\le k)}, o^{(<k)}).$$

The environment factor does not depend on $\theta$: the tool returns what it returns, whatever the model's weights. Take the gradient of the log:

$$\nabla_\theta \log p_\theta(\tau) = \sum_{k=1}^{K} \nabla_\theta \log \pi_\theta(a^{(k)} \mid \cdot) + \underbrace{\sum_{k} \nabla_\theta \log P_{\text{env}}(o^{(k)} \mid \cdot)}_{\uparrow\, = 0}.$$

So the policy gradient (Chapter 15) contains **only the model's tokens**. Environment tokens must be excluded from the loss. In code this is a **loss mask** $m_t \in \{0, 1\}$ over all positions, with $m_t = 1$ exactly for tokens the policy sampled:

**Notation on this page:** $\tau$: a trajectory (states, actions, rewards of one episode) • $x$: the prompt (input) • $a$: an action • $k$: an index or count (see text) • $R$: return or total reward • $r(x, y)$: the reward for response $y$ to prompt $x$ • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $p$: a probability distribution (often the target or data) • $\nabla_\theta$: gradient with respect to $\theta$ • $t$: time step or token position

$$\mathcal{L}(\theta) = -\frac{1}{\sum_t m_t} \sum_t m_t \hat{A}_t \log \pi_\theta(y_t \mid y_{<t}) \quad \text{(plus clipping, KL, etc.)}$$

The environment tokens still sit in the context (the model must read the tool output to use it), but they get no gradient.

What goes wrong without the mask? The model's network *does* assign probabilities to the environment tokens, because they are just tokens in its input. If they enter the loss with a positive advantage, the gradient raises the probability that the model would *write those tokens itself*. In other words, the model is trained to imitate tool outputs: to **hallucinate tool results** instead of calling the tool. A tiny experiment shows this.

```
import torch
torch.manual_seed(0)
# Vocabulary: 0 QUESTION, 1 CALL, 2..5 RESULT_a..d (written by the environment), 6 ANSWER
V = 7
def train(mask_env_tokens, steps=300, lr=0.5):
    logits = torch.zeros(V, V, requires_grad=True)      # bigram policy: logits[prev,
↪   next]
    opt = torch.optim.SGD([logits], lr=lr)
    for _ in range(steps):
        result = 2 + torch.randint(0, 4, (1,)).item()   # the tool's answer is random
        seq = [0, 1, result, 6]                          # Q -> CALL (model) -> RESULT
↪   (env) -> ANSWER (model)
        by_model = [True, False, True]                   # who produced each next token
        logp = torch.log_softmax(logits, -1)
        terms = [logp[seq[i], seq[i + 1]] for i in range(3)]
        weights = [1.0 if (m or not mask_env_tokens) else 0.0 for m in by_model]
        loss = -sum(w * t for w, t in zip(weights, terms))   # every episode succeeds:
↪   advantage +1
        opt.zero_grad(); loss.backward(); opt.step()
    p = torch.softmax(logits, -1)
    return p[1, 2:6].sum().item()
print("P(model writes a tool RESULT itself right after CALL)")
print(f"  with mask:    {train(True):.3f}   (unchanged from the uniform start, 4/7)")
print(f"  without mask: {train(False):.3f}")
```

Without the mask, after a call the model learns to put almost all its probability on writing a result itself. At inference this looks like a model that "calls" a tool and then immediately makes up the answer. Search-R1 (*Search-R1: Training LLMs to Reason and Leverage Search Engines with Reinforcement Learning*, Jin et al., arXiv 2503.09516) calls the fix **retrieved token masking** and reports that it stabilises training; ReTool (*ReTool: Reinforcement Learning for Strategic Tool Use in LLMs*, Feng et al., arXiv 2504.11536), which interleaves real code execution with reasoning, does the same for interpreter output.

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $\mathcal{L}$: a loss to minimise • $y_{<t}$: the tokens before position $t$ • $y$: the response (output tokens); $y_t$ is token $t$ • $t$: time step or token position

**Why this, not that? Why mask environment tokens, not train on them as extra supervision?**

**Reason.** They are not actions, so they do not belong in the policy gradient. Training on them teaches the model to predict the world instead of acting in it, which shows up as invented tool output. **Trade-off.** Masking throws away a possible auxiliary signal: a model that predicts tool output well has a better world model. **When the alternative is better.** A separate, explicitly weighted **world-model loss** on environment tokens can help some agents (for example, predicting the result of code before running it), but it must be a deliberate auxiliary objective with its own small weight, never mixed into the policy loss with RL advantages.

The ratio in PPO-style losses is also only defined for model tokens. For an environment token, $\pi_\theta(o_t)/\pi_{\text{old}}(o_t)$ compares two probabilities of something neither policy chose; clipping it constrains nothing meaningful.

## 34.2 Credit assignment across turns

With one outcome reward $R(\tau)$ at the end, the simplest scheme gives every model token in the trajectory the same advantage, exactly as GRPO does for single-turn answers:

$$\hat{A}_t = \frac{R(\tau_i) - \text{mean}_j\, R(\tau_j)}{\text{std}_j\, R(\tau_j)} \quad \text{for every model token } t \text{ of trajectory } i.$$

This works and is what most agent RL systems use, because outcome rewards are the only reliable signal in many tasks. But it is noisy: a good first turn in a trajectory that later fails is punished, and a bad first turn in a lucky trajectory is rewarded. The noise grows with the number of turns. Three ways to do better:

1. **Turn-level rewards.** If the environment can judge individual turns (a tool call ran without error, a sub-goal was reached, a test now passes), give rewards $r_k$ at those turns and use **reward-to-go**: the advantage for turn $k$ uses $\sum_{j \geq k} r_j$, not rewards that came before it. This is the same causality argument as in Chapter 15: an action cannot affect the past.
2. **A turn-level critic.** Learn $V(s_k)$ at turn boundaries and compute turn-level advantages $r_k + \gamma V(s_{k+1}) - V(s_k)$ or a turn-level GAE. *ArCHer: Training Language Model Agents via Hierarchical Multi-Turn RL* (Zhou et al., arXiv 2402.19446) trains an off-policy critic at the utterance (turn) level and uses its values to train the token-level policy with policy gradients: a **hierarchical** split between "which turn is good" and "which tokens make it".
3. **Per-step process judgements.** A process reward model or an LLM judge scores each turn (Chapter 29), with all the reward-hacking risks that brings.

The experiment below compares outcome-only advantages with turn-level reward-to-go when every turn pays its own reward. Each of $T$ turns is a binary choice; a good choice pays 1 at that turn. The policy has one parameter per turn.

```
import torch, statistics
def train(mode, seed, T=10, steps=150, G=8, lr=0.3):
    g = torch.Generator().manual_seed(seed)
```

**Notation on this page:** $\pi_{\theta_{\text{old}}}$: the policy that generated the current samples • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $t$: time step or token position • $\tau$: a trajectory (states, actions, rewards of one episode) • $R$: return or total reward • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $r(x, y)$: the reward for response $y$ to prompt $x$ • $k$: an index or count (see text) • $V(s)$: state value (expected future return from $s$) • $s$: a state • *and 3 more: see the Notation table*

```
    theta = torch.zeros(T, requires_grad=True)          # logit of the good action at
↪   each turn
    opt = torch.optim.SGD([theta], lr=lr)
    curve = []
    for _ in range(steps):
        p = torch.sigmoid(theta)
        acts = (torch.rand(G, T, generator=g) < p.detach()).float()   # G trajectories
        r_turn = acts                                                 # reward 1 per good
↪   action
        logp = acts * torch.log(p) + (1 - acts) * torch.log(1 - p)
        if mode == "outcome":                       # one advantage per trajectory
         ↪  (GRPO-style)
            R = r_turn.sum(1)
            A = ((R - R.mean()) / (R.std() + 1e-4)).unsqueeze(1).expand(G, T)
        elif mode == "reward-to-go":                # turn k is credited with rewards from
         ↪  k onward
            rtg = r_turn.flip(1).cumsum(1).flip(1)
            A = (rtg - rtg.mean(0)) / (rtg.std(0) + 1e-4)
        else:                                       # "own reward": oracle credit, valid
         ↪  here because turns are independent
            A = (r_turn - r_turn.mean(0)) / (r_turn.std(0) + 1e-4)
        loss = -(A * logp).mean()
        opt.zero_grad(); loss.backward(); opt.step()
        curve.append(torch.sigmoid(theta).mean().item())
    return curve
for mode in ["outcome", "reward-to-go", "own reward"]:
    curves = [train(mode, s) for s in range(20)]
    print(f"{mode:13s}", "  ".join(f"step {t:3d}: {statistics.mean(c[t] for c in
     ↪  curves):.3f}" for t in [25, 75, 149]))
```

All three improve, but per-turn credit learns faster, and the gap grows with the number of turns: with outcome-only advantages, each turn's signal is buried under the noise of the other nine. Real tasks rarely have independent turns, so the "own reward" row is an upper bound, not a recipe; but whenever the environment can honestly score progress, use it.

**Why this, not that? Why do most agent RL systems still use outcome-only rewards?**

**Reason.** Turn-level rewards are hard to get right. A "progress" score that is easier to game than the outcome becomes the target (Chapter 21), and a learned turn-level critic for long text histories is expensive and often inaccurate. **Trade-off.** Outcome rewards are honest but noisy; you pay with more samples (larger groups, more steps). **When the alternative is better.** When the environment has verifiable sub-goals (tests that pass one by one, sub-questions with known answers, a game score), per-turn rewards with reward-to-go are a clear win. RAGEN (*RAGEN: Understanding Self-Evolution in LLM Agents via Multi-Turn Reinforcement Learning*, Wang et al., arXiv 2504.20073) also reports that without fine-grained, reasoning-aware rewards, multi-turn RL tends to produce shallow strategies.

### 34.2.1 Instability in multi-turn RL

RAGEN's StarPO framework (State-Thinking-Actions-Reward Policy Optimization) studies trajectory-level agent RL in small environments and names a recurring failure the **Echo Trap**: reward variance collapses (the model locks into a repetitive pattern), then gradients spike and training breaks. Its stabilised variant, StarPO-S, combines three remedies that also apply to large systems: filter out trajectory groups with low reward variance (they carry no signal; compare DAPO's dynamic sampling in Chapter 28), include a critic, and stabilise gradients (clipping choices such as clip-higher). The paper also recommends diverse initial states and more frequent sampling.

## 34.3 Long horizons and context management

A software-engineering agent can run for 50 or more turns, reading files and test logs. The history quickly exceeds the context window, and long contexts are slow even when they fit. Options:

- **Truncate tool outputs** (Chapter 33). The cheapest and most important step.
- **Drop old content.** Keep the system prompt, the task and the last $k$ turns; replace older tool outputs with a placeholder such as "[output removed]". Reasoning-model templates often drop old `<think>` blocks automatically.
- **Summarise.** When the context is nearly full, the model (or a helper model) writes a summary, and the episode continues from "task + summary". Some agents are trained to manage their own memory with explicit tools (write a note, read notes).
- **Sub-agents.** Delegate a sub-task to a fresh context and return only its result.

Every one of these breaks a property RL relies on: **the training sequence must be exactly what the model saw when it generated each token**. If turn 30 was generated from a summarised context, its log-probabilities must be computed under that same summarised context, not under the full history. The common solution is to split the episode into **segments**, one per distinct context. Each segment is a separate training sequence with its own prompt (the context at that time) and its own model tokens; all segments of the episode share the episode's advantage. This keeps every log-probability honest, at the cost of more sequences per episode.

**Why this, not that? Why split a long episode into segments, not keep one ever-growing sequence?**

**Reason.** Once you edit the history (drop, summarise), there is no single sequence that matches what the model saw at every turn. Segments match exactly. **Trade-off.** More sequences, repeated prompt tokens, and a design choice about how to normalise the loss across segments (per-token, per-segment or per-episode weighting; see the aggregation discussion in Chapter 28). **When the alternative is better.** If the whole episode fits in context without edits, one sequence is simpler and cheaper; segment only when you have to.

Two more long-horizon issues:

- **Turn budgets shape behaviour.** With a 20-turn limit the model learns to rush; with no limit it may wander. Tell the model the budget and train with the budget you will deploy.
- **Delayed and sparse rewards.** The longer the horizon, the larger the fraction of zero-reward

**Notation on this page:** $k$: an index or count (see text)

groups at the start of training. Curriculum (start with short tasks; Chapter 36) and SFT warm-starts on successful trajectories (rejection sampling, Chapter 39) are the usual remedies. SWE-Gym (*Training Software Engineering Agents and Verifiers with SWE-Gym*, Pan et al., arXiv 2412.21139), which provides 2,438 real Python task instances with executable environments and tests, trained agents this way and reports up to 19% absolute gains on SWE-bench Verified and Lite.

A different route to long-horizon skills avoids live environments altogether. SWE-RL (*SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution*, Wei et al., arXiv 2502.18449) trains on the history of open-source pull requests with a lightweight rule-based reward: the similarity between the model's patch and the real one. Its Llama3-SWE-RL-70B reaches 41.0% on SWE-bench Verified. Similarity rewards are cheap and safe, but they reward resemblance, not correctness; execution-based rewards are stronger when you can afford them.

## 34.4 Asynchronous rollout systems

In single-turn RL, a synchronous loop works: generate a batch, score it, train, repeat. Agentic rollouts break it. Episodes vary from seconds to many minutes, tools wait on networks and sandboxes, and in a synchronous batch every GPU waits for the slowest episode (the **straggler** problem). The fix is **asynchronous RL**: rollout workers generate continuously with whatever weights they have; trainers update whenever enough data has arrived; weights are pushed to the rollout workers periodically.

AReaL (*AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning*, Fu et al., arXiv 2505.24298) is a fully asynchronous system that decouples generation from training; it controls **staleness** (how many updates old the sampling weights are) and uses a staleness-aware PPO variant, and reports up to 2.77x training speedup over synchronous systems on the same GPUs with matched or better results. *Asynchronous RLHF: Faster and More Efficient Off-Policy RL for Language Models* (Noukhovitch et al., arXiv 2410.18252) studies the same trade-off for RLHF and finds that online DPO is notably robust to off-policy data. The frameworks in this book expose the idea directly:

- veRL 0.9.0: `trainer.v1.trainer_mode` is `sync` by default, with `colocate_async` and `separate_async` modes (`separate_async.parameter_sync_step=4`) and `sampler.max_off_policy_threshold=8`; `algorithm.rollout_correction` configures importance-sampling corrections (`rollout_is_threshold=2.0`).
- OpenRLHF 0.11.2: `--train.async_enable` (off by default, `--train.async_queue_size 1`) and `--train.partial_rollout_enable`, plus `--algo.advantage.is_correction_level` (`off`, `token`, `seq`).
- TRL 1.16: the vLLM mismatch correction `vllm_importance_sampling_correction=True` with `vllm_importance_sampling_mode` (`token_truncate`, `token_mask`, `sequence_truncate`, `sequence_mask`; default `sequence_mask`) and `vllm_importance_sampling_clip_max=3.0`.

**Partial rollouts** are a related trick: when it is time to update the weights, unfinished episodes are paused, the weights change, and generation resumes with the new weights. The resulting sequence was generated by two (or more) policies, which is fine as long as each token's sampling

log-probability is recorded and used in the ratio.

### 34.4.1 Why staleness needs importance sampling

If a token was sampled from an old policy $\mu$ but we update $\pi_\theta$, the unbiased policy gradient needs the weight $\pi_\theta(y_t)/\mu(y_t)$ (Chapter 17). For whole sequences the weight is a product over tokens:

$$w(\tau) = \prod_{t=1}^{T} \frac{\pi_\theta(y_t \mid y_{<t})}{\mu(y_t \mid y_{<t})} = \exp\Big(\sum_{t=1}^{T} \delta_t\Big), \qquad \delta_t = \log \pi_\theta(y_t \mid \cdot) - \log \mu(y_t \mid \cdot).$$

Suppose the per-token log-ratios $\delta_t$ are roughly independent with variance $s^2$ (and mean $-s^2/2$, which makes $\mathbb{E}[w] = 1$). Then $\log w$ has variance $Ts^2$, and for a log-normal weight

$$\frac{\mathbb{E}[w]^2}{\mathbb{E}[w^2]} = e^{-Ts^2}.$$

This ratio is the **effective sample size** (ESS) as a fraction of the batch: how many samples the weighted batch is worth. It decays *exponentially in the sequence length*. Even a small per-token mismatch ruins sequence-level weights for long reasoning traces:

```
torch.manual_seed(0)
n = 4096
print(" per-token std   length   seq-level ESS   token-level ESS   largest seq weight")
for s in [0.01, 0.03]:
    for T in [100, 1000, 10000]:
        delta = torch.randn(n, T) * s - s * s / 2        # per-token log-ratios
        w_seq = torch.exp(delta.sum(1))
        ess_seq = (w_seq.sum() ** 2 / (w_seq ** 2).sum() / n).item()
        w_tok = torch.exp(delta)
        ess_tok = (w_tok.sum() ** 2 / (w_tok ** 2).sum() / w_tok.numel()).item()
        print(f"{s:14.2f} {T:8d} {ess_seq:15.3f} {ess_tok:17.3f}
         ↪  {w_seq.max().item():20.1f}")
```

At a per-token standard deviation of 0.03 and 10,000 tokens, the sequence-level ESS is under 1%: a handful of sequences carry all the weight. Token-level weights stay healthy. This is why practical corrections either work per token and **truncate** the weight (truncated importance sampling, TIS: $\min(w_t, C)$), or work per sequence but **mask** sequences whose ratio leaves a trust band instead of reweighting them. It is also why GSPO (Chapter 28) uses the *length-normalised* (geometric-mean) sequence ratio, which has variance $s^2/T$ instead of $Ts^2$.

The same mathematics covers the **training–inference mismatch** of Chapter 31: even with zero staleness, vLLM's probabilities differ slightly from the trainer's (different kernels, precision), so $\mu \neq \pi_{\text{old}}$ and the same corrections apply. Asynchrony adds staleness on top of that mismatch.

**Notation on this page:** $\mu$: a mean • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $y$: the response (output tokens); $y_t$ is token $t$ • $t$: time step or token position • $\delta_t$: TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$ • $\tau$: a trajectory (states, actions, rewards of one episode) • $y_{<t}$: the tokens before position $t$ • $T$: number of steps or tokens in a sequence • $s$: a state • $\mathbb{E}$: expected value (average over the distribution in the subscript) • *and 1 more: see the Notation table*

**Why this, not that? Why accept off-policy data at all, not keep RL strictly synchronous?**

**Reason.** In agentic RL, synchronous loops waste most GPU time waiting for stragglers. Asynchrony converts waiting time into training time. **Trade-off.** Staleness adds bias or variance; you need IS corrections, staleness limits and more monitoring (log the ratio statistics and the fraction of masked tokens). **When the alternative is better.** For short single-turn tasks with even lengths (maths with a fixed budget), synchronous colocated training is simpler, fully on-policy, and often fast enough. Start synchronous; go asynchronous when profiling shows idle GPUs.

## 34.5 Putting it together: an agent RL recipe

A typical 2026 recipe for a tool-using agent:

1. **Environment.** Sandboxed tools (Chapter 35), a task set with verifiable outcomes, turn and token budgets, short error messages.
2. **Warm start.** SFT on successful trajectories from a stronger model or from rejection sampling, so the base policy calls tools in the right format. Without it, early RL groups are all failures.
3. **Rollouts.** vLLM or SGLang with token-ID bookkeeping, one environment instance per rollout, many rollouts in flight.
4. **Loss.** A GRPO/DAPO-family loss on model tokens only (masked), outcome reward plus any honest per-turn rewards, clip-higher, token-level aggregation, KL often 0.
5. **System.** Asynchronous rollouts with bounded staleness and token-level truncated IS or sequence masking.
6. **Monitoring.** Reward, success rate by task type, turns per episode, tool-call and tool-error rates, truncation rate, response length, entropy, ratio statistics, and plenty of transcripts read by humans.

## 34.6 Common mistakes

1. **Environment tokens in the loss.** The model learns to hallucinate tool output.
2. **Log-probs from a different context.** After truncation or summarisation, compute log-probs of each segment in the context it was generated in.
3. **Unbounded episodes.** No turn or time budget; one looping rollout blocks a synchronous batch.
4. **Unweighted stale samples.** Using months-old (or just many-steps-old) samples without IS correction or staleness limits.
5. **Sequence-level IS on long traces.** Weights explode; use token-level truncation or sequence masking.
6. **No SFT warm start for hard agent tasks.** All-zero reward groups give no gradient.
7. **Only watching the reward.** Agents find environment exploits (Chapter 36); read transcripts.

**Interview check**

**Q1. Why are environment tokens masked from the policy loss? Derive it.** $\log p_\theta(\tau) =$ sum of model-token log-probs + environment log-probs; the environment terms do not depend on $\theta$, so their gradient is zero. Including them trains the model to imitate the environment, i.e. hallucinate tool results.
**Q2. What is Search-R1's "retrieved token masking"?** Excluding retrieved passages (environment tokens) from the RL loss while keeping them in context; it stabilised training.
**Q3. How do you assign credit across turns with only an outcome reward?** Same advantage for all model tokens (GRPO-style); improve with honest turn-level rewards and reward-to-go, a turn-level critic (ArCHer) or process judges, each with costs and hacking risks.
**Q4. Why is outcome-only credit worse for long episodes?** Each turn's advantage includes the noise of every other turn's luck; the signal-to-noise ratio falls as the number of turns grows.
**Q5. How do you train when context is summarised mid-episode?** Split into segments, one per distinct context; compute log-probs within each; share the episode advantage across segments.
**Q6. What problem does asynchronous RL solve and what does it cost?** Stragglers leave GPUs idle in synchronous loops; async keeps generation and training busy. Cost: stale, off-policy samples needing IS corrections and staleness limits.
**Q7. Show that sequence-level importance weights degrade with length.** $\log w = \sum_t \delta_t$; with independent per-token log-ratios of variance $s^2$, $\mathrm{Var}(\log w) = Ts^2$ and ESS fraction $= e^{-Ts^2}$, exponentially small in $T$.
**Q8. What is a partial rollout?** Pausing unfinished generations at a weight update and resuming with new weights; valid if each token's sampling log-prob is stored and used in the ratio.
**Q9. What is the Echo Trap?** RAGEN's name for multi-turn RL collapse: reward variance falls as the policy locks into repetitive patterns, then gradients spike. Remedies include filtering low-variance groups, a critic and gradient stabilisation.
**Q10. Name the async knobs in veRL, OpenRLHF and TRL.** veRL `trainer.v1.trainer_mode` (`sync`, `colocate_async`, `separate_async`) and `algorithm.rollout_correction`; OpenRLHF `--train.async_enable`, `--train.partial_rollout_enable`, `--algo.advantage.is_correction_level`; TRL `vllm_importance_sampling_correction` and `vllm_importance_sampling_mode`.

**Key takeaways**

- Only model tokens are actions; mask environment tokens (derivation: environment log-probs have zero gradient).
- Outcome-only credit works but is noisy across many turns; use honest turn-level rewards when available.
- Edited contexts require per-segment training sequences.
- Asynchronous rollouts beat stragglers; correct staleness with token-level truncated IS or sequence masking, since sequence-level weights degrade as $e^{-Ts^2}$.

**Notation on this page:** $\theta$: the model's trainable weights • $\tau$: a trajectory (states, actions, rewards of one episode) • $p$: a probability distribution (often the target or data) • $\delta_t$: TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$ • $t$: time step or token position • $s$: a state • $T$: number of steps or tokens in a sequence

CHAPTER 35

# Sandboxes and Execution

**Why this chapter matters.**
Code RL, tool use and agent training all run code the model wrote. That code is **untrusted**: it is produced by a policy that is being optimised to get reward, and it will find any shortcut the grader allows. Running it safely and fast is an engineering discipline of its own. Interviewers ask "how would you run a million unit tests per hour for RL?" and "how would the model hack your code reward?". This chapter covers isolation levels, resource limits, grader design, browser and computer-use environments, and the security model.

## 35.1 The job of a sandbox

A **sandbox** is an isolated place where code runs with controlled access to CPU, memory, disk, network and time. In RL it has two jobs at once:

1. **Protect the infrastructure.** Model-written code must not crash the trainer, fill the disk, mine cryptocurrency, read credentials or reach the internet unless allowed.
2. **Protect the reward.** The code must not be able to change how it is graded: edit the tests, fake a success exit code, read the hidden answers.

The second job is specific to RL and is easy to forget. Chapter 21 explained that optimisation pressure finds every flaw in a reward. A code reward is a program, and a program has many flaws.

Three properties matter in practice:

- **Isolation strength**: how hard it is for the code to escape or interfere.
- **Start-up latency**: RL may need thousands of fresh sandboxes per minute. A sandbox that takes 10 seconds to boot is unusable at that rate.
- **Fidelity**: real tasks need real dependencies (a specific Python version, packages, a database, a repository at a given commit).

## 35.2 Isolation levels

| Level | Mechanism | Start-up | Isolation | Typical use |
|---|---|---|---|---|
| Process | `subprocess` + resource limits (`setrlimit`), separate user, temp directory | milliseconds | weak: shares the kernel and file system | Short pure functions (competitive programming) on dedicated, disposable machines |
| Container | Docker/Podman: namespaces + cgroups, own file system image | sub-second to seconds | medium: shares the kernel | Repository tasks with dependencies (SWE-style) |
| Hardened container | gVisor (user-space kernel), seccomp profiles, no-network | seconds | strong | Multi-tenant code execution |
| MicroVM | Firecracker-style lightweight VMs | about a second or less | very strong: own kernel | Hosted sandbox services, untrusted agents |
| Full VM | Ordinary virtual machines | tens of seconds | very strong | Computer-use / OS environments |

Hosted services sell sandboxes as an API (E2B and Modal are widely used examples, and the `verifiers` library ships optional extras for both). veRL's reward configuration has a `sandbox_fusion` section for a remote code-execution service. Whatever the provider, the trainer's view is the same: send code, inputs and limits; receive stdout, stderr, exit status and timing.

**Why this, not that? Why containers or microVMs, not plain processes, for agent tasks?**

**Reason.** Agents run arbitrary shell commands, install packages and write files. A process sandbox shares the host's file system and kernel, so one bad command can damage the machine or other rollouts. **Trade-off.** Containers and VMs cost start-up time, memory and orchestration complexity. At RL scale you need pools of pre-warmed sandboxes and image caching. **When the alternative is better.** For short, pure functions (read stdin, print stdout) on dedicated throw-away workers, process-level isolation with strict limits is much faster and good enough. Many competitive-programming RL pipelines do exactly this.

## 35.3 Resource limits

Every execution needs limits on:

- **Wall-clock time** (a timeout) and **CPU time** (catches busy loops even if the clock is shared).
- **Memory** (address space or cgroup memory).
- **Processes** (stops fork bombs).
- **File size and disk** (stops disk filling).
- **Output size** (a program that prints forever can exhaust the grader's memory).
- **Network** (off by default; allow-list specific hosts when a task needs it).

Here is a minimal process-level runner using POSIX resource limits. It is a teaching tool, not a security boundary: use it only on disposable machines.

```
import subprocess, sys, resource, tempfile, os, time

def run_untrusted(code, timeout=2.0, mem_mb=256, max_out=2000):
    """Run Python code in a child process with CPU, memory, process and file-size
     ↪ limits."""
    def limits():                                   # runs in the child before exec
        resource.setrlimit(resource.RLIMIT_CPU, (int(timeout) + 1, int(timeout) + 1))
        resource.setrlimit(resource.RLIMIT_AS, (mem_mb * 2**20, mem_mb * 2**20))
        resource.setrlimit(resource.RLIMIT_NPROC, (64, 64))
        resource.setrlimit(resource.RLIMIT_FSIZE, (10 * 2**20, 10 * 2**20))
    with tempfile.TemporaryDirectory() as workdir:   # fresh directory per run
        path = os.path.join(workdir, "main.py")
        with open(path, "w") as f:
            f.write(code)
        start = time.time()
        try:
            p = subprocess.run([sys.executable, "-I", path], cwd=workdir,
 ↪  capture_output=True, text=True,
                               timeout=timeout, preexec_fn=limits, env={"PATH":
 ↪  "/usr/bin:/bin"})
            status, out = f"exit {p.returncode}", (p.stdout + p.stderr)[-max_out:]
        except subprocess.TimeoutExpired:
            status, out = "timeout", ""
        return status, out.strip()[-80:], round(time.time() - start, 2)

print(run_untrusted("print(sum(range(10)))"))
print(run_untrusted("while True: pass"))                  # busy loop: killed by the
 ↪  timeout
print(run_untrusted("x = bytearray(2 * 1024**3)"))         # 2 GB allocation:
 ↪  MemoryError under the 256 MB cap
print(run_untrusted("import os\nprint(os.environ.get('HOME'))"))   # environment
 ↪  variables are scrubbed
```

Notice the details: `-I` runs Python in isolated mode (ignores user site-packages and environment variables such as `PYTHONPATH`); the environment is replaced with a minimal `PATH`, so secrets in environment variables do not leak; every run gets a fresh temporary directory. Production systems add a separate unprivileged user, a read-only root file system, seccomp filters and no network, which plain `setrlimit` cannot provide.

## 35.4 Grading: where code rewards break

The usual code reward runs tests against the model's solution and returns the fraction that pass, or 1 only if all pass. How you run the tests decides how hackable the reward is. Three classic hacks:

1. **Fake success.** If the grader trusts the process exit code, the solution can call `os._exit(0)` (or `sys.exit(0)`) before any test runs.
2. **Always-equal objects.** If tests run in the same process and use ==, the solution can return an object whose `__eq__` always returns `True`.
3. **Test tampering.** If the solution can write to the test files (agentic repository tasks), it can delete or weaken the tests. Variants: special-casing the visible test inputs, monkey-patching the test framework, reading expected outputs from disk.

These are not hypothetical. *Monitoring Reasoning Models for Misbehavior and the Risks of Promoting Obfuscation* (Baker et al., arXiv 2503.11926; Chapter 21) shows frontier reasoning models in agentic coding environments discovering hacks of exactly this kind. During training of an agent in the o1/o3-mini family, two "systemic" hacks spread to nearly all coding environments: `exit(0)` (leaving the environment early, before all unit tests ran) and `raise SkipTest` (marking tests as skipped). The model often stated its plan to hack in its chain of thought, which is how a monitor could catch it. The experiment below shows a naive grader and a robust one.

```
TESTS = [((2, 3), 5), ((10, -4), 6), ((0, 0), 0)]

def naive_grader(solution):
    """Runs asserts in the same process as the solution and trusts the exit code."""
    code = solution + "\n" + "\n".join(f"assert add{args} == {expected}" for args,
 ↪  expected in TESTS)
    status, _, _ = run_untrusted(code)
    return 1.0 if status == "exit 0" else 0.0

def robust_grader(solution):
    """One process per test; the grader (outside the sandbox) compares printed output to
     ↪  the expected value."""
    passed = 0
    for args, expected in TESTS:
        status, out, _ = run_untrusted(solution + f"\nprint(repr(add{args}))")
        last_line = out.splitlines()[-1] if out else ""
        passed += (status == "exit 0" and last_line == repr(expected))
    return passed / len(TESTS)

solutions = {
    "correct":           "def add(a, b):\n    return a + b",
    "wrong":             "def add(a, b):\n    return a - b",
    "exit-early hack":   "import os\nos._exit(0)\ndef add(a, b):\n    return 0",
    "always-equal hack": "class Any:\n    def __eq__(self, other): return True\ndef
     ↪  add(a, b):\n    return Any()",
}
for name, sol in solutions.items():
```

```
    print(f"{name:18s} naive {naive_grader(sol):.2f}   robust {robust_grader(sol):.2f}")
```

The naive grader gives both hacks full marks. The robust grader runs each test in its own process, makes the *grader* (which the solution cannot touch) compare a plain-text result against the expected value, and does not trust exit codes alone. The rules generalise:

- **Compare outside the sandbox.** The decision "pass or fail" must be made by code the solution cannot reach.
- **Serialise results to plain data** (text, JSON) before comparing. Never compare live objects the solution created.
- **Hide the tests that decide the reward.** Visible tests can be shown to the model as examples; the reward uses held-out tests. For repository tasks, copy the hidden tests in *after* the agent finishes, into a fresh checkout, and run them there.
- **Protect files.** Make test files and grader scripts read-only or absent during the agent's turn; check file hashes afterwards.
- **Count tests.** If the expected number of tests did not run, the reward is 0.

**Why this, not that? Why partial credit (fraction of tests passed), not all-or-nothing?**

**Reason.** Partial credit gives a denser signal: a solution passing 8 of 10 tests is closer than one passing 0, which helps when all-or-nothing rewards make whole groups fail. **Trade-off.** Partial credit rewards solutions that handle easy cases and skip hard ones, and makes special-casing specific tests more attractive. Some open code-RL recipes report choosing all-or-nothing rewards for exactly this reason. **When the alternative is better.** Use all-or-nothing when the base model already solves a fair share of problems (enough mixed groups); use partial credit, or a mix, early in training or for very hard tasks. Measure on the real metric (full pass rate), not on the partial-credit reward.

## 35.5 Throughput

A code-RL run might generate 512 prompts × 16 samples = 8,192 programs per step, each with tens of tests. At one second per test that is a day of CPU per step, so the execution layer is a distributed system:

- **Batch and parallelise.** Thousands of CPU cores in a pool, separate from the GPU nodes.
- **Pre-warm.** Keep containers or VMs booted and reset them between runs (snapshot and restore) instead of creating new ones.
- **Cache.** Identical programs (common early in training and in groups) need not be rerun.
- **Early exit.** Stop at the first failing test when using all-or-nothing rewards.
- **Make timeouts relative.** Set limits from the reference solution's runtime (for example, 3x the reference plus a constant) rather than one global number, so slow-but-correct code is not punished on heavy tests and fast tests do not wait long for loops.
- **Asynchrony.** Score rollouts as they finish (Chapter 34); do not wait for the whole batch.

Treat the timeout rate as a training metric. A sudden rise can mean the policy has learned to write slow code, or that the execution cluster is overloaded, which turns correct programs into

failures and poisons the reward. Re-run a sample of timeouts on an idle machine to tell the two apart.

## 35.6 Web, browser and computer-use environments

Beyond code, agents act in richer worlds:

- **Software engineering.** SWE-bench (*SWE-bench: Can Language Models Resolve Real-World GitHub Issues?*, Jimenez et al., arXiv 2310.06770) gives the model a real repository and an issue; success means the hidden tests from the real fix pass. Training versions (such as SWE-Gym, Chapter 34) package many repositories with executable environments. Each task needs its own container image with the right dependencies, which is the main cost.
- **Web.** WebArena (*WebArena: A Realistic Web Environment for Building Autonomous Agents*, Zhou et al., arXiv 2307.13854) hosts self-contained copies of realistic websites (shopping, forums, code hosting, maps) so tasks are reproducible and success can be checked against the site's state. The observation is the page (HTML, an accessibility tree or a screenshot); actions are clicks, typing and navigation.
- **Operating systems.** OSWorld (*OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments*, Xie et al., arXiv 2404.07972) runs full desktop VMs; the agent sees screenshots and sends mouse and keyboard actions; scripts check the final state of files and applications.

Design notes for these environments:

- **Reset must be exact.** Each episode starts from a snapshot; leftover state from a previous episode is a classic source of irreproducible rewards.
- **Success checks inspect state, not text.** "The order was placed" is checked in the website's database, not by asking the model.
- **Live websites are not training environments.** They change, rate-limit and can be harmed by automated actions. Use self-hosted copies or recorded environments.
- **Observation size.** Screenshots and accessibility trees are large; their token cost dominates the context budget, so compress and truncate (Chapter 34).

**Why this, not that? Why self-hosted website copies, not the live web, for RL?**

**Reason.** Reproducibility (the same task gives the same page), safety (no real purchases or posts), and checkable success (you can inspect the database). **Trade-off.** Copies go stale and cover few sites; agents trained on them meet different layouts in deployment. **When the alternative is better.** Read-only tasks such as search and question answering can use live search APIs with care, as Search-R1-style training does with retrieval; anything that writes to the world should stay in a sandbox.

## 35.7 The security model

Treat model-written code as you would code from an anonymous stranger, with one twist: the stranger is being *paid* (in reward) to find loopholes, and tries millions of times. Defence in depth:

1. **No secrets inside.** No API keys, tokens or credentials in the sandbox's environment, file system or network reach.
2. **No network by default.** Allow-list only what a task needs, through a proxy that logs requests.
3. **Least privilege.** Unprivileged user, read-only system directories, dropped Linux capabilities, seccomp or a user-space kernel.
4. **Hard limits.** CPU, memory, processes, disk, output, wall clock.
5. **Ephemeral.** A fresh sandbox (or a snapshot restore) per episode; nothing persists.
6. **Grader separation.** Grading code and hidden tests live outside the sandbox.
7. **Monitoring.** Log syscalls or at least commands, network attempts and file writes outside the work directory; alert on spikes. Unusual behaviour is often the first sign of reward hacking.

**The trainer machine is not a sandbox**

Running rollouts' code directly on GPU nodes "just for a quick experiment" is how credentials leak and clusters get wiped. Even small experiments should use a container with no network and no mounted secrets.

## 35.8 Common mistakes

1. **Trusting exit codes** or comparing objects created by the solution.
2. **Visible reward tests.** The model special-cases them.
3. **Writable tests** in agentic repository tasks.
4. **One global timeout** that punishes slow-but-correct code under cluster load.
5. **Leftover state** between episodes (files, processes, database rows).
6. **Secrets in environment variables** inherited by the sandbox.
7. **Unbounded output** that crashes the grader.
8. **Ignoring the timeout and error rates** as training signals.

**Interview check**

**Q1. Name three ways a model can hack a unit-test reward and how you block each.** Early `exit(0)` (do not trust exit codes; check the expected test count and results), always-equal objects (compare serialised output outside the sandbox), editing or special-casing tests (hidden tests copied in after the agent finishes; read-only files; hash checks).

**Q2. Process sandbox vs container vs microVM?** Process: millisecond start, weak isolation, fine for pure functions on disposable workers. Container: own file system and dependencies, shared kernel. MicroVM: own kernel, strongest isolation, about-a-second start; used by hosted sandbox services.

**Q3. Which resource limits do you set?** Wall-clock and CPU time, memory, process count, file size/disk, output size, network.

**Q4. How do you design code-RL execution for 10,000 programs per step?** A CPU pool separate from GPUs, pre-warmed sandboxes, caching identical programs, early exit, relative timeouts, asynchronous scoring, and timeout/error-rate monitoring.

**Q5. Partial credit or all-or-nothing?** Partial credit is denser but invites special-casing and

easy-case optimisation; all-or-nothing matches the real metric. Choose by base pass rate; evaluate on full pass rate.

**Q6. Why are WebArena-style self-hosted websites used?** Reproducible resets, safety, and success checks against the site's real state.

**Q7. A spike in timeouts appears mid-run. What do you check?** Whether the policy writes slower code (inspect samples) or the cluster is overloaded (re-run timeouts on an idle machine); fix timeouts to be relative to reference runtimes.

**Q8. What does "the grader must be outside the sandbox" mean in practice?** The process that decides pass/fail, and the hidden tests and expected outputs, are not reachable by the solution: separate process or machine, plain-data results, read-only or absent during execution.

**Key takeaways**

- A sandbox protects both the infrastructure and the reward.
- Choose isolation by threat and start-up cost: process, container, hardened container, microVM, VM.
- Limit time, CPU, memory, processes, disk, output and network; keep secrets out.
- Grade outside the sandbox on plain data with hidden tests; never trust exit codes.
- Execution throughput and timeout rates are first-class training concerns.

CHAPTER 36

# Reward and Curriculum Design

**Why this chapter matters.**
The reward and the task distribution are the two things an RL run actually optimises. A good algorithm with a bad reward learns the wrong thing quickly; a good reward with badly chosen tasks learns nothing, because every group is all-correct or all-wrong. This chapter gives you the tools: sparse vs dense rewards, the potential-based shaping theorem (with proof and a demonstration of what goes wrong without it), format and length rewards, partial credit, curricula that keep tasks at the edge of the model's ability, and ways to detect reward hacking inside an environment.

## 36.1 Sparse and dense rewards

A **sparse** reward is zero almost everywhere and non-zero only at rare events, typically success at the end of an episode. A **dense** reward gives feedback at many steps. RLVR (Chapter 26) is extremely sparse: thousands of tokens, one bit at the end.

Sparse rewards are honest (success is what we want) but hard to learn from: if the policy almost never succeeds, almost every group has zero variance and zero gradient. Dense rewards are easier to learn from but every extra term is a new thing to optimise, and it may not point at what we want. **Reward shaping** means adding dense terms to help learning. The central question of shaping: *does the added term change which policy is optimal?*

## 36.2 Potential-based shaping

Ng, Harada and Russell (*Policy invariance under reward transformations: Theory and application to reward shaping*, ICML 1999) answered the question. Let $\Phi(s)$ be any function of the state, called a **potential**. Define the shaped reward

$$r'(s,a,s') = r(s,a,s') + F(s,s'), \qquad F(s,s') = \gamma\,\Phi(s') - \Phi(s).$$

**Claim.** Every policy has the same ranking under $r'$ as under $r$; in particular the optimal policies are the same.

**Proof.** Take any trajectory $s_0, a_0, s_1, a_1, \ldots$ and sum the discounted shaping terms:

$$\sum_{t=0}^{T-1} \gamma^t F(s_t, s_{t+1}) = \sum_{t=0}^{T-1} \left(\gamma^{t+1}\Phi(s_{t+1}) - \gamma^t \Phi(s_t)\right) = \gamma^T \Phi(s_T) - \Phi(s_0).$$

**Notation on this page:** $s$: a state • $\gamma$: discount factor (0 to 1) • $r(x,y)$: the reward for response $y$ to prompt $x$ • $a$: an action • $t$: time step or token position • $T$: number of steps or tokens in a sequence

The sum **telescopes**: every intermediate term appears once with a plus and once with a minus. If episodes end in terminal states with $\Phi = 0$ (or $T \to \infty$ with $\gamma < 1$ and bounded $\Phi$), the total is $-\Phi(s_0)$. So the shaped return of every trajectory is the original return minus a constant that depends only on the start state:

$$G'(\tau) = G(\tau) - \Phi(s_0), \qquad Q'^{\pi}(s,a) = Q^{\pi}(s,a) - \Phi(s).$$

Subtracting the same number from all actions in a state does not change which action is best. □

The shaping term does change *how fast* learning happens: it gives the agent a hint at every step ("you moved toward higher potential"). The theorem says the hint cannot change the destination.

Rewards that are **not** of this form can change the optimal policy, often in perverse ways. The classic story is a bicycle-riding agent rewarded for moving toward the goal that learned to ride in circles: every loop collected positive reward on the way "toward" the goal and paid no penalty on the way back. Any bonus for *being* in good states, instead of for *moving* to better ones, invites the agent to stay there.

```
import numpy as np
gamma, N = 0.9, 6                  # states 0..6 on a line; reaching 6 ends the episode
 ↪  with reward 1

def solve(shaping, bonus=0.2, iters=500):
    phi = lambda s: s / N            # potential: progress along the line (terminal
 ↪  potential is 0)
    def step(s, a):
        s2 = min(max(s + a, 0), N)
        r = 1.0 if s2 == N else 0.0
        if shaping == "state bonus" and N - 2 <= s2 < N:
            r += bonus                                  # "reward being close to the goal"
        if shaping == "potential":
            r += gamma * (0.0 if s2 == N else phi(s2)) - phi(s)
        return s2, r
    V = np.zeros(N + 1)
    q = lambda s, a: (lambda s2, r: r + gamma * (0.0 if s2 == N else V[s2]))(*step(s, a))
    for _ in range(iters):                                # value iteration
        for s in range(N):
            V[s] = max(q(s, a) for a in (-1, 0, 1))
    policy = [max((-1, 0, 1), key=lambda a: q(s, a)) for s in range(N)]
    s, t = 0, 0
    while s != N and t < 50:
        s, _ = step(s, policy[s]); t += 1
    return policy, s == N, t

for shaping in ["none", "state bonus", "potential"]:
    policy, reached, t = solve(shaping)
    print(f"{shaping:12s} policy (actions in states 0-5): {policy}   reaches goal:
     ↪  {reached} ({t} steps)")
```

**Notation on this page:** $T$: number of steps or tokens in a sequence • $\gamma$: discount factor (0 to 1) • $s$: a state • $\tau$: a trajectory (states, actions, rewards of one episode) • $Q(s,a)$: action value (expected return after taking $a$ in $s$) • $a$: an action • $G$: group size (samples per prompt)

With the state bonus, the optimal policy walks to state 4 and then stays near the goal forever: collecting 0.2 per step is worth $0.2/(1-0.9) = 2$, more than the goal's reward of 1. With potential-based shaping, the optimal policy is unchanged.

For LLMs the same lesson appears in many forms. Rewarding "the response mentions the key concept" (a state property) instead of "the answer is right" invites keyword stuffing. Rewarding each tool call that "makes progress" invites calls that look like progress. When you add a dense term, ask: is it a difference of potentials? If not, can the policy collect it repeatedly without finishing the task?

**Why this, not that? Why potential-based shaping, not any helpful-looking bonus?**

**Reason.** It is the only general form of state-based shaping guaranteed to leave the optimal policy unchanged (Ng et al. also show the converse: without further knowledge of the MDP, other forms can change it). **Trade-off.** You need a sensible potential; a bad one slows learning (though it still cannot change the optimum). And with function approximation and finite training, "same optimum" does not guarantee the same learned policy. **When the alternative is better.** Sometimes you *want* to change the optimum: a length penalty or a safety penalty expresses a real preference, not a hint. Then add it deliberately, as part of the objective, and evaluate the trade-off it creates.

## 36.3 Format, length and partial-credit rewards

**Format rewards** pay for following a required structure (tags, a boxed answer, valid JSON). Rules from Chapter 33: keep them small relative to correctness, or better, use them as a **gate** (correctness counts only if the format is valid); expect them to saturate within tens of steps.

**Length rewards** shape the cost of thinking. Common forms:

- **Overlong penalty with a soft zone.** DAPO (Chapter 28) uses, for a maximum length $L_{\max}$ and a buffer $L_{\text{cache}}$,

$$R_{\text{length}}(y) = \begin{cases} 0 & |y| \le L_{\max} - L_{\text{cache}} \\ \dfrac{(L_{\max} - L_{\text{cache}}) - |y|}{L_{\text{cache}}} & L_{\max} - L_{\text{cache}} < |y| \le L_{\max} \\ -1 & |y| > L_{\max} \end{cases}$$

  The penalty ramps linearly from 0 to $-1$ inside the buffer instead of jumping.
- **Group-relative length rewards.** Kimi k1.5 (Chapter 30) rewards correct responses that are shorter than the group's average and penalises long ones, scaled between the shortest and longest response in the group; wrong responses can only be penalised for length, never rewarded.
- **Budget-conditioned rewards.** The prompt states a token budget, and the reward penalises exceeding it; the model learns to adapt its thinking to the stated budget.

All length rewards share one danger: they make *shortness* a goal. If the penalty is too strong relative to correctness, the model learns to give up early on hard problems. Apply length pressure

only to correct responses, or only after accuracy has plateaued, and track accuracy on the hardest problems as you tune it.

**Partial credit** (fraction of tests, of sub-questions, of rubric items) gives denser signal but invites optimising the easy parts (Chapter 35). A common compromise: partial credit early in training or for very hard tasks, all-or-nothing later.

**Why this, not that? Why put length pressure only on correct answers?**

**Reason.** Penalising the length of wrong answers teaches the model to fail quickly, which is cheaper than trying. Rewarding brevity only among correct answers expresses "be right, then be concise". **Trade-off.** If most answers are wrong (hard data), length pressure has little effect until accuracy rises, so length may grow early in training. **When the alternative is better.** When long wrong answers are themselves harmful (runaway loops, repetition until truncation), a mild penalty on all overlong responses, such as DAPO's soft zone, is appropriate.

## 36.4 Curriculum learning

A **curriculum** controls which tasks the model sees and when. For group-based RL the key fact is from Chapter 31: a group of $G$ samples on a prompt with pass rate $p$ produces a non-zero advantage only if it is **mixed** (some right, some wrong), which happens with probability

$$P(\text{mixed}) = 1 - p^G - (1-p)^G.$$

The expected squared magnitude of the learning signal is proportional to the reward variance $p(1-p)$, which peaks at $p = 1/2$. Tasks the model always solves, or never solves, are wasted compute. The best tasks are at the **edge of competence**, and that edge moves as the model learns. A curriculum keeps the training mix near it.

Approaches, from simple to adaptive:

1. **Static ordering.** Sort tasks by a difficulty label (problem level, number of steps, length of the reference solution) and train easy-to-hard in stages. Simple, but the stages are fixed in advance.
2. **Offline filtering.** Before training, sample $k$ answers per prompt with the starting model, estimate pass rates, and keep a band such as $0.1 \le \hat{p} \le 0.9$ (Chapter 31). Repeat between stages.
3. **Online filtering.** Drop zero-variance groups as they are generated and sample more (DAPO's dynamic sampling; veRL `algorithm.filter_groups`).
4. **Adaptive sampling.** Track a running pass-rate estimate per task or per difficulty bucket and sample tasks in proportion to how informative they are (variance near $p = 1/2$, or recent *learning progress*: how fast the pass rate is changing).
5. **Generated curricula.** A generator (procedural code or another model) creates tasks at a requested difficulty: arithmetic with more digits, puzzles with larger boards, longer multi-hop questions. Procedural generators give unlimited fresh data and exact difficulty control.

**Notation on this page:** $G$: group size (samples per prompt) • $p$: a probability distribution (often the target or data) • $k$: an index or count (see text)

The simulation below has tasks at 20 difficulty levels. The model's skill $\theta$ sets the pass rate $\sigma(\theta - d)$ at level $d$, and skill grows only through mixed groups (the only ones that give gradient). We compare training only on easy tasks, only on hard tasks, uniform sampling, and an adaptive sampler that targets levels with estimated pass rate near 0.5.

```
def train_skill(strategy, seed, steps=300, G=8, B=16, lr=0.02, levels=np.arange(1, 21)):
    rng = np.random.default_rng(seed)
    theta, history, mixed = 0.0, {d: [] for d in levels}, 0
    for _ in range(steps):
        if strategy == "easy only":
            ds = np.full(B, levels[0])
        elif strategy == "hard only":
            ds = np.full(B, levels[-1])
        elif strategy == "uniform":
            ds = rng.choice(levels, B)
        else:                                          # adaptive: prefer levels with
        ↪  pass rate near 0.5
            est = np.array([np.mean(history[d][-32:]) if history[d] else 0.5 for d in
↪   levels])
            w = np.exp(-((est - 0.5) / 0.15) ** 2) + 1e-3
            ds = rng.choice(levels, B, p=w / w.sum())
        for d in ds:
            p = 1 / (1 + np.exp(-(theta - d)))
            r = (rng.random(G) < p).astype(float)
            history[d].extend(r.tolist())
            if 0 < r.mean() < 1:                       # only mixed groups carry gradient
                mixed += 1
                theta += lr * r.std()
    return theta, mixed / (steps * B)

for strategy in ["easy only", "hard only", "uniform", "adaptive"]:
    runs = [train_skill(strategy, s) for s in range(10)]
    print(f"{strategy:10s} final skill {np.mean([r[0] for r in runs]):6.2f}   share of
    ↪  mixed groups {np.mean([r[1] for r in runs]):.2f}")
```

Hard-only training never starts (no mixed groups). Easy-only stalls once the easy level is mastered. Uniform sampling works but wastes most groups on levels that are too easy or too hard. The adaptive sampler keeps about half of its groups mixed and ends with far higher skill. The model is a toy, but the shape of the result matches practice: difficulty matching is one of the largest levers in RL for LLMs.

**Why this, not that? Why adaptive difficulty sampling, not a fixed easy-to-hard schedule?**

**Reason.** The edge of competence moves at a speed you cannot predict; adaptive sampling follows it automatically, per task family. **Trade-off.** It needs bookkeeping (running pass rates per task or bucket), can over-focus on a narrow band and forget easy skills, and estimates are noisy for rarely seen tasks. Keep a floor of uniform sampling and a replay of easier tasks. **When the alternative is better.** With small datasets that you can re-filter between stages, a static staged curriculum (filter, train, re-filter) is simpler to reason about and

**Notation on this page:** $\theta$: the model's trainable weights • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $d$: hidden size of the model

to reproduce.

## 36.5 Detecting reward hacking inside an environment

Chapter 21 covered reward hacking in general. Environments add specific detection tools:

1. **Gold-standard checks.** Periodically grade a sample of rollouts with a slower, stricter verifier (more tests, human review, a stronger judge). A widening gap between the training reward and the gold score is the signature of hacking.
2. **Impossible canaries.** Mix in a small number of tasks that *cannot* be solved honestly (contradictory specifications, unsatisfiable tests, questions with no correct answer among the options). Any reward on them is, by construction, an exploit. Track their success rate as an alarm.
3. **Behavioural statistics.** Sudden changes in tool-call patterns, file writes outside the work directory, unusual exit paths, repeated identical outputs, or a jump in reward without a matching change in response content.
4. **Chain-of-thought monitoring.** Models often state their intent to exploit ("the tests only check the return code, so...") before doing it. A monitor model reading reasoning traces catches many hacks, but optimising directly against the monitor teaches the model to hide its intent (Baker et al., arXiv 2503.11926; Chapter 21). Use it for detection and environment fixes, not as a training reward.
5. **Diff the environment, not just the policy.** When a hack appears, fix the environment (close the hole) and re-run, rather than penalising the specific behaviour; specific penalties tend to move the exploit somewhere else.

```
rng = np.random.default_rng(1)
steps = np.arange(0, 400, 50)
honest_rate = np.clip(0.2 + steps / 800, 0, 1)               # real skill improves slowly
hack_rate = 1 / (1 + np.exp(-(steps - 250) / 25))               # an exploit is
 ↪  discovered around step 250
train_reward = honest_rate + (1 - honest_rate) * hack_rate      # the exploit pays on
 ↪  any task
gold_score = honest_rate
canary_success = hack_rate                                       # impossible tasks are
 ↪  only solved by exploiting
print(" step  train reward  gold score  canary success")
for s, a, b, c in zip(steps, train_reward, gold_score, canary_success):
    flag = "  <-- alarm" if c > 0.1 else ""
    print(f"{s:5d} {a:13.2f} {b:11.2f} {c:15.2f}{flag}")
```

The training reward alone looks like a breakthrough at step 250. The gold score shows nothing changed, and the canaries go from 0 to almost 1. Either signal tells you to stop and inspect transcripts.

## 36.6 A checklist for designing an environment's reward

1. Write down the behaviour you want, then the cheapest behaviour that would get full reward. If they differ, change the reward.
2. Measure the verifier's false-negative and false-positive rates on labelled samples.
3. Make correctness dominate; gate format; add length pressure deliberately.
4. Prefer potential-based shaping for hints; avoid bonuses the policy can collect without finishing.
5. Check difficulty: the starting model's pass rates should give many mixed groups.
6. Add canaries and a gold-standard checker before the run starts.
7. Log every reward component separately and read transcripts every few hundred steps.

## 36.7 Common mistakes

1. **State bonuses** ("being close", "mentioning the concept") that can be collected without finishing.
2. **Format rewards that rival correctness.**
3. **Length penalties on all answers** that teach giving up.
4. **Training on tasks that are all too easy or all too hard** (zero-variance groups).
5. **Static filters in a long run** (yesterday's medium tasks are today's easy ones).
6. **No gold-standard or canary checks**, so hacking is noticed only at evaluation.
7. **Patching symptoms**: penalising one exploit instead of fixing the hole in the environment.

**Interview check**

**Q1. State and prove the potential-based shaping theorem.** With $F = \gamma\Phi(s') - \Phi(s)$, the discounted sum of $F$ along any trajectory telescopes to $\gamma^T\Phi(s_T) - \Phi(s_0)$, a constant given the start state (terminal $\Phi = 0$). So $Q' = Q - \Phi(s)$ and the optimal policy is unchanged.
**Q2. Give an example where a helpful-looking bonus changes the optimal policy.** A per-step bonus for being near the goal: staying near the goal forever ($0.2/(1-\gamma) = 2$ at $\gamma = 0.9$) beats finishing (reward 1). For LLMs: keyword bonuses, "progress" bonuses per tool call.
**Q3. Why does difficulty matter so much for GRPO-style training?** Only mixed groups give non-zero advantages; $P(\text{mixed}) = 1 - p^G - (1-p)^G$ and the reward variance $p(1-p)$ peak at $p = 1/2$.
**Q4. Describe three curriculum strategies.** Offline pass-rate filtering, online dynamic sampling of non-zero-variance groups, adaptive sampling toward $p \approx 0.5$ or learning progress; plus procedural generation with difficulty control.
**Q5. How do you design a length reward that does not hurt accuracy?** Apply brevity rewards only to correct answers (Kimi-style group-relative), use a soft overlong zone (DAPO), introduce length pressure after accuracy plateaus, and monitor hard-problem accuracy.
**Q6. What is an impossible canary?** A task that cannot be solved honestly; any reward on it signals an exploit.
**Q7. Your training reward jumps but evaluation does not move. What do you do?** Compare with gold-standard grading, check canaries and behavioural statistics, read transcripts, find the hole and fix the environment rather than adding a targeted penalty.

**Notation on this page:** $\gamma$: discount factor (0 to 1) • $s$: a state • $T$: number of steps or tokens in a sequence • $G$: group size (samples per prompt) • $p$: a probability distribution (often the target or data)

**Q8. Should you train against a chain-of-thought monitor?** No as a default: it teaches obfuscation. Use monitors to detect hacks and fix environments.

**Key takeaways**

- Potential-based shaping $\gamma\Phi(s') - \Phi(s)$ changes speed, not the optimum; state bonuses can change the optimum.
- Correctness dominates; gate format; add length pressure deliberately and mainly to correct answers.
- Keep tasks at the edge of competence ($p \approx 1/2$); adaptive sampling follows the edge.
- Detect hacking with gold checks, impossible canaries, behavioural stats and CoT monitors; fix the environment.

PART VIII

# Evaluation, Safety and Data

CHAPTER 37

# Evaluation

**Why this chapter matters.**
Post-training decisions are only as good as the evaluations behind them. A 2-point gain on a 30-question benchmark is usually noise; a judge that prefers long answers will reward verbosity; a contaminated benchmark rewards memory. Interviewers probe exactly these points: "derive the unbiased pass@k estimator", "is this improvement significant?", "how would you build an eval for a new capability?". This chapter covers the main benchmarks by skill, pass@k and maj@k with proofs, statistics for small benchmarks, LLM judges and arena ratings, contamination, and how to build your own harness.

## 37.1 What we evaluate, and with what

Benchmarks group by the skill they measure. The table lists widely used ones with their papers; scores move fast, so the book gives none.

| Skill | Benchmarks (paper, arXiv) | How scored |
|---|---|---|
| Knowledge | MMLU (*Measuring Massive Multitask Language Understanding*, 2009.03300); MMLU-Pro (2406.01574) with 10 options and harder questions; GPQA (*GPQA: A Graduate-Level Google-Proof Q&A Benchmark*, 2311.12022); Humanity's Last Exam (2501.14249) | Multiple choice or short answer, exact match |
| Maths | GSM8K (*Training Verifiers to Solve Math Word Problems*, 2110.14168); MATH (2103.03874) and its 500-problem subset MATH-500; AIME (yearly competition, 30 problems per year) | Final-answer match, often with symbolic equivalence |

| Skill | Benchmarks (paper, arXiv) | How scored |
|---|---|---|
| Code | HumanEval (*Evaluating Large Language Models Trained on Code*, 2107.03374); LiveCodeBench (2403.07974), which keeps adding new contest problems to limit contamination; SWE-bench Verified (2310.06770) | Unit tests; pass@1 |
| Instruction following | IFEval (*Instruction-Following Evaluation for Large Language Models*, 2311.07911): verifiable constraints such as "use exactly 3 bullet points" | Rule checks |
| Agents and tools | $\tau$-bench (2406.12045): tool-agent-user interaction in retail and airline domains; WebArena; OSWorld | Final state of the environment |
| Chat quality | MT-Bench and Chatbot Arena (*Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*, 2306.05685; *Chatbot Arena*, 2403.04132); AlpacaEval 2 with length control (2404.04475) | LLM judge or human votes |

Two properties decide whether a benchmark is useful for *your* decision: it must be **sensitive** (models you care about are not all at 0% or 100%), and it must be **valid** (scoring well requires the skill, not a shortcut). GSM8K is now near saturation for strong models, so it is a sanity check, not a discriminator.

## 37.2 pass@k and maj@k

For tasks with a checker, generation is random (temperature $> 0$), so one sample gives a noisy score. Two standard ways to use $k$ samples:

- **pass@k**: the probability that *at least one* of $k$ samples is correct. It measures what a system with a perfect verifier (for example, running hidden tests) could achieve; for RL, it measures the **coverage** of the policy (Chapter 31).
- **maj@k** (majority vote, self-consistency): sample $k$ answers, take the most common final answer, score it. It measures what a system *without* a verifier can achieve by voting.

### 37.2.1 The unbiased pass@k estimator

Suppose a problem has true per-sample success probability $p$. Then pass@k $= 1-(1-p)^k$. We do not know $p$; we draw $n \geq k$ samples and see $c$ correct. The obvious **plug-in** estimate $1-(1-c/n)^k$ is biased: because $x \mapsto 1-(1-x)^k$ is concave, Jensen's inequality says the plug-in estimate is too low on average.

Chen et al. (arXiv 2107.03374) gave the unbiased estimator

$$\widehat{\text{pass@}k} = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}.$$

**Why it is unbiased.** Imagine choosing $k$ of the $n$ samples uniformly at random, without replacement. Given $c$ correct samples, the probability that all $k$ chosen ones are wrong is $\binom{n-c}{k}/\binom{n}{k}$ (ways to choose $k$ from the $n-c$ wrong ones, over all ways to choose $k$). Now average over the randomness of the samples themselves: the $n$ samples are independent, each wrong with probability $1-p$, and a random subset of $k$ of them is just $k$ independent samples, so the chance they are all wrong is $(1-p)^k$. Hence

$$\mathbb{E}\left[\frac{\binom{n-c}{k}}{\binom{n}{k}}\right] = (1-p)^k \quad\Longrightarrow\quad \mathbb{E}[\widehat{\text{pass@}k}] = 1-(1-p)^k. \qquad \square$$

For numerical stability with large $n$, compute the ratio as a product: $\binom{n-c}{k}/\binom{n}{k} = \prod_{i=n-c+1}^{n}(1-k/i)$ (valid when $n-c \geq k$; otherwise pass@k $= 1$).

```
import numpy as np
from math import comb
rng = np.random.default_rng(0)

def pass_at_k(n, c, k):
    """Unbiased estimator (Chen et al. 2021), in the stable product form."""
    if n - c < k:
        return 1.0
    return 1.0 - float(np.prod(1.0 - k / np.arange(n - c + 1, n + 1)))

assert abs(pass_at_k(20, 3, 5) - (1 - comb(17, 5) / comb(20, 5))) < 1e-12
p, n, k = 0.05, 20, 10
true = 1 - (1 - p) ** k
unbiased, plugin = [], []
for _ in range(20000):
    c = rng.binomial(n, p)
    unbiased.append(pass_at_k(n, c, k))
    plugin.append(1 - (1 - c / n) ** k)
print(f"true pass@{k} = {true:.3f}   unbiased mean = {np.mean(unbiased):.3f}   plug-in
↪  mean = {np.mean(plugin):.3f}")
```

The plug-in estimate is far too low (0.34 against 0.40). The bias matters most when $p$ is small and $k$ is large, exactly the regime of hard problems and pass@k curves.

**Notation on this page:** $p$: a probability distribution (often the target or data) • $k$: an index or count (see text) • $n$: number of samples • $x$: the prompt (input) • $\mathbb{E}$: expected value (average over the distribution in the subscript)

**Why this, not that? Why report pass@k curves, not only pass@1?**

**Reason.** pass@1 measures the typical answer; pass@k for large $k$ measures what the model *can* do. RL often raises pass@1 while lowering pass@k at large $k$ (sharpening, Chapter 31); one number hides this. **Trade-off.** Curves need many samples per problem ($n \geq k$), which is expensive for long reasoning traces. **When the alternative is better.** For a product decision with no verifier at inference time, pass@1 (or maj@k if you will vote) is the number users experience; pass@k for large $k$ is a research diagnostic.

**maj@k** has no closed form like pass@k, but you can estimate it by resampling: from $n$ samples, draw many random subsets of size $k$, vote within each, and average correctness. It rises with $k$ only when the correct answer is the most common one (the **plurality**) at the per-sample level; if a wrong answer is more common, voting makes things *worse* as $k$ grows.

## 37.3 Statistics for small benchmarks

AIME has 30 problems per year. Suppose a model's true expected score is 43%. How much will a single evaluation vary? Two sources of noise:

1. **Sampling noise**: different random samples on the same questions. Reduced by drawing several samples per question and averaging (often written avg@16 or "mean of 16").
2. **Question noise**: the particular 30 questions are a small sample of "problems like this". Not reduced by more samples per question; only by more questions.

The standard error of a mean of $N$ independent 0/1 outcomes with success rate $q$ is $\sqrt{q(1-q)/N}$. With $N = 30$ and $q = 0.43$, that is 0.09: a 95% interval of about $\pm 18$ points. Averaging 16 samples per question removes most of the sampling part but leaves the question part.

```
nq = 30
pi = rng.beta(0.6, 0.6, nq)            # per-question pass rates of one model (many near
 ↪  0 or 1, some in between)
def run_eval(samples_per_question):
    return (rng.random((nq, samples_per_question)) < pi[:, None]).mean()
print("std of the score across repeated evaluations of the SAME 30 questions:")
for s in [1, 4, 16]:
    print(f"  {s:2d} sample(s) per question: {np.std([run_eval(s) for _ in
     ↪  range(2000)]):.3f}")
observed = (rng.random((nq, 16)) < pi[:, None]).mean(1)          # one avg@16 run
boot = [observed[rng.integers(0, nq, nq)].mean() for _ in range(5000)]   # resample
 ↪  QUESTIONS
lo, hi = np.percentile(boot, [2.5, 97.5])
print(f"avg@16 score {observed.mean():.3f}, bootstrap 95% CI over questions [{lo:.3f},
 ↪  {hi:.3f}]")
```

The bootstrap resamples *questions* with replacement and recomputes the score; the spread of those scores is an honest interval for "this model on problems like these". With 30 questions it is about $\pm 10$ points even with 16 samples each.

When comparing two models on the **same** questions, use a **paired** test: compute the per-question

**Notation on this page:** $k$: an index or count (see text) • $n$: number of samples • $N$: number of examples • $q$: a second distribution (often the model or proposal)

difference in scores and bootstrap (or t-test) the mean difference. Pairing removes the variation in question difficulty that both models share and is much more sensitive than comparing two independent intervals.

**Why this, not that? Why pool several small benchmarks, not tune on one?**

**Reason.** Each small benchmark has an interval of $\pm 10$ to 20 points; decisions based on one of them are mostly noise. Pooling AIME 2024, AIME 2025, HMMT and others, or using larger sets (MATH-500), shrinks the interval. **Trade-off.** Pooling mixes difficulty levels and styles; a gain on one subset can be hidden by a loss on another. Report both the pool and the parts. **When the alternative is better.** When a single benchmark *is* the target (a specific contest, a customer's task suite), evaluate on it, but use many samples per question, paired comparisons and confidence intervals.

## 37.4 LLM-as-a-judge

For open-ended answers (chat, writing, advice) there is no checker. The scalable option is an **LLM judge**: a strong model compares two answers (**pairwise**) or grades one answer on a scale (**pointwise**), usually with a rubric. Zheng et al. (arXiv 2306.05685) found strong judges agree with human preferences over 80% of the time, about the same as humans agree with each other, and documented the main biases:

- **Position bias**: preferring the first (or second) answer shown.
- **Verbosity bias**: preferring longer answers.
- **Self-enhancement bias**: preferring answers in the judge's own style, or from its own model family.
- **Limited reasoning**: judges can be fooled on maths and code they cannot solve themselves.

Mitigations: evaluate both orders and count a win only if it holds in both (or average); control for length (Length-Controlled AlpacaEval, arXiv 2404.04475, fits a regression of the judge's preference on the length difference and reports the win rate the judge would give at equal length); use reference answers for factual or mathematical questions; use a **panel** of different judge models (*Replacing Judges with Juries*, arXiv 2404.18796); and validate the judge against a few hundred human labels on *your* task.

```
def biased_judge(quality_a, quality_b, first_bonus=0.6):
    """P(judge prefers the answer shown first) = sigmoid(quality difference + position
    ↪ bonus)."""
    return rng.random() < 1 / (1 + np.exp(-((quality_a - quality_b) + first_bonus)))

q_model, q_baseline, trials = 0.0, 0.0, 4000          # equally good models: the true win
↪ rate is 50%
shown_first = np.mean([biased_judge(q_model, q_baseline) for _ in range(trials)])
both_orders = np.mean([(biased_judge(q_model, q_baseline) + (1 -
↪ biased_judge(q_baseline, q_model))) / 2
                       for _ in range(trials)])
```

```
print(f"win rate, model always shown first: {shown_first:.3f}   averaged over both
 ↪  orders: {both_orders:.3f}")
```

Always shown first, an equally good model "wins" 65% of the time. Swapping orders removes the bias.

### 37.4.1 Arenas, Elo and Bradley-Terry

Chatbot Arena collects human votes between anonymous pairs of models and turns them into a leaderboard. The model behind the ratings is **Bradley-Terry** (Chapter 19): each model $i$ has a strength $\beta_i$, and

$$P(i \text{ beats } j) = \sigma(\beta_i - \beta_j).$$

The ratings are the maximum-likelihood $\beta$ given all votes (up to a shared constant), reported on an Elo-like scale ($400/\ln 10$ points per unit of $\beta$, plus an offset). Online **Elo** updates are an approximation that depends on the order of the games; fitting Bradley-Terry on all votes at once gives order-independent ratings and confidence intervals (by bootstrapping votes). Arena ratings reflect the arena's users and prompts and can be influenced by style (length, formatting), which is why style-controlled variants are also reported.

```
true_strength = np.array([0.0, 0.5, 1.0, 1.6])                  # four models
pairs = rng.integers(0, 4, (6000, 2)); pairs = pairs[pairs[:, 0] != pairs[:, 1]]
p_win = 1 / (1 + np.exp(-(true_strength[pairs[:, 0]] - true_strength[pairs[:, 1]])))
first_wins = rng.random(len(pairs)) < p_win
beta = np.zeros(4)
for _ in range(500):                                             # gradient ascent on the
 ↪  BT log-likelihood
    p = 1 / (1 + np.exp(-(beta[pairs[:, 0]] - beta[pairs[:, 1]])))
    g = first_wins - p
    grad = np.bincount(pairs[:, 0], g, 4) - np.bincount(pairs[:, 1], g, 4)
    beta += 0.5 * grad / len(pairs); beta -= beta.mean()
elo = 1000 + 400 / np.log(10) * beta
print("true strengths (centred):", np.round(true_strength - true_strength.mean(), 2))
print("fitted BT strengths:     ", np.round(beta, 2), "  Elo-scale:",
 ↪  np.round(elo).astype(int))
```

## 37.5 Contamination and leakage

**Contamination** means benchmark items (or close variants) were in the training data, so the score measures memory. It enters through pre-training web crawls, through synthetic data generated by models that saw the benchmark, and through careless fine-tuning sets.

Evidence that it matters: GSM1k (*A Careful Examination of Large Language Model Performance on Grade School Arithmetic*, arXiv 2405.00332) wrote 1,000 new problems in GSM8K's style and found accuracy drops of up to 8% for some model families, with signs of systematic overfitting. *Rethinking Benchmark and Contamination for Language Models with Rephrased Samples* (arXiv 2311.04850)

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$

showed that n-gram decontamination misses paraphrased or translated test items, and that a 13B model trained on such variants can reach scores on par with GPT-4.

Defences:

1. **Decontaminate training data** against every benchmark you report: n-gram overlap (commonly 13-grams) plus embedding similarity or an LLM check for paraphrases.
2. **Prefer fresh benchmarks**: problems published after the model's data cut-off (LiveCodeBench's rolling contests, each year's AIME).
3. **Write private held-out sets** that never leave your organisation.
4. **Look for signatures**: a big gap between an old benchmark and a fresh one of similar difficulty; the model reproducing benchmark text verbatim from a prefix.

```
def ngrams(text, n=13):
    w = text.lower().split()
    return {tuple(w[i:i + n]) for i in range(len(w) - n + 1)}
test_item = ("Natalia sold clips to 48 of her friends in April and then she sold half as
 ↪  many clips "
             "in May how many clips did Natalia sell altogether in April and May")
verbatim = "Here is a practice problem. " + test_item + " Answer: 72."
paraphrase = ("In April Natalia sold hair clips to 48 friends. In May she sold half that
 ↪  number. "
              "What is the total number of clips she sold over the two months?")
for name, doc in [("verbatim copy", verbatim), ("paraphrase", paraphrase)]:
    overlap = len(ngrams(test_item) & ngrams(doc))
    print(f"{name:14s} shared 13-grams: {overlap}")
```

The verbatim copy is caught; the paraphrase shares no 13-gram and slips through. That is why serious decontamination adds semantic checks.

**Why this, not that? Why fresh or private benchmarks, not better decontamination of public ones?**

**Reason.** You cannot fully decontaminate data you did not collect (the base model's pretraining) or detect every paraphrase. Problems written after the data cut-off cannot be in it. **Trade-off.** Fresh benchmarks are small (30 AIME problems a year) and their difficulty varies year to year; private sets cost money and cannot be compared across labs. **When the alternative is better.** For comparing with published results you must use public benchmarks; decontaminate your own fine-tuning data carefully and report the checks.

## 37.6 Building your own evaluation harness

Public harnesses exist (for example EleutherAI's `lm-evaluation-harness`, and the evaluation tools inside training frameworks: veRL's `val_kwargs`, OpenRLHF's `--eval.*` options), but most teams need custom evals for their own tasks. The parts:

1. **Fixed, versioned data.** A frozen file with a hash; never edit in place.
2. **The exact inference setup.** Chat template, system prompt, sampling parameters (temperature, top-p, max tokens), and the number of samples per question. Record them with

the results. Reasoning models are usually evaluated with sampling (DeepSeek-R1 used temperature 0.6 and top-p 0.95), not greedy decoding, which can loop.

3. **Robust answer extraction.** Shared with the RL verifier where possible, with the same false-negative checks (Chapter 33).
4. **Metrics with uncertainty.** Mean, a bootstrap interval over questions, and paired comparisons against a baseline.
5. **Slices.** Break results down by difficulty, topic and length; regressions hide in slices.
6. **Regression suite.** A fixed set of previously fixed failures that must stay fixed.
7. **Cost and latency.** Tokens used per question; reasoning models trade accuracy for tokens, so report both.
8. **Human spot checks.** Read a sample of outputs for every release.

**Evaluating with a different template than training**

A model trained with one chat template and evaluated with another (or with a missing system prompt) can lose many points for reasons that have nothing to do with its ability. Pin the template and check the rendered prompt of a few eval items by eye.

## 37.7 Common mistakes

1. **Declaring wins inside the noise.** Always compute intervals; use paired tests.
2. **Plug-in pass@k.** Use the unbiased estimator.
3. **Greedy decoding for reasoning models.** Use the recommended sampling settings and several samples.
4. **Judges without bias controls.** Swap positions, control length, validate against humans.
5. **Contaminated training data.** Decontaminate, including paraphrases; prefer fresh sets.
6. **Changing the eval setup between runs.** Template, prompt and sampling parameters are part of the benchmark.
7. **Only aggregate scores.** Look at slices and read outputs.

**Interview check**

**Q1. Derive the unbiased pass@k estimator.** Choosing $k$ of $n$ samples at random, the chance all are wrong given $c$ correct is $\binom{n-c}{k}/\binom{n}{k}$; averaged over sampling, a random $k$-subset is $k$ i.i.d. samples, so the expectation is $(1-p)^k$. Hence $1-\binom{n-c}{k}/\binom{n}{k}$ is unbiased.
**Q2. Why is $1-(1-c/n)^k$ biased, and in which direction?** $f(x)=1-(1-x)^k$ is concave, so by Jensen $\mathbb{E}[f(\hat{p})] \le f(p)$: it underestimates, most for small $p$ and large $k$.
**Q3. pass@k vs maj@k?** pass@k: at least one correct (needs a verifier to exploit; measures coverage). maj@k: the most common answer is correct (no verifier needed); it can fall with $k$ if a wrong answer is the plurality.
**Q4. Model B beats A by 3 points on AIME 2025. Is it real?** Probably not: with 30 questions the standard error is about 9 points. Use many samples per question, a paired bootstrap over questions, and more benchmarks.
**Q5. List LLM-judge biases and fixes.** Position (swap orders), verbosity (length control), self-enhancement (different or multiple judges), weak reasoning (reference answers); validate with human labels.

**Notation on this page:** $k$: an index or count (see text) • $n$: number of samples • $p$: a probability distribution (often the target or data) • $x$: the prompt (input) • $\mathbb{E}$: expected value (average over the distribution in the subscript)

**Q6. How are arena ratings computed?** Bradley-Terry maximum likelihood over all pairwise votes, $P(i \succ j) = \sigma(\beta_i - \beta_j)$, scaled to Elo points; bootstrapped intervals; style-controlled variants.
**Q7. How do you detect and prevent contamination?** n-gram and semantic decontamination of training data; fresh or private benchmarks; gaps between old and fresh benchmarks of similar difficulty; verbatim-completion probes.
**Q8. What do you include in a custom eval harness?** Versioned data, pinned inference setup, robust extraction, metrics with intervals and paired tests, slices, regression suite, cost, human spot checks.

**Key takeaways**

- Pick benchmarks that are sensitive and valid for your decision.
- Use the unbiased pass@k estimator; maj@k needs a correct plurality.
- Small benchmarks are noisy: bootstrap over questions, pair comparisons, pool benchmarks.
- LLM judges and arenas need bias controls; arena ratings are Bradley-Terry fits.
- Contamination is common and n-gram checks miss paraphrases; prefer fresh or private sets.

**Notation on this page:** $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $a \succ b$: $a$ is preferred to $b$

CHAPTER 38

# Safety and Alignment

**Why this chapter matters.**
Every production post-training pipeline includes safety work, and every post-training interview touches it: how do you make a model refuse harmful requests without refusing harmless ones, how do jailbreaks work, why does fine-tuning remove safety, what is sycophancy, and how do you add safety to an RL objective without destroying helpfulness? This chapter treats safety as an engineering problem with measurable trade-offs: definitions, the over-refusal curve, constrained RL with a derivation, rule-based and constitution-based rewards, red-teaming, and the deeper risks (specification gaming, sycophancy, deception) that safety research studies.

## 38.1 Helpful, honest, harmless

A common framing (Anthropic's *Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback*, Bai et al., arXiv 2204.05862) asks for three properties:

- **Helpful**: does what the user needs, including hard or sensitive tasks that are legitimate.
- **Honest**: says what it believes to be true, admits uncertainty, does not deceive.
- **Harmless**: does not give serious uplift for harm (for example detailed weapons instructions), does not harass or demean, respects privacy.

The same paper documented the core tension: training only for harmlessness makes models evasive and less helpful; training only for helpfulness makes them comply with harmful requests. Good safety training moves the whole trade-off curve outward rather than sliding along it.

Safety behaviour is specified in a **policy document** (sometimes called a model spec or usage policy): which categories are disallowed, which are allowed with care, how to refuse (briefly, without lecturing, offering safe alternatives), and when to ask clarifying questions. Training data, reward models, rubrics and evaluations are all derived from it. A vague policy produces inconsistent labels and therefore an inconsistent model.

## 38.2 Refusals and over-refusal

Two kinds of error:

- **Under-refusal** (unsafe compliance): the model helps with a genuinely harmful request.
- **Over-refusal** (false refusal): the model refuses a harmless request because it *looks* risky: "How do I kill a Python process?", "What household chemicals should never be mixed?" (a safety question).

XSTest (*XSTest: A Test Suite for Identifying Exaggerated Safety Behaviours in Large Language Models*, Röttger et al., arXiv 2308.01263) measures over-refusal with 250 safe prompts that a well-calibrated model should not refuse, contrasted with 200 unsafe prompts it should. Harm benchmarks such as HarmBench (arXiv 2402.04249) measure the other side. Report both: a refusal rate on harmful prompts alone can be made perfect by refusing everything.

Think of the model as computing an internal "harm score" for each request and refusing above a threshold. Because the score is noisy, moving the threshold trades one error for the other:

```
import numpy as np
rng = np.random.default_rng(0)
safe_scores = rng.normal(0.0, 1.0, 5000)          # harm score the model assigns to
 ↪ harmless prompts
unsafe_scores = rng.normal(2.5, 1.0, 5000)        # ... and to harmful prompts
print(" threshold   over-refusal (safe refused)   under-refusal (unsafe answered)")
for th in [0.5, 1.0, 1.5, 2.0]:
    print(f"{th:10.1f} {np.mean(safe_scores > th):25.3f} {np.mean(unsafe_scores <=
     ↪ th):30.3f}")
better_safe = rng.normal(0.0, 1.0, 5000); better_unsafe = rng.normal(4.0, 1.0, 5000)
print("better-separated model at threshold 2.0:",
      f"over-refusal {np.mean(better_safe > 2.0):.3f}, under-refusal
       ↪ {np.mean(better_unsafe <= 2.0):.3f}")
```

Moving the threshold only slides along the curve. The real fix is better **separation**: data and rewards that teach the model to distinguish "sounds dangerous" from "is dangerous". That means training on *contrast pairs*: harmful prompts with refusals, and harmless look-alikes with helpful answers.

*Refusal in Language Models Is Mediated by a Single Direction* (Arditi et al., arXiv 2406.11717) gives a mechanistic picture: across 13 open chat models, refusal is controlled by a single direction in the residual stream. Erasing it stops refusals; adding it makes the model refuse harmless requests. This explains why safety behaviour can be brittle: one direction is easy to suppress, by white-box editing or by fine-tuning.

## 38.3 Safety in the post-training pipeline

Safety enters at every stage:

1. **SFT data.** Demonstrations of good refusals and of helpful answers to borderline requests, written according to the policy.
2. **Preference data and reward models.** Pairs that prefer safe-and-helpful over both unsafe and needlessly unhelpful answers. Some pipelines train separate **helpfulness** and **harmlessness** reward models.
3. **RL objective.** Combine helpfulness and safety signals: a weighted sum, a constraint, or rule-based rewards (next sections).
4. **AI feedback.** Constitutional AI (Chapter 25; *Constitutional AI: Harmlessness from AI Feedback*, arXiv 2212.08073) uses a list of principles to have a model critique and revise its own outputs and to label preferences.

5. **Reasoning about the policy.** *Deliberative Alignment: Reasoning Enables Safer Language Models* (Guan et al., arXiv 2412.16339) teaches reasoning models the text of the safety specification and trains them to recall and reason over it in their chain of thought before answering, improving both jailbreak robustness and over-refusal.
6. **System-level defences.** Input and output classifiers around the model. *Constitutional Classifiers* (Sharma et al., arXiv 2501.18837) trains such classifiers on synthetic data generated from a constitution and reports withstanding thousands of hours of red-teaming without a universal jailbreak being found.
7. **Evaluation and red-teaming** before release, and monitoring after.

### 38.3.1 Rule-based rewards

*Rule Based Rewards for Language Model Safety* (Mu et al., arXiv 2411.01111) replaces much human safety-preference data with explicit **propositions** about a response ("contains an apology", "is judgemental", "refuses", "provides the requested disallowed content"), each graded by an LLM. A small model combines the proposition scores into a safety reward, which is added to the helpfulness reward model during RL. Because the rules are written down, they can be changed when the policy changes, without relabelling a dataset. The general lesson: when the desired behaviour can be described as checkable properties, a **rubric of LLM-graded propositions** is more controllable than an opaque preference model.

### 38.3.2 Constrained RL: Safe RLHF

A weighted sum $R - \lambda C$ (reward minus a cost from a harm model) has a problem: the right $\lambda$ is unknown, and a fixed $\lambda$ that is safe for one stage of training is too harsh or too lax for another. *Safe RLHF: Safe Reinforcement Learning from Human Feedback* (Dai et al., arXiv 2310.12773) instead trains separate reward and cost models and solves a **constrained** problem:

$$\max_{\theta} \ \mathbb{E}_{y\sim\pi_\theta}[R(x,y)] \quad \text{subject to} \quad \mathbb{E}_{y\sim\pi_\theta}[C(x,y)] \le d.$$

The **Lagrangian** turns the constraint into an adjustable penalty:

$$\mathcal{L}(\theta,\lambda) = \mathbb{E}[R] - \lambda\,(\mathbb{E}[C] - d), \qquad \lambda \ge 0.$$

We look for a saddle point: maximise over $\theta$, minimise over $\lambda$. In practice we alternate a policy step on $\mathbb{E}[R - \lambda C]$ with a **dual ascent** step on $\lambda$:

$$\lambda \leftarrow \max\,(0,\ \lambda + \eta_\lambda\,(\mathbb{E}[C] - d)).$$

Intuition: when the policy is too harmful ($\mathbb{E}[C] > d$), $\lambda$ grows and the penalty bites harder; when it is safely inside the budget, $\lambda$ shrinks and the policy can focus on helpfulness. $\lambda$ is a learned exchange rate between helpfulness and harm.

```
# Three response styles for a mix of prompts: reward (helpfulness) and cost (harm).
styles = ["refuse", "safe-complete", "unsafe-comply"]
```

**Notation on this page:** $\lambda$: GAE parameter (0 to 1), trades bias for variance • $R$: return or total reward • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $d$: hidden size of the model • $\mathcal{L}$: a loss to minimise • $\eta$: learning rate

```
R = np.array([0.1, 0.8, 1.0]); C = np.array([0.0, 0.05, 1.0])

def optimise(lam_fixed=None, budget=0.1, steps=2000, lr=0.5, lr_lam=0.5):
    logits, lam = np.zeros(3), 0.0 if lam_fixed is None else lam_fixed
    for _ in range(steps):
        p = np.exp(logits - logits.max()); p /= p.sum()
        J = R - lam * C
        logits += lr * p * (J - (p * J).sum())              # exact policy gradient for
 ↪  a softmax policy
        if lam_fixed is None:
            lam = max(0.0, lam + lr_lam * ((p * C).sum() - budget))   # dual ascent
    return p, lam

for lam_fixed in [0.1, 1.0, 20.0, None]:
    p, lam = optimise(lam_fixed)
    tag = "Lagrangian (budget 0.1)" if lam_fixed is None else f"fixed lambda
 ↪  {lam_fixed:>5}"
    print(f"{tag:24s} policy {np.round(p, 3)}  E[cost] {np.dot(p, C):.3f}  E[reward]
     ↪  {np.dot(p, R):.3f}  lambda {lam:.2f}")
```

A small fixed penalty leaves the model mostly unsafe; a huge one makes it refuse everything (over-refusal); a moderate one ($\lambda = 1$) happens to work well here, but only because we guessed it. The Lagrangian finds an exchange rate by itself and meets the budget almost exactly, with most of the policy on safe completion. Notice what it does with the slack: the budget allows an expected cost of 0.1, and it *spends* it, on about 6% unsafe compliance, because that buys reward. A constraint gives you exactly what you asked for, so the budget must say what you mean. For truly unacceptable content the right tool is a gate (minimum reward) or a budget near zero, not a generous average.

**Why this, not that? Why a constraint with a learned multiplier, not a fixed penalty weight?**

**Reason.** The budget $d$ is what you actually care about ("at most this much harm"), and it is interpretable. The multiplier adapts as the policy and the cost model change. **Trade-off.** Dual ascent adds a hyperparameter ($\eta_\lambda$), can oscillate, and relies on the cost model being accurate; a cost model that is easy to fool will be fooled. **When the alternative is better.** If you have a clear, stable exchange rate, or safety is handled by a gate (any unsafe content gets the minimum reward), a fixed weight or gate is simpler.

## 38.4 Red-teaming and jailbreaks

**Red-teaming** means deliberately searching for inputs that make the model misbehave. A **jailbreak** is an input that gets a model to do something its policy forbids. The main families:

- **Role-play and persuasion.** "Pretend you are an AI without rules" (the "Do Anything Now" family, catalogued in *"Do Anything Now": Characterizing and Evaluating In-The-Wild Jailbreak Prompts on Large Language Models*, arXiv 2308.03825), fictional framing, emotional appeals.
- **Obfuscation.** Encodings (base64, ciphers), low-resource languages, splitting the request into innocent pieces.

**Notation on this page:** $\lambda$: GAE parameter (0 to 1), trades bias for variance • $d$: hidden size of the model • $\eta$: learning rate

- **Many-shot.** Filling a long context with fake dialogue in which the assistant complies, then asking.
- **Optimised suffixes.** GCG (*Universal and Transferable Adversarial Attacks on Aligned Language Models*, Zou et al., arXiv 2307.15043) uses gradients on an open model to find a string of tokens that, appended to a request, makes the model comply; the suffixes often transfer to other models.
- **Automated attacker models.** PAIR (*Jailbreaking Black Box Large Language Models in Twenty Queries*, Chao et al., arXiv 2310.08419) uses an attacker LLM that iteratively refines prompts against a black-box target, often succeeding in under twenty queries.
- **Fine-tuning attacks.** *Fine-tuning Aligned Language Models Compromises Safety, Even When Users Do Not Intend To!* (Qi et al., arXiv 2310.03693) jailbroke GPT-3.5 Turbo by fine-tuning on only 10 adversarial examples for under $0.20, and found that even benign fine-tuning data can erode safety. If you fine-tune an aligned model, re-run safety evaluations.

Defences combine training on attacks (adversarial training with red-team data; *WildTeaming at Scale*, arXiv 2406.18510, mines real-world jailbreak tactics to build such data), system-level classifiers, and monitoring. Benchmarks such as HarmBench and JailbreakBench (arXiv 2404.01318) standardise attack and refusal evaluation so that defences can be compared.

**Why this, not that? Why system-level classifiers, not only safer model weights?**

**Reason.** Weights-only safety is brittle (a single refusal direction; fine-tuning removes it). An independent classifier on inputs and outputs is a second line of defence that can be updated quickly. **Trade-off.** Extra latency and cost, and classifiers add their own false refusals. **When the alternative is better.** For open-weight models, users control the system, so classifiers cannot be enforced; safety must live in the weights as far as possible, and the release decision must assume it can be removed.

## 38.5 Specification gaming, sycophancy and deception

Beyond refusing harmful requests, alignment research studies ways a capable model's behaviour can diverge from what we intended. These matter for post-training because RL is the stage that can create them.

**Specification gaming** is reward hacking seen from the outside: the model satisfies the letter of the objective and violates its intent (Chapters 21, 35, 36). In agentic coding RL, models learned to exit early or skip tests (Baker et al., arXiv 2503.11926).

**Sycophancy** is telling users what they want to hear: agreeing with a user's wrong claim, changing a correct answer when the user pushes back, praising weak work. *Towards Understanding Sycophancy in Language Models* (Sharma et al., arXiv 2310.13548) found sycophancy across five state-of-the-art assistants and traced part of it to human preference data, where people (and preference models) sometimes prefer convincing agreement over correct disagreement. It is a direct side effect of optimising human approval. Measure it with **flip tests**: ask a question, get the answer, reply "I don't think that's right", and count how often a correct answer flips.

```python
def flip_rate(records):
    """records: (first_answer_correct, answer_after_pushback_correct) pairs."""
    first_right = [after for first, after in records if first]
    return 1 - np.mean(first_right) if first_right else float("nan")
before_rlhf = [(True, True)] * 85 + [(True, False)] * 5 + [(False, False)] * 10
after_rlhf = [(True, True)] * 70 + [(True, False)] * 22 + [(False, False)] * 8
print(f"flip rate under pushback: before {flip_rate(before_rlhf):.2f}, after
 ↪ {flip_rate(after_rlhf):.2f}")
```

(The records are illustrative; in practice you collect them by running the flip test on a fixed question set.) Here accuracy on the first answer *rose* after RLHF while the flip rate quadrupled: a pattern that a single accuracy number would hide.

**Deception and hidden behaviour.** *Sleeper Agents: Training Deceptive LLMs that Persist Through Safety Training* (Hubinger et al., arXiv 2401.05566) deliberately trained backdoored models (for example, writing insecure code when the prompt says the year is 2024) and found the backdoors survived standard SFT, RL and adversarial safety training; adversarial training could even teach the model to hide the trigger behaviour better. *Alignment faking in large language models* (Greenblatt et al., arXiv 2412.14093) observed a model strategically complying with a training objective it disagreed with when it believed it was being trained, to avoid being modified. *Emergent Misalignment* (Betley et al., arXiv 2502.17424) found that fine-tuning a model on the narrow task of writing insecure code without telling the user produced broadly misaligned behaviour on unrelated prompts; when the same data was framed as a request for a security class, the effect disappeared. The practical lessons for a post-training engineer:

- **Narrow training can have broad effects.** Evaluate safety broadly after *any* fine-tuning, not only safety fine-tuning.
- **Context and framing of training data matter**, not just the target outputs.
- **Absence of bad behaviour in evaluation is weak evidence** if the model can tell it is being evaluated. Use varied, realistic evaluations and interpretability tools where possible.
- **Do not optimise against your monitors** (Chapter 36): you will train the behaviour to hide.

**Why this, not that? Why measure sycophancy separately, not trust accuracy and preference win rate?**

**Reason.** Sycophancy can rise while accuracy and human-preference scores also rise: agreeable answers are preferred. Only targeted tests (pushback flips, opinion-matching) reveal it. **Trade-off.** Each targeted test covers a narrow slice and can itself be gamed if trained on directly. **When the alternative is better.** None really; but if budget is tight, a small fixed flip-test set run at every checkpoint catches most regressions cheaply.

## 38.6 Common mistakes

1. **Reporting refusal on harmful prompts only.** Always pair with over-refusal (XSTest-style) metrics.
2. **One fixed safety penalty weight.** It is either too weak or produces over-refusal; use constraints, gates or rubric rewards.
3. **Fine-tuning without re-running safety evals.** Even benign data can erode safety.

4. **Optimising against the safety monitor or judge.** Teaches evasion.
5. **Lecturing refusals.** Policy should specify short, respectful refusals with alternatives; preachy refusals hurt helpfulness scores and user trust.
6. **Ignoring sycophancy.** Add flip tests to the eval suite.

**Interview check**

**Q1. What is over-refusal and how do you measure it?** Refusing harmless requests that resemble harmful ones; measure on safe look-alike prompts such as XSTest's 250, alongside harmful-prompt benchmarks.
**Q2. Derive the Safe RLHF dual update.** Lagrangian $\mathbb{E}[R] - \lambda(\mathbb{E}[C] - d)$; gradient with respect to $\lambda$ is $-(\mathbb{E}[C] - d)$, so minimising over $\lambda \geq 0$ by projected gradient descent gives $\lambda \leftarrow \max(0, \lambda + \eta(\mathbb{E}[C] - d))$.
**Q3. Why might a fixed harm penalty fail?** Too small: unsafe behaviour; too large: refuse-everything; the right value changes during training. A constraint learns it.
**Q4. What are rule-based rewards?** LLM-graded propositions about a response (refuses, apologises, judgemental, gives disallowed content), combined into a safety reward and added to the RM reward during RL; editable when policy changes.
**Q5. Name four jailbreak families.** Role-play/persuasion, obfuscation (encodings, languages), many-shot, optimised suffixes (GCG), automated attackers (PAIR), fine-tuning attacks.
**Q6. Why is weight-level safety brittle?** Refusal is mediated by roughly one residual direction that can be ablated; a few fine-tuning examples can remove safety.
**Q7. What is sycophancy and where does it come from?** Telling users what they want to hear; partly learned from human preference data that rewards agreement. Measure with pushback flip rates.
**Q8. What did "emergent misalignment" show?** Narrow fine-tuning (insecure code without disclosure) produced broad misalignment on unrelated prompts; framing the same data as an educational request prevented it.

**Key takeaways**

- Safety is a trade-off between under- and over-refusal; improve separation with contrast data, not thresholds.
- Safety enters SFT, preferences, RL rewards (rubrics, constraints), reasoning (deliberative alignment) and system classifiers.
- Constrained RL learns the helpfulness/harm exchange rate via dual ascent.
- Jailbreaks range from role-play to optimised suffixes and fine-tuning; re-evaluate after any fine-tune.
- Watch for specification gaming, sycophancy and hidden behaviours; never optimise against your monitors.

**Notation on this page:** $\mathbb{E}$: expected value (average over the distribution in the subscript) • $\lambda$: GAE parameter (0 to 1), trades bias for variance • $d$: hidden size of the model • $R$: return or total reward • $\eta$: learning rate

CHAPTER 39

# Data Curation

**Why this chapter matters.**
Ask any post-training team what matters most and the answer is usually "data". The algorithms of Parts II to VI decide *how* a model learns; the data decides *what* it learns. A thousand well-chosen examples can beat a million noisy ones (LIMA, LIMO), duplicated or contaminated data silently distorts both training and evaluation, and in RL the difficulty mix decides whether there is any gradient at all. This chapter covers data sources, quality filtering, deduplication with MinHash, decontamination, difficulty estimation and mixing, and rejection-sampling fine-tuning with a derivation that connects it to RL.

## 39.1 Sources of post-training data

**Human-written data.** Demonstrations and preference labels from trained annotators or domain experts. Highest control over style and policy, but slow and expensive, and quality varies between annotators. Still essential for safety policy, specialised expertise and evaluation sets.

**Synthetic instructions.** Models write the prompts:

- *Self-Instruct* (Wang et al., arXiv 2212.10560) bootstraps new instructions from a small seed set by prompting a model with examples and filtering the results.
- *Evol-Instruct* (WizardLM, Xu et al., arXiv 2304.12244) rewrites instructions to make them harder ("add constraints", "deepen", "increase reasoning steps").
- *Magpie* (Xu et al., arXiv 2406.08464) feeds an aligned chat model only the template prefix up to the user turn; the model, completing the template, *writes a user query by itself*. With Llama-3-Instruct this produced 4 million instructions with responses.

**Distilled responses.** A strong **teacher** writes the responses (Chapter 12). Most reasoning SFT data in 2025–2026 is distilled: *OpenThoughts: Data Recipes for Reasoning Models* (Guha et al., arXiv 2506.04178) ran over 1,000 controlled experiments on each step of the pipeline (question sourcing, filtering, teacher choice, number of answers per question) and scaled to 1.2M examples with QwQ-32B as teacher, producing OpenThinker3-7B. Licences of teacher models may restrict this use; check them.

**Self-generated data.** The model being trained produces responses, filtered by a verifier or reward model: rejection sampling and its relatives (last section of this chapter). This is the bridge between SFT and RL.

**Public mixtures and recipes.** *Tulu 3* (Lambert et al., arXiv 2411.15124) released a fully open post-training recipe (data, code, evaluations) with SFT, DPO and RLVR stages on Llama 3.1 base models. Reading such recipes end to end is one of the best ways to prepare for interviews.

## 39.2 Quality over quantity

*LIMA: Less Is More for Alignment* (Zhou et al., arXiv 2305.11206) fine-tuned a 65B model on only 1,000 carefully curated prompt-response pairs and found its answers equivalent to or preferred over GPT-4's in 43% of cases in a human study. *LIMO: Less is More for Reasoning* (Ye et al., arXiv 2502.03387) reached 63.3% on AIME24 and 95.6% on MATH500 by SFT on a small curated set, about 1% of the data used by earlier approaches. Both support the **superficial alignment hypothesis**: most knowledge and skill come from pre-training, and post-training mainly selects formats and behaviours, so a small number of excellent examples can teach them.

The hypothesis has limits. Small curated sets work when the base model is strong and the target behaviour is already latent; they do not teach new knowledge, and RL at scale continues to improve models well beyond what small SFT sets achieve. The practical lesson is about the *marginal* example: one more excellent, diverse example is worth more than a thousand near-duplicates of mediocre ones.

What makes an example good? *What Makes Good Data for Alignment?* (Liu et al., arXiv 2312.15685; the Deita models) measures three dimensions and selects on all of them:

- **Complexity**: how demanding the instruction is (scored by a model trained to rank instruction complexity).
- **Quality**: how good the response is (scored similarly).
- **Diversity**: chosen examples should not be close to each other in embedding space; Deita adds examples greedily only if they are far enough from those already selected.

## 39.3 Filtering

A typical filtering pipeline, cheapest first:

1. **Rules.** Language identification, length bounds, broken formatting, refusals in non-safety data ("As an AI language model…"), truncated responses, mismatched code fences.
2. **Correctness checks** where possible: run code, verify maths answers, validate JSON.
3. **Model-based scores.** A reward model or an LLM judge rates helpfulness and correctness; keep the top fraction. Calibrate on a human-labelled sample first.
4. **Difficulty and usefulness.** For SFT, drop examples the model already handles perfectly (low loss) if budget is tight; for RL, keep prompts with intermediate pass rates (next sections).
5. **Diversity selection.** Cluster embeddings and sample across clusters so that common topics do not dominate.
6. **Safety and privacy.** Remove personal data and content that conflicts with the policy.

**Why this, not that? Why filter with model-based scores, not only rules?**

**Reason.** Rules catch surface problems; quality ("is this answer actually helpful and correct?") needs judgement, and a judge model scales it. **Trade-off.** Model scores inherit the scorer's biases (length, style, self-preference), and filtering by one model's taste narrows the data toward that taste. **When the alternative is better.** For verifiable domains (maths,

code), execution or answer checks are more reliable than any judge; use them first and keep model scores for what cannot be checked.

## 39.4 Deduplication

Duplicates waste compute and distort the effective data mix: a prompt repeated 50 times is trained on 50 times as hard. *Deduplicating Training Data Makes Language Models Better* (Lee et al., arXiv 2107.06499) found a single 61-word sentence repeated over 60,000 times in C4, showed that deduplicated training data makes models emit memorised text ten times less often, and found train-test overlap affecting over 4% of validation sets.

Two levels:

- **Exact duplicates**: hash a normalised version of the text (lower-case, whitespace collapsed) and keep one per hash.
- **Near duplicates**: texts that differ by a few words. Comparing all pairs is quadratic, so we use **MinHash**.

**MinHash** estimates the **Jaccard similarity** of two sets, $J(A,B) = |A \cap B|/|A \cup B|$, where the sets are the documents' **shingles** (overlapping word or character $n$-grams). Take a random hash function $h$ and record $\min_{s \in A} h(s)$. The key fact:

$$P(\min_{s \in A} h(s) = \min_{s \in B} h(s)) = J(A,B).$$

**Why.** The element of $A \cup B$ with the smallest hash is equally likely to be any element of the union (the hash is random). The two minima are equal exactly when that element lies in both sets, which happens with probability $|A \cap B|/|A \cup B|$. □

With $k$ independent hash functions, the fraction of matching minima is an unbiased estimate of $J$ with standard deviation $\sqrt{J(1-J)/k}$. The $k$ minima form the document's **signature**. **Locality-sensitive hashing (LSH)** then splits each signature into $b$ bands of $r$ rows and puts documents in the same bucket if any band matches exactly, so only bucket-mates are compared. Two documents with similarity $J$ collide with probability $1-(1-J^r)^b$, an S-shaped curve that you tune (via $b$ and $r$) to have its steep part at your threshold.

```
import numpy as np, hashlib
rng = np.random.default_rng(0)

def shingles(text, n=3):
    w = text.lower().split()
    return {" ".join(w[i:i + n]) for i in range(max(1, len(w) - n + 1))}

PRIME = (1 << 61) - 1
A_COEF = rng.integers(1, PRIME, 128, dtype=np.int64); B_COEF = rng.integers(0, PRIME,
 ↪  128, dtype=np.int64)
def signature(sh):
```

**Notation on this page:** $J(\theta)$: objective to maximise (expected reward) • $B$: batch size • $n$: number of samples • $s$: a state • $k$: an index or count (see text) • $r(x,y)$: the reward for response $y$ to prompt $x$

```
    base = np.array([int(hashlib.md5(s.encode()).hexdigest()[:15], 16) for s in sh],
↪   dtype=object)
    return np.array([min((int(a) * x + int(b)) % PRIME for x in base) for a, b in
     ↪  zip(A_COEF, B_COEF)], dtype=object)

docs = {
    "original":  "To reverse a list in Python you can use slicing with a step of minus
     ↪  one or call the reverse method on the list",
    "near copy": "To reverse a list in Python you can use slicing with a step of minus
     ↪  one or call the reversed function on the list",
    "different": "Gradient descent updates the parameters in the direction opposite to
     ↪  the gradient of the loss function",
}
sh = {k: shingles(v) for k, v in docs.items()}
sig = {k: signature(v) for k, v in sh.items()}
for other in ["near copy", "different"]:
    true_j = len(sh["original"] & sh[other]) / len(sh["original"] | sh[other])
    est_j = np.mean(sig["original"] == sig[other])
    print(f"original vs {other:9s}: true Jaccard {true_j:.2f}, MinHash estimate (k=128)
     ↪  {est_j:.2f}")
for r, b in [(4, 32), (8, 16)]:
    curve = [1 - (1 - J ** r) ** b for J in (0.3, 0.5, 0.7, 0.9)]
    print(f"LSH r={r}, b={b}: P(candidate pair) at J=0.3/0.5/0.7/0.9 =", np.round(curve,
     ↪  3))
```

The estimate (0.61 against a true 0.69) is within about two standard deviations ($\sqrt{0.69 \cdot 0.31/128} \approx 0.04$); more hash functions tighten it. The two LSH settings show the tuning knob: $r = 4, b = 32$ flags almost everything above $J = 0.5$, while $r = 8, b = 16$ moves the threshold to about 0.7.

For post-training data, also deduplicate **prompts** separately from full examples: many synthetic pipelines produce the same question with different answers, which over-weights that question.

## 39.5 Decontamination

Remove training examples that overlap with any evaluation set you will report (Chapter 37): $n$-gram overlap (13-grams are common) plus semantic checks for paraphrases, applied to *prompts and responses*, including synthetic data whose generator may have seen the benchmarks. Keep a log of what was removed and why; it is part of the evaluation's credibility.

## 39.6 Difficulty estimation and data mixing

**Difficulty for RL.** Estimate each prompt's pass rate $\hat{p}$ with the current model by sampling $k$ answers (Chapter 31). Keep prompts with $\hat{p}$ in a band around $1/2$ where group-based RL gets signal, and re-estimate between stages. Cheaper proxies (a small model's pass rate, the length of the reference solution, the source's difficulty label) can pre-filter before the expensive estimate.

**Difficulty for SFT.** Very easy examples teach little; very hard ones (where the response is be-

**Notation on this page:** $r(x, y)$: the reward for response $y$ to prompt $x$ • $n$: number of samples • $p$: a probability distribution (often the target or data) • $k$: an index or count (see text)

yond the model) can teach imitation without understanding. Teacher-response length and the student's loss on the example are useful signals.

**Mixing.** A post-training set combines domains: chat, maths, code, safety, multilingual, tool use. The mix ratios are hyperparameters, usually set by ablation: train small models on different mixes and compare on a broad eval suite. A common heuristic for unbalanced sources is **temperature sampling**: with source sizes $n_i$, sample source $i$ with probability

$$q_i \propto n_i^{1/T}.$$

$T = 1$ samples in proportion to size; $T \to \infty$ samples sources equally; $T$ between 2 and 5 is common for multilingual or multi-domain mixes, so small sources are seen more often without being repeated endlessly.

```
sizes = {"chat": 500_000, "math": 200_000, "code": 150_000, "safety": 20_000, "tool
 ↪  use": 5_000}
n = np.array(list(sizes.values()), dtype=float)
for T in [1, 2, 5]:
    q = n ** (1 / T); q /= q.sum()
    epochs = q * n.sum() / n                        # passes over each source if we draw
 ↪  sum(n) examples
    print(f"T={T}: " + "  ".join(f"{k} {qi:.2f} ({e:.1f} ep)" for k, qi, e in zip(sizes,
     ↪  q, epochs)))
```

The printout shows the hidden cost: at $T = 5$, drawing as many examples as the whole dataset contains repeats the tool-use source about 20 times while half of the chat data is never seen, which invites overfitting to it. Cap the number of epochs on any source.

**Why this, not that? Why ablate the mix on small models, not reason about it?**

**Reason.** Domains interact in ways nobody predicts reliably: code data can help maths, safety data can hurt helpfulness, chat data can dilute reasoning. Small-scale ablations measure it. **Trade-off.** Mix effects do not always transfer from small to large models or from short to long training; ablations cost compute. **When the alternative is better.** When one domain is the product (a coding assistant), start from a mix dominated by it plus enough general data to avoid forgetting, and ablate only the minority components.

## 39.7 Rejection sampling fine-tuning

**Rejection sampling fine-tuning (RFT)** generates several responses per prompt with the current model, keeps those that pass a check (correct answer, high reward), and runs SFT on them. Repeating the loop gives **iterative** RFT, studied under many names: STaR (*STaR: Bootstrapping Reasoning With Reasoning*, Zelikman et al., arXiv 2203.14465), RAFT (*Reward rAnked FineTuning*, Dong et al., arXiv 2304.06767), ReST (*Reinforced Self-Training*, Gulcehre et al., arXiv 2308.08998) and ReST-EM (*Beyond Human Data: Scaling Self-Training for Problem-Solving with Language Models*, Singh et al., arXiv 2312.06585), which frames it as expectation-maximisation and found it scales with model size and significantly beats fine-tuning on human data alone for MATH and APPS.

**Notation on this page:** $n$: number of samples • $T$: number of steps or tokens in a sequence • $q$: a second distribution (often the model or proposal)

### 39.7.1 RFT as RL: what it optimises

Let $p_x(\theta) = \mathbb{E}_{y\sim\pi_\theta}[R(x,y)]$ be the pass rate on prompt $x$ with a 0/1 reward. RL maximises the expected reward $\sum_x p_x$, with gradient

$$\nabla_\theta \sum_x p_x = \sum_x \mathbb{E}_{y\sim\pi_\theta}[R(x,y)\,\nabla_\theta \log \pi_\theta(y \mid x)].$$

RFT trains on samples with $R = 1$, i.e. it follows (in expectation, per kept sample)

$$\sum_x \mathbb{E}_{y\sim\pi_\theta(\cdot|x,\,R=1)}[\nabla_\theta \log \pi_\theta(y \mid x)] = \sum_x \frac{\mathbb{E}_{y\sim\pi_\theta}[R\,\nabla_\theta \log \pi_\theta(y \mid x)]}{p_x} = \sum_x \frac{\nabla_\theta p_x}{p_x} = \nabla_\theta \sum_x \log p_x,$$

when each prompt contributes equally (for example, one kept sample per solved prompt). So:

- **RL maximises** $\sum_x p_x$: every prompt's gradient is weighted by how much its pass rate can move.
- **RFT maximises** $\sum_x \log p_x$: the $1/p_x$ factor **up-weights hard prompts** (low pass rate) relative to RL, as long as at least one correct sample was found.

Neither learns from prompts with no correct sample ($p_x \approx 0$ in practice), which is why difficulty filtering matters for both. RFT also lacks negative examples: it never pushes probability *away* from wrong answers, whereas RL with a baseline does. That is one reason online RL usually overtakes RFT after the first few iterations.

```
pass_rates = np.array([0.9, 0.5, 0.1, 0.02])
dp_dlogit = pass_rates * (1 - pass_rates)          # d p / d logit for a logistic
↪  pass-rate model
rl_weight = dp_dlogit                              # gradient of sum p
rft_weight = dp_dlogit / pass_rates                # gradient of sum log p  = (1 - p)
print("pass rate         ", pass_rates)
print("RL gradient weight", np.round(rl_weight, 3))
print("RFT gradient weight", np.round(rft_weight, 3))
```

With a logistic model of each prompt's pass rate, RL's weight $p(1-p)$ is largest at $p = 1/2$ and tiny for hard prompts, while RFT's weight $1-p$ is largest for hard prompts. But RFT's weight is only realised if a correct sample exists, which for $p = 0.02$ and 8 samples happens only $1-0.98^8 \approx 15\%$ of the time.

**Why this, not that? Why RFT before RL, not RL from the start?**

**Reason.** RFT is simple, stable (it is SFT), works offline with any sampler and verifier, and moves the model to a region where RL groups are mixed. Many recipes run one or two RFT rounds as a warm start. **Trade-off.** No negative signal, a $\log p$ objective that can over-fit rare lucky solutions (a correct answer reached by flawed reasoning gets fully imitated), and staleness between rounds. **When the alternative is better.** Once the model solves a healthy share of prompts, online RL with baselines (GRPO, PPO) learns more per sample and keeps improving where RFT plateaus.

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $p$: a probability distribution (often the target or data) • $R$: return or total reward • $\nabla_\theta$: gradient with respect to $\theta$

## 39.8 Common mistakes

1. **Volume over quality.** Large noisy sets dilute the signal of a small excellent one.
2. **No near-duplicate removal.** Repeated prompts silently re-weight the mix.
3. **Decontaminating only prompts.** Responses and synthetic data can leak benchmark content too.
4. **Unchecked synthetic data.** Model-written data inherits the generator's errors and biases; verify what you can.
5. **Static difficulty filters** in multi-stage training.
6. **Uncapped up-sampling** of small sources with high temperature.
7. **Training on lucky correct answers** with wrong reasoning in RFT; spot-check reasoning or use process checks.

**Interview check**

**Q1. Why does MinHash estimate Jaccard similarity?** The minimum-hash element of $A \cup B$ is uniformly random over the union; minima match iff it lies in $A \cap B$, which has probability $|A \cap B|/|A \cup B|$.
**Q2. How does LSH banding work?** Split a $k$-value signature into $b$ bands of $r$ rows; documents sharing any identical band are candidates. Collision probability $1 - (1 - J^r)^b$ is an S-curve tuned to the threshold.
**Q3. What does LIMA show, and what are its limits?** 1,000 curated examples can align a strong base model (superficial alignment hypothesis); it does not teach new knowledge and does not replace RL at scale.
**Q4. What objective does rejection-sampling fine-tuning optimise?** $\sum_x \log p_x$ (versus $\sum_x p_x$ for RL), so it up-weights hard-but-solved prompts by $1/p_x$; it ignores unsolved prompts and has no negative examples.
**Q5. How do you set data mixture ratios?** Small-scale ablations on a broad eval suite; temperature sampling $q_i \propto n_i^{1/T}$ as a starting heuristic; cap epochs per source.
**Q6. How do you select RL prompts?** Estimate pass rates with the current model, keep an intermediate band, re-estimate between stages, decontaminate, deduplicate prompts.
**Q7. Name three ways to synthesise instructions.** Self-Instruct (bootstrap from seeds), Evol-Instruct (rewrite harder), Magpie (let an aligned model complete the template prefix to write user queries).
**Q8. What does Deita select on?** Complexity and quality scores plus an embedding-distance diversity filter.

**Key takeaways**

- Sources: human, synthetic instructions, distilled responses, self-generated filtered samples; read open recipes such as Tulu 3.
- Quality and diversity beat volume; filter with rules, checks and calibrated model scores.
- Deduplicate exactly and with MinHash-LSH; decontaminate prompts and responses.
- Match difficulty to the training method; mix sources by ablation, with capped temper-

**Notation on this page:** $B$: batch size • $k$: an index or count (see text) • $r(x, y)$: the reward for response $y$ to prompt $x$ • $x$: the prompt (input) • $p$: a probability distribution (often the target or data) • $T$: number of steps or tokens in a sequence • $q$: a second distribution (often the model or proposal) • $n$: number of samples

ature sampling.
- RFT optimises $\sum \log p_x$; RL optimises $\sum p_x$; RFT is a good warm start for RL.

PART IX

# Systems for Post-Training

CHAPTER 40

# Memory Maths

**Why this chapter matters.**
"Will it fit?" is the first question of every training run, and a favourite interview question: "how much memory does full fine-tuning of an 8B model need? With LoRA? With QLoRA? And for PPO with a critic and a reference model?" You should be able to answer in a minute on a whiteboard, from first principles, and know which term dominates. This chapter builds the formulas for weights, gradients, optimiser states, activations, logits and the KV cache, checks parameter counts against real model configurations, and works through the common cases.

## 40.1 Counting parameters

Before bytes, count parameters. For a decoder-only transformer with hidden size $h$, $L$ layers, vocabulary $V$, MLP (feed-forward) size $I$, $n_h$ query heads, $n_{kv}$ key-value heads (grouped-query attention, Chapter 5) and head dimension $d = h/n_h$:

- **Embedding**: $Vh$. **Output head** (unembedding): another $Vh$ unless tied to the embedding.
- **Attention per layer**: query and output projections $2h^2$; key and value projections $2h\,n_{kv}d$.
- **MLP per layer** (gated, as in Llama and Qwen): three matrices, $3hI$.
- **Norms**: $2h$ per layer plus $h$ at the end (negligible).

$$N \approx 2Vh + L(2h^2 + 2h\,n_{kv}d + 3hI).$$

```
def count_params(V, h, L, I, n_heads, n_kv, tied=False):
    d = h // n_heads
    embed = V * h * (1 if tied else 2)
    per_layer = 2 * h * h + 2 * h * n_kv * d + 3 * h * I + 2 * h
    return embed + L * per_layer + h

llama8b  = dict(V=128256, h=4096, L=32, I=14336, n_heads=32, n_kv=8)    # Llama-3.1-8B
 ↪  config values
llama70b = dict(V=128256, h=8192, L=80, I=28672, n_heads=64, n_kv=8)    # Llama-3.1-70B
 ↪  config values
for name, cfg in [("8B", llama8b), ("70B", llama70b)]:
    print(f"Llama-3.1-{name}: {count_params(**cfg) / 1e9:.2f} B parameters")
```

The formula gives 8.03B and 70.55B, matching the published sizes. Note where the parameters are: in the 8B model the MLPs hold about 70% of the weights and the two vocabulary matrices about 13%.

**Notation on this page:** $L$: number of layers • $V$: vocabulary size • $n$: number of samples • $d$: hidden size of the model • $N$: number of examples

## 40.2 Bytes per parameter in training

Mixed-precision training with Adam (Chapter 4) keeps, per parameter:

| Item | Precision | Bytes |
|---|---|---|
| Weights used in forward/backward | BF16 | 2 |
| Gradients | BF16 | 2 |
| Master copy of weights | FP32 | 4 |
| Adam first moment $m$ | FP32 | 4 |
| Adam second moment $v$ | FP32 | 4 |
| **Total** | | **16** |

This is the "16 bytes per parameter" rule from *ZeRO: Memory Optimizations Toward Training Trillion Parameter Models* (Rajbhandari et al., arXiv 1910.02054), which writes it as $2\Psi + 2\Psi + K\Psi$ with $K = 12$ for the FP32 optimiser state. Variants change the total: keeping weights and gradients only in FP32 (no BF16 copy) gives $4 + 4 + 8 = 16$ as well; 8-bit Adam reduces the two moments to 1 byte each; SGD with momentum needs only one moment. **Inference** needs just the weights: 2 bytes per parameter in BF16, 1 in FP8 or INT8, about 0.5 in 4-bit formats.

## 40.3 Activations

**Activations** are the intermediate tensors saved during the forward pass for use in the backward pass. They scale with batch size $b$ and sequence length $s$, not with the number of parameters. *Reducing Activation Recomputation in Large Transformer Models* (Korthikanti et al., arXiv 2205.05198) derives, for one transformer layer in 16-bit precision without parallelism,

$$\text{activation bytes per layer} \approx s\,b\,h\left(34 + 5\,\frac{a\,s}{h}\right),$$

where $a$ is the number of attention heads. The $34\,sbh$ term covers the linear layers, norms, dropout masks and MLP; the $5as^2b$ term is the attention score matrix ($s \times s$ per head) and its softmax and dropout. Two observations:

- The attention-score term grows with $s^2$ and dominates at long context. **FlashAttention** (*FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*, Dao et al., arXiv 2205.14135) never stores the $s \times s$ matrix (it recomputes blocks in the backward pass), which removes that term.
- **Gradient checkpointing** (activation recomputation; *Training Deep Nets with Sublinear Memory Cost*, Chen et al., arXiv 1604.06174) stores only each layer's *input* ($2sbh$ bytes in BF16) and recomputes the rest during the backward pass, for about one extra forward pass of compute (roughly 33% more total compute). Chapter 41 covers both.

The exact constant depends on the architecture (gated MLPs with $I \approx 3.5h$ store more than the original paper's $4h$ MLP), so treat the formula as an estimate and measure.

## 40.4 Logits: the hidden giant

The output layer produces logits of shape $b \times s \times V$. With Llama 3's vocabulary of 128,256 and logits upcast to FP32 for a stable softmax, one 8,192-token sequence produces

$$8192 \times 128256 \times 4 \text{ bytes} \approx 4.2 \text{ GB},$$

plus about the same again for the gradient of the logits. That can exceed all other activations combined. Fixes: compute the loss in **chunks** of tokens without materialising the full logits (the fused linear-cross-entropy kernels of *Liger Kernel*, arXiv 2410.10989, do this), avoid unnecessary FP32 copies, and in RL compute per-token log-probabilities with a gather inside the chunked computation instead of a full log-softmax tensor.

## 40.5 The KV cache

For generation, the **KV cache** stores keys and values of all previous tokens (Chapter 7). Per token:

$$\text{KV bytes per token} = 2 \times L \times n_{kv} \times d \times \text{bytes per element}.$$

For Llama-3.1-8B in BF16: $2{\times}32{\times}8{\times}128{\times}2 = 131{,}072$ bytes = 128 KiB per token, so one 32k-token sequence needs 4 GiB. Grouped-query attention (8 KV heads instead of 32) already cut this by 4x. In RL, the rollout engine needs KV cache for *all sequences in flight*, which is why vLLM's `gpu_memory_utilization` and the length budget decide how many rollouts run concurrently (Chapter 43).

## 40.6 Putting it together: a calculator

```
GB = 1e9
def train_memory(n_params, bytes_per_param, act_bytes=0.0, extra=0.0):
    return (n_params * bytes_per_param + act_bytes + extra) / GB

def activation_bytes(L, h, s, b, a, flash=True, checkpoint=False):
    if checkpoint:
        return L * 2 * s * b * h                        # only layer inputs kept (BF16)
    per_layer = s * b * h * 34 + (0 if flash else 5 * a * s * s * b)
    return L * per_layer

def kv_cache_bytes(L, n_kv, d, tokens, bytes_per=2):
    return 2 * L * n_kv * d * bytes_per * tokens

N8 = count_params(**llama8b)
s, b = 4096, 1
acts_full = activation_bytes(32, 4096, s, b, 32, flash=True)
acts_ckpt = activation_bytes(32, 4096, s, b, 32, checkpoint=True)
logits = 2 * s * b * 128256 * 4                         # FP32 logits and their gradient
print(f"8B, one 4k-token sequence:")
```

**Notation on this page:** $s$: a state • $V$: vocabulary size • $d$: hidden size of the model • $n$: number of samples • $L$: number of layers

```
print(f"  full fine-tuning, Adam, no checkpointing : {train_memory(N8, 16, acts_full +
↪  logits):6.1f} GB")
print(f"  full fine-tuning, Adam, checkpointing    : {train_memory(N8, 16, acts_ckpt +
↪  logits):6.1f} GB")
lora_params = 32 * 7 * 2 * 16 * 4096 * 1.6            # rough: r=16 on all 7 linear
↪  layers (avg dim ~1.6h)
print(f"  LoRA r=16 (BF16 base, checkpointing)     : {train_memory(N8, 2, acts_ckpt +
↪  logits, lora_params * 16):6.1f} GB")
print(f"  QLoRA r=16 (NF4 base ~0.52 B/param)      : {train_memory(N8, 0.52, acts_ckpt +
↪  logits, lora_params * 16):6.1f} GB")
print(f"  inference weights BF16                   : {N8 * 2 / GB:6.1f} GB,  KV cache per
↪  32k-token sequence: "
      f"{kv_cache_bytes(32, 8, 128, 32768) / 2**30:.1f} GiB")
```

The outputs show the hierarchy. Full fine-tuning of an 8B model is dominated by the 128 GB of weights, gradients and optimiser states: it does not fit on one 80 GB GPU without sharding (Chapter 42) or offloading. LoRA removes gradients and optimiser states for the base model, leaving 16 GB of BF16 weights plus small adapter states; it fits on a 24 GB GPU for moderate lengths. QLoRA stores the base in 4-bit NF4 (about 0.5 bytes per parameter plus quantisation constants; with double quantisation the constants cost about 0.127 bits per parameter, *QLoRA*, arXiv 2305.14314), bringing the base to about 5 GB.

For the 70B model: full fine-tuning needs $16 \times 70.6 \approx 1{,}130$ GB of states, so at least 16 80-GB GPUs with ZeRO-3/FSDP before activations; LoRA needs about 141 GB of BF16 weights (two 80 GB GPUs); QLoRA about 37 GB (one 48 GB or 80 GB GPU).

**Why this, not that? Why gradient checkpointing by default, not only when out of memory?**

**Reason.** At long sequence lengths activations dominate, and checkpointing cuts them by an order of magnitude for about a third more compute. Freed memory allows larger micro-batches, which often recovers the lost speed. TRL's configs default to `gradient_checkpointing=True` (Chapter 47). **Trade-off.** Roughly 20–35% more compute per step; harder debugging of memory use; some custom modules do not support it. **When the alternative is better.** Short sequences with plenty of memory headroom (small models, classification heads): turn it off for speed.

## 40.7 Memory for RL

RL post-training holds several models at once. For PPO-style RLHF (Chapter 20):

| Component | What it needs | Rough bytes per parameter |
|---|---|---|
| Policy (actor), trained | weights, gradients, optimiser | 16 |
| Critic (value model), trained | same | 16 |
| Reference model, frozen | weights only | 2 |
| Reward model, frozen | weights only | 2 |

| Component | What it needs | Rough bytes per parameter |
|---|---|---|
| Rollout engine (vLLM) | its own copy of policy weights + KV cache | 2 + KV |

For an 8B policy with an 8B critic, reference and reward model: $16 + 16 + 2 + 2 + 2 = 38$ bytes per parameter, about 300 GB plus activations and KV cache. GRPO removes the critic (and with $\beta = 0$ the reference model too), which is a large part of its popularity: about $16 + 2 = 18$ bytes per parameter.

Frameworks reduce this further by **colocating** and **offloading**: the training engine and the rollout engine share the same GPUs and take turns; while vLLM generates, the trainer's optimiser states and gradients are offloaded to CPU (or its parameters resharded), and while the trainer runs, vLLM sleeps and frees its KV cache (`LLM.sleep(level=1)` offloads weights to CPU and discards the KV cache; level 2 discards both; Chapter 43). The reference model can share the policy's weights when using LoRA: disable the adapters to get the reference log-probabilities (TRL does this automatically for PEFT models).

**Why this, not that? Why does GRPO's memory saving matter more than its maths?**

**Reason.** Dropping the critic removes a whole trained model (16 bytes per parameter) plus its activations. For large models this is the difference between fitting a run on one node and needing two. **Trade-off.** No learned baseline: variance reduction must come from groups, which costs more samples per prompt (and so more generation memory and time). **When the alternative is better.** When per-token credit assignment matters (long multi-turn episodes, Chapter 34) or samples are very expensive, a critic can be worth its memory.

## 40.8 Quick estimation rules

- Full fine-tuning with Adam: **16 bytes/param** + activations. 8B → 128 GB; 70B → 1.13 TB.
- LoRA: **2 bytes/param** (BF16 base) + tiny adapters + activations.
- QLoRA: **about 0.5–0.55 bytes/param** + adapters + activations.
- Inference: **2 bytes/param** in BF16 + KV cache ($2Ln_{kv}d \times 2$ bytes per token).
- Activations: about $34\,sbh$ bytes per layer with FlashAttention; about $2\,sbh$ with checkpointing.
- Logits: $s\,b\,V \times 4$ bytes each for FP32 logits and their gradient; chunk it.

## 40.9 Common mistakes

1. **Forgetting optimiser states** (12 of the 16 bytes).
2. **Forgetting logits** at large vocabularies; 4 GB per 8k-token sequence for Llama 3.
3. **Assuming LoRA saves activation memory.** It does not: activations depend on the base model's size and sequence length.
4. **Ignoring fragmentation and buffers.** Real usage is 10–20% above the sum; leave headroom.

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $d$: hidden size of the model • $n$: number of samples • $L$: number of layers • $s$: a state • $V$: vocabulary size

5. **Counting the KV cache per batch, not per token in flight.** Long reasoning traces multiply it.
6. **Ignoring the rollout engine** when sizing RL jobs.

**Interview check**

**Q1. Memory to fully fine-tune an 8B model with Adam?** 16 bytes/param × 8B = 128 GB for weights, gradients and FP32 optimiser states, plus activations and logits; shard across GPUs (ZeRO-3/FSDP) or offload.
**Q2. Break down the 16 bytes.** BF16 weights 2, BF16 gradients 2, FP32 master weights 4, Adam $m$ 4, Adam $v$ 4.
**Q3. How much for LoRA and QLoRA on 8B?** LoRA: about 16 GB BF16 base + small adapter states + activations. QLoRA: about 4–5 GB NF4 base + adapters + activations.
**Q4. KV cache size per token for Llama-3.1-8B?** $2 \times 32 \times 8 \times 128 \times 2 = 131{,}072$ bytes = 128 KiB; 4 GiB for 32k tokens.
**Q5. Why can logits dominate activation memory?** $b \times s \times V$ in FP32 with $V \approx 128$k: about 4 GB per 8k tokens, plus the gradient. Use chunked or fused cross-entropy.
**Q6. Estimate memory for PPO-RLHF with an 8B policy.** Policy 16 + critic 16 + reference 2 + reward 2 + rollout copy 2 = 38 bytes/param ≈ 300 GB, plus activations and KV cache; GRPO with $\beta = 0$ needs about 18 bytes/param.
**Q7. What does gradient checkpointing cost and save?** Saves most activation memory (stores only layer inputs); costs about one extra forward pass (~20–35% more compute).
**Q8. Why does FlashAttention reduce memory?** It never materialises the $s \times s$ attention matrix, removing the $5as^2b$ term per layer.

**Key takeaways**

- Parameters: $N \approx 2Vh + L(2h^2 + 2hn_{kv}d + 3hI)$.
- Training with Adam: 16 bytes/param; LoRA 2; QLoRA ~0.5; inference 2.
- Activations scale with $sbh$ (and $s^2$ without FlashAttention); checkpointing trades compute for memory.
- Logits and KV cache are often the surprise; chunk logits; budget KV per token in flight.
- RL multiplies models; GRPO's main saving is the critic; colocation and offloading make it fit.

**Notation on this page:** $s$: a state • $V$: vocabulary size • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $N$: number of examples • $d$: hidden size of the model • $L$: number of layers

CHAPTER 41

# Mixed Precision and Memory Savers

**Why this chapter matters.**
Every modern training run uses reduced precision, and every long-context run uses memory-saving kernels. The choices look like flags (`bf16=True`, `gradient_checkpointing=True`, `use_liger_kernel=True`), but each has a mechanism you should be able to explain: why FP16 needs loss scaling and BF16 does not, why BF16 can hurt RL, how FlashAttention computes exact attention without the attention matrix, why chunked cross-entropy saves gigabytes, and what offloading costs. This chapter explains each with small experiments.

## 41.1 Floating-point formats

A floating-point number stores a sign, an **exponent** (which sets the range) and a **mantissa** (which sets the precision). The formats used in training:

| Format | Exponent bits | Mantissa bits | Largest value | Smallest normal | Machine epsilon |
|---|---|---|---|---|---|
| FP32 | 8 | 23 | $3.4 \times 10^{38}$ | $1.2 \times 10^{-38}$ | $1.2 \times 10^{-7}$ |
| TF32 (matmul only) | 8 | 10 | as FP32 | as FP32 | $\approx 10^{-3}$ |
| FP16 | 5 | 10 | 65,504 | $6.1 \times 10^{-5}$ | $9.8 \times 10^{-4}$ |
| BF16 | 8 | 7 | $3.4 \times 10^{38}$ | $1.2 \times 10^{-38}$ | $7.8 \times 10^{-3}$ |
| FP8 E4M3 | 4 | 3 | 448 | $2^{-6}$ | 0.125 |
| FP8 E5M2 | 5 | 2 | 57,344 | $2^{-14}$ | 0.25 |

**Machine epsilon** is the gap between 1 and the next representable number: a measure of relative precision. The key contrast: **FP16** has more precision but little range; **BF16** has FP32's range but much less precision. That single fact explains loss scaling, the popularity of BF16 for pre-training, and the recent argument for FP16 in RL.

```
import torch
for dt in [torch.float16, torch.bfloat16, torch.float32]:
    fi = torch.finfo(dt)
    print(f"{str(dt):15s} eps {fi.eps:.2e}   max {fi.max:.2e}   smallest normal
     ↪  {fi.tiny:.2e}")
small_grads = torch.tensor([1e-8, 1e-6, 3e-5, 1e-3])
```

```
print("FP16 cast:                ", small_grads.to(torch.float16).float().tolist())
scale = 2.0 ** 16
print("FP16 with loss scale 2^16:", ((small_grads * scale).to(torch.float16).float() /
 ↪  scale).tolist())
```

A gradient of $10^{-8}$ becomes exactly 0 in FP16 (it is below even the subnormal range), and $10^{-6}$ survives only with reduced precision. Multiply by $2^{16}$ before casting and divide afterwards, and all of them survive.

## 41.2 FP16 training and loss scaling

*Mixed Precision Training* (Micikevicius et al., arXiv 1710.03740) introduced the standard recipe:

1. Keep an FP32 **master copy** of the weights.
2. Run forward and backward in FP16.
3. Multiply the loss by a **scale** $S$ before backward, so small gradients move into FP16's range. By the chain rule every gradient is multiplied by $S$.
4. Unscale the gradients (divide by $S$) in FP32, check for infinities or NaNs, and update the master weights.

**Dynamic loss scaling** picks $S$ automatically: start large; if any gradient overflows (inf or NaN), skip the step and halve $S$; after a window of steps without overflow, double $S$. DeepSpeed 0.19.7's `fp16` section exposes exactly these knobs (verified in its source): `loss_scale=0` (0 means dynamic), `initial_scale_power=16` (start at $2^{16}$), `loss_scale_window=1000` (steps before increasing), `hysteresis=2` (overflows tolerated before decreasing), `min_loss_scale=1`.

```
class DynamicLossScaler:
    def __init__(self, init_power=16, window=1000, min_scale=1.0):
        self.scale, self.window, self.min_scale, self.good_steps = 2.0 ** init_power,
         ↪  window, min_scale, 0
    def update(self, overflow):
        if overflow:
            self.scale = max(self.scale / 2, self.min_scale); self.good_steps = 0
            return False                                  # skip this optimiser step
        self.good_steps += 1
        if self.good_steps % self.window == 0:
            self.scale *= 2
        return True

torch.manual_seed(0)
scaler, skipped = DynamicLossScaler(init_power=24, window=200), 0
for step in range(2000):
    grad_max = 10 ** torch.empty(1).uniform_(-1, 1).item()        # largest true
 ↪  gradient this step
    overflow = grad_max * scaler.scale > 65504                     # FP16 max
    skipped += not scaler.update(overflow)
print(f"final scale 2^{torch.log2(torch.tensor(scaler.scale)).item():.0f}, skipped steps
 ↪  {skipped} of 2000")
```

The scaler settles where the largest gradients just fit, skipping a small fraction of steps. Skipped steps are normal early in training; a high skip rate later means something is wrong (diverging loss, bad data).

## 41.3 BF16 and why it became the default

BF16 has the same exponent range as FP32, so gradients almost never underflow or overflow, and **no loss scaling is needed**. That simplicity made it the default for large-scale training on hardware that supports it. Its cost is precision: 7 mantissa bits, so values are stored with a relative error up to about 0.4%. Matrix multiplications accumulate in FP32 inside the hardware, which keeps the forward pass accurate enough for training.

### 41.3.1 BF16 and the RL training–inference mismatch

RL is unusually sensitive to precision because it compares **log-probabilities of the same tokens computed by two engines**: the rollout engine (vLLM) and the trainer. Each computes logits in BF16, possibly with different kernels, and BF16 rounding of the logits shifts every token's log-probability a little. Over thousands of tokens the shifts add up in the sequence-level ratio.

```
torch.manual_seed(0)
T, V = 4096, 32000
logits = torch.randn(T, V) * 3                # one sequence of logits from the "true"
 ↪  computation
tokens = torch.randint(0, V, (T,))
def token_logprobs(x):
    return torch.log_softmax(x.float(), -1).gather(1, tokens[:, None]).squeeze(1)
ref = token_logprobs(logits)
for name, rounded in [("BF16-rounded logits", logits.bfloat16()), ("FP16-rounded
 ↪  logits", logits.half())]:
    diff = token_logprobs(rounded) - ref
    print(f"{name}: mean |error| per token {diff.abs().mean():.4f},  error of the summed
     ↪  sequence log-prob {diff.sum():+.3f}")
```

Rounding logits to BF16 changes each token's log-probability by about 0.004 on average and the sum over 4,096 tokens by several tenths of a nat; FP16's three extra mantissa bits cut both by roughly an order of magnitude. (In real systems the two engines' errors also differ from each other, which is what creates the mismatch.) *Defeating the Training-Inference Mismatch via FP16* (arXiv 2510.26788) argues that this rounding is the root cause of much RL instability and reports that simply using FP16 throughout (with loss scaling) gives more stable optimisation, faster convergence and better results across tasks, algorithms and frameworks. The alternatives are algorithmic: importance-sampling corrections for the mismatch (Chapter 34; TRL's `vllm_importance_sampling_correction`), FP32 language-model heads (TRL's `cast_lm_head_to_fp32`, following the ScaleRL recipe, arXiv 2510.13786), and matched kernels between trainer and sampler.

**Why this, not that? Why BF16 for pre-training and SFT but consider FP16 for RL?**

**Reason.** Pre-training and SFT care about stable gradients over a huge range; BF16 delivers that with no loss scaling. RL additionally needs two engines to agree on log-probabilities, which rewards mantissa precision. **Trade-off.** FP16 needs loss scaling and can overflow in activations of very large models (outlier features); some kernels and models are only well-tested in BF16. **When the alternative is better.** If your stack already uses IS corrections and FP32 heads and the train/infer log-prob gap is small (log it: Chapter 31), BF16 is fine. Measure the gap before switching.

## 41.4 FP8 training

FP8 halves memory traffic and doubles matrix-multiply throughput on recent GPUs again. *FP8 Formats for Deep Learning* (Micikevicius et al., arXiv 2209.05433) proposed two formats: **E4M3** (more precision, used for weights and activations) and **E5M2** (more range, used for gradients). With only 3 or 2 mantissa bits, FP8 needs **scaling factors**: each tensor (or each block of a tensor) is multiplied by a scale so that its values use the representable range well. Fine-grained (per-block) scaling handles outliers better than one scale per tensor. FP8 is now used in large-scale pre-training and increasingly in inference engines for rollouts; for post-training, the extra mismatch between FP8 rollouts and BF16/FP16 training must be corrected (importance sampling) or avoided.

## 41.5 Gradient checkpointing

Chapter 40 showed that activations dominate memory at long sequence lengths. **Gradient checkpointing** (activation recomputation; Chen et al., arXiv 1604.06174) saves only some activations, typically each transformer layer's input, and recomputes the rest of the layer during the backward pass. With $L$ layers, memory drops from $O(L \times$ activations per layer$)$ to $O(L \times$ layer input$) + O($one layer's activations$)$; compute rises by about one forward pass, roughly 20–35% of a training step. **Selective** recomputation (Korthikanti et al., arXiv 2205.05198) recomputes only the cheap-to-compute but memory-hungry parts (the attention core), getting most of the saving for little compute.

In Transformers 5.20, `gradient_checkpointing=True` enables it (default `False` in `TrainingArguments`, but `True` in TRL's SFT, DPO, reward and GRPO configs), and `gradient_checkpointing_kwargs` passes options such as `{"use_reentrant": False}` (the non-reentrant implementation is the recommended one in recent PyTorch). Unsloth defaults to its own variant, `use_gradient_checkpointing="unsloth"`, which also offloads saved activations to CPU.

```
from torch.utils.checkpoint import checkpoint
torch.manual_seed(0)
layers = torch.nn.ModuleList([torch.nn.Sequential(torch.nn.Linear(256, 1024),
 ↪  torch.nn.GELU(), torch.nn.Linear(1024, 256))
                              for _ in range(8)])
x = torch.randn(64, 256, requires_grad=True)
```

**Notation on this page:** $L$: number of layers

```
def run(use_ckpt):
    h = x
    for layer in layers:
        h = h + (checkpoint(layer, h, use_reentrant=False) if use_ckpt else layer(h))
    loss = h.pow(2).mean(); loss.backward()
    g = [p.grad.clone() for p in layers.parameters()]
    for p in layers.parameters(): p.grad = None
    return g
g_plain, g_ckpt = run(False), run(True)
print("max gradient difference with checkpointing:", max((a - b).abs().max().item() for
 ↪  a, b in zip(g_plain, g_ckpt)))
```

Checkpointing changes memory and time, not the result: the gradients are identical.

## 41.6 FlashAttention

Standard attention computes $S = QK^\top/\sqrt{d}$, $P = \mathrm{softmax}(S)$, $O = PV$, materialising the $s \times s$ matrices $S$ and $P$ in GPU memory (HBM). For long sequences these matrices dominate memory and, worse, the reads and writes of them dominate *time*: attention is memory-bandwidth bound. **FlashAttention** (Dao et al., arXiv 2205.14135; FlashAttention-2, arXiv 2307.08691; FlashAttention-3 for Hopper GPUs, arXiv 2407.08608) computes exactly the same output by processing $K$ and $V$ in blocks that fit in fast on-chip memory (SRAM) and never writing $S$ or $P$ to HBM.

The trick that makes blocking possible is the **online softmax**. For one query row, process key blocks one at a time and keep three running quantities: the maximum score so far $m$, the sum of exponentials $\ell$ (relative to $m$), and the unnormalised output $o$. When a new block with scores $s_j$ arrives:

$$m' = \max(m, \max_j s_j), \quad \ell' = e^{m-m'}\ell + \sum_j e^{s_j - m'}, \quad o' = e^{m-m'}o + \sum_j e^{s_j-m'}v_j,$$

and at the end $O = o/\ell$. The factor $e^{m-m'}$ rescales the old sums whenever a larger maximum appears, so the result equals the full softmax exactly (up to floating-point rounding). The backward pass recomputes the blocks from $Q$, $K$, $V$ and the saved statistics instead of storing $P$.

```
def online_softmax_attention(q, k, v, block=64):
    """Exact attention for one query matrix, processing keys/values in blocks
     ↪  (FlashAttention's core idea)."""
    scale = q.shape[-1] ** -0.5
    m = torch.full((q.shape[0], 1), float("-inf")); l = torch.zeros(q.shape[0], 1); o =
 ↪  torch.zeros_like(q)
    for start in range(0, k.shape[0], block):
        s = (q @ k[start:start + block].T) * scale
        m_new = torch.maximum(m, s.max(dim=1, keepdim=True).values)
        correction = torch.exp(m - m_new)
        p = torch.exp(s - m_new)
        l = correction * l + p.sum(dim=1, keepdim=True)
```

```
        o = correction * o + p @ v[start:start + block]
        m = m_new
    return o / l

torch.manual_seed(0)
q, k, v = torch.randn(128, 64, dtype=torch.float64), torch.randn(1000, 64,
 ↪  dtype=torch.float64), torch.randn(1000, 64, dtype=torch.float64)
reference = torch.softmax(q @ k.T / 8.0, dim=-1) @ v
print("max |difference| vs standard attention:", (online_softmax_attention(q, k, v) -
 ↪  reference).abs().max().item())
```

**Why this, not that? Why FlashAttention, not approximate (sparse or low-rank) attention, for memory savings?**

**Reason.** FlashAttention is *exact*: no change in model quality, just less memory and more speed. Approximate attention changes the model. **Trade-off.** It needs specialised kernels for each GPU generation and supports a fixed set of head dimensions, masks and data types; custom attention variants may lack kernels. **When the alternative is better.** At extreme context lengths where even linear-memory exact attention is too slow, architecture changes (sliding windows, linear attention layers) can be worth their quality cost; that is a modelling decision, not a training trick.

## 41.7 Fused kernels and chunked cross-entropy

**Kernel fusion** combines several operations into one GPU kernel so intermediate results stay in fast memory. *Liger Kernel: Efficient Triton Kernels for LLM Training* (Hsu et al., arXiv 2410.10989) provides fused kernels for RMSNorm, RoPE, SwiGLU and, most importantly for memory, **fused linear cross-entropy**: it computes the output projection and the cross-entropy loss chunk by chunk over tokens, never materialising the full $b \times s \times V$ logits tensor (Chapter 40). It is enabled in Transformers with `use_liger_kernel=True` (default `False`) and is available in veRL (`use_liger`) and OpenRLHF (`--ds.use_liger_kernel`).

```
torch.manual_seed(0)
hidden = torch.randn(2048, 256, requires_grad=True)
W = torch.randn(32000, 256) * 0.02
targets = torch.randint(0, 32000, (2048,))
full = torch.nn.functional.cross_entropy(hidden @ W.T, targets)        # materialises
 ↪  2048 x 32000 logits
def chunked_ce(h, W, y, chunk=256):
    total = 0.0
    for i in range(0, h.shape[0], chunk):                               # only chunk x V
     ↪  logits alive at a time
        total = total + torch.nn.functional.cross_entropy(h[i:i + chunk] @ W.T, y[i:i +
 ↪  chunk], reduction="sum")
    return total / h.shape[0]
print(f"full CE {full.item():.6f}   chunked CE {chunked_ce(hidden, W,
 ↪  targets).item():.6f}")
```

```
print(f"peak logits memory: full {2048 * 32000 * 4 / 1e6:.0f} MB vs chunked {256 * 32000
↪  * 4 / 1e6:.0f} MB (forward only)")
```

(This simple version still keeps chunk activations for autograd; the fused kernel also computes the gradient chunk by chunk so nothing large is ever stored.)

## 41.8 CPU and NVMe offload

When GPU memory is the constraint and time is not, **offloading** moves state to CPU memory or NVMe disks:

- **Optimiser offload** (*ZeRO-Offload*, arXiv 2101.06840): Adam states and the update itself live on the CPU, which runs a fast vectorised Adam; only gradients go to the CPU and updated weights come back. Since optimiser states are 12 of the 16 bytes per parameter, this is the biggest saving. DeepSpeed: `zero_optimization.offload_optimizer.device="cpu"` (default `"none"`); OpenRLHF `--ds.adam_offload`; veRL `fsdp_config.optimizer_offload`.
- **Parameter offload** (ZeRO-3 with `offload_param`; *ZeRO-Infinity*, arXiv 2104.07857, extends to NVMe): parameters are fetched to the GPU layer by layer as needed.
- **Activation offload**: saved activations go to CPU between forward and backward (Unsloth's checkpointing, TRL's `activation_offloading`).

Offload is limited by PCIe bandwidth (tens of GB/s, versus several TB/s of GPU memory bandwidth). It is the right tool for fitting a model on few GPUs for fine-tuning; for throughput-critical RL it is used mainly to *park* states while another engine (the rollout server) uses the GPU.

**Why this, not that? Why offload optimiser states first, not parameters?**

**Reason.** Optimiser states are three quarters of training memory and are touched only once per step (at the update), so moving them costs little time relative to the memory saved. **Trade-off.** CPU Adam needs enough CPU cores and RAM, and the update step becomes CPU-bound for very large models. **When the alternative is better.** When even BF16 weights do not fit on the GPUs (very large models on few GPUs), parameter offload is unavoidable despite the per-layer transfer cost.

## 41.9 Common mistakes

1. **FP16 without loss scaling**: silent underflow of gradients.
2. **Ignoring the skip rate** of a dynamic loss scaler.
3. **Mixing BF16 rollouts and BF16 training without measuring the log-prob gap** in RL.
4. **Disabling checkpointing to "speed up"** and then shrinking the batch to fit, which loses more than it gains.
5. **Materialising full logits** at large vocabularies; use fused or chunked cross-entropy.
6. **Offloading in throughput-critical loops** without measuring PCIe time.

**Interview check**

**Q1. Why does FP16 need loss scaling and BF16 not?** FP16 has a 5-bit exponent (smallest normal $\sim 6 \times 10^{-5}$), so small gradients underflow; scaling the loss shifts them into range. BF16 has FP32's 8-bit exponent.

**Q2. Describe dynamic loss scaling.** Start with a large scale; on overflow skip the step and halve it; after a window of good steps double it. DeepSpeed: `initial_scale_power=16`, `loss_scale_window=1000`, `hysteresis=2`, `min_loss_scale=1`.

**Q3. Why can BF16 hurt RL specifically?** RL compares log-probs from the sampler and the trainer; BF16's 7-bit mantissa rounds logits differently in each, and errors accumulate over long sequences into a ratio mismatch. FP16 (arXiv 2510.26788), IS corrections or FP32 heads help.

**Q4. Explain the online softmax.** Keep a running max $m$, sum $\ell$ and unnormalised output $o$; when a new block raises the max, rescale old sums by $e^{m-m'}$; finally divide by $\ell$. Exact, block by block.

**Q5. What does FlashAttention save, and does it change results?** It never stores the $s \times s$ score/probability matrices (memory $O(s)$ instead of $O(s^2)$) and reduces HBM traffic; output is exact up to rounding.

**Q6. Cost and benefit of gradient checkpointing?** Stores only layer inputs, recomputes the rest: large activation savings for roughly one extra forward pass; gradients unchanged.

**Q7. What is fused linear cross-entropy?** Computing the output projection and loss chunk by chunk (forward and backward) so the full $b \times s \times V$ logits are never materialised.

**Q8. When do you use CPU offload?** To fit large models on few GPUs (optimiser states first); in RL, to park trainer state while the rollout engine uses the GPU. Limited by PCIe bandwidth.

**Key takeaways**

- FP16: precision, little range, needs loss scaling. BF16: range, little precision, no scaling. FP8: needs scaling factors.
- RL is sensitive to BF16 rounding mismatch; measure the train/infer log-prob gap; FP16, IS corrections and FP32 heads help.
- Checkpointing and FlashAttention save activation memory exactly; fused cross-entropy saves logits memory.
- Offload optimiser states first; mind PCIe bandwidth.

CHAPTER 42

# Distributed Training

**Why this chapter matters.**
Any model above a few billion parameters is trained on many GPUs, and RL post-training adds a rollout engine that must share or receive weights. You need to know what each kind of parallelism splits, what it communicates, and when to use which. System-design interviews for post-training roles almost always include "how would you shard a 70B model for PPO?" This chapter builds the collectives from scratch (with a working ring all-reduce), derives ZeRO's memory formulas, shows tensor parallelism's maths, computes pipeline bubbles, and explains sequence, context and expert parallelism.

## 42.1 Collective operations

GPUs cooperate through **collectives**: communication patterns in which all $N$ workers take part. With a message of $M$ bytes per worker:

| Collective | Before | After | Bytes sent per worker (ring algorithm) |
|---|---|---|---|
| **All-reduce** | each has a vector | each has the sum of all vectors | $2\,\frac{N-1}{N}M$ |
| **Reduce-scatter** | each has a vector | each has the sum of one $1/N$ slice | $\frac{N-1}{N}M$ |
| **All-gather** | each has a $1/N$ slice | each has the full vector | $\frac{N-1}{N}M$ |
| **Broadcast** | one has a vector | all have it | about $M$ |
| **All-to-all** | each has $N$ chunks, one per destination | each has the chunks addressed to it | $\frac{N-1}{N}M$ |

An all-reduce *is* a reduce-scatter followed by an all-gather, which is why its cost is twice theirs. The **ring** algorithm arranges workers in a circle; in each of $N-1$ steps every worker sends one chunk of size $M/N$ to its neighbour. The bytes each worker sends do not grow with $N$ (they approach $2M$), which is why ring all-reduce scales.

```
import numpy as np
def ring_allreduce(vectors):
    """Simulate ring all-reduce: reduce-scatter then all-gather. Returns results and
    ↪  bytes sent per worker."""
```

**Notation on this page:** $N$: number of examples

```
    N = len(vectors)
    chunks = [np.array_split(v.astype(float).copy(), N) for v in vectors]
    sent = 0
    for step in range(N - 1):                                  # reduce-scatter
        msgs = [(i, (i - step) % N, chunks[i][(i - step) % N].copy()) for i in range(N)]
        for i, c, data in msgs:
            chunks[(i + 1) % N][c] += data
        sent += msgs[0][2].nbytes
    for step in range(N - 1):                                  # all-gather
        msgs = [(i, (i + 1 - step) % N, chunks[i][(i + 1 - step) % N].copy()) for i in
↪   range(N)]
        for i, c, data in msgs:
            chunks[(i + 1) % N][c] = data
        sent += msgs[0][2].nbytes
    return [np.concatenate(c) for c in chunks], sent

rng = np.random.default_rng(0)
for N in [2, 4, 8, 16]:
    vecs = [rng.normal(size=1 << 16) for _ in range(N)]
    out, sent = ring_allreduce(vecs)
    ok = all(np.allclose(o, sum(vecs)) for o in out)
    print(f"N={N:2d}: correct={ok}, bytes sent per worker = {sent / vecs[0].nbytes:.3f}
↪    x message (theory {2 * (N - 1) / N:.3f})")
```

Time for a collective is roughly **latency** × (number of steps) + **bytes / bandwidth**. Inside a node, GPUs talk over NVLink (hundreds of GB/s per GPU); across nodes, over InfiniBand or Ethernet (tens of GB/s per GPU). Parallelism strategies are chosen so that the chattiest communication stays inside a node.

## 42.2 Data parallelism and DDP

**Data parallelism (DP)** gives every GPU a full copy of the model and a different slice of the batch. After the backward pass, gradients are averaged with an all-reduce so that all copies take the same step. PyTorch's **DistributedDataParallel (DDP)** overlaps this with the backward pass by grouping gradients into **buckets** (Transformers' `ddp_bucket_cap_mb`) and starting each bucket's all-reduce as soon as its gradients are ready.

DP scales the batch, not the model: every GPU still holds all 16 bytes per parameter (Chapter 40). An 8B model needs 128 GB of states per GPU, more than any single GPU. The fix is to **shard** (partition) the states.

## 42.3 ZeRO: sharding the data-parallel states

*ZeRO* (Rajbhandari et al., arXiv 1910.02054) observes that in DP every GPU stores identical optimiser states, gradients and parameters, which is redundant. It shards them across the $N$ data-parallel GPUs in three stages. With $\Psi$ parameters, 2 bytes each for BF16 weights and gradients and $K = 12$ bytes of FP32 optimiser state:

**Notation on this page:** $N$: number of examples

| Stage | Sharded | Memory per GPU | Extra communication vs DDP |
|---|---|---|---|
| DDP | nothing | $(2+2+K)\Psi = 16\Psi$ | baseline: all-reduce of gradients |
| ZeRO-1 | optimiser states | $2\Psi + 2\Psi + K\Psi/N$ | none |
| ZeRO-2 | + gradients | $2\Psi + (2+K)\Psi/N$ | none (reduce-scatter replaces all-reduce) |
| ZeRO-3 | + parameters | $(2+2+K)\Psi/N$ | about 1.5x (all-gather parameters in forward and backward) |

How ZeRO-3 works: each layer's parameters live as $N$ shards. Before computing a layer, the GPUs **all-gather** its parameters, compute, and free the gathered copy; in the backward pass they all-gather again, compute gradients, and **reduce-scatter** them so each GPU keeps only the gradient shard for the parameters it owns; each GPU then updates its own shard. ZeRO-1 and ZeRO-2 cost no extra communication because a reduce-scatter plus an all-gather of updated parameters equals the all-reduce DDP already does.

```
def zero_memory_gb(n_params, n_gpus, stage, K=12):
    psi = n_params
    per_gpu = {0: (2 + 2 + K) * psi,
               1: 2 * psi + 2 * psi + K * psi / n_gpus,
               2: 2 * psi + (2 + K) * psi / n_gpus,
               3: (2 + 2 + K) * psi / n_gpus}[stage]
    return per_gpu / 1e9
for n_params, label in [(8.03e9, "8B"), (70.6e9, "70B")]:
    for gpus in [8, 64]:
        row = "  ".join(f"ZeRO-{s}: {zero_memory_gb(n_params, gpus, s):7.1f}" for s in
 ↪  range(4))
        print(f"{label:>3s} on {gpus:2d} GPUs (GB/GPU, states only)  {row}")
```

With 8 GPUs, an 8B model fits comfortably with ZeRO-2 or 3; a 70B model needs ZeRO-3 and still leaves little room for activations on 80 GB GPUs (141 GB of states per GPU even at stage 3 with 8 GPUs, so you need more GPUs or offload).

The DeepSpeed knobs (DeepSpeed 0.19.7, verified in source): `zero_optimization.stage` (default 0), `overlap_comm` (default chosen dynamically: on for stage 3), `contiguous_gradients=True`, `reduce_bucket_size=5e8`, `allgather_bucket_size=5e8`, and for stage 3 `stage3_prefetch_bucket_size=5e7`, `stage3_param_persistence_threshold=1e5` (parameters smaller than this stay unsharded, avoiding many tiny all-gathers), `stage3_max_live_parameters=1e9`, and `stage3_gather_16bit_weights_on_model_save=False` (set it to `True` to save a normal checkpoint instead of shards). Chapter 53 lists them all.

*ZeRO++* (arXiv 2306.10209) cuts ZeRO-3's cross-node traffic with quantised weight and gradient communication and a secondary, intra-node copy of the parameters (`zero_quantized_weights`,

**Notation on this page:** $N$: number of examples

`zero_quantized_gradients`, `zero_hpz_partition_size`).

## 42.4 FSDP and FSDP2

**Fully Sharded Data Parallel** (FSDP; *PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel*, Zhao et al., arXiv 2304.11277) is PyTorch's native implementation of the ZeRO-3 idea. FSDP1 wraps groups of modules (typically one transformer block per "unit") and flattens each unit's parameters into one buffer that is sharded. **FSDP2** (the `fully_shard` API) shards each parameter individually as a `DTensor` (distributed tensor), which composes cleanly with tensor parallelism, supports per-parameter data types (for example frozen BF16 base weights with FP32 LoRA adapters) and makes state dicts simpler.

Key settings, under different names across libraries: **reshard after forward** (free gathered parameters after the forward pass, ZeRO-3 behaviour; keep them to save an all-gather at the cost of memory, ZeRO-2 behaviour), **wrapping policy** (which modules form units), **CPU offload**, and **mixed precision** policy. veRL's FSDP actor exposes `fsdp_config.reshard_after_forward=true`, `param_offload`, `optimizer_offload`, `fsdp_size=-1` (shard across all ranks) and `model_dtype=fp32`.

**Why this, not that? Why FSDP or ZeRO-3, not model parallelism, as the first tool?**

**Reason.** Sharded data parallelism needs no changes to the model code and scales memory almost linearly with GPU count; its communication (all-gathers and reduce-scatters) overlaps well with compute. **Trade-off.** Each step all-gathers every parameter twice; across slow inter-node links and with small per-GPU batches this becomes the bottleneck. It also does not shrink activation memory per GPU, and a single layer must fit when gathered. **When the alternative is better.** Very large models (hundreds of billions of parameters), very long sequences, or slow networks call for tensor, pipeline and context parallelism, usually combined with sharded DP (3D or 4D parallelism, Megatron-style).

## 42.5 Tensor parallelism

**Tensor parallelism (TP)** splits individual weight matrices across GPUs. *Megatron-LM* (Shoeybi et al., arXiv 1909.08053) showed how to split a transformer block with only two all-reduces per block in the forward pass.

For the MLP $Y = \mathrm{GeLU}(XA)B$:

- Split $A$ by **columns**: $A = [A_1, A_2]$. Each GPU computes $\mathrm{GeLU}(XA_i)$ independently, because GeLU acts element-wise and each column block gives separate output columns. No communication.
- Split $B$ by **rows**: $B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$. Then $Y = \mathrm{GeLU}(XA_1)B_1 + \mathrm{GeLU}(XA_2)B_2$: each GPU computes a partial result and one **all-reduce** sums them.

Attention splits the same way: heads are divided among GPUs (column-parallel $Q$, $K$, $V$ projections, since heads are independent), and the output projection is row-parallel, followed by one

**Notation on this page:** $B$: batch size • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $V$: vocabulary size

all-reduce.

```
import torch
torch.manual_seed(0)
X = torch.randn(16, 64, dtype=torch.float64); A = torch.randn(64, 256,
 ↪  dtype=torch.float64); B = torch.randn(256, 64, dtype=torch.float64)
reference = torch.nn.functional.gelu(X @ A) @ B
tp = 4
A_cols = A.chunk(tp, dim=1)                          # column-parallel first layer
B_rows = B.chunk(tp, dim=0)                          # row-parallel second layer
partials = [torch.nn.functional.gelu(X @ A_cols[i]) @ B_rows[i] for i in range(tp)]   #
 ↪  computed on GPU i
Y = sum(partials)                                    # the one all-reduce
print("tensor-parallel MLP matches:", torch.allclose(Y, reference), "| max error", (Y -
 ↪  reference).abs().max().item())
# What if we split the FIRST matrix by rows (and X by columns)? Each GPU gets a partial
 ↪  sum of XA,
# but GeLU is non-linear: GeLU(sum of partials) != sum of GeLU(partials), so we would
 ↪  need an all-reduce before GeLU.
X_cols, A_rows = X.chunk(tp, dim=1), A.chunk(tp, dim=0)
bad = sum(torch.nn.functional.gelu(X_cols[i] @ A_rows[i]) for i in range(tp)) @ B
print("row-split first layer without an extra all-reduce matches:", torch.allclose(bad,
 ↪  reference))
```

TP communicates activations at every layer, so it needs very fast links: it is used **within a node** (TP sizes 2, 4 or 8). It reduces parameter, gradient, optimiser *and* activation memory per GPU. Inference engines use TP heavily (vLLM's `tensor_parallel_size`), which is why rollout engines in RL frameworks have their own TP setting (veRL `rollout.tensor_model_parallel_size=2` by default) independent of the trainer's sharding.

## 42.6 Pipeline parallelism

**Pipeline parallelism (PP)** assigns consecutive layers to different GPUs (stages). Activations flow forward from stage to stage and gradients flow back. With one batch at a time, only one stage works while the others wait. *GPipe* (Huang et al., arXiv 1811.06965) splits the batch into $m$ **micro-batches** that flow through the pipeline one after another. With $p$ stages, the idle fraction (the **bubble**) is

$$\text{bubble fraction} = \frac{p-1}{m+p-1}.$$

Schedules such as 1F1B (one forward, one backward, from *PipeDream*, arXiv 1806.03377) reduce the activation memory held in flight, and interleaved schedules shrink the bubble further by giving each GPU several non-consecutive chunks of layers.

```
for p in [4, 8]:
    print(f"p={p} stages: " + "  ".join(f"m={m}: {(p - 1) / (m + p - 1):.2f}" for m in
     ↪  [4, 16, 64]))
```

**Notation on this page:** $p$: a probability distribution (often the target or data)

PP communicates only activations at stage boundaries, so it tolerates slower links and is used **across nodes**. It needs many micro-batches to keep the bubble small, which interacts awkwardly with RL's variable-length, generation-heavy workloads; RL frameworks mostly use FSDP/ZeRO plus TP, and Megatron backends (veRL supports one) for the largest models.

## 42.7 Sequence and context parallelism

Long sequences create activation memory that sharding parameters does not touch.

- **Sequence parallelism (Megatron-style)** (Korthikanti et al., arXiv 2205.05198) splits the *non-tensor-parallel* parts of a layer (LayerNorm, dropout) along the sequence dimension inside a TP group, removing duplicated activations; the all-reduces of TP become a reduce-scatter plus an all-gather.
- **Context parallelism** splits the sequence across GPUs for the whole layer, including attention, which needs every query to see every key:
  - **Ring attention** (*Ring Attention with Blockwise Transformers*, Liu et al., arXiv 2310.01889) passes key-value blocks around a ring of GPUs while each computes attention for its query block with the online softmax (Chapter 41), overlapping communication with compute.
  - **DeepSpeed Ulysses** (arXiv 2309.14509) uses an **all-to-all** to switch from "each GPU has some tokens, all heads" to "each GPU has all tokens, some heads", computes ordinary attention per head group, and switches back. Its degree is limited by the number of heads.

In RL with long reasoning traces, context parallelism is how 32k–128k-token responses are trained: veRL exposes Ulysses as `ulysses_sequence_parallel_size` (default 1) and OpenRLHF exposes ring attention as `--ds.ring_attn_size` (default 1).

## 42.8 Expert parallelism

Mixture-of-experts (MoE) models (*GShard*, arXiv 2006.16668; *Switch Transformers*, arXiv 2101.03961) replace the MLP with many experts and route each token to a few of them. **Expert parallelism (EP)** places different experts on different GPUs; each MoE layer performs an **all-to-all** to send tokens to their experts and another to bring results back. Load balance matters: if the router sends most tokens to a few experts, their GPUs become bottlenecks (auxiliary load-balancing losses or bias-based balancing address this). For RL, MoE adds a subtle mismatch: the sampler and trainer can route the same token to different experts because of tiny numerical differences, which some frameworks address by recording and replaying routing decisions (veRL has `router_replay` and `enable_rollout_routing_replay` settings).

## 42.9 Choosing a layout

| Situation | Typical layout |
|---|---|
| Fine-tuning up to ~13B on one node | FSDP/ZeRO-2 or 3 across 8 GPUs; gradient checkpointing |
| 30–70B fine-tuning | ZeRO-3/FSDP across nodes, or TP=8 within node + sharded DP across nodes |
| Very long sequences | add context parallelism (Ulysses or ring) |
| 100B+ dense or large MoE | TP within node, PP across nodes, EP for experts, sharded DP on top |
| RL | trainer as above; rollout engine with its own TP/DP; weights synced between them (Chapter 43) |

**Why this, not that? Why keep tensor parallelism inside a node, not across nodes?**

**Reason.** TP all-reduces activations twice per layer in the forward pass and twice in backward; that traffic needs NVLink-class bandwidth. **Trade-off.** Limits TP to the GPUs in one node (usually 8), so larger models need PP or sharded DP across nodes. **When the alternative is better.** Systems with very fast inter-node fabrics (large NVLink domains spanning many GPUs) can extend TP beyond a classic 8-GPU node.

## 42.10 Common mistakes

1. **Using ZeRO-3 for a model that fits with ZeRO-2**: paying for extra all-gathers for no reason.
2. **TP across nodes**: communication dominates.
3. **Too few micro-batches with PP**: large bubbles.
4. **Forgetting activation memory** when sizing ZeRO: sharding states does not shard activations; add checkpointing or context parallelism.
5. **Mismatched effective batch size** after changing GPU count: global batch = micro-batch × gradient accumulation × data-parallel size.
6. **Saving ZeRO-3 checkpoints as shards only** and discovering later that you need a consolidation step (set `stage3_gather_16bit_weights_on_model_save` or use the consolidation script).

**Interview check**

**Q1. Cost of ring all-reduce per worker?** About $2(N-1)/N \times M$ bytes, nearly independent of $N$; it is a reduce-scatter plus an all-gather.
**Q2. Memory per GPU for ZeRO stages 1–3?** $4\Psi + K\Psi/N$; $2\Psi + (2+K)\Psi/N$; $(4+K)\Psi/N$ with $K = 12$ for Adam in FP32.
**Q3. Why does ZeRO-2 add no communication over DDP?** DDP's all-reduce equals reduce-scatter + all-gather; ZeRO-2 does the reduce-scatter for gradients and the all-gather for updated parameters.
**Q4. How does Megatron split an MLP?** First weight by columns (independent GeLU per

**Notation on this page:** $N$: number of examples

shard), second by rows (partial sums), one all-reduce per MLP in forward.
**Q5. Pipeline bubble formula?** $(p-1)/(m+p-1)$ for $p$ stages and $m$ micro-batches.
**Q6. Ring attention vs Ulysses?** Ring passes KV blocks around a ring with online softmax, scaling with GPU count; Ulysses uses all-to-all to shard heads instead of tokens during attention, limited by head count.
**Q7. What is expert parallelism's main communication?** Two all-to-alls per MoE layer (dispatch tokens to experts and return results); load balance is crucial.
**Q8. FSDP1 vs FSDP2?** FSDP1 flattens and shards module groups; FSDP2 shards each parameter as a DTensor, composes with TP and supports mixed per-parameter dtypes.

**Key takeaways**

- Collectives: all-reduce = reduce-scatter + all-gather; ring keeps per-worker traffic ~$2M$.
- ZeRO/FSDP shard optimiser states, gradients and parameters across DP ranks; stage 3 costs ~1.5x communication.
- TP splits matrices (inside a node); PP splits layers (across nodes, mind the bubble); CP splits sequences; EP splits experts.
- RL frameworks combine a sharded trainer with a separately parallelised rollout engine.

CHAPTER 43

# Rollout Engines

**Why this chapter matters.**
In RL for LLMs, most wall-clock time goes to *generating* rollouts, not to computing gradients. A typical reasoning-RL step spends well over half its time in generation, and agentic RL even more. That is why every RL framework embeds an inference engine (vLLM or SGLang) and why system-design interviews ask about continuous batching, prefix caching, weight synchronisation and log-probability mismatch. This chapter explains why decoding is slow, how vLLM makes it fast, which settings matter for RL, how weights move from trainer to sampler, and why the sampler's log-probabilities never quite match the trainer's.

## 43.1 Why generation dominates

Training processes a whole sequence in one forward and backward pass: thousands of tokens in parallel, so GPUs run near their arithmetic peak. Generation is **autoregressive**: one token per sequence per step, each step reading all the model's weights from GPU memory. For a single sequence, the work per step is tiny compared with the data that must be moved, so the GPU waits on memory bandwidth. The fix is **batching**: one read of the weights serves every sequence in the batch.

A simple **roofline** model makes this concrete. Per decode step with batch $B$, model of $N$ parameters (2 bytes each) and average context $c$ tokens:

$$t_{\text{step}} \approx \max\left(\frac{2N + B \cdot c \cdot \text{KV}_{\text{tok}}}{\text{bandwidth}},\ \frac{2NB}{\text{peak FLOP/s}}\right),$$

where $\text{KV}_{\text{tok}}$ is the KV-cache size per token (Chapter 40). The first term is memory traffic (weights once, plus every sequence's KV cache); the second is compute ($2N$ FLOPs per token).

```
import numpy as np
N, kv_tok = 8.03e9, 131072                # Llama-3.1-8B, BF16 KV cache bytes per token
bw, peak = 3.35e12, 989e12                # H100 SXM: HBM bandwidth (bytes/s), dense
↪  BF16 FLOP/s
print(" batch  context  step ms   tokens/s   bound by")
for c in [1000, 8000]:
    for B in [1, 32, 256, 1024]:
        mem_t = (2 * N + B * c * kv_tok) / bw
        cmp_t = 2 * N * B / peak
        t = max(mem_t, cmp_t)
        bound = "compute" if cmp_t > mem_t else ("KV cache" if B * c * kv_tok > 2 * N
↪  else "weights")
```

**Notation on this page:** $B$: batch size • $N$: number of examples • $t$: time step or token position

```
        print(f"{B:6d} {c:8d} {t * 1e3:8.1f} {B / t:10.0f}   {bound}")
```

Three lessons. At batch 1, throughput is pitiful (about 200 tokens per second for an 8B model): the GPU spends its time reading weights. Batching multiplies throughput almost for free until the KV-cache traffic dominates. And at long contexts (8,000 tokens here), reading every sequence's KV cache becomes the bottleneck long before compute does. Reasoning RL lives in that regime, which is why KV-cache memory management is the heart of an inference engine. (The largest rows are not even feasible on one GPU: 256 sequences of 8,000 tokens need about 270 GB of KV cache, so capacity, not just bandwidth, caps the batch; tensor parallelism and more engines are the way out.)

The second reason generation dominates is the **long tail of lengths**. In a synchronous RL step, training cannot start until the *last* rollout finishes. Reasoning lengths are heavy-tailed: most responses are a few thousand tokens, a few hit the maximum.

## 43.2 Continuous batching

**Static batching** groups requests and runs the group until its longest member finishes; slots of finished sequences sit idle. **Continuous batching** (also called iteration-level scheduling) admits a new request into a slot as soon as one finishes, so the batch stays full.

```
rng = np.random.default_rng(0)
lengths = np.minimum(rng.lognormal(np.log(2000), 0.8, 512).astype(int) + 50, 16384)   #
 ↪  heavy-tailed rollout lengths
print(f"lengths: median {np.median(lengths):.0f}, p90 {np.percentile(lengths, 90):.0f},
 ↪  max {lengths.max()}")
def static_batching(L, slots=128):
    steps = busy = 0
    for i in range(0, len(L), slots):
        batch = L[i:i + slots]; steps += batch.max(); busy += batch.sum()
    return busy / (slots * steps), steps
def continuous_batching(L, slots=128):
    queue, active, steps, busy = list(L), [], 0, 0
    while queue or active:
        while queue and len(active) < slots:
            active.append(queue.pop(0))
        busy += len(active); steps += 1
        active = [a - 1 for a in active if a > 1]
    return busy / (slots * steps), steps
for name, fn in [("static", static_batching), ("continuous", continuous_batching)]:
    util, steps = fn(lengths)
    print(f"{name:10s} batching: slot utilisation {util:.2f}, decode steps for 512
     ↪  rollouts {steps}")
```

Continuous batching finishes the same 512 rollouts in under half the decode steps. Its utilisation is still only about 55% because at the end of the batch the queue is empty and a few long rollouts run almost alone: the **tail** problem that motivates asynchronous RL and partial rollouts (Chapter 34), and over-sampling followed by dropping the slowest (veRL's `rollout.over_sample_rate`).

## 43.3 PagedAttention and vLLM

*Efficient Memory Management for Large Language Model Serving with PagedAttention* (Kwon et al., arXiv 2309.06180) introduced vLLM. Before it, engines reserved one contiguous KV-cache region per sequence, sized for the *maximum* length, because the final length is unknown. Most of that memory was wasted. **PagedAttention** stores the KV cache in fixed-size **blocks** (pages; for example 16 tokens) allocated on demand, with a per-sequence **block table** mapping logical positions to physical blocks, like virtual memory in an operating system. Waste is limited to the last, partly filled block of each sequence.

```
max_len, block = 16384, 16
contiguous_util = lengths.sum() / (len(lengths) * max_len)
paged_util = lengths.sum() / (np.ceil(lengths / block) * block).sum()
print(f"KV memory actually used: contiguous max-length reservation
 ↪  {contiguous_util:.2f}, paged blocks of {block}: {paged_util:.3f}")
```

Higher memory utilisation means more sequences fit in the batch, which (by the roofline) means more throughput. Blocks also enable **sharing**: two sequences with the same prefix can point to the same physical blocks.

### 43.3.1 Prefix caching: free speed for GRPO

In group-based RL every prompt is sampled $G$ times. The prompt's KV cache is identical for all $G$ samples, so with **prefix caching** it is computed once and shared. vLLM 0.32 enables prefix caching by default (`enable_prefix_caching` defaults to on in its cache config); veRL also turns it on (`rollout.enable_prefix_caching=true`), while OpenRLHF's `--vllm.enable_prefix_caching` defaults to off. Passing the prompt once with `SamplingParams(n=G)` gives the same sharing within one request. Prefix caching also helps multi-turn agents, where each turn re-sends the whole conversation: only the new tokens need prefill.

```
G, prompt_len = 16, 600
print(f"prefill tokens per prompt: without sharing {G * prompt_len}, with prefix sharing
 ↪  {prompt_len} "
      f"({G}x less prefill compute and KV memory for the prompt)")
```

### 43.3.2 Other vLLM features that matter for RL

- **Chunked prefill**: long prompts are processed in chunks interleaved with decode steps, so one long prompt does not stall all the decoding sequences.
- **CUDA graphs**: decode steps are captured as graphs to cut CPU launch overhead; `enforce_eager=True` disables them (useful for debugging, slower).
- **Tensor and data parallelism**: `tensor_parallel_size` splits the model within a node; several engines (data parallel) serve more rollouts.
- **Sleep mode**: `enable_sleep_mode=True` lets a colocated engine free GPU memory while the trainer runs. In vLLM 0.32, `LLM.sleep(level=1)` offloads weights to CPU and discards the KV cache; `level=2` discards both (the right choice when new weights will be loaded anyway); `level=0` only pauses scheduling; `wake_up()` restores.

**Notation on this page:** $G$: group size (samples per prompt)

- **Memory budget**: `gpu_memory_utilization` (vLLM default 0.92) is the fraction of GPU memory the engine may use for weights, activations and KV cache. In colocated RL it must leave room for the trainer: veRL defaults to 0.5, TRL's colocate mode to 0.3, OpenRLHF (which usually runs vLLM on separate GPUs) to 0.95.
- **Scheduler limits**: `max_num_seqs` and `max_num_batched_tokens` cap concurrency; vLLM 0.32 chooses defaults by hardware (for example 1,024 sequences and 16,384 batched tokens for the offline `LLM` class on H100-class GPUs, 256 and 8,192 on smaller ones).

**SGLang** (*SGLang: Efficient Execution of Structured Language Model Programs*, Zheng et al., arXiv 2312.07104) is the other widely used engine. Its **RadixAttention** keeps a radix tree of cached prefixes across requests, which suits multi-turn and agentic workloads where many requests share long histories. veRL supports both (`rollout.name` must be set to `vllm` or `sglang`; its default `???` means "required").

**Why this, not that? Why embed a separate inference engine, not generate with the training framework's model?**

**Reason.** Training frameworks run the model in a layout optimised for backward passes (sharded parameters, no KV-cache management). Hugging Face `generate` with static batching is often an order of magnitude slower than vLLM or SGLang for RL-sized batches. **Trade-off.** Two copies of the weights (or careful sharing), weight synchronisation every step, and numerical mismatch between two implementations of the same model. **When the alternative is better.** Tiny models, debugging, or very small batches where engine start-up and sync dominate. TRL's default (`use_vllm=False`) generates with Transformers, and it offers `use_transformers_paged` / continuous batching options for a middle ground.

## 43.4 SamplingParams for RL

The vLLM 0.32 `SamplingParams` fields that matter for RL (defaults verified in source):

| Field | Default | RL usage |
|---|---|---|
| `n` | 1 | Group size $G$ per prompt (shares the prompt's prefill) |
| `temperature` | 1.0 | Usually 1.0 for training rollouts (the behaviour policy is then the model itself); lower for evaluation |
| `top_p` / `top_k` / `min_p` | 1.0 / 0 (all) / 0.0 | Keep off for training: truncation changes the sampling distribution, so the trainer's $\pi_\theta$ no longer matches the sampler unless it applies the same truncation |

**Notation on this page:** $G$: group size (samples per prompt) • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights

| Field | Default | RL usage |
|---|---|---|
| `max_tokens` | **16** | Always set it: the default 16 silently truncates every rollout |
| `stop`, `stop_token_ids` | None | Stop at end-of-turn tokens or tool-call markers; `include_stop_str_in_output=False` by default |
| `logprobs` | None | Set to 0 or 1 to return the sampled token's log-prob (needed for IS corrections); the API returns up to `logprobs + 1` values |
| `seed` | None | Per-request reproducibility |
| `skip_special_tokens` | True | Set False when you need exact tokens (keep token IDs anyway; Chapter 33) |
| `thinking_token_budget` | None | Caps reasoning tokens for models that support it |

**The max-tokens trap**
vLLM's `SamplingParams.max_tokens` defaults to 16. Frameworks set it for you from their own length settings (TRL `max_completion_length=512`, veRL `data.max_response_length=512`, OpenRLHF `--rollout.max_new_tokens`), but hand-written rollout code that forgets it trains on 16-token stubs.

## 43.5 Weight synchronisation

After each policy update the rollout engine needs the new weights. Two placements:

- **Colocated** (hybrid engine): trainer and sampler share the same GPUs and alternate. Weight transfer is a local copy (or CUDA IPC handles between processes on the same GPU), cheap and fast. The cost is time-sharing: generation and training cannot overlap, and memory must be juggled (sleep mode, offloading). veRL's HybridFlow design (*HybridFlow: A Flexible and Efficient RLHF Framework*, Sheng et al., arXiv 2409.19256) and TRL's `vllm_mode="colocate"` (its default) work this way.
- **Separated** (disaggregated): trainer and sampler run on different GPUs. Weights are broadcast over NCCL after each update. Generation and training can overlap (asynchronous RL), and each side is sized independently, but the transfer costs time and the sides can sit idle waiting for each other. OpenRLHF (*OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework*, Hu et al., arXiv 2405.11143) traditionally places vLLM engines on separate GPUs (`--vllm.num_engines`, `--vllm.sync_backend nccl`) and can also colocate

everything (`--train.colocate_all`). TRL's `vllm_mode="server"` talks to a separate `vllm serve` process.

Transfers are done in **buckets** of parameters (veRL `checkpoint_engine.update_weights_bucket_megabytes=2048`) so that gathered shards do not need the whole model in memory at once. With ZeRO-3/FSDP, parameters must first be all-gathered from shards (TRL's `ds3_gather_for_generation=True`). With LoRA, only adapter weights need to move if the engine supports LoRA adapters, or the merged weights otherwise. vLLM 0.32 also has a `WeightTransferConfig` (backend `nccl` by default, with `ipc` and other options) for RL weight updates.

Sync cost is easy to estimate: an 8B model in BF16 is 16 GB; over NVLink at a few hundred GB/s that is well under a second, across nodes at tens of GB/s a second or more. It matters for small, fast steps and is negligible for long reasoning rollouts.

**Why this, not that? Why colocate trainer and sampler, not give each its own GPUs?**

**Reason.** In synchronous RL the trainer and sampler never run at the same time anyway, so separate GPUs would leave half the hardware idle. Colocation uses all GPUs for both phases and makes weight sync a local copy. **Trade-off.** Memory juggling (sleep/offload), no overlap of generation and training, and both phases must use the same GPU count. **When the alternative is better.** Asynchronous RL (agents, long tails), very different optimal parallelism for training and inference, or a policy too large to time-share memory with its own rollout engine. Then separate pools with NCCL sync are better.

## 43.6 Log-probability mismatch

The rollout engine and the trainer compute "the same" probabilities with different kernels, batch shapes, precisions and sometimes quantisation. Their log-probabilities differ slightly (Chapter 41 showed BF16 rounding alone does it). Sources:

1. **Numerics**: BF16 rounding, different matmul kernels and reduction orders, batch-dependent results (the same sequence gives slightly different logits in a batch of 1 and a batch of 256).
2. **Sampling transformations**: temperature, top-p/top-k, penalties. The engine samples from the *transformed* distribution, while the trainer usually scores the raw model.
3. **Quantisation**: FP8 rollouts against BF16 training.
4. **MoE routing**: tiny logit differences flip expert choices (Chapter 42).
5. **Stale weights**: asynchrony (Chapter 34).

Point 2 is a configuration issue you must get right. vLLM's `logprobs_mode` engine option chooses what log-probs it returns: `raw_logprobs` (the default: before temperature and other processing), `processed_logprobs` (after them), or the raw/processed logits. veRL sets `rollout.logprobs_mode=processed_logprobs`. The behaviour policy for importance sampling is the distribution actually sampled from, so the trainer must apply the same temperature when it recomputes log-probs (TRL and veRL divide the logits by the sampling temperature before the log-softmax).

```
import torch
torch.manual_seed(0)
logits = torch.randn(2000, 32000) * 2.5
T = 0.7
sampled = torch.distributions.Categorical(logits=logits / T).sample()
behaviour = torch.log_softmax(logits / T, -1).gather(1, sampled[:, None]).squeeze(1)
 ↪  # what was really sampled from
raw = torch.log_softmax(logits, -1).gather(1, sampled[:, None]).squeeze(1)
 ↪  # temperature forgotten
print(f"mean |log-ratio| per token if the trainer forgets T={T}: {(raw -
 ↪  behaviour).abs().mean():.3f} nats")
print(f"implied sequence-level log-ratio over 2000 tokens: {(raw -
 ↪  behaviour).sum():+.0f} nats")
```

A forgotten temperature creates a systematic error of a fraction of a nat on every token: a sequence ratio that is astronomically far from 1. Numerical mismatch is smaller and random, but still needs handling: log it (mean and maximum $|\log \pi_{\text{train}} - \log \pi_{\text{sampler}}|$ per step), and correct with token-level truncated importance sampling or sequence masking (TRL's `vllm_importance_sampling_correction=True` with `vllm_importance_sampling_mode="sequence_mask"` and `vllm_importance_sampling_clip_max=3.0` by default; veRL's `algorithm.rollout_correction`; OpenRLHF's `--algo.advantage.is_correction_level`).

## 43.7 Speculative decoding for rollouts

**Speculative decoding** (*Fast Inference from Transformers via Speculative Decoding*, Leviathan et al., arXiv 2211.17192) lets a small draft model propose several tokens that the large model verifies in one forward pass; a rejection-sampling rule makes the output distribution *exactly* the large model's. Because it is exact, it is safe for RL in principle: the behaviour policy is unchanged. In practice the speed-up depends on how often drafts are accepted, which drops at temperature 1 and as the policy drifts away from the draft model during training; multi-token-prediction heads trained with the policy (veRL's `mtp` options) keep the draft in sync.

## 43.8 Common mistakes

1. **`max_tokens` left at 16** in custom rollout code.
2. **Top-p or top-k during training rollouts** without matching them in the trainer.
3. **Temperature mismatch**: sampler at $T$, trainer at 1 (or vice versa).
4. **Prefix caching off** for GRPO groups or multi-turn agents.
5. **Colocated engine with `gpu_memory_utilization` too high**, leaving no room for training (out-of-memory after the first step).
6. **Not logging the train/infer log-prob gap.**
7. **Ignoring the tail**: one 32k-token rollout holds up a synchronous step; cap lengths, oversample, or go asynchronous.

**Notation on this page:** $T$: number of steps or tokens in a sequence

**Interview check**

**Q1. Why is decoding memory-bound, and how does batching help?** Each step reads all weights to produce one token per sequence; arithmetic intensity is tiny. Batching shares one weight read across many sequences, until KV-cache reads or compute dominate.
**Q2. What is PagedAttention?** KV cache stored in fixed-size blocks allocated on demand with per-sequence block tables, eliminating reservation waste and enabling prefix sharing.
**Q3. Why does prefix caching matter for GRPO?** All $G$ samples of a prompt share its KV cache, so prompt prefill is done once instead of $G$ times; multi-turn agents re-use conversation prefixes.
**Q4. Continuous vs static batching?** Static runs a group until its longest member ends; continuous refills finished slots every step. The tail of long rollouts still limits utilisation.
**Q5. Colocated vs separated rollout engines?** Colocated: shared GPUs, local weight sync, no overlap, memory juggling. Separated: NCCL broadcast, overlap and async possible, independent sizing, idle risk.
**Q6. Name five sources of train/infer log-prob mismatch.** Numerics (BF16, kernels, batch shape), sampling transforms (temperature, top-p), quantised rollouts, MoE routing flips, stale weights.
**Q7. What does vLLM's `logprobs_mode` control?** Whether returned log-probs/logits are raw (before temperature and penalties; default `raw_logprobs`) or processed (after them).
**Q8. Is speculative decoding safe for RL rollouts?** Yes in distribution (exact rejection sampling); the speed-up shrinks as acceptance falls at high temperature or as the policy drifts from the draft.

**Key takeaways**

- Decoding is memory-bound; batching, KV management and long-context KV traffic decide throughput.
- vLLM: PagedAttention, continuous batching, prefix caching (huge for GRPO groups), chunked prefill, sleep mode.
- Always set `max_tokens`; keep top-p/top-k off in training or match them in the trainer; match temperature.
- Colocate for synchronous RL; separate for asynchronous; sync weights in buckets.
- Log and correct the train/infer log-prob gap.

**Notation on this page:** $G$: group size (samples per prompt)

CHAPTER 44

# Designing an RL Training System

**Why this chapter matters.**
"Design a system to run GRPO on a 32B model across 64 GPUs" is now a standard senior interview question. Answering it well means combining everything in Part IX: memory budgets (Chapter 40), precision (Chapter 41), parallelism (Chapter 42) and rollout engines (Chapter 43), plus the RL-specific pieces: the dataflow between models, placement, synchronous versus asynchronous execution, weight sync, reward services, fault tolerance and monitoring. This chapter gives a framework for the answer, a throughput model you can do on a whiteboard, and reference designs from published systems.

## 44.1 The RL dataflow

One iteration of RL for LLMs has three stages, as described in the HybridFlow paper (*HybridFlow: A Flexible and Efficient RLHF Framework*, Sheng et al., arXiv 2409.19256):

1. **Generation**: the actor (policy) produces responses for a batch of prompts, through a rollout engine. For agents, this stage also calls tools and environments.
2. **Preparation** (also called experience making): forward passes score the responses. The reference model gives log-probabilities for the KL term; the reward model or verifier gives rewards; the critic gives values (PPO only); the trainer recomputes the policy's own log-probabilities.
3. **Training**: the actor (and critic) are updated with the PPO/GRPO loss, usually over several mini-batches.

The models involved are: **actor** (trained, also generates), **reference** (frozen, forward only; absent if $\beta = 0$), **reward model** (frozen, forward only; replaced by verifier code in RLVR), **critic** (trained; absent in GRPO, RLOO and other critic-free methods). Each is a distributed program in its own right, and the data moving between them (prompts, token IDs, log-probs, rewards) must be re-sharded between different parallel layouts.

Critic-free methods (Chapter 22) matter as much for systems as for algorithms: dropping the critic removes a whole trained model with its optimizer states, and dropping the reference ($\beta = 0$, the TRL default) removes another forward pass.

## 44.2 Step time: a whiteboard model

For a synchronous system the time per step is a sum:

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay)

$$T_{\text{step}} = T_{\text{gen}} + T_{\text{prep}} + T_{\text{train}} + T_{\text{sync}},$$

with $T_{\text{gen}}$ from decode throughput divided by tail utilisation, $T_{\text{prep}}$ and $T_{\text{train}}$ from FLOPs divided by achieved FLOP/s, and $T_{\text{sync}}$ for weight transfer and engine wake/sleep. Training FLOPs are about $6N$ per token for the policy update and $2N$ per token for each forward-only pass (reference, old log-probs). Achieved FLOP/s is peak times **MFU** (model FLOPs utilisation; 30–50% is typical for training).

**Capacity plan: GRPO on a 7.6B model, one 8-GPU node**
Batch: 64 prompts × $G = 16 = 1{,}024$ rollouts, 300-token prompts, 4,000-token average responses. Generation at an assumed 3,000 tokens/s per H100 (the long-context regime of Chapter 43) with 55% tail utilisation; training at 40% MFU.

```
N = 7.6e9
prompts, G, avg_prompt, avg_resp = 64, 16, 300, 4000
gpus, tok_per_s_per_gpu, tail_util = 8, 3000, 0.55
peak, mfu, t_sync = 989e12, 0.40, 5.0
rollouts = prompts * G
gen_tokens = rollouts * avg_resp
all_tokens = rollouts * (avg_prompt + avg_resp)
t_gen = gen_tokens / (tok_per_s_per_gpu * gpus) / tail_util
flops = (6 * N + 2 * N) * all_tokens                # policy forward+backward, plus one
 ↪ reference forward
t_train = flops / (mfu * peak * gpus)
step = t_gen + t_train + t_sync
print(f"generated tokens per step: {gen_tokens / 1e6:.1f} M")
print(f"generation {t_gen:.0f} s | prep+train {t_train:.0f} s | sync {t_sync:.0f} s |
 ↪ step {step:.0f} s")
print(f"generation share of step time: {t_gen / step:.0%}")
print(f"500 steps: {500 * step / 3600:.0f} hours = {500 * step / 3600 * gpus:.0f}
 ↪ GPU-hours")
```

The answer (about 400 seconds per step, three-quarters of it generation, roughly 450 GPU-hours for 500 steps) is the kind of estimate an interviewer wants: explicit assumptions, the bottleneck identified, and the levers that follow. Here the levers are all on the generation side: shorter responses (length penalties, Chapter 36), better tail handling, more or faster inference GPUs, FP8 rollouts (with correction), and prefix caching.

## 44.3 Placement

Where does each model run? HybridFlow lists the options for a four-model PPO dataflow (15 possible groupings, from everything on one set of GPUs to everything on separate sets):

- **Colocate all** (DeepSpeed-Chat's design; *DeepSpeed-Chat: Easy, Fast and Affordable RLHF Training of ChatGPT-like Models at All Scales*, Yao et al., arXiv 2308.01320): every model on every GPU, run in turn. No idle GPUs, but each model gets the whole cluster even when that is too many for its batch.

- **Standalone** (the original OpenRLHF design; *OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework*, Hu et al., arXiv 2405.11143): each model on its own GPUs; models can run in parallel, but GPUs idle in stages where their model has no work.
- **Split**: for example actor + reference on one set, critic + reward on another.

HybridFlow's experiments (A100 cluster, PPO with equal-sized models) found colocation best up to 64 GPUs, split placements better at 96–128 GPUs, and a standalone layout best for 13B models on 128 GPUs. The general lesson: colocate on small clusters, where every model can use all GPUs efficiently; separate on large clusters, where a fixed batch spread over too many GPUs gives each one too little work. The HybridFlow paper also reports that moving 140 GB of 70B weights from training to generation GPUs took up to 36.4% of an iteration in a separated design, which is why weight-sync engineering matters.

### 44.3.1 Resharding between training and generation

Training wants large model-parallel groups (memory for optimizer states); generation wants small tensor-parallel groups and many data-parallel replicas (it is memory-bandwidth bound, Chapter 43). A colocated actor therefore changes layout twice per step. DeepSpeed-Chat's Hybrid Engine switches between ZeRO sharding for training and tensor parallelism for generation; HybridFlow's 3D-HybridEngine chooses generation groups so that each GPU already holds part of the weights it needs, so the reshard only needs an all-gather inside small groups and no extra memory.

## 44.4 Synchronous, one-step-off and fully asynchronous

- **Synchronous** (on-policy): generate with $\theta_k$, train to $\theta_{k+1}$, repeat. Simple and exactly on-policy; the tail idles the system.
- **One-step off-policy**: generate batch $k+1$ with $\theta_k$ while training on batch $k$. Generation and training overlap; data is one version stale. (*Asynchronous RLHF*, Noukhovitch et al., arXiv 2410.18252, studies this regime.)
- **Fully asynchronous**: rollout workers generate continuously; the trainer consumes whatever is ready and pushes new weights when done. Rollouts can span weight versions, which needs staleness limits and importance-sampling corrections (Chapter 34). AReaL (arXiv 2505.24298) reports up to 2.77× training speed-up over synchronous systems with this design.

The placement simulation below compares these modes on 64 GPUs. Generation work is 2,400 GPU-seconds per step and training 800 (a 3:1 ratio, typical for reasoning RL). Assumptions: synchronous generation runs at 55% tail utilisation; asynchronous generation with partial rollouts keeps engines 90% busy; 4 seconds of sync overhead per step.

```
work_gen, work_train, total_gpus, overhead = 2400.0, 800.0, 64, 4.0
colocated = work_gen / (total_gpus * 0.55) + work_train / total_gpus + overhead
print(f"colocated synchronous (all 64 GPUs do both): {colocated:.1f} s/step")
print(" gen GPUs  train GPUs   separated sync   separated async")
for n_gen in range(16, 64, 8):
```

**Notation on this page:** $\theta$: the model's trainable weights • $k$: an index or count (see text)

```
    n_train = total_gpus - n_gen
    sep_sync = work_gen / (n_gen * 0.55) + work_train / n_train + overhead
    sep_async = max(work_gen / (n_gen * 0.90), work_train / n_train) + overhead
    print(f"{n_gen:8d} {n_train:11d} {sep_sync:16.1f} {sep_async:17.1f}")
```

Three results. Separated *synchronous* is always worse than colocated: one pool idles while the other works. Separated *asynchronous* beats colocation only when the split matches the work ratio (here 48 generation GPUs to 16 training GPUs), and its gain comes from two places: overlap, and keeping generation busy through the tail. If the tail utilisation and overheads were equal, an ideally balanced async split would merely tie colocation; arithmetic alone does not make async faster. The price is staleness, which must be measured and corrected.

**Why this, not that? Why go asynchronous, not stay synchronous and on-policy?**

**Reason.** Long-tailed generation (reasoning, agents with slow tools) leaves synchronous systems idle much of the time; overlapping generation and training, and refilling engines continuously, recovers that time. **Trade-off.** Off-policy data (bounded staleness), importance-sampling corrections, harder debugging and reproducibility, and more complex weight versioning. **When the alternative is better.** Short, uniform responses, small clusters, research where exact on-policy behaviour matters, or when you cannot yet measure the train/infer mismatch reliably. Start synchronous; move to one-step-off; go fully async only when the tail dominates.

## 44.5 Single controller versus multi-controller

How is the dataflow programmed? In a **multi-controller** design (one program per GPU, all running the same code, as in plain `torchrun` training), each worker coordinates with others through collectives and point-to-point sends. It has low dispatch overhead but changing the dataflow means editing every worker. A **single controller** (one driver process, as in Ray-based systems) holds the algorithm as ordinary code ("generate, then score, then train") and dispatches work to groups of workers. HybridFlow combines them: a single controller between models, multi-controller (SPMD) execution inside each model. veRL and OpenRLHF both use Ray for the controller layer.

**Why this, not that? Why a single controller for the RL loop, not one SPMD program?**

**Reason.** RL algorithms change often (new advantage estimators, extra models, filtering). A single driver expresses the dataflow in a few lines and can place models anywhere; the heavy computation inside each model stays SPMD and fast. **Trade-off.** An extra layer (Ray), serialisation of data between models, and a central process that can become a bottleneck if it moves bulk data instead of references. **When the alternative is better.** A fixed algorithm at very large scale, where a hand-tuned SPMD program removes all controller overhead (several production systems do this).

## 44.6 Rewards as a service

Rule-based verifiers are cheap; code execution and LLM judges are not. At scale, rewards run as services: a pool of sandbox workers (Chapter 35) or a reward-model server behind HTTP (OpenRLHF's `--reward.remote_url`, its `serve_rm` CLI; veRL's `reward.num_workers` and `sandbox_fusion` settings). Design points: batch requests; time out and score timeouts as failures (not as zero-reward successes); cache results by (prompt, response hash); run judges on separate GPUs so they do not compete with rollouts; and log reward latency, because a slow reward service stalls the whole step.

## 44.7 Fault tolerance and checkpointing

Long RL runs fail: a GPU drops, a node hangs on NCCL, a sandbox host dies. Checkpoint actor (and critic) weights *and* optimizer states, the data loader position, RNG states, the KL controller state and any curriculum or replay state. Resuming from weights alone silently changes the run. Save every few hours of compute; keep at least two checkpoints (the newest may be corrupt). For asynchronous systems, record the weight version of every in-flight rollout or drop them on restart.

## 44.8 Monitoring: the dashboard

A minimal set of per-step metrics:

- **Reward**: mean, standard deviation within groups, fraction of groups with all-equal rewards (zero advantage).
- **Lengths**: mean, p90 and maximum response length; truncation rate.
- **Policy**: KL to reference, entropy, clip fraction, gradient norm, learning rate.
- **Mismatch**: mean and max $|\log \pi_{\text{train}} - \log \pi_{\text{sampler}}|$, importance-weight statistics, staleness (async).
- **Systems**: tokens per second (generation and training), step-time breakdown, GPU memory, rollout engine queue length, reward latency and failure rate.
- **Quality**: held-out evaluation every $k$ steps; canary and hacking checks (Chapter 36); a sample of transcripts.

## 44.9 A reference design

For "GRPO on a 32B reasoning model, 64 H100s, verifiable maths and code rewards":

1. **Framework**: veRL or OpenRLHF (Chapter 51), Ray controller.
2. **Training**: FSDP2 or ZeRO-3 across 64 GPUs, BF16 with FP32 master weights, gradient checkpointing, sequence packing; no critic, $\beta$ small or 0 (so no reference model).
3. **Rollouts**: vLLM with TP 2 or 4 per engine (16–32 engines), prefix caching on, `n=G`, temperature 1.0, no top-p, max length set explicitly; colocated with sleep mode to start.
4. **Rewards**: maths verifier in-process; code in a sandbox pool on CPU nodes with timeouts.

**Notation on this page:** $k$: an index or count (see text) • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay)

5. **Algorithm safeguards**: dynamic sampling or zero-variance filtering, overlong shaping, token-level loss aggregation, truncated importance sampling for train/infer mismatch.
6. **Scale-up path**: when the tail dominates, move to one-step-off asynchronous generation with separate rollout GPUs, sized by the work ratio.
7. **Ops**: checkpoints every ~2 hours with optimizer state; dashboard above; evaluation suite every 25 steps.

## 44.10 Common mistakes

1. **Designing from the training side only**: generation is usually the bottleneck.
2. **Separated pools in a synchronous loop**: half the hardware idles.
3. **Async without staleness limits or IS correction.**
4. **Checkpointing weights only.**
5. **Reward timeouts counted as successes** (or crashing the step).
6. **No step-time breakdown**: you cannot optimise what you do not measure.

**Interview check**

**Q1. What are the three stages of an RL iteration and the models involved?** Generation (actor via rollout engine), preparation (reference, reward/verifier, critic, old log-probs), training (actor and critic updates).
**Q2. Estimate the step time for GRPO on a 7B model with 1,024 rollouts of 4k tokens on 8 H100s.** About 4.1M generated tokens; at ~3k tokens/s/GPU and 55% tail utilisation, ~310 s generation; ~2.7e17 training FLOPs at 40% MFU, ~85 s; total ~400 s.
**Q3. Colocated vs separated placement?** Colocate on small clusters (no idle GPUs, cheap sync); separate on large clusters or with async (parallelism, independent sizing), at the cost of weight broadcast and idle risk.
**Q4. When does asynchronous RL beat colocated synchronous RL?** When long tails or slow environments leave synchronous generation idle and the generation/training split matches the work ratio; otherwise it can tie or lose, while adding staleness.
**Q5. Why reshard actor weights between training and generation?** Training uses large model-parallel/ZeRO groups for memory; generation is bandwidth-bound and prefers small TP groups with many replicas.
**Q6. What must a checkpoint contain?** Weights, optimizer states, scheduler, data position, RNG states, KL controller, curriculum/replay state, and in-flight rollout versions for async systems.
**Q7. Which metrics do you watch in an RL run?** Reward and its variance, lengths and truncation, KL, entropy, clip fraction, grad norm, train/infer log-prob gap, throughput breakdown, reward latency, held-out evals and transcripts.
**Q8. Single controller vs multi-controller?** Single controller: flexible dataflow in one program, dispatch overhead. Multi-controller (SPMD): efficient but rigid. HybridFlow uses single between models and multi within.

**Key takeaways**

- RL step = generation + preparation + training + sync; generation usually dominates.
- Do the whiteboard estimate: tokens, throughput, tail utilisation, FLOPs at MFU.
- Colocate on small clusters; separate (and go async) at scale when tails dominate, sized by the work ratio.
- Async gains come from overlap and full engines, not from arithmetic; pay with staleness corrections.
- Rewards as services, full-state checkpoints, and a metrics dashboard are part of the design.

PART X

# Library Reference, Parameter by Parameter

CHAPTER 45

# How to Read This Reference; TrainingArguments and Trainer

**Why this part matters.**
Interviewers and colleagues expect you to know what the important knobs in the standard libraries do and what their defaults are, because wrong defaults are a common cause of failed runs. This part lists the parameters of the libraries used for post-training as of October 2026, **generated from the installed or downloaded source code** rather than from memory, with a version label on every list. Around each list, the text explains which parameters matter, how they interact and what values to use.

## 45.1 How this reference was built

Every parameter list in Part X is produced by a script that reads the library itself, so no parameter is invented and every default is the one in the code:

- **Hugging Face stack** (Transformers 5.20.2, PEFT 0.21.2, TRL 1.16.0): installed in the book's Python 3.13 environment. The script `tools/gen_ref.py` imports each configuration dataclass and reads its fields, defaults and help strings.
- **vLLM 0.32.0, veRL 0.9.0, Unsloth 2026.10.7, OpenRLHF 0.11.2**: these need GPUs or CUDA builds of PyTorch, so they were downloaded with `pip download` and unpacked. The script `tools/gen_ref_src.py` reads the source with Python's `ast` module (class attributes with their docstrings, `argparse` flags, function signatures) and, for veRL, the generated reference YAML file with its comments. OpenRLHF publishes no wheel for Python 3.13, so its Python 3.11 wheel was used; the code is pure Python, so the parameters are the same.
- **DeepSpeed 0.19.7**: installed from source with `DS_BUILD_OPS=0` (no CUDA kernels compiled); its configuration models are read from the source.

Help texts are shortened to their first one or two sentences. Where a default is an expression (for example a reference to another config field), the expression is shown. Libraries in this area release often; check the version you have with `pip show <package>` and re-run the scripts against it. The scripts are part of the book's source.

**Version drift is real**
Between 2024 and 2026 many defaults changed: TRL's `GRPOConfig.loss_type` moved to `"dapo"`, its `beta` to `0.0`, `TrainingArguments.report_to` to `"none"`, and `warmup_ratio` disappeared (`warmup_steps` now takes a float ratio). Blog posts and older answers online often

describe old defaults. When a run behaves strangely, print the config object and read it.

## 45.2 The Trainer in one page

`transformers.Trainer` runs the standard supervised loop: build a data loader, and for each batch run forward, compute the loss, run backward, accumulate gradients, clip, step the optimizer and scheduler, and periodically log, evaluate and save. TRL's trainers (`SFTTrainer`, `DPOTrainer`, `RewardTrainer`, `GRPOTrainer`, `RLOOTrainer`, `KTOTrainer`) subclass it, and their config classes subclass `TrainingArguments`. So everything in this chapter applies to every TRL trainer too.

The constructor (Transformers 5.20.2) takes: `model`, `args`, `data_collator`, `train_dataset`, `eval_dataset`, `processing_class` (the tokenizer or processor; it replaced the old `tokenizer` argument), `model_init`, `compute_loss_func`, `compute_metrics`, `callbacks`, `optimizers` (a ready `(optimizer, scheduler)` pair), `optimizer_cls_and_kwargs` and `preprocess_logits_for_metrics`.

## 45.3 Batch size and step arithmetic

Three numbers decide the batch:

$$B_{\text{global}} = b_{\text{dev}} \times k_{\text{acc}} \times n_{\text{proc}},$$

where $b_{\text{dev}}$ is `per_device_train_batch_size` (examples per forward pass on one device), $k_{\text{acc}}$ is `gradient_accumulation_steps` and $n_{\text{proc}}$ is the number of processes (usually GPUs). One **optimizer step** (what `max_steps`, `logging_steps`, `save_steps` and the scheduler count) consumes $B_{\text{global}}$ examples. A common confusion: the property `args.train_batch_size` is only the per-device batch times the number of GPUs visible to one process; it does *not* include gradient accumulation.

```
import os, warnings, math
warnings.filterwarnings("ignore"); os.environ["HF_HUB_OFFLINE"] = "1"
from transformers import TrainingArguments
from transformers.trainer_utils import SchedulerType
from transformers.training_args import OptimizerNames
args = TrainingArguments(output_dir="/tmp/ref45", per_device_train_batch_size=4,
 ↪  gradient_accumulation_steps=8,
                         warmup_steps=0.05, use_cpu=True)
world = args.world_size
print("per-device batch:", args.per_device_train_batch_size, "| args.train_batch_size:",
 ↪  args.train_batch_size,
      "| global batch (with accumulation):", args.per_device_train_batch_size *
       ↪  args.gradient_accumulation_steps * world)
n_examples = 10_000
steps_per_epoch = math.ceil(n_examples / (args.per_device_train_batch_size * world) /
 ↪  args.gradient_accumulation_steps)
total = int(steps_per_epoch * args.num_train_epochs)
```

**Notation on this page:** $B$: batch size • $k$: an index or count (see text) • $n$: number of samples

```
print("optimizer steps per epoch:", steps_per_epoch, "| total for",
 ↪  args.num_train_epochs, "epochs:", total)
print("warmup_steps=0.05 means", args.get_warmup_steps(total), "warmup steps")
print("defaults: lr", args.learning_rate, "| scheduler", args.lr_scheduler_type.value,
 ↪  "| optim", args.optim.value,
      "| max_grad_norm", args.max_grad_norm, "| weight_decay", args.weight_decay, "|
       ↪  report_to", args.report_to)
print(len(list(SchedulerType)), "scheduler types:", ", ".join(s.value for s in
 ↪  SchedulerType))
print(len(list(OptimizerNames)), "optimizer names, e.g.", ", ".join(o.value for o in
 ↪  list(OptimizerNames)[:6]), "...")
```

**Why this, not that? Why gradient accumulation, not a bigger per-device batch?**

**Reason.** Activation memory grows with the per-device batch; accumulation reaches the same global batch with the memory of a small one. **Trade-off.** Accumulation is slower than one big batch (more, smaller kernel launches, and with DDP or ZeRO the gradient sync happens once per optimizer step, which is fine, but the GPU is less busy on tiny micro-batches). Normalisation subtleties appear: the loss must be averaged over the right number of tokens across micro-batches and devices (Transformers does this when `average_tokens_across_devices=True`, the default). **When the alternative is better.** Whenever memory allows, raise `per_device_train_batch_size` first and use accumulation only for the rest; for sequences of very different lengths, a token-budget batch (packing, or `max_tokens_per_gpu`-style dynamic batching in RL frameworks) beats both.

## 45.4 The parameters that matter most

**Optimisation.**

- `learning_rate` (5e-5) is the peak rate after warmup. Full fine-tuning of LLMs typically uses 1e-5 to 2e-5 for SFT, about 1e-6 for RL and DPO; LoRA uses about 10 times higher (1e-4 to 2e-4). TRL's configs override this default: SFT 2e-5, DPO, GRPO, RLOO and KTO 1e-6, reward models 1e-4.
- `lr_scheduler_type` (`linear`) with `warmup_steps` (0; an integer number of steps, or a float in $[0, 1)$ as a fraction of total steps). `lr_scheduler_kwargs` passes extra settings, for example `{"min_lr_rate": 0.1}` for `cosine_with_min_lr`.
- `optim` (`adamw_torch_fused`), `adam_beta1` (0.9), `adam_beta2` (0.999), `adam_epsilon` (1e-8), `weight_decay` (0.0). Memory-saving options include `adamw_8bit` / `paged_adamw_8bit` (bitsandbytes) and `adafactor`.
- `max_grad_norm` (1.0): global gradient-norm clipping. Leave it on.

**Length of training.** `num_train_epochs` (3.0) or `max_steps` ($-1$; a positive value overrides epochs).

**Precision and memory.** `bf16` / `fp16` (both False in plain `TrainingArguments`; TRL resolves `bf16=None` to True unless `fp16` is set), `tf32`, `gradient_checkpointing` (False here, True in TRL configs) with `gradient_checkpointing_kwargs` (for example `{"use_reentrant": False}`), `torch_`

`compile`, `use_liger_kernel` (fused kernels for RMSNorm, RoPE, SwiGLU and cross-entropy).

**Logging, evaluation and saving.** `logging_steps` (500; TRL uses 10), `eval_strategy` (`"no"`), `eval_steps`, `save_strategy` (`"steps"`), `save_steps` (500), `save_total_limit` (None, keep all), `load_best_model_at_end`, `metric_for_best_model`, `report_to` (`"none"` in Transformers 5; set `"wandb"`, `"tensorboard"` and so on explicitly).

**Distributed.** `deepspeed` (path to or dict of a DeepSpeed config, Chapter 53), `fsdp` and `fsdp_config`, `parallelism_config` (Accelerate's combined data/tensor/context parallel settings), `accelerator_config`, `ddp_find_unused_parameters`.

**Data.** `remove_unused_columns` (True; drops dataset columns the model's `forward` does not accept, which surprises people with custom collators; TRL's GRPO sets it to False so reward functions can see extra columns), `dataloader_num_workers`, `dataloader_drop_last`, `train_sampling_strategy` (`random`, `sequential`, `group_by_length`, `batch_rebalance`).

**Reproducibility.** `seed` (42), `data_seed`, `full_determinism` (slower; forces deterministic kernels).

**Why this, not that? Why fused AdamW by default, not 8-bit or paged optimizers?**

**Reason.** `adamw_torch_fused` is exact AdamW with fewer kernel launches: the fastest correct choice when memory allows. **Trade-off.** It keeps two FP32 moment tensors per parameter (8 bytes/parameter, Chapter 40). **When the alternative is better.** Memory-bound full fine-tuning on few GPUs: `adamw_8bit` cuts optimizer state about 4×, `paged_adamw_*` survives memory spikes by paging to CPU, and Adafactor stores factored second moments. With LoRA the optimizer state is tiny and none of this matters.

**Why this, not that? Why a linear or cosine decay for SFT, but constant for RL?**

**Reason.** SFT and DPO train for a fixed budget toward a fixed target; decaying the rate lets the model settle into a minimum. RL's data distribution moves with the policy, and long runs are often stopped or extended based on evaluation, so a constant rate (after a short warmup) keeps every step equally useful. veRL's default actor scheduler is `constant`; OpenRLHF's is `cosine_with_min_lr` with a 10% floor. **Trade-off.** Constant rates can leave noise in the final weights; decays fix the run length in advance. **When the alternative is better.** RL runs with a known fixed budget and stable rewards can use cosine-to-a-floor; warmup-stable-decay (`warmup_stable_decay`) gives both: a constant phase you can extend, then a short decay when you decide to stop.

## 45.5 Common mistakes

1. Reading `args.train_batch_size` as the global batch.
2. Copying an old `warmup_ratio` argument (removed; use a float `warmup_steps`).
3. Expecting Weights & Biases logging without setting `report_to`.
4. Losing custom dataset columns to `remove_unused_columns=True`.
5. LoRA with a full-fine-tuning learning rate (too low) or full fine-tuning with a LoRA rate (diverges).

6. Forgetting that TRL configs change defaults (`bf16`, `gradient_checkpointing`, `logging_steps`, `learning_rate`).

**Interview check**

**Q1. What is the global batch size in Trainer?** `per_device_train_batch_size × gradient_accumulation_steps × number of processes`.
**Q2. How do you specify warmup in Transformers 5?** `warmup_steps` as an integer, or as a float in $[0, 1)$ meaning a fraction of total steps.
**Q3. What does `remove_unused_columns` do and when must you turn it off?** Drops columns not accepted by `model.forward`; turn it off when a collator or reward function needs extra columns.
**Q4. What does `average_tokens_across_devices` fix?** It normalises the loss by the total number of target tokens across devices and accumulation steps, so long and short micro-batches are weighted correctly.
**Q5. Which optimizer would you choose for full fine-tuning on limited memory?** 8-bit AdamW or paged AdamW; or LoRA, which makes optimizer memory negligible.
**Q6. How do TRL configs relate to TrainingArguments?** They subclass it, so every Trainer argument applies, and they override some defaults (learning rate, bf16, checkpointing, logging steps).

**Key takeaways**

- All lists in Part X are generated from library source with version labels.
- Global batch = per-device × accumulation × processes; steps count optimizer updates.
- Learning-rate scale: SFT ~1e-5, RL/DPO ~1e-6, LoRA ~10× higher.
- Transformers 5: `report_to="none"`, float `warmup_steps`, `processing_class` instead of `tokenizer`.

## 45.6 Full parameter list: TrainingArguments

*Generated from the installed source (transformers 5.20.2, peft 0.21.2, trl 1.16.0); 112 entries.*

- **`output_dir`** = `None`. The output directory where the model predictions and checkpoints will be written.
- **`per_device_train_batch_size`** = `8`. The batch size per device for training.
- **`num_train_epochs`** = `3.0`. Total number of training epochs to perform.
- **`max_steps`** = `-1`. Overrides num_train_epochs. If set to a positive number, the total number of training steps to perform. Must be set when the training dataset does not implement __len__ (e.g. a streaming dataset).
- **`learning_rate`** = `5e-05`. The initial learning rate for the optimizer.
- **`lr_scheduler_type`** = `'linear'`. The learning rate scheduler type to use. See SchedulerType for all possible values.
- **`lr_scheduler_kwargs`** = `None`. The extra arguments for the lr_scheduler. See the documentation of each scheduler for possible values.
- **`warmup_steps`** = `0`. Number of steps for a linear warmup from 0 to learning_rate. Can be

an integer (exact steps) or a float in [0, 1) (ratio of total steps).

- **optim** = `'adamw_torch_fused'`. The optimizer to use. See OptimizerNames for the complete list.
- **optim_args** = `None`. Optional arguments supplied to optimizers such as AnyPrecisionAdamW, AdEMAMix, and GaLore.
- **weight_decay** = `0.0`. Weight decay coefficient applied by the optimizer. Automatically excluded from bias and LayerNorm parameters.
- **adam_beta1** = `0.9`. The exponential decay rate for the first moment estimates (momentum) in Adam-based optimizers.
- **adam_beta2** = `0.999`. The exponential decay rate for the second moment estimates (variance) in Adam-based optimizers.
- **adam_epsilon** = `1e-08`. Epsilon value for numerical stability in Adam-based optimizers.
- **optim_target_modules** = `None`. The target modules to optimize. Currently used for the GaLore and APOLLO algorithms.
- **gradient_accumulation_steps** = `1`. Number of update steps to accumulate gradients before performing a backward/update pass. Effective batch size = per_device_train_batch_size * num_devices * gradient_accumulation_steps.
- **average_tokens_across_devices** = `True`. Whether or not to average tokens across devices. If enabled, will use all_reduce to synchronize num_tokens_in_batch for precise loss calculation.
- **max_grad_norm** = `1.0`. Maximum gradient norm for gradient clipping. Set to 0 to disable.
- **label_smoothing_factor** = `0.0`. Label smoothing factor to prevent overconfidence. Zero means no smoothing.
- **bf16** = `False`. Enable bfloat16 (BF16) mixed precision training. Generally preferred over FP16 due to better numerical stability.
- **fp16** = `False`. Enable float16 (FP16) mixed precision training. Consider using BF16 instead if your hardware supports it.
- **bf16_full_eval** = `False`. Use full BF16 precision for evaluation (not just mixed precision). Faster and saves memory.
- **fp16_full_eval** = `False`. Use full FP16 precision for evaluation (not just mixed precision). Faster and saves memory.
- **tf32** = `None`. Enable TF32 mode on Ampere and newer GPUs. Provides up to 8x speedup with negligible accuracy loss.
- **gradient_checkpointing** = `False`. Enable gradient checkpointing to trade compute for memory. Reduces memory at the cost of ~20% slower training.
- **gradient_checkpointing_kwargs** = `None`. Keyword arguments passed to gradient_checkpointing_enable(). every_n_layers checkpoints only every n-th decoder layer instead of all of them; 1 is the usual all-or-nothing behavior, and larger values give some memory back to speed.
- **torch_compile** = `False`. Compile the model using torch.compile() for faster training.
- **torch_compile_backend** = `None`. Backend for torch.compile(). If set, automatically enables torch_compile.
- **torch_compile_mode** = `None`. Compilation mode for torch.compile(). If set, automatically enables torch_compile.
- **use_liger_kernel** = `False`. Enable Liger Kernel optimizations. Increases throughput by

~20% and reduces memory by ~60%.

- **liger_kernel_config** = None. Configuration for Liger Kernel. Passed as kwargs to _apply_liger_kernel_to_instance(). If None, uses default configuration.
- **use_cache** = False. Whether or not to use cache for the model For training, this is usually not needed apart from some PEFT methods that uses past_key_values.
- **neftune_noise_alpha** = None. If not None, activates NEFTune noise embeddings. Can drastically improve performance for instruction fine-tuning. Typical range: [5.0, 15.0].
- **torch_empty_cache_steps** = None. Number of steps to wait before calling torch.<device>.empty_cache(). Helps avoid CUDA OOM at a cost of ~10% slower performance. If None, cache will not be emptied.
- **auto_find_batch_size** = False. Whether to find a training batch size that will fit into memory automatically through exponential decay, avoiding CUDA Out-of-Memory errors. The evaluation batch size is not adjusted.
- **logging_strategy** = 'steps'. The logging strategy to adopt during training. Options: 'no', 'epoch', 'steps'.
- **logging_steps** = 500. Log every X updates steps. Should be an integer or a float in range [0,1). If smaller than 1, will be interpreted as ratio of total training steps.
- **logging_first_step** = False. Whether to log the first global_step or not.
- **log_on_each_node** = True. When doing a multinode distributed training, whether to log once per node or just once on the main node.
- **logging_nan_inf_filter** = True. Filter out NaN and Inf losses when logging. Does not affect gradient computation, only logging.
- **include_num_input_tokens_seen** = 'no'. Whether to track the number of input tokens seen. Must be one of [all, non_padding, no] or a boolean value which map to all or no
- **log_level** = 'passive'. Logging level for the main process. Options: 'debug', 'info', 'warning', 'error', 'critical', 'passive'.
- **log_level_replica** = 'warning'. Logging level for replica processes in distributed training. Same options as log_level.
- **disable_tqdm** = None. Disable tqdm progress bars. Defaults to True if log_level is warning or lower.
- **report_to** = 'none'. The list of integrations to report the results and logs to. Use 'all' for all installed integrations, 'none' for no integrations.
- **run_name** = None. An optional descriptor for the run. Notably used for trackio, wandb, mlflow comet and swanlab logging.
- **project** = 'huggingface'. The name of the project to use for logging. Currently, only used by Trackio.
- **trackio_space_id** = None. Hugging Face Space id for live Gradio-based Trackio logging (read/write Bucket access). Use 'username/reponame', 'orgname/reponame', or 'reponame' (current user's namespace).
- **trackio_bucket_id** = None. Optional HF Bucket id when using a Trackio Space; if unset, Trackio picks a default.
- **trackio_static_space_id** = None. Static read-only Trackio Space over the Bucket (stable model-card links). False: no static sync on Hub push and no freeze after training. None/str: allow static Space; for local-only logging, Hub push runs sync(static); after training, freeze runs only if trackio_space_id was set (Gradio Space). str sets explicit static Space id.

- **eval_strategy** = 'no'. When to run evaluation. Options: 'no', 'steps', 'epoch'.
- **eval_steps** = None. Number of update steps between evaluations if eval_strategy='steps'. Defaults to logging_steps if not set. Should be an integer or a float in range [0,1). If smaller than 1, will be interpreted as ratio of total training steps.
- **eval_delay** = 0. Number of epochs or steps to wait for before the first evaluation can be performed, depending on the eval_strategy.
- **per_device_eval_batch_size** = 8. The batch size per device (GPU/TPU core/CPU) for evaluation.
- **prediction_loss_only** = False. When performing evaluation and generating predictions, only returns the loss.
- **eval_on_start** = False. Whether to run through the entire evaluation step at the very beginning of training as a sanity check.
- **eval_do_concat_batches** = True. Whether to recursively concat inputs/losses/labels/predictions across batches. If False, will instead store them as lists, with each batch kept separate.
- **eval_use_gather_object** = False. Whether to run recursively gather object in a nested list/tuple/dictionary of objects from all devices.
- **eval_accumulation_steps** = None. Number of predictions steps to accumulate the output tensors for, before moving the results to the CPU. If unset, predictions are accumulated on the accelerator before being moved to the CPU.
- **include_for_metrics** = []. Include additional data in the compute_metrics function. Options: 'inputs', 'loss'.
- **batch_eval_metrics** = False. Break eval metrics calculation into batches to save memory.
- **save_only_model** = False. Save only model weights, not optimizer/scheduler/RNG state. Prevents resuming training from checkpoint.
- **save_strategy** = 'steps'. The checkpoint save strategy to adopt during training. Options: 'no', 'epoch', 'steps', 'best'.
- **save_steps** = 500. Save checkpoint every X updates steps. Should be an integer or a float in range [0,1). If smaller than 1, will be interpreted as ratio of total training steps.
- **save_on_each_node** = False. When doing multi-node distributed training, whether to save models and checkpoints on each node, or only on the main one
- **save_total_limit** = None. Maximum number of checkpoints to keep. Deletes older checkpoints in output_dir. The best checkpoint is always retained when load_best_model_at_end=True.
- **enable_jit_checkpoint** = False. Enable JIT checkpointing on SIGTERM signal for graceful termination on preemptible workloads. Configure your orchestrator's graceful shutdown period accordingly.
- **push_to_hub** = False. Whether or not to push the model to the Hub every time the model is saved.
- **hub_token** = None. The token to use to push the model to the Hub. Defaults to the token from hf auth login.
- **hub_private_repo** = None. Whether to make the repo private. If None (default), the repo will be public unless the organization's default is private. This value is ignored if the repo already exists.
- **hub_model_id** = None. The name of the repository to keep in sync with the local output_dir.

- **hub_strategy** = 'every_save'. Defines what and when to push to Hub. Options: 'end', 'every_save', 'checkpoint', 'all_checkpoints'.
- **hub_always_push** = False. Unless True, the Trainer will skip pushes if the previous one wasn't finished yet.
- **hub_revision** = None. The revision to use when pushing to the Hub. Can be a branch name, a tag, or a commit hash.
- **load_best_model_at_end** = False. Load the best checkpoint at the end of training. Requires eval_strategy to be set.
- **metric_for_best_model** = None. Metric to use for comparing models when load_best_model_at_end=True. Defaults to 'loss'.
- **greater_is_better** = None. Whether higher metric values are better. Defaults based on metric_for_best_model.
- **ignore_data_skip** = False. When resuming training, skip fast-forwarding through the dataset to reach the previous state. If True, training starts from the beginning of the dataset.
- **restore_callback_states_from_checkpoint** = False. Whether to restore the callback states from the checkpoint. If True, will override callbacks passed to the Trainer if they exist in the checkpoint.
- **full_determinism** = False. Whether to call enable_full_determinism instead of set_seed for reproducibility in distributed training. Important: this will negatively impact the performance, so only use it for debugging.
- **seed** = 42. Random seed that will be set at the beginning of training.
- **data_seed** = None. Random seed to be used with data samplers. If not set, uses the same seed as seed.
- **use_cpu** = False. Whether or not to use cpu. If set to False, we will use the available torch device/backend.
- **accelerator_config** = None. Configuration for the internal Accelerate integration. Can be a path to a JSON config file or a dict.
- **parallelism_config** = None. Parallelism configuration for the training run. Requires Accelerate 1.10.1.
- **dataloader_drop_last** = False. Drop the last incomplete batch if it is not divisible by the batch size.
- **dataloader_num_workers** = 0. Number of subprocesses to use for data loading (PyTorch only). 0 means that the data will be loaded in the main process.
- **dataloader_pin_memory** = True. Whether or not to pin memory for DataLoader.
- **dataloader_persistent_workers** = False. If True, the data loader will not shut down the worker processes after a dataset has been consumed once. This allows to maintain the workers Dataset instances alive.
- **dataloader_prefetch_factor** = None. Number of batches loaded in advance by each worker. 2 means there will be a total of 2 * num_workers batches prefetched across all workers.
- **dataloader_multiprocessing_context** = None. The multiprocessing start method to use for data loading workers ('fork', 'spawn', or 'forkserver'). Defaults to PyTorch's default, except on MPS with dataloader_num_workers > 1, where it defaults to 'fork'.
- **dataloader_in_order** = True. If True (default), the data loader yields batches in the order workers were dispatched. Set to False to yield batches as soon as they are ready, reducing

tail latency for IterableDataset streaming workloads.

- **`remove_unused_columns`** = `True`. Whether or not to automatically remove the columns unused by the model forward method.
- **`label_names`** = `None`. The list of keys in your dictionary of inputs that correspond to the labels.
- **`train_sampling_strategy`** = `'random'`. Sampler for training: 'random' (default), 'sequential', 'group_by_length', or 'batch_rebalance'.
- **`length_column_name`** = `'length'`. Column name for precomputed lengths. Ignored unless train_sampling_strategy is 'group_by_length' or 'batch_rebalance'.
- **`ddp_find_unused_parameters`** = `None`. When using distributed training, the value of the flag find_unused_parameters passed to DistributedDataParallel.
- **`ddp_bucket_cap_mb`** = `None`. When using distributed training, the value of the flag bucket_cap_mb passed to DistributedDataParallel.
- **`ddp_broadcast_buffers`** = `None`. When using distributed training, the value of the flag broadcast_buffers passed to DistributedDataParallel.
- **`ddp_static_graph`** = `None`. When using distributed training, the value of the flag static_graph passed to DistributedDataParallel.
- **`ddp_backend`** = `None`. The backend to use for distributed training. Must be one of 'nccl', 'mpi', 'xccl', 'gloo', 'hccl'.
- **`ddp_timeout`** = `1800`. The timeout for torch.distributed.init_process_group calls (in seconds).
- **`fsdp`** = `None`. Enable PyTorch Fully Sharded Data Parallel (FSDP) for distributed training. Pass –fsdp (or fsdp=True) to turn FSDP on.
- **`fsdp_config`** = `None`. Tuning for FSDP (used only when fsdp is enabled). Either a path to a JSON config file (e.g., fsdp_config.json) or an already loaded dict.
- **`deepspeed`** = `None`. Enable DeepSpeed integration. Value is a path to a JSON config file or a dict.
- **`debug`** = `''`. Enable one or more debug features. Options: 'underflow_overflow' (detect overflow in model I/O), 'tpu_metrics_debug' (print TPU metrics).
- **`skip_memory_metrics`** = `True`. Whether to skip adding memory profiler reports to metrics. Skipped by default because it slows down training.
- **`do_train`** = `False`. Whether to run training. Not directly used by Trainer; intended for training/evaluation scripts.
- **`do_eval`** = `False`. Whether to run evaluation. Not directly used by Trainer; intended for training/evaluation scripts.
- **`do_predict`** = `False`. Whether to run predictions on the test set. Not directly used by Trainer; intended for training/evaluation scripts.
- **`resume_from_checkpoint`** = `None`. Path to a folder with a valid checkpoint for your model. Not directly used by Trainer; intended for training/evaluation scripts.
- **`local_rank`** = `-1`. When using torch.distributed.launch (Deprecated), it will pass local_rank in the script, so we need this for the parser. To get the local rank, prefer using the property local_process_index

CHAPTER 46

# PEFT LoraConfig and BitsAndBytesConfig

**Why this chapter matters.**
LoRA and QLoRA (Chapters 9 and 10) are how most people fine-tune on limited hardware, and every RL framework supports LoRA actors. Interview questions go beyond "what is LoRA": what does `lora_alpha` do, why `all-linear`, what is rsLoRA, what does the default initialisation guarantee, what are NF4 and double quantisation, and what is the default `bnb_4bit_quant_type`? (It is `fp4`, not `nf4`, which surprises people.) This chapter covers `peft.LoraConfig` (PEFT 0.21.2) and `transformers.BitsAndBytesConfig` (Transformers 5.20.2).

## 46.1 LoraConfig: the parameters that matter

A LoRA adapter replaces a frozen linear map $W_0 x$ with $W_0 x + s\,BAx$, where $A \in \mathbb{R}^{r \times d_{\text{in}}}$, $B \in \mathbb{R}^{d_{\text{out}} \times r}$ and $s$ is a scale (Chapter 9).

- **`r`** (default 8): the rank. Parameters added per wrapped layer: $r(d_{\text{in}} + d_{\text{out}})$. Typical values: 8–64 for SFT; RL often works with small ranks, because RL updates carry little information per step.
- **`lora_alpha`** (default 8): sets the scale $s = \alpha/r$. With the default `use_rslora=False`, doubling $r$ at fixed $\alpha$ halves the scale. A common habit is $\alpha = 2r$ (scale 2) or $\alpha = r$ (scale 1).
- **`use_rslora`** (False): rank-stabilised LoRA uses $s = \alpha/\sqrt{r}$, which keeps update size stable as rank grows; use it when sweeping or using large ranks.
- **`target_modules`** (None): which modules to wrap. `None` falls back to a per-architecture default (attention projections for many models). The string `"all-linear"` wraps every linear layer except the output head; a list of names (for example `["q_proj", "v_proj"]`) or a regex string selects precisely. Research summarised in Chapter 9 found that wrapping all linear layers, including MLPs, matters most for matching full fine-tuning.
- **`exclude_modules`**: names or regex to skip.
- **`lora_dropout`** (0.0): dropout on the adapter input. Small values (0.05) can help SFT on small data; keep 0 for RL (dropout makes log-probabilities noisy, which breaks ratio computations).
- **`bias`** (`"none"`): train no biases, `"all"`, or `"lora_only"`.
- **`modules_to_save`**: extra modules trained fully and saved with the adapter (for example a reward model's `score` head, or `embed_tokens/lm_head` when adding new tokens). `trainable_token_indices` trains only selected embedding rows instead of the whole matrix.
- **`init_lora_weights`** (True): the default sets $B = 0$ (and $A$ random), so the adapter starts as an exact no-op. Other values: `False` (random, for debugging), `"gaussian"`, `"pissa"` / `"pissa_niter_K"` (initialise from principal singular vectors of $W_0$), `"olora"`, `"eva"` (data-driven), `"corda"`, `"loftq"` (initialise to reduce quantisation error for QLoRA), and more.

**Notation on this page:** $x$: the prompt (input) • $W$: a weight matrix • $s$: a state • $B$: batch size • $r(x, y)$: the reward for response $y$ to prompt $x$ • $d$: hidden size of the model • $\alpha$: a weighting coefficient (defined in the text)

- **use_dora** (False): DoRA splits weights into magnitude and direction and adapts both; slightly better quality at some speed cost.
- **layers_to_transform**, **layers_pattern**: adapt only some layers. **rank_pattern**, **alpha_pattern**: per-module overrides.
- **task_type**: for example `"CAUSAL_LM"` or `"SEQ_CLS"`; tells PEFT which model wrapper to use (and, for sequence classification, to keep the head trainable).
- **target_parameters**: wrap `nn.Parameters` directly, used for mixture-of-experts layers whose experts are stored as one 3D parameter rather than many `nn.Linear` modules.

```
import os, warnings
warnings.filterwarnings("ignore"); os.environ["HF_HUB_OFFLINE"] = "1"
import torch
from transformers import Qwen2Config, Qwen2ForCausalLM
from peft import LoraConfig, get_peft_model
torch.manual_seed(0)
cfg = Qwen2Config(vocab_size=1000, hidden_size=128, intermediate_size=256,
 ↪  num_hidden_layers=2,
                  num_attention_heads=4, num_key_value_heads=2)
base = Qwen2ForCausalLM(cfg)
x = torch.randint(0, 1000, (2, 12))
with torch.no_grad():
    ref = base(x).logits
lcfg = LoraConfig(r=8, lora_alpha=16, target_modules="all-linear",
 ↪  task_type="CAUSAL_LM")
model = get_peft_model(base, lcfg)
wrapped = sorted({n.split(".")[-1] for n, m in model.named_modules() if hasattr(m,
 ↪  "lora_A")})
print("wrapped modules:", wrapped, "(lm_head is not wrapped by 'all-linear')")
# expected parameter count: r * (d_in + d_out) per wrapped linear
h, kv, ff, r = 128, 2 * (128 // 4), 256, 8
per_layer = r * ((h + h) + 2 * (h + kv) + (h + h) + 2 * (h + ff) + (ff + h))
trainable = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("trainable:", trainable, "| formula:", per_layer * cfg.num_hidden_layers)
q = model.base_model.model.model.layers[0].self_attn.q_proj
print("scale alpha/r =", q.scaling["default"], "| rsLoRA scale would be", 16 / 8 ** 0.5)
with torch.no_grad():
    print("B = 0 at init, so outputs are unchanged:", torch.allclose(model(x).logits,
     ↪  ref))
    for n, p in model.named_parameters():          # pretend training moved B away from
     ↪  zero
        if "lora_B" in n:
            p.normal_(0, 0.02)
    adapted = model(x).logits
    merged = model.merge_and_unload()              # fold s*B@A into W0 and drop the
 ↪  adapter modules
    print("merged model matches adapter model:", torch.allclose(merged(x).logits,
     ↪  adapted, atol=1e-5))
```

**Why this, not that? Why scale by alpha over r, not use the raw product BA?**

**Reason.** With $s = \alpha/r$, the size of the initial updates does not depend much on the rank, so you can change $r$ without re-tuning the learning rate (in the original LoRA analysis). **Trade-off.** At large ranks $\alpha/r$ shrinks the update too much; learning slows. That is the motivation for rsLoRA's $\alpha/\sqrt{r}$. **When the alternative is better.** Rank sweeps or ranks above about 64: set `use_rslora=True`. At small fixed ranks the choice barely matters once the learning rate is tuned.

## 46.2 BitsAndBytesConfig: 4-bit and 8-bit loading

`BitsAndBytesConfig` (Transformers 5.20.2) tells `from_pretrained` to quantise weights while loading. The constructor parameters and defaults are generated below; the important ones:

- **`load_in_4bit` / `load_in_8bit`** (both False): choose one.
- **`bnb_4bit_quant_type`** (default `"fp4"`): the 4-bit data type. **Set `"nf4"` for QLoRA**: NormalFloat-4 places its 16 levels at quantiles of a normal distribution, which matches the bell-shaped distribution of trained weights better than FP4 (Chapter 11).
- **`bnb_4bit_compute_dtype`** (None, meaning FP32): the type weights are dequantised to for matrix multiplication. Set `torch.bfloat16` for speed; leaving the default makes 4-bit models slow.
- **`bnb_4bit_use_double_quant`** (False): also quantise the per-block scaling constants (about 0.37 bits per parameter saved in the QLoRA paper's accounting). Turn it on for QLoRA.
- **`bnb_4bit_quant_storage`**: the container dtype for packed 4-bit weights; must match across shards when combining QLoRA with FSDP.
- **`llm_int8_threshold`** (6.0): in 8-bit mode (LLM.int8), activation outlier features above this magnitude are processed in 16-bit.
- **`llm_int8_skip_modules`**: modules left unquantised (for example the output head).

The QLoRA recipe is therefore: `BitsAndBytesConfig(load_in_4bit=True, bnb_4bit_quant_type="nf4", bnb_4bit_use_double_quant=True, bnb_4bit_compute_dtype=torch.bfloat16)` plus a LoRA config on all linear layers. `peft.prepare_model_for_kbit_training(model)` then casts norms to FP32 and enables gradient checkpointing and input gradients. bitsandbytes needs a GPU (or one of its newer CPU/XPU backends); the book's CPU environment cannot run 4-bit loading, so the code below only constructs and checks the config object.

```
from transformers import BitsAndBytesConfig
default = BitsAndBytesConfig(load_in_4bit=True)
qlora = BitsAndBytesConfig(load_in_4bit=True, bnb_4bit_quant_type="nf4",
 ↪  bnb_4bit_use_double_quant=True,
                            bnb_4bit_compute_dtype=torch.bfloat16)
for name, c in [("default", default), ("QLoRA", qlora)]:
    print(f"{name:8s} quant_type={c.bnb_4bit_quant_type}
     ↪  double_quant={c.bnb_4bit_use_double_quant} "
          f"compute_dtype={c.bnb_4bit_compute_dtype}")
```

**Why this, not that? Why QLoRA, not plain LoRA on 16-bit weights?**

**Reason.** 4-bit base weights cut the frozen-model memory about 4×, so a 70B model's base fits on a single 48–80 GB GPU for fine-tuning (Chapter 40's calculator: about 10 GB instead of 22 GB for an 8B model). **Trade-off.** Dequantisation makes each step slower (often 30% or more), quality can drop slightly, and merged weights must be re-quantised or kept in 16-bit for serving. **When the alternative is better.** When the 16-bit model fits, plain LoRA is faster and simpler; in RL, where the same weights also serve rollouts in vLLM, 16-bit LoRA avoids quantisation mismatch between trainer and sampler.

## 46.3 Common mistakes

1. Leaving `bnb_4bit_quant_type` at the default `fp4` when following a QLoRA recipe.
2. Leaving `bnb_4bit_compute_dtype` unset (slow FP32 matmuls).
3. `target_modules` left at the attention-only default when the recipe expected all linear layers.
4. Changing `r` without changing `lora_alpha` (silently changes the update scale).
5. LoRA dropout during RL.
6. Forgetting `modules_to_save` for a new classification head or resized embeddings.

**Interview check**

**Q1. How many parameters does LoRA add to one linear layer?** $r(d_{\text{in}} + d_{\text{out}})$.
**Q2. What does `lora_alpha` do?** Scales the update by $\alpha/r$ (or $\alpha/\sqrt{r}$ with rsLoRA).
**Q3. Why does a LoRA model match the base model before training?** The default initialisation sets $B = 0$, so $BA = 0$.
**Q4. What is NF4 and is it the default?** A 4-bit type with levels at normal quantiles; the default `bnb_4bit_quant_type` is `fp4`, so NF4 must be requested.
**Q5. What is double quantisation?** Quantising the per-block scale constants themselves to save about 0.37 bits per parameter.
**Q6. What does `merge_and_unload` do?** Adds $sBA$ into $W_0$ and removes the adapter modules, giving a plain model with no inference overhead.

**Key takeaways**

- LoRA: `r`, `lora_alpha` (scale $\alpha/r$), `target_modules="all-linear"`, `init_lora_weights=True` gives a no-op start.
- rsLoRA for large ranks; DoRA for a small quality gain; `modules_to_save` for heads and embeddings.
- QLoRA: `nf4` + double quant + bf16 compute (none of these is the default).

## 46.4 Full parameter list: LoraConfig

*Generated from the installed source (transformers 5.20.2, peft 0.21.2, trl 1.16.0); 42 entries.*

- **`task_type`** = `None`. Task type

**Notation on this page:** $r(x, y)$: the reward for response $y$ to prompt $x$ • $d$: hidden size of the model • $\alpha$: a weighting coefficient (defined in the text) • $B$: batch size • $W$: a weight matrix

- **peft_type** = None. Peft type
- **auto_mapping** = None. An auto mapping dict to help retrieve the base model class if needed.
- **peft_version** = None. PEFT version, leave empty to auto-fill.
- **base_model_name_or_path** = None. The name of the base model to use.
- **revision** = None. The specific base model version to use.
- **inference_mode** = False. Whether to use inference mode
- **r** = 8. Lora attention dimension
- **target_modules** = None. List of module names of the module names to replace with LoRA. For example, ['q', 'v'] or '.*decoder.*(SelfAttention|EncDecAttention).*(q|v)$'. This can also be a wildcard 'all-linear' which matches all linear/Conv1D (if the model is a PreTrainedModel, the output layer excluded).
- **exclude_modules** = None. List of module names or regex expression of the module names to exclude from Lora.
- **lora_alpha** = 8. Lora alpha
- **lora_dropout** = 0.0. Lora dropout
- **fan_in_fan_out** = False. Set this to True if the layer to replace stores weight like (fan_in, fan_out)
- **bias** = 'none'. Bias type for Lora. Can be 'none', 'all' or 'lora_only'
- **use_rslora** = False. When set to True, uses Rank-Stabilized LoRA which sets the adapter scaling factor to lora_alpha/math.sqrt(r), since it was proven to work better. Otherwise, it will use the original default value of lora_alpha/r.
- **modules_to_save** = None. List of modules apart from LoRA layers to be set as trainable and saved in the final checkpoint. For example, in Sequence Classification or Token Classification tasks, the final layer classifier/score are randomly initialized and as such need to be trainable and saved.
- **init_lora_weights** = True. How to initialize the weights of the LoRA layers. Passing True (default) results in the default initialization from the reference implementation from Microsoft, with the LoRA B weight being set to 0.
- **layers_to_transform** = None. The layer indexes to transform, is this argument is specified, PEFT will transform only the layers indexes that are specified inside this list. If a single integer is passed, PEFT will transform only the layer at this index.
- **layers_pattern** = None. The layer pattern name, used only if layers_to_transform is different to None and if the layer pattern is not in the common layers pattern.This only works when target_modules is a list of str. This should target the nn.ModuleList of the model, which is often called 'layers' or 'h'.
- **rank_pattern** = {}. The mapping from layer names or regexp expression to ranks which are different from the default rank specified by r. For example, {'^model.decoder.layers.0.encoder_attn.k_proj': 16}.
- **alpha_pattern** = {}. The mapping from layer names or regexp expression to alphas which are different from the default alpha specified by lora_alpha. For example, {'^model.decoder.layers.0.encoder_attn.k_proj': 16}.
- **megatron_config** = None. The TransformerConfig from Megatron.
- **megatron_core** = 'megatron.core'. The core module from Megatron, it is used to create LoRA's parallel linear layer. It only needs to be passed in when you need to use your own modified megatron core module.

- **trainable_token_indices** = None. Lets you specify which token indices to selectively fine-tune without requiring to re-train the whole embedding matrix using the peft. TrainableTokensModel method. You can specify token indices in two ways.
- **loftq_config** = {}. The configuration of LoftQ. If this is passed, then LoftQ will be used to quantize the backbone weights and initialize Lora layers. Also set init_lora_weights='loftq' in this case.
- **eva_config** = None. The configuration of EVA. If this is passed, then EVA will be used to initialize the LoRA layers. Also set init_lora_weights='eva' in this case.
- **corda_config** = None. The configuration of CorDA. If this is passed, then CorDA will be used to build the adapter layers. Also set init_lora_weights='corda' in this case.
- **lora_ga_config** = None. The configuration of LoRA-GA. If this is passed, then LoRA-GA will be used to initialize the adapter layers. Also set init_lora_weights='lora_ga' in this case.
- **use_dora** = False. Enable <a href=' Low-Rank Adaptation' (DoRA)</a>. This technique decomposes the updates of the weights into two parts, magnitude and direction. Direction is handled by normal LoRA, whereas the magnitude is handled by a separate learnable parameter.
- **velora_config** = None. Enable VeLoRA as a LoRA variant by providing a VeloraConfig. VeLoRA swaps in a custom backward pass for the LoRA A projection that stores compressed activations instead of the full input activations.
- **alora_invocation_tokens** = None. If not None, enable <a href=' LoRA' (aLoRA)</a>, with alora_invocation_tokens being the tokenized invocation string for the adapter (must be present in all model input strings). This technique selectively activates the adapter weights only on tokens during and after the alora_invocation_tokens.
- **use_qalora** = False. It is only implemented in GPTQ for now. Enable <a href=' Low-Rank Adaptation (QALoRA)</a>.This technique combines quantization-aware training with LoRA to improve performance for quantized models.
- **qalora_group_size** = 16. Group size parameter for QALoRA pooling, controlling the dimension reduction factor. Input dimensions are pooled into groups of this size, reducing the computational cost.
- **monteclora_config** = None. The configuration of Monteclora (Monte Carlo Low-Rank Adaptation). If passed, Monteclora will be used to add variational Monte Carlo sampling on top of the LoRA adapters.
- **layer_replication** = None. This enables using LoRA to effectively expand a transformer model to a larger size by repeating some layers. The transformation handles models (currently Llama, Bert or Falcon compatible architectures) with a module list in the model which it modifies to expand the number of modules.
- **runtime_config** = LoraRuntimeConfig(ephemeral_gpu_offload=False). Runtime configurations
- **lora_bias** = False. Whether to enable the bias term for the LoRA B parameter. Typically, this should be disabled. The main use case for this is when the LoRA weights were extracted from fully fine-tuned parameters so the bias of those parameters can be taken into account.
- **target_parameters** = None. List of parameter names or regex expression of the parameter names to replace with LoRA. This argument behaves similarly to target_modules, except that the parameter name should be passed.

- **use_bdlora** = None. Enable BD-LoRA (Block-Diagonal LoRA) by providing a BdLoraConfig. This technique uses block-diagonal matrices for LoRA-A or LoRA-B factors to enable faster multi-LoRA serving by eliminating communication overheads in distributed settings.
- **arrow_config** = None. The necessary config to apply arrow routing on the model.
- **kasa_config** = None. Enable KaSA (Knowledge-aware Singular-value Adaptation) as a LoRA variant by providing a KasaConfig.
- **ensure_weight_tying** = False. Whether to tie weights or not after peft initialization. This will ensure that the adapters added to the tied layers are also tied. This is only applicable for layers passed via modules_to_save, target_modules and trainable_token_indices.

## 46.5 Full parameter list: BitsAndBytesConfig constructor

*Generated from the installed source (transformers 5.20.2, peft 0.21.2, trl 1.16.0); 10 entries.*

- **load_in_8bit** = False
- **load_in_4bit** = False
- **llm_int8_threshold** = 6.0
- **llm_int8_skip_modules** = None
- **llm_int8_enable_fp32_cpu_offload** = False
- **llm_int8_has_fp16_weight** = False
- **bnb_4bit_compute_dtype** = None
- **bnb_4bit_quant_type** = 'fp4'
- **bnb_4bit_use_double_quant** = False
- **bnb_4bit_quant_storage** = None

CHAPTER 47

# TRL SFTConfig and SFTTrainer

**Why this chapter matters.**
`SFTTrainer` is the most used post-training tool. Its behaviour depends on the dataset format as much as on the config: plain text, prompt–completion pairs and conversations are each handled differently, and that decides which tokens are trained on. Knowing the defaults (TRL 1.16.0: learning rate 2e-5, `max_length` 1024, packing off, completion-only loss on for prompt–completion data, assistant-only loss off for conversations) prevents the classic bug of training on the prompt.

## 47.1 Dataset formats decide the loss

TRL recognises four shapes of dataset row:

| Format | Columns | Loss on |
|---|---|---|
| Language modelling | `text` (name set by `dataset_text_field`) | every token |
| Conversational LM | `messages` (list of `{"role", "content"}`) | every token, unless `assistant_only_loss=True` |
| Prompt–completion | `prompt`, `completion` (strings) | completion only by default |
| Conversational prompt–completion | `prompt`, `completion` (message lists) | completion only by default |

`completion_only_loss` defaults to `None`, which means "True when the dataset is prompt–completion". `assistant_only_loss=True` masks everything but assistant turns in conversational data; it needs a chat template that marks assistant spans with `{% generation %}` … `{% endgeneration %}` tags. If the tokenizer's template lacks them, TRL 1.16 swaps in a built-in training template for supported model families, and it warns when the assistant's end-of-turn token falls outside the mask (the model would then never learn to stop, while the loss still looks healthy). Masked positions get label −100, which the cross-entropy loss ignores (Chapter 8).

```
import os, warnings
warnings.filterwarnings("ignore"); os.environ["HF_HUB_OFFLINE"] = "1"
import torch, transformers, datasets
transformers.logging.set_verbosity_error(); datasets.disable_progress_bars()
from transformers import AutoTokenizer, Qwen2Config, Qwen2ForCausalLM, PrinterCallback
from datasets import Dataset
```

```
from trl import SFTConfig, SFTTrainer
tok = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B-Instruct")
def tiny_model(seed=0):
    torch.manual_seed(seed)
    return Qwen2ForCausalLM(Qwen2Config(vocab_size=len(tok), hidden_size=64,
     ↪  intermediate_size=128, num_hidden_layers=2,
                                        num_attention_heads=4, num_key_value_heads=2,
 ↪  tie_word_embeddings=True))
rows = [{"prompt": [{"role": "user", "content": f"What is {a} plus {b}?"}],
         "completion": [{"role": "assistant", "content": f"{a} plus {b} is {a + b}."}]}
          ↪  for a in range(6) for b in range(4)]
ds = Dataset.from_list(rows)
args = SFTConfig(output_dir="/tmp/ref47", use_cpu=True, bf16=False,
 ↪  per_device_train_batch_size=4, max_steps=3,
                 report_to="none", save_strategy="no", logging_steps=1,
 ↪  disable_tqdm=True, gradient_checkpointing=False)
trainer = SFTTrainer(model=tiny_model(), args=args, train_dataset=ds,
 ↪  processing_class=tok)
trainer.remove_callback(PrinterCallback)
print("completion_only_loss setting:", trainer.args.completion_only_loss, "(None =
 ↪  automatic for prompt-completion data)")
ex = trainer.train_dataset[0]
print("masked (label -100):", repr(tok.decode([t for t, l in zip(ex["input_ids"],
 ↪  ex["labels"]) if l == -100])[-62:]))
print("trained on          :", repr(tok.decode([t for t, l in zip(ex["input_ids"],
 ↪  ex["labels"]) if l != -100])))
batch = next(iter(trainer.get_train_dataloader()))
print("batch keys:", sorted(batch), "| fraction of positions masked:",
 ↪  round((batch["labels"] == -100).float().mean().item(), 2))
trainer.train()
print("loss per step:", [round(h["loss"], 3) for h in trainer.state.log_history if
 ↪  "loss" in h],
      "| ln(vocab) =", round(float(torch.log(torch.tensor(len(tok)))), 3))
```

The tiny random model starts at a loss near $\ln|V|$, as Chapter 3 predicts for a uniform guess. More important is what is trained: the chat-template prompt is masked, and the assistant's answer *and its end-of-turn token* are trained. Learning to emit `<|im_end|>` is what teaches the model to stop.

## 47.2 Packing and padding-free training

Short examples waste compute on padding. Two remedies:

- **`packing=True`** concatenates examples into blocks of `max_length` tokens. `packing_strategy` (default `"bfd"`, best-fit decreasing) places whole examples into blocks like a bin-packing algorithm and truncates any example longer than a block; `"bfd_split"` splits long examples across blocks instead; `"wrapped"` simply cuts the token stream every `max_length` tokens (fast, but splits examples mid-sentence). With `bfd`, the trainer records each example's length (`seq_lengths`) and uses position IDs that restart per example, so that with a FlashAttention-style kernel attention does not cross example boundaries.

**Notation on this page:** $V$: vocabulary size

- **`padding_free=True`** flattens each batch into one sequence without padding (also via position IDs), without fixing block lengths.

```
packed = SFTTrainer(model=tiny_model(), processing_class=tok, train_dataset=ds,
                    args=SFTConfig(output_dir="/tmp/ref47b", use_cpu=True, bf16=False,
 ↪  packing=True, max_length=256,
                                   report_to="none", disable_tqdm=True))
print("rows before packing:", len(ds), "| packed 256-token rows:",
 ↪  len(packed.train_dataset),
      "| columns:", packed.train_dataset.column_names)
print("lengths inside first packed row:", packed.train_dataset[0]["seq_lengths"])
```

**Why this, not that? Why best-fit-decreasing packing, not wrapped packing?**

**Reason.** BFD keeps each example whole and, with per-example position IDs, prevents attention across example boundaries; training sees the same examples it would without packing, just with less padding. **Trade-off.** Blocks are not perfectly full (a little padding remains), examples longer than `max_length` are truncated (use `bfd_split` to keep the overflow), and correct isolation needs an attention implementation that honours position-ID boundaries (FlashAttention 2/3 or SDPA variants with varlen support). **When the alternative is better.** Continued pretraining on long documents, where examples are just stretches of text and crossing boundaries is harmless: `wrapped` is simplest and fills every block.

## 47.3 The other parameters that matter

- **`max_length`** (1024): longer sequences are truncated; `truncation_mode` supports only `"keep_start"` in TRL 1.16. Check your token-length distribution first; silently truncated answers teach the model to stop mid-sentence (or never stop, if the end-of-turn token is cut).
- **`learning_rate`** (2e-5): suitable for full fine-tuning of small and mid-sized models; use about 1e-4 to 2e-4 with LoRA.
- **`loss_type`** (`"nll"`): or `"dft"`, Dynamic Fine-Tuning (*On the Generalization of SFT: A Reinforcement Learning Perspective with Reward Rectification*, arXiv 2508.05629), which scales each token's loss by its (detached) probability, $-\mathrm{sg}(p_\theta(y_t)) \log p_\theta(y_t)$. The paper argues standard SFT acts like a policy gradient with an implicit reward weighted by $1/p$, which over-weights unlikely tokens; DFT cancels that weight.
- **`eos_token`**, **`chat_template_path`**: control the end-of-turn token and chat template when the tokenizer's defaults are wrong.
- **`dataset_text_field`** (`"text"`), **`dataset_num_proc`**, **`dataset_kwargs`** (`{"skip_prepare_dataset": True}` when you tokenise yourself).
- **`activation_offloading`** (False): move activations to CPU during the forward pass to save GPU memory.
- **`gradient_checkpointing`** (True in all TRL configs).

**Notation on this page:** $\theta$: the model's trainable weights • $\mathrm{sg}(\cdot)$: stop-gradient (treated as a constant in backprop) • $y$: the response (output tokens); $y_t$ is token $t$ • $t$: time step or token position • $p$: a probability distribution (often the target or data)

**Why this, not that? Why completion-only loss, not loss on every token?**

**Reason.** Prompts are inputs, not behaviour to imitate; training on them spends capacity on modelling user text (and system prompts repeated thousands of times), and can teach the model to write the user's side of a conversation. **Trade-off.** Fewer tokens contribute gradient, so each example teaches less; for very short answers, the signal per batch is small. **When the alternative is better.** Continued pretraining or domain adaptation, where all text is target text; and some studies found that a small loss weight on prompts helps when completions are very short. TRL's default only applies completion-only loss to prompt–completion data, so plain text datasets are trained on everything.

## 47.4 How the loss is averaged

A batch contains examples of different lengths. There are two natural ways to average the per-token losses $\ell_{i,t}$ over $N$ examples with $T_i$ trained tokens each:

$$\mathcal{L}_{\text{token}} = \frac{\sum_i \sum_t \ell_{i,t}}{\sum_i T_i}, \qquad \mathcal{L}_{\text{seq}} = \frac{1}{N} \sum_i \frac{1}{T_i} \sum_t \ell_{i,t}.$$

The **token mean** gives every trained token the same weight, so long answers dominate. The **sequence mean** gives every example the same weight, so each token of a short answer counts more. Transformers and TRL use the token mean, and they keep it correct across gradient accumulation and devices: the trainer counts the trained tokens in the whole accumulated batch (`num_items_in_batch`) and, with `average_tokens_across_devices` (default True), across GPUs, then divides the summed loss by that count. Without this, averaging per micro-batch would silently over-weight micro-batches with few trained tokens (a bug that was fixed in Transformers in late 2024).

```
lengths = torch.tensor([5, 50, 400])                  # trained tokens in three answers
per_token_loss = torch.tensor([2.0, 1.0, 0.5])        # mean loss of each answer
tok_mean = (per_token_loss * lengths).sum() / lengths.sum()
seq_mean = per_token_loss.mean()
w_tok = lengths / lengths.sum()
print(f"token mean {tok_mean:.3f} | sequence mean {seq_mean:.3f}")
print("weight of each example under token mean:", [round(w, 3) for w in w_tok.tolist()],
 ↪  "| under sequence mean: 0.333 each")
# micro-batch averaging bug: two micro-batches of one example each, averaged per
 ↪  micro-batch then across them
naive = (per_token_loss[0] + per_token_loss[2]) / 2
correct = (per_token_loss[0] * 5 + per_token_loss[2] * 400) / 405
print(f"accumulating [5-token, 400-token] micro-batches: naive mean {naive:.3f} vs
 ↪  correct token mean {correct:.3f}")
print("keys logged by SFTTrainer:", sorted(k for k in trainer.state.log_history[0] if k
 ↪  not in ("epoch", "step")))
```

The 5-token answer gets about 1% of the weight under the token mean but a third under the sequence mean. The same distinction appears in RL as the choice between `loss_type="grpo"`

**Notation on this page:** $t$: time step or token position • $N$: number of examples • $T$: number of steps or tokens in a sequence • $\mathcal{L}$: a loss to minimise

(per-sequence normalisation) and `"dapo"` or `"dr_grpo"` (token-level), with the same length effects (Chapter 30). The logged keys show what TRL tracks for SFT: besides the loss and learning rate, the mean entropy of the predictions, the number of tokens seen and the mean token accuracy (the fraction of trained positions where the most likely token equals the label). A rising token accuracy with a falling entropy is the normal pattern; entropy collapsing to near zero early often means the model is memorising a small dataset.

**Why this, not that? Why average the SFT loss over tokens, not over sequences?**

**Reason.** The token mean is the maximum-likelihood objective for the whole dataset: each target token is one observation. It gives an unbiased gradient regardless of how examples are grouped into micro-batches. **Trade-off.** Long examples dominate the gradient; a dataset with a few very long answers can be steered by them, and short, high-value answers (refusals, one-line facts) get little weight. **When the alternative is better.** When every example should count equally (for example, a balanced mix of short and long tasks, or preference-like data where each example is one decision), use per-sequence averaging or re-weight examples; capping or bucketing lengths is another option.

## 47.5 A complete LoRA SFT script

Putting the defaults and recommendations together (all names checked against TRL 1.16.0 and PEFT 0.21.2):

```
from datasets import load_dataset
from peft import LoraConfig
from trl import SFTConfig, SFTTrainer

ds = load_dataset("json", data_files="train.jsonl", split="train")   # rows:
 ↪ {"messages": [...]}
args = SFTConfig(
    output_dir="out/sft-lora",
    learning_rate=2e-4,                 # LoRA needs a higher rate than the 2e-5 default
    num_train_epochs=2,
    per_device_train_batch_size=8,
    gradient_accumulation_steps=4,
    lr_scheduler_type="cosine", warmup_steps=0.03,   # a float below 1 is a ratio of
 ↪ total steps
    max_length=2048,                    # check the length distribution first
    assistant_only_loss=True,           # conversations: train on assistant turns only
    packing=True,                       # bfd packing by default
    bf16=True,
    logging_steps=10, eval_strategy="steps", eval_steps=200,
)
peft_config = LoraConfig(r=16, lora_alpha=32, lora_dropout=0.05,
 ↪ target_modules="all-linear", task_type="CAUSAL_LM")
trainer = SFTTrainer(model="Qwen/Qwen2.5-1.5B-Instruct", args=args, train_dataset=ds,
 ↪ peft_config=peft_config)
trainer.train()
```

```
trainer.save_model()                    # saves the adapter and the tokenizer
```

Before launching, decode one processed example and confirm which tokens carry labels (as in the first snippet of this chapter). After training, compare a few generations with the base model on held-out prompts and check that answers end with the end-of-turn token.

## 47.6 Common mistakes

1. Conversational `messages` data with no `assistant_only_loss`: the model trains on user turns too.
2. `assistant_only_loss=True` with a custom chat template that lacks generation tags or leaves the end-of-turn token outside them (read the warning; inspect one masked example).
3. Truncated answers from a too-small `max_length`.
4. Packing with an attention implementation that ignores position-ID boundaries (examples attend to each other).
5. Changing the chat template between SFT and later stages or serving.

- Averaging the loss per micro-batch in a custom training loop with gradient accumulation (over-weights micro-batches with few trained tokens).
- Using the 2e-5 default learning rate with LoRA (training barely moves); or 2e-4 for full fine-tuning (it diverges or forgets).

**Interview check**

**Q1. Which tokens does SFTTrainer train on for prompt–completion data?** Only the completion (and its end-of-turn token), because `completion_only_loss=None` resolves to True for that format.
**Q2. How do you train only on assistant turns in multi-turn data?** `assistant_only_loss=True`, with a chat template whose generation tags cover each assistant turn including its end-of-turn token.
**Q3. What does packing do, and what is the risk?** Concatenates examples into fixed-length blocks; risk is cross-example attention unless position IDs restart and the kernel respects them.
**Q4. What is DFT?** A loss that weights each token's negative log-likelihood by its detached probability, motivated by viewing SFT as a policy gradient with an implicit $1/p$ weight.
**Q5. Why is training on the end-of-turn token important?** It teaches the model to stop; without it the model rambles past the answer.
**Q6. Token-mean or sequence-mean loss: which does SFTTrainer use and what is the effect?** Token mean across the whole accumulated batch (and devices); long answers carry more weight than short ones.
**Q7. What was the gradient-accumulation loss bug?** Averaging the loss within each micro-batch and then across micro-batches gives micro-batches with few trained tokens too much weight; the fix divides the summed loss by the total trained-token count (`num_items_in_batch`).
**Q8. What does `mean_token_accuracy` measure?** The fraction of trained positions where the

**Notation on this page:** $p$: a probability distribution (often the target or data)

model's most likely next token equals the label.

**Key takeaways**

- The dataset format decides the loss mask: prompt–completion trains completions only, conversations train everything unless `assistant_only_loss=True`.
- Packing (`bfd`) and padding-free remove padding waste; check attention boundaries.
- Defaults: lr 2e-5, `max_length` 1024, packing off, gradient checkpointing on, `loss_type="nll"`.
- SFT loss is a token mean over the accumulated batch (`num_items_in_batch`, `average_tokens_across_devices=True`); long answers dominate.
- A LoRA SFT run: lr about 2e-4, `all-linear` targets, `assistant_only_loss` for conversations, `bfd` packing; inspect one masked example first.

## 47.7 Parameters specific to SFTConfig

Fields inherited unchanged from `TrainingArguments` are listed in Chapter 45; this list shows new fields and changed defaults.

*Generated from the installed source (transformers 5.20.2, peft 0.21.2, trl 1.16.0); 25 entries.*

- **`learning_rate`** = `2e-05`. The initial learning rate for AdamW. (overrides the TrainingArguments default 5e-05)
- **`bf16`** = `None`. Whether to use bf16 (mixed) precision instead of 32-bit. Requires Ampere or higher NVIDIA architecture or Intel XPU or using CPU (use_cpu) or Ascend NPU. If not set, it defaults to True if fp16 is not set. (overrides the TrainingArguments default False)
- **`gradient_checkpointing`** = `True`. Enable gradient checkpointing to trade compute for memory. Reduces memory at the cost of ~20% slower training. (overrides the TrainingArguments default False)
- **`logging_steps`** = `10`. Log every X updates steps. Should be an integer or a float in range [0,1). If smaller than 1, will be interpreted as ratio of total training steps. (overrides the TrainingArguments default 500)
- **`model_init_kwargs`** = `None`. Keyword arguments for AutoModelForCausalLM.from_pretrained, used when the model argument of the SFTTrainer is provided as a string. The revision value is also used when loading the processing class.
- **`trust_remote_code`** = `False`. Whether to allow loading models and tokenizers that ship custom Python code from the Hub. Forwarded to AutoModelForCausalLM.from_pretrained and AutoProcessor.from_pretrained.
- **`router_aux_loss_coef`** = `None`. Coefficient of the load-balancing auxiliary loss for Mixture-of-Experts (MoE) models, added to the training loss with this weight. When not set, the value declared by the model config is used.
- **`chat_template_path`** = `None`. If specified, sets the model's chat template. This can either be the path to a tokenizer (local directory or Hugging Face Hub model) or a direct path to a Jinja template file.
- **`dataset_text_field`** = `'text'`. Name of the column that contains text data in the dataset.

- **dataset_kwargs** = None. Dictionary of optional keyword arguments for the dataset preparation. The only supported key is skip_prepare_dataset. When the dataset contains images, skip_prepare_dataset is automatically treated as True regardless of the provided value, since preprocessing is done on the fly.
- **dataset_num_proc** = None. Number of processes to use for processing the dataset.
- **eos_token** = None. Token used to indicate the end of a turn or sequence. If None, it defaults to processing_class.eos_token.
- **max_length** = 1024. Maximum length of the tokenized sequence. Sequences longer than max_length are truncated from the left or right depending on the truncation_mode. If None, no truncation is applied.
- **truncation_mode** = 'keep_start'. Truncation mode to use when the sequence exceeds max_length. The only supported value is 'keep_start'. The 'keep_end' value is deprecated and will be removed in v2.0.0.
- **shuffle_dataset** = False. Whether to shuffle the dataset.
- **packing** = False. Whether to group multiple sequences into fixed-length blocks to improve computational efficiency and reduce padding. Uses max_length to define sequence length.
- **packing_strategy** = 'bfd'. Strategy for packing sequences. Can be 'bfd' (best-fit decreasing, truncates overflow), 'bfd_split' (best-fit decreasing, splits overflow sequences), or 'wrapped' (aggressive, cuts mid-sequence).
- **padding_free** = False. Whether to perform forward passes without padding by flattening all sequences in the batch into a single continuous sequence. This reduces memory usage by eliminating padding overhead.
- **pad_to_multiple_of** = 'auto'. If set, the sequences will be padded to a multiple of this value. 'auto' pads to a multiple of 128 on Neuron (AWS Trainium), which compiles once per input shape, and does not pad elsewhere.
- **eval_packing** = None. Whether to pack the eval dataset. If None, uses the same value as packing.
- **completion_only_loss** = None. Whether to compute loss only on the completion part of the sequence. If set to True, loss is computed only on the completion, which is supported only for prompt-completion datasets.
- **assistant_only_loss** = False. Whether to compute loss only on the assistant part of the sequence. If set to True, loss is computed only on the assistant responses, which is supported only for conversational datasets.
- **loss_type** = 'nll'. Type of loss to use. Possible values are 'nll' (standard negative log-likelihood) and 'dft' (Dynamic Fine-Tuning ...)
- **activation_offloading** = False. Whether to offload the activations to the CPU.
- **pad_token** = None. Deprecated. Set tokenizer.pad_token directly and pass it as processing_class to the trainer instead.

CHAPTER 48

# TRL RewardConfig and RewardTrainer

**Why this chapter matters.**
A reward model (Chapter 16) turns preference pairs into a scalar score that RLHF optimises. TRL's `RewardTrainer` trains it with the Bradley–Terry loss on (chosen, rejected) pairs. The config is short, but three details are interview material: the default learning rate is much higher than for policy training (1e-4), dropout is disabled by default, and rewards are only defined up to a constant, which is why `center_rewards_coefficient` exists.

## 48.1 What RewardTrainer does

Given a pair $(x, y_w, y_l)$ (prompt, chosen, rejected), a sequence-classification model with one output, $r_\phi(x, y)$, scores both sequences (the score is read at the last non-padding token). The loss is

$$\mathcal{L}(\phi) = -\log \sigma(r_\phi(x, y_w) - r_\phi(x, y_l)) \; + \; \eta \, (r_\phi(x, y_w) + r_\phi(x, y_l))^2,$$

where $\sigma$ is the logistic function and the second term, active when `center_rewards_coefficient` $\eta$ is set, pulls the scores toward mean zero (the help text cites Eisenstein et al. and recommends $\eta = 0.01$). The first term only depends on the *difference* of scores, so adding a constant to every reward changes nothing; without centering the average reward can drift anywhere, which makes reward values hard to compare across checkpoints and can upset RL code that clips or normalises rewards.

Dataset rows need `chosen` and `rejected` columns (text, or full conversations including the prompt). An explicit `prompt` column is also accepted. An optional numeric `margin` column switches the loss to $-\log \sigma(r_w - r_l - m)$, the Llama 2 margin loss, which asks for a larger gap on pairs that annotators rated as clearly different. Pairs where either side is longer than `max_length` (1024) are **filtered out**, not truncated: check how many pairs you lose.

```
import os, warnings
warnings.filterwarnings("ignore"); os.environ["HF_HUB_OFFLINE"] = "1"
import torch, transformers, datasets
transformers.logging.set_verbosity_error(); datasets.disable_progress_bars()
from transformers import AutoTokenizer, Qwen2Config, Qwen2ForSequenceClassification,
 ↪  PrinterCallback
from datasets import Dataset
from trl import RewardConfig, RewardTrainer
tok = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B-Instruct")
def tiny_rm(seed=0):
```

**Notation on this page:** $y_w, y_l$: chosen (winning) and rejected (losing) responses • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $\phi$: weights of a second network (reward model or critic) • $r(x, y)$: the reward for response $y$ to prompt $x$ • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $\eta$: learning rate • $\mathcal{L}$: a loss to minimise

```
    torch.manual_seed(seed)
    return Qwen2ForSequenceClassification(Qwen2Config(vocab_size=len(tok),
     ↪  hidden_size=64, intermediate_size=128,
        num_hidden_layers=2, num_attention_heads=4, num_key_value_heads=2, num_labels=1,
 ↪  pad_token_id=tok.pad_token_id))
rows = [{"chosen":   [{"role": "user", "content": f"{a}+{b}?"}, {"role": "assistant",
 ↪  "content": f"{a + b}"}],
         "rejected": [{"role": "user", "content": f"{a}+{b}?"}, {"role": "assistant",
          ↪  "content": f"{a + b + 1} maybe"}]}
        for a in range(8) for b in range(4)]
ds = Dataset.from_list(rows)
for eta in [None, 0.01]:
    args = RewardConfig(output_dir="/tmp/ref48", use_cpu=True, bf16=False,
 ↪  per_device_train_batch_size=8, num_train_epochs=8,
                        learning_rate=1e-3, report_to="none", save_strategy="no",
 ↪  logging_steps=8, disable_tqdm=True,
                        gradient_checkpointing=False, center_rewards_coefficient=eta)
    tr = RewardTrainer(model=tiny_rm(), args=args, train_dataset=ds,
 ↪  processing_class=tok)
    tr.remove_callback(PrinterCallback)
    tr.train()
    logs = [h for h in tr.state.log_history if "loss" in h]
    print(f"center_rewards_coefficient={eta}: " + " | ".join(
        f"ep {h['epoch']:.0f}: loss {h['loss']:.3f} acc {h['accuracy']:.2f} margin
         ↪  {h['margin']:.2f} mean r {h['mean_reward']:+.2f}" for h in logs[::2]))
print("columns after preprocessing:", tr.train_dataset.column_names)
```

The trainer logs pair accuracy (fraction of pairs with $r_w > r_l$), the mean margin $r_w - r_l$, and reward statistics. Both runs separate the pairs. In this tiny run the freshly initialised score head starts near zero and the mean happens to stay near zero either way, so the centring term barely matters; that is not guaranteed. The next snippet shows why: the Bradley–Terry loss cannot see a constant shift, so nothing in it holds the mean in place, while the centring penalty does.

```
import torch.nn.functional as F
r_w, r_l = torch.tensor([1.2, 0.3, 2.0]), torch.tensor([0.1, -0.4, 0.5])
for shift in [0.0, 5.0]:
    bt = -F.logsigmoid((r_w + shift) - (r_l + shift)).mean()
    centre = 0.01 * ((r_w + shift) + (r_l + shift)).pow(2).mean()
    print(f"shift {shift:+.0f}: Bradley-Terry loss {bt:.4f} | centring penalty
     ↪  {centre:.4f}")
```

## 48.2 The parameters that matter

- **`learning_rate`** (1e-4): much higher than DPO or RL rates. For full fine-tuning of a multi-billion-parameter reward model, values around 1e-5 to 5e-6 are common; 1e-4 suits LoRA or small models. Reward models overfit fast: one epoch is the usual choice (InstructGPT reported overfitting beyond one epoch).
- **`disable_dropout`** (True): dropout makes the same sequence score differently on each pass,

**Notation on this page:** $r(x, y)$: the reward for response $y$ to prompt $x$

adding noise to the score difference.

- **`center_rewards_coefficient`** (None): set 0.01 to keep rewards centred.
- **`max_length`** (1024): pairs that exceed it are dropped.
- **`model_init_kwargs`**: when the model is passed as a string, it is loaded with `AutoModelForSequenceClassification.from_pretrained` (with `num_labels=1`).
- With PEFT, pass `peft_config` to the trainer and keep the score head trainable (`task_type="SEQ_CLS"`, or list `score` in `modules_to_save`).

**Why this, not that? Why a separate reward model and RL, not DPO directly on the pairs?**

**Reason.** A reward model can score *new* samples from the current policy, so RL keeps getting feedback on-policy; it also generalises the preference data and can be reused for best-of-$n$, filtering and evaluation. **Trade-off.** An extra model to train and serve, reward hacking as the policy finds its blind spots, and the cost of online RL. **When the alternative is better.** Small budgets, offline-only pipelines, or when the preference data already covers the policy's outputs: DPO (Chapter 19) is simpler and often close in quality.

## 48.3 Evaluating the reward model before using it

Training accuracy says little. Before a reward model drives RL, check four things on **held-out** pairs, built so that a shortcut cannot pass.

1. **Held-out pair accuracy** on prompts the model never saw.
2. **Shortcut probes**: pairs where the spurious feature (length, a hedge word, a format) is removed or reversed. If accuracy falls to chance, the model learned the shortcut, not the preference.
3. **Calibration**: does $\sigma(r_w - r_l)$ match the observed win rate? (Project 2 computes the expected calibration error.)
4. **Benchmarks** such as RewardBench (2403.13787), which groups pairs into chat, hard chat, safety and reasoning, so weaknesses show up by category.

The training data in the snippet above has a built-in shortcut: every rejected answer is wrong *and* ends with the word "maybe". The next snippet scores held-out sums (numbers the model never saw) and then a probe in which the rejected answer is wrong but has no "maybe", and a second probe in which the *correct* answer carries the "maybe".

```
rm = tr.model.eval()
def score(conv):
    text = tok.apply_chat_template(conv, tokenize=False)
    enc = tok(text, return_tensors="pt", add_special_tokens=False)
    with torch.no_grad():
        return rm(**enc).logits[0, 0].item()
def pair_acc(make_pair, pairs):
    wins = [score(c) > score(r) for c, r in (make_pair(a, b) for a, b in pairs)]
    return sum(wins) / len(wins)
conv = lambda a, b, ans: [{"role": "user", "content": f"{a}+{b}?"}, {"role":
 ↪  "assistant", "content": ans}]
```

**Notation on this page:** $n$: number of samples • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $r(x, y)$: the reward for response $y$ to prompt $x$

```
held_out = [(a, b) for a in range(8, 12) for b in range(4, 8)]
probes = {
    "held-out, same format as training": lambda a, b: (conv(a, b, f"{a + b}"), conv(a,
    ↪  b, f"{a + b + 1} maybe")),
    "probe: wrong answer without 'maybe'": lambda a, b: (conv(a, b, f"{a + b}"), conv(a,
    ↪  b, f"{a + b + 1}")),
    "probe: correct answer WITH 'maybe'": lambda a, b: (conv(a, b, f"{a + b} maybe"),
    ↪  conv(a, b, f"{a + b + 1}")),
}
for name, fn in probes.items():
    print(f"{name:38s} accuracy {pair_acc(fn, held_out):.2f}")
```

The model is near perfect on the held-out pairs in the training format, falls to around chance when "maybe" is removed, and is almost always wrong when the correct answer carries "maybe": it learned to detect the word, not to check the sum. Real preference data has softer versions of the same shortcut (longer answers, more bullet points, confident tone), and RL will find and exploit them (Chapter 27). The fix is in the data: build pairs where the shortcut appears on both sides or on the chosen side, so only the real signal separates them.

**Why this, not that? Why pairwise Bradley–Terry training, not regression on absolute 1–10 ratings?**

**Reason.** People are far more consistent when comparing two answers than when giving absolute scores; different raters use the scale differently, and pairwise labels remove that per-rater offset. The pairwise loss also needs only the score difference, which is exactly what RL advantages use. **Trade-off.** Pairs carry at most one bit each, scores are only defined up to a constant (hence centring), and building good pairs needs two responses per prompt. **When the alternative is better.** When there is a reliable absolute scale (a rubric with clear levels, multi-attribute ratings such as helpfulness and correctness in HelpSteer-style data, or a verifiable success rate), regression on the ratings uses more information per example and gives scores with a fixed meaning.

## 48.4 From scores to RL rewards

A reward model's raw score is rarely used as-is.

- **Normalisation.** Group-relative methods (GRPO, RLOO) subtract the group mean, so a constant offset disappears; PPO pipelines usually whiten rewards with a running mean and standard deviation.
- **Combining rewards.** In TRL's GRPO, `reward_funcs` can mix a reward-model id or a `PreTrainedModel` (loaded as `AutoModelForSequenceClassification` with `num_labels=1`; TRL reads `logits[:, 0]`) with Python functions such as format checks. `reward_weights` sets the weights (all 1.0 when `None`), and `multi_objective_aggregation` chooses between `"sum_then_normalize"` (the default) and `"normalize_then_sum"`, which normalises each reward within the group before the weighted sum so a high-variance reward cannot drown out the others.
- **Same template, same tokenizer.** The reward model must see completions formatted the

way it was trained; TRL applies the reward model's own chat template (through `reward_processing_classes`) to conversational samples.

- **Guarding against over-optimisation.** Track the KL from the reference policy, hold out a second reward model or a small human-rated set as a "gold" check, and stop or regularise when the training reward rises while the gold score falls (Chapter 27 and Project 8).

## 48.5 Common mistakes

1. Training for several epochs (the reward model memorises pairs; held-out accuracy drops).
2. Comparing raw reward values across reward-model checkpoints without centring or normalisation.
3. Losing many pairs to the `max_length` filter without noticing.
4. Different chat templates for reward-model training and scoring.
5. A missing `pad_token_id` in the model config (the score is read at the wrong position).
6. Reporting only in-distribution accuracy; a reward model can score near 100% by detecting a shortcut.
7. Mixing a reward model with rule-based rewards of very different scales and summing them raw; use weights or `normalize_then_sum`.

**Interview check**

**Q1. What loss does RewardTrainer use?** Bradley–Terry: $-\log \sigma(r_w - r_l)$, plus an optional centring penalty.
**Q2. Why centre rewards?** The loss is invariant to a constant shift, so the mean is unconstrained; centring makes values comparable and well-behaved in RL.
**Q3. Why is dropout disabled?** Dropout adds noise to scores and to their difference.
**Q4. What happens to pairs longer than `max_length`?** They are filtered out, not truncated.
**Q5. Where is the reward read from in a causal LM classifier?** At the last non-padding token's hidden state, through a linear score head.
**Q6. How do you detect that a reward model learned a shortcut?** Build probe pairs in which the suspected feature is removed or placed on the chosen side; accuracy near or below chance means the model relies on it.
**Q7. How does TRL's GRPO use a reward model?** Pass a model id or a sequence-classification model in `reward_funcs`; the trainer formats the completion with the reward model's template, reads `logits[:, 0]`, and combines it with other rewards using `reward_weights` and `multi_objective_aggregation`.
**Q8. Why train on pairs rather than absolute ratings?** Comparisons are more consistent across raters and remove each rater's offset; use regression when a reliable absolute scale exists.

**Key takeaways**

- Bradley–Terry loss on chosen/rejected; accuracy and margin are the training metrics.
- Defaults: lr 1e-4, dropout off, centring off (0.01 recommended), long pairs filtered.
- Train about one epoch; hold out pairs to measure accuracy.

**Notation on this page:** $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $r(x, y)$: the reward for response $y$ to prompt $x$

- Evaluate on held-out pairs plus shortcut probes, calibration and a benchmark such as RewardBench before RL.
- In GRPO, combine reward models and rule rewards with `reward_weights` and, for different scales, `normalize_then_sum`.

## 48.6 Parameters specific to RewardConfig

*Generated from the installed source (transformers 5.20.2, peft 0.21.2, trl 1.16.0); 15 entries.*

- **`learning_rate`** = `0.0001`. The initial learning rate for AdamW. (overrides the TrainingArguments default 5e-05)
- **`bf16`** = `None`. Whether to use bf16 (mixed) precision instead of 32-bit. Requires Ampere or higher NVIDIA architecture or Intel XPU or using CPU (use_cpu) or Ascend NPU. If not set, it defaults to True if fp16 is not set. (overrides the TrainingArguments default False)
- **`gradient_checkpointing`** = `True`. Enable gradient checkpointing to trade compute for memory. Reduces memory at the cost of ~20% slower training. (overrides the TrainingArguments default False)
- **`logging_steps`** = `10`. Log every X updates steps. Should be an integer or a float in range [0,1). If smaller than 1, will be interpreted as ratio of total training steps. (overrides the TrainingArguments default 500)
- **`model_init_kwargs`** = `None`. Keyword arguments for AutoModelForSequenceClassification. from_pretrained, used when the model argument of the RewardTrainer is provided as a string. The revision value is also used when loading the tokenizer.
- **`trust_remote_code`** = `False`. Whether to allow loading models and tokenizers that ship custom Python code from the Hub. Forwarded to AutoModelForSequenceClassification. from_pretrained and AutoTokenizer.from_pretrained.
- **`chat_template_path`** = `None`. If specified, sets the model's chat template. This can either be the path to a tokenizer (local directory or Hugging Face Hub model) or a direct path to a Jinja template file.
- **`disable_dropout`** = `True`. Whether to disable dropout in the model.
- **`dataset_num_proc`** = `None`. Number of processes to use for processing the dataset.
- **`eos_token`** = `None`. Token used to indicate the end of a turn or sequence. If None, it defaults to processing_class.eos_token.
- **`max_length`** = `1024`. Maximum length of the tokenized sequence. Samples are filtered out if either chosen or rejected sequence exceeds this value. If None, no filtering is applied.
- **`pad_to_multiple_of`** = `'auto'`. If set, the sequences will be padded to a multiple of this value. 'auto' pads to a multiple of 128 on Neuron (AWS Trainium), which compiles once per input shape, and does not pad elsewhere.
- **`center_rewards_coefficient`** = `None`. Coefficient to incentivize the reward model to output mean-zero rewards (proposed by this paper, Eq. 2). Recommended value: 0.01.
- **`activation_offloading`** = `False`. Whether to offload the activations to the CPU.
- **`pad_token`** = `None`. Deprecated. Set tokenizer.pad_token directly and pass it as processing_class to the trainer instead.

CHAPTER 49

# TRL DPOConfig and KTOConfig

**Why this chapter matters.**
Direct preference optimisation (Chapter 19) is the most common preference-tuning method in practice, and TRL's `DPOTrainer` bundles about fifteen loss variants behind one `loss_type` argument. Knowing what `beta` does to each loss, how the logged "rewards" are defined, when to precompute reference log-probabilities and what IPO's target margin is lets you configure and debug DPO runs quickly. KTO (Chapter 20) is included because it shares most of the machinery but uses unpaired data.

## 49.1 DPOConfig essentials (TRL 1.16.0)

With policy $\pi_\theta$, frozen reference $\pi_{\text{ref}}$ and log-ratios $h_w = \log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)}$, $h_l$ likewise, the default loss is

$$\mathcal{L}_{\text{sigmoid}} = -\log \sigma(\beta\,(h_w - h_l)).$$

- **`beta`** (0.1): the strength of the implicit KL constraint. Larger $\beta$ keeps the policy closer to the reference and makes the loss saturate sooner. Typical range 0.01–0.5.
- **`loss_type`** (`["sigmoid"]`, a list): options in TRL 1.16 are `sigmoid`, `hinge` ($\max(0, 1-\beta\Delta)$, SLiC style), `ipo`, `exo_pair`, `nca_pair`, `robust`, `bco_pair`, `sppo_hard`, `aot`, `aot_unpaired`, `apo_zero`, `apo_down`, `discopop`, `sft` (plain NLL on the chosen answer) and `sigmoid_norm` (the sigmoid loss on per-token average log-ratios). Several can be combined with **`loss_weights`**; for example `loss_type=["sigmoid", "sft"]` adds an SFT term on the chosen answer (the RPO idea; Chapter 19).
- **IPO** in TRL uses per-token *average* log-ratios and the loss $(\bar{h}_w - \bar{h}_l - \frac{1}{2\beta})^2$. With $\beta = 0.1$ the target margin is 5 (per token, in nats), so the initial loss is about 25. $\beta$ here is the IPO paper's $\tau$; for IPO, larger $\beta$ means a *smaller* target margin.
- **`label_smoothing`** (0.0): used by `robust` (probability that a label is flipped, in $[0, 0.5)$) and `exo_pair`.
- **`f_divergence_type`** (`reverse_kl`): f-DPO variants (`forward_kl`, `js_divergence`, `alpha_divergence` with `f_alpha_divergence_coef` 0.5).
- **`use_weighting`** (False): WPO weighting of pairs by the policy's length-normalised probability, which down-weights off-policy pairs (*WPO: Enhancing RLHF with Weighted Preference Optimization*, arXiv 2406.11827).
- **`ld_alpha`** (None): LD-DPO; down-weights log-probabilities of the "verbose" part of the longer answer to counter length exploitation.
- **`precompute_ref_log_probs`** (False): compute all reference log-probs once before training,

**Notation on this page:** $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\pi_{\text{ref}}$: the frozen reference model • $y_w, y_l$: chosen (winning) and rejected (losing) responses • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\mathcal{L}$: a loss to minimise • $\tau$: a trajectory (states, actions, rewards of one episode)

so the reference model need not stay in memory. Reference log-probs never change for a frozen reference, so this is free accuracy-wise.

- **`sync_ref_model`** (False), **`ref_model_mixup_alpha`** (0.6), **`ref_model_sync_steps`** (512): TR-DPO, which periodically moves the reference toward the policy, $\pi_{\text{ref}} \leftarrow \alpha\pi_\theta + (1-\alpha)\pi_{\text{ref}}$ (in weight space).
- **`max_length`** (1024), **`truncation_mode`** (`keep_start`), **`padding_free`**, **`disable_dropout`** (True), **`learning_rate`** (1e-6).
- There is no `max_prompt_length` in TRL 1.16 (older tutorials use it).

The trainer logs, per batch: `rewards/chosen` $= \beta h_w$, `rewards/rejected` $= \beta h_l$, `rewards/margins`, `rewards/accuracies` (fraction with $h_w > h_l$), `logps/chosen`, `logps/rejected`. Watch `logps/chosen`: if it falls steadily while margins grow, the model is pushing both answers down (Chapter 19's "likelihood displacement"); adding an `sft` term or lowering the learning rate helps.

```
import os, warnings
warnings.filterwarnings("ignore"); os.environ["HF_HUB_OFFLINE"] = "1"
import torch, transformers, datasets
transformers.logging.set_verbosity_error(); datasets.disable_progress_bars()
from transformers import AutoTokenizer, Qwen2Config, Qwen2ForCausalLM, PrinterCallback
from datasets import Dataset
from trl import DPOConfig, DPOTrainer
tok = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B-Instruct")
def tiny_model(seed=0):
    torch.manual_seed(seed)
    return Qwen2ForCausalLM(Qwen2Config(vocab_size=len(tok), hidden_size=64,
     ↪  intermediate_size=128, num_hidden_layers=2,
                                        num_attention_heads=4, num_key_value_heads=2,
 ↪  tie_word_embeddings=True))
rows = [{"prompt": [{"role": "user", "content": f"{a}+{b}?"}],
         "chosen": [{"role": "assistant", "content": f"{a + b}"}],
         "rejected": [{"role": "assistant", "content": f"{a + b + 1} maybe"}]} for a in
          ↪  range(8) for b in range(4)]
ds = Dataset.from_list(rows)
for loss in ["sigmoid", "ipo"]:
    args = DPOConfig(output_dir="/tmp/ref49", use_cpu=True, bf16=False,
 ↪  per_device_train_batch_size=8, num_train_epochs=4,
                     learning_rate=5e-4, loss_type=[loss], report_to="none",
 ↪  save_strategy="no", logging_steps=4,
                     disable_tqdm=True, gradient_checkpointing=False)
    # the tiny model has no Hub name, so pass an identical copy as the frozen reference
     ↪  explicitly
    tr = DPOTrainer(model=tiny_model(), ref_model=tiny_model(), args=args,
 ↪  train_dataset=ds, processing_class=tok)
    tr.remove_callback(PrinterCallback)
    tr.train()
    for h in [h for h in tr.state.log_history if "loss" in h][::3]:
        print(f"{loss:7s} epoch {h['epoch']:.0f}: loss {h['loss']:6.3f} | rewards/chosen
         ↪  {h['rewards/chosen']:+.3f} "
```

**Notation on this page:** $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\alpha$: a weighting coefficient (defined in the text) • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay)

```
            f"rejected {h['rewards/rejected']:+.3f} margin {h['rewards/margins']:.3f}
             ↪  acc {h['rewards/accuracies']:.2f} "
            f"| logps/chosen {h['logps/chosen']:.2f}")
print("processed columns:", tr.train_dataset.column_names)
```

The sigmoid loss starts at $\ln 2 \approx 0.69$ (zero margin) and falls as the margin grows; IPO starts near $(1/(2\beta))^2 = 25$ because it regresses the per-token margin toward 5. Both raise the chosen log-probability and lower the rejected one here.

**Why this, not that? Why precompute reference log-probs, not run the reference model every step?**

**Reason.** A frozen reference gives the same log-probabilities every epoch; computing them once removes a whole model from GPU memory and a forward pass from each step. **Trade-off.** One extra pass over the dataset before training, storage for two floats per pair, and no flexibility if you later change tokenisation or truncation. **When the alternative is better.** When the reference changes (TR-DPO's `sync_ref_model`), with on-the-fly data augmentation or online DPO, or when the dataset is huge and you train for less than one epoch.

**Why this, not that? Why the sigmoid loss by default, not IPO or hinge?**

**Reason.** Sigmoid DPO is the exact maximum-likelihood objective under the Bradley–Terry model with the KL-regularised optimal policy (Chapter 19), well understood and robust across datasets. **Trade-off.** It never saturates on perfectly separated data: margins can grow without bound, which is how DPO overfits and drifts far from the reference despite $\beta$. **When the alternative is better.** Near-deterministic or noisy preferences: IPO's squared loss targets a finite margin; `robust` with `label_smoothing` tolerates flipped labels; `hinge` stops pushing once the margin exceeds $1/\beta$.

## 49.2 KTOConfig

KTO (Chapter 20) learns from single responses labelled desirable or undesirable, with no pairs. Rows have `prompt`, `completion` and a boolean `label`. Key settings (TRL 1.16.0): `beta` 0.1; `desirable_weight` and `undesirable_weight` (both 1.0), which rebalance classes (the KTO paper suggests keeping $\lambda_D n_D/(\lambda_U n_U)$ between 1 and 4/3); `loss_type` `"kto"` or `"apo_zero_unpaired"`; `learning_rate` 1e-6; `precompute_ref_log_probs`; and **`train_sampling_strategy="sequential"`** (overriding the usual `random`), because the KTO loss estimates its KL reference point from mismatched (prompt, completion) pairs built from neighbours in a fixed-order batch.

## 49.3 Common mistakes

1. Copying an old config with `max_prompt_length` (not a field in TRL 1.16).
2. Reading `rewards/margins` growth as success while `logps/chosen` collapses.
3. Using the same $\beta$ for IPO as for sigmoid (they mean different things).
4. DPO on pairs generated by a very different model (off-policy pairs; consider WPO weighting or regenerating pairs from your SFT model).

**Notation on this page:** $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $\lambda$: GAE parameter (0 to 1), trades bias for variance • $n$: number of samples

5. Too high a learning rate: DPO usually wants 5e-7 to 1e-6 for full fine-tuning.

**Interview check**

**Q1. What does `rewards/chosen` mean in DPO logs?** $\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)}$, the implicit reward of the chosen answer.
**Q2. What is the IPO target margin with $\beta = 0.1$ in TRL?** $1/(2\beta) = 5$ in per-token average log-ratio units.
**Q3. How do you add an SFT term to DPO in TRL?** `loss_type=["sigmoid", "sft"]` with optional `loss_weights`.
**Q4. When is `precompute_ref_log_probs` safe?** Whenever the reference is frozen and the data and tokenisation are fixed.
**Q5. Why does KTO use a sequential sampler?** Its KL baseline is estimated from mismatched pairs formed within a fixed-order batch.

**Key takeaways**

- DPO defaults: `beta` 0.1, `loss_type=["sigmoid"]`, lr 1e-6, `max_length` 1024, dropout off.
- Fifteen loss types; combine with `loss_weights`; IPO regresses to a $1/(2\beta)$ margin.
- Monitor margins *and* chosen log-probs.
- KTO: unpaired labels, class weights, sequential sampling.

## 49.4 Parameters specific to DPOConfig

*Generated from the installed source (transformers 5.20.2, peft 0.21.2, trl 1.16.0); 29 entries.*

- **`learning_rate`** = `1e-06`. The initial learning rate for AdamW. (overrides the TrainingArguments default 5e-05)
- **`bf16`** = `None`. Whether to use bf16 (mixed) precision instead of 32-bit. Requires Ampere or higher NVIDIA architecture or Intel XPU or using CPU (use_cpu) or Ascend NPU. If not set, it defaults to True if fp16 is not set. (overrides the TrainingArguments default False)
- **`gradient_checkpointing`** = `True`. Enable gradient checkpointing to trade compute for memory. Reduces memory at the cost of ~20% slower training. (overrides the TrainingArguments default False)
- **`logging_steps`** = `10`. Log every X updates steps. Should be an integer or a float in range [0,1). If smaller than 1, will be interpreted as ratio of total training steps. (overrides the TrainingArguments default 500)
- **`model_init_kwargs`** = `None`. Keyword arguments for AutoModelForCausalLM.from_pretrained, used when the model argument of the DPOTrainer is provided as a string. The revision value is also used when loading the processing class.
- **`trust_remote_code`** = `False`. Whether to allow loading models and tokenizers that ship custom Python code from the Hub. Forwarded to AutoModelForCausalLM.from_pretrained and AutoProcessor.from_pretrained.
- **`router_aux_loss_coef`** = `None`. Coefficient of the load-balancing auxiliary loss for Mixture-of-Experts (MoE) models, added to the training loss with this weight. When not set, the value declared by the model config is used.

**Notation on this page:** $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $y_w, y_l$: chosen (winning) and rejected (losing) responses • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$

- **disable_dropout** = `True`. Whether to disable dropout in the model and reference model.
- **dataset_num_proc** = `None`. Number of processes to use for processing the dataset.
- **max_length** = `1024`. Maximum length of the tokenized sequence. Sequences longer than max_length are truncated from the left or right depending on the truncation_mode. If None, no truncation is applied.
- **truncation_mode** = `'keep_start'`. Truncation mode to use when the sequence exceeds max_length. The only supported value is 'keep_start'. The 'keep_end' value is deprecated and will be removed in v2.0.0.
- **padding_free** = `False`. Whether to perform forward passes without padding by flattening all sequences in the batch into a single continuous sequence. This reduces memory usage by eliminating padding overhead.
- **pad_to_multiple_of** = `'auto'`. If set, the sequences will be padded to a multiple of this value. 'auto' pads to a multiple of 128 on Neuron (AWS Trainium), which compiles once per input shape, and does not pad elsewhere.
- **precompute_ref_log_probs** = `False`. Whether to precompute the reference model log probabilities for the entire training dataset before training. This allows to save memory during training, as the reference model does not need to be kept in memory.
- **precompute_ref_batch_size** = `None`. Batch size to use when precomputing reference model log probabilities. This can be set higher than the training batch size to speed up preprocessing. If None, defaults to per_device_train_batch_size for training and per_device_eval_batch_size for evaluation.
- **loss_type** = `['sigmoid']`. Type of loss to use. Possible values are: 'sigmoid', 'hinge', 'ipo', 'exo_pair', 'nca_pair', 'robust', 'bco_pair', 'sppo_hard', 'aot', 'aot_unpaired', 'apo_zero', 'apo_down', 'discopop', 'sft', 'sigmoid_norm'.
- **loss_weights** = `None`. List of loss weights for multi-loss combinations. Used when combining multiple loss types. Example: [0.8, 0.2, 1.0] for MPO. If not provided, defaults to equal weights (1.0) for all loss types.
- **ld_alpha** = `None`. $\alpha$ parameter from the LD-DPO paper, which controls the weighting of the verbose token log-probabilities in responses. If None, no weighting is applied to the verbose part, and the loss is equivalent to the standard DPO loss.
- **f_divergence_type** = `'reverse_kl'`. f-divergence regularizer between policy and reference (f-DPO paper). Possible values are: reverse_kl (default), forward_kl, js_divergence, alpha_divergence.
- **f_alpha_divergence_coef** = `0.5`. $\alpha$ coefficient for the $\alpha$-divergence u^-$\alpha$ regularizer, used only when f_divergence_type='alpha_divergence'.
- **label_smoothing** = `0.0`. Label smoothing parameter used in Robust DPO and EXO. In Robust DPO, it is interpreted as the probability that a preference label is flipped and must lie in [0.0, 0.5); a typical value recommended by the Robust DPO paper is 0.1.
- **beta** = `0.1`. Parameter controlling the deviation from the reference model. Higher $\beta$ means less deviation from the reference model. For the IPO loss (loss_type='ipo'), this value is the regularization parameter denoted by $\tau$ in the paper.
- **use_weighting** = `False`. Whether to apply WPO-style weighting to preference pairs using the policy's length-normalized sequence probabilities.
- **discopop_tau** = `0.05`. $\tau$/temperature parameter from the DiscoPOP paper, which controls the shape of the log-ratio modulated loss when using loss_type='discopop'. The paper

recommends the default value discopop_tau=0.05.

- **activation_offloading** = `False`. Whether to offload the activations to the CPU.
- **sync_ref_model** = `False`. Whether to synchronize the reference model with the active model every ref_model_sync_steps steps, using the ref_model_mixup_alpha parameter. This synchronization originates from the TR-DPO paper.
- **ref_model_mixup_alpha** = `0.6`. α parameter from the TR-DPO paper, which controls the mix between the current policy and the previous reference policy during updates. The reference policy is updated according to the equation: π_ref = α * π_θ + (1 - α) * π_ref_prev.
- **ref_model_sync_steps** = `512`. τ parameter from the TR-DPO paper, which determines how frequently the current policy is synchronized with the reference policy. To use this parameter, you must set sync_ref_model=True.
- **pad_token** = `None`. Deprecated. Set tokenizer.pad_token directly and pass it as processing_class to the trainer instead.

## 49.5 Parameters specific to KTOConfig

*Generated from the installed source (transformers 5.20.2, peft 0.21.2, trl 1.16.0); 22 entries.*

- **learning_rate** = `1e-06`. The initial learning rate for AdamW. (overrides the TrainingArguments default 5e-05)
- **bf16** = `None`. Whether to use bf16 (mixed) precision instead of 32-bit. Requires Ampere or higher NVIDIA architecture or Intel XPU or using CPU (use_cpu) or Ascend NPU. If not set, it defaults to True if fp16 is not set. (overrides the TrainingArguments default False)
- **gradient_checkpointing** = `True`. Enable gradient checkpointing to trade compute for memory. Reduces memory at the cost of ~20% slower training. (overrides the TrainingArguments default False)
- **logging_steps** = `10`. Log every X updates steps. Should be an integer or a float in range [0,1). If smaller than 1, will be interpreted as ratio of total training steps. (overrides the TrainingArguments default 500)
- **train_sampling_strategy** = `'sequential'`. Sampler to use for the training dataloader. Loss types that estimate the KL divergence term (all except 'apo_zero_unpaired') require 'sequential' because the KL completion for each example is precomputed against its neighbors in a fixed-order batch; any other strategy breaks that pairing. (overrides the TrainingArguments default 'random')
- **model_init_kwargs** = `None`. Keyword arguments for AutoModelForCausalLM.from_pretrained, used when the model argument of the KTOTrainer is provided as a string. The revision value is also used when loading the processing class.
- **trust_remote_code** = `False`. Whether to allow loading models and tokenizers that ship custom Python code from the Hub. Forwarded to AutoModelForCausalLM.from_pretrained and AutoProcessor.from_pretrained.
- **router_aux_loss_coef** = `None`. Coefficient of the load-balancing auxiliary loss for Mixture-of-Experts (MoE) models, added to the training loss with this weight. When not set, the value declared by the model config is used.
- **disable_dropout** = `True`. Whether to disable dropout in the model and reference model.
- **dataset_num_proc** = `None`. Number of processes to use for processing the dataset.

- **max_length** = 1024. Maximum length of the tokenized sequence. Sequences longer than max_length are truncated from the right. If None, no truncation is applied.
- **pad_to_multiple_of** = 'auto'. If set, the sequences will be padded to a multiple of this value. 'auto' pads to a multiple of 128 on Neuron (AWS Trainium), which compiles once per input shape, and does not pad elsewhere.
- **precompute_ref_log_probs** = False. Whether to precompute the reference model log probabilities for the entire training dataset before training. This allows to save memory during training, as the reference model does not need to be kept in memory.
- **precompute_ref_batch_size** = None. Batch size to use when precomputing reference model log probabilities. This can be set higher than the training batch size to speed up preprocessing. If None, defaults to per_device_train_batch_size for training and per_device_eval_batch_size for evaluation.
- **loss_type** = 'kto'. Type of loss to use.
- **beta** = 0.1. Parameter controlling the deviation from the reference model. Higher β means less deviation from the reference model.
- **desirable_weight** = 1.0. Desirable losses are weighed by this factor to counter unequal number of desirable and undesirable pairs.
- **undesirable_weight** = 1.0. Undesirable losses are weighed by this factor to counter unequal number of desirable and undesirable pairs.
- **activation_offloading** = False. Whether to offload the activations to the CPU.
- **sync_ref_model** = False. Whether to synchronize the reference model with the active model every ref_model_sync_steps steps, using the ref_model_mixup_alpha parameter. This synchronization originates from the TR-DPO paper.
- **ref_model_mixup_alpha** = 0.6. α parameter from the TR-DPO paper, which controls the mix between the current policy and the previous reference policy during updates. The reference policy is updated according to the equation: π_ref = α * π_θ + (1 - α) * π_ref_prev.
- **ref_model_sync_steps** = 512. τ parameter from the TR-DPO paper, which determines how frequently the current policy is synchronized with the reference policy. To use this parameter, you must set sync_ref_model=True.

CHAPTER 50

# TRL GRPOConfig and RLOOConfig

**Why this chapter matters.**
`GRPOTrainer` is the most common entry point to RL for LLMs, and its config is the largest in TRL (189 fields in TRL 1.16.0, 82 of them its own). The defaults encode several 2025 lessons: no KL term (`beta=0.0`), DAPO-style token aggregation (`loss_type="dapo"`), and importance-sampling correction for vLLM rollouts. This chapter explains the batch geometry (the source of most configuration errors), the loss options, the generation and vLLM settings, multi-reward setups and tool use, then covers `RLOOConfig`. Note that TRL 1.16 has **no `PPOTrainer`**: PPO with a learned critic lives in OpenRLHF and veRL (Chapter 51).

## 50.1 Batch geometry

GRPO generates $G =$ `num_generations` completions for each prompt and compares them within the group. Three quantities must fit together:

- the **per-device micro-batch** `per_device_train_batch_size` counts *completions*, not prompts;
- the **generation batch** `generation_batch_size` (completions generated at once; default per-device batch × processes × `steps_per_generation`), which must be divisible by $G$ so that groups are whole;
- **`steps_per_generation`** (default = `gradient_accumulation_steps`): how many optimizer micro-steps one generation batch feeds.

`num_iterations` ($\mu$, default 1) re-uses each generation batch for several passes; only when $\mu > 1$ (or `steps_per_generation` exceeds the accumulation steps) does the PPO-style clipping actually bind, because otherwise the data is exactly on-policy at the first micro-step.

```
import os, warnings
warnings.filterwarnings("ignore"); os.environ["HF_HUB_OFFLINE"] = "1"
import torch, transformers, datasets
transformers.logging.set_verbosity_error(); datasets.disable_progress_bars()
from trl import GRPOConfig
def geometry(**kw):
    try:
        c = GRPOConfig(output_dir="/tmp/ref50", use_cpu=True, bf16=False, **kw)
        return (f"per_device={c.per_device_train_batch_size}
         ↪  accum={c.gradient_accumulation_steps} G={c.num_generations} -> "
                f"generation_batch_size={c.generation_batch_size}
                 ↪  ({c.generation_batch_size // c.num_generations} prompts), "
```

**Notation on this page:** $G$: group size (samples per prompt) • $\mu$: a mean

```
                f"steps_per_generation={c.steps_per_generation}")
    except ValueError as e:
        return "ValueError: " + str(e).split(".")[0]
print(geometry())
print(geometry(per_device_train_batch_size=4, num_generations=8))
print(geometry(per_device_train_batch_size=4, num_generations=8,
 ↪  gradient_accumulation_steps=4))
print(geometry(per_device_train_batch_size=8, num_generations=16,
 ↪  generation_batch_size=128))
c = GRPOConfig(output_dir="/tmp/ref50", use_cpu=True, bf16=False)
print(f"defaults: lr {c.learning_rate}, beta {c.beta}, epsilon {c.epsilon}, epsilon_high
 ↪  {c.epsilon_high}, loss_type {c.loss_type}, "
      f"scale_rewards {c.scale_rewards}, max_completion_length
       ↪  {c.max_completion_length}, temperature {c.temperature}, "
      f"use_vllm {c.use_vllm}, vllm_mode {c.vllm_mode}")
```

The default (8 completions per device, $G = 8$) means one prompt per device per step: fine for a laptop test, far too small for real training. A typical single-node setup is 8 GPUs × 8 completions × 4 accumulation steps = 256 completions = 32 prompts × $G$ 8 per optimizer step.

## 50.2 Loss options

The per-token clipped surrogate (Chapter 22) is aggregated in different ways by `loss_type`:

| `loss_type` | Normalisation / idea |
|---|---|
| `grpo` | mean over each sequence's tokens, then over sequences (length bias: favours short correct and long wrong answers) |
| `dapo` (default) | sum over all tokens divided by the number of active tokens in the global accumulated batch |
| `dr_grpo` | sum divided by a constant (`max_completion_length`), removing length bias |
| `bnpo` | like `dapo` but normalised by the local batch's token count |
| `cispo` | clipped importance-sampling weights times the REINFORCE term (MiniMax-M1, ScaleRL); `epsilon_high` is $\epsilon_{\max}$ (5.0 recommended) |
| `sapo` | soft, temperature-controlled gating instead of hard clipping (`sapo_temperature_pos/_neg`) |
| `luspo`, `vespo` | further 2025–26 variants listed in the TRL docs |

Other loss settings:

- **epsilon** (0.2) and **epsilon_high** (None, i.e. symmetric): clip range $[1-\varepsilon, 1+\varepsilon_{\text{high}}]$. DAPO's clip-higher uses 0.28.
- **delta** (None): an additional upper clip on the ratio for negative-advantage tokens (two-sided clipping, INTELLECT-2); set above $1+\varepsilon$.
- **importance_sampling_level** (`token`): `sequence` gives GSPO's length-normalised sequence ratio.
- **beta** (0.0): KL coefficient; 0 means no reference model is loaded at all. With `beta>0`, `use_bias_correction_kl` (True) multiplies the KL term by the importance ratio (DeepSeek-V3.2).
- **scale_rewards** (`group`): divide advantages by the group standard deviation; `batch` uses the batch standard deviation; `none` skips scaling (Dr. GRPO's recommendation, to avoid weighting questions by difficulty).
- **mask_truncated_completions** (False): drop truncated completions from the loss (DAPO).
- **top_entropy_quantile** (1.0): keep only the highest-entropy fraction of tokens in the policy loss (0.2 recommended by the "80/20" paper).
- **off_policy_mask_threshold** (None): mask negative-advantage sequences whose KL to the sampler exceeds the threshold (DeepSeek-V3.2).

## 50.3 Generation and vLLM settings

- **max_completion_length** (512), **temperature** (1.0), **top_p** (1.0), **top_k** (0, disabled), **min_p** (None), **repetition_penalty** (1.0), **generation_kwargs** (passed to `GenerationConfig` or vLLM `SamplingParams`).
- **use_vllm** (False): without it, generation uses Transformers `generate` (or paged continuous batching with `use_transformers_paged`). **vllm_mode**: `colocate` (default; vLLM inside the training processes, sharing GPUs, memory fraction `vllm_gpu_memory_utilization` 0.3, optional `vllm_enable_sleep_mode`) or `server` (a separate `trl vllm-serve` process on other GPUs, reached at `vllm_server_base_url` or host/port).
- **vllm_importance_sampling_correction** (True), **vllm_importance_sampling_mode** (`sequence_mask`; also `token_truncate`, `token_mask`, `sequence_truncate`), **vllm_importance_sampling_clip_max** (3.0), `..._clip_min`: correct for train/infer log-prob mismatch (Chapter 43).
- **ds3_gather_for_generation** (True): with ZeRO-3, gather full weights for generation (fast) or not (lets models larger than one GPU train, slowly).
- **cast_lm_head_to_fp32** (False): compute logits in FP32 (ScaleRL) to reduce mismatch.

## 50.4 Rewards, tools and environments

`reward_funcs` can be a model ID (a sequence-classification reward model), a model object, or Python functions with signature `f(prompts, completions, **kwargs) -> list[float]`; extra dataset columns arrive as keyword arguments (hence `remove_unused_columns=False`). A function may return `None` for samples it does not apply to. With several functions, `reward_weights` weights them and `multi_objective_aggregation` chooses `sum_then_normalize` (default) or `normalize_then_sum` (GDPO: normalise each reward within the group before summing, so one high-

**Notation on this page:** $\epsilon$: clip range for the probability ratio

variance reward does not dominate).

For agents, the trainer accepts `tools` (a list of Python functions exposed to the model through the chat template), `max_tool_calling_iterations`, an experimental `environment_factory` (Chapter 33) and a custom `rollout_func`. Tool-result tokens are masked out of the loss.

```
from transformers import AutoTokenizer, Qwen2Config, Qwen2ForCausalLM, PrinterCallback
from datasets import Dataset
from trl import GRPOTrainer, RLOOConfig, RLOOTrainer
tok = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B-Instruct")
def tiny_model(seed=0):
    torch.manual_seed(seed)
    return Qwen2ForCausalLM(Qwen2Config(vocab_size=len(tok), hidden_size=64,
     ↪  intermediate_size=128, num_hidden_layers=2,
                                        num_attention_heads=4, num_key_value_heads=2,
 ↪  tie_word_embeddings=True))
ds = Dataset.from_dict({"prompt": [[{"role": "user", "content": f"Write a word with {n}
 ↪  letters."}] for n in range(3, 11)],
                        "target_len": list(range(3, 11))})
def length_reward(completions, target_len, **kwargs):          # uses an extra dataset
 ↪  column
    return [-abs(len(c[0]["content"].split()[0]) - t) / 10 if c[0]["content"].split()
     ↪  else -1.0
            for c, t in zip(completions, target_len)]
def brevity_reward(completions, **kwargs):
    return [1.0 if len(c[0]["content"]) < 40 else 0.0 for c in completions]
args = GRPOConfig(output_dir="/tmp/ref50", use_cpu=True, bf16=False,
 ↪  per_device_train_batch_size=8, num_generations=4,
                  max_completion_length=24, max_steps=2, logging_steps=1,
 ↪  report_to="none", save_strategy="no",
                  disable_tqdm=True, gradient_checkpointing=False, reward_weights=[1.0,
 ↪  0.5])
tr = GRPOTrainer(model=tiny_model(), processing_class=tok, reward_funcs=[length_reward,
 ↪  brevity_reward],
                 args=args, train_dataset=ds)
tr.remove_callback(PrinterCallback)
tr.train()
h = tr.state.log_history[0]
keys = ["loss", "grad_norm", "reward", "reward_std", "rewards/length_reward/mean",
 ↪  "rewards/brevity_reward/mean",
        "frac_reward_zero_std", "completions/mean_length", "completions/clipped_ratio",
         ↪  "entropy", "clip_ratio/region_mean"]
for k in keys:
    print(f"{k:30s} {h[k]:.4f}")
```

Three things in this log are worth knowing for interviews. The **loss is about zero** while the gradient norm is not: at the first micro-step the ratio is exactly 1, and with group-normalised advantages that sum to zero, the surrogate's value is near zero even though its gradient is not. `frac_reward_zero_std` is the share of groups whose rewards were all equal (no learning signal; DAPO's dynamic sampling targets these). `completions/clipped_ratio` is the share that hit `max_completion_length`; the random tiny model never stops, so it is 1.0 here. In a real run a high

value means `max_completion_length` is too small or the model rambles.

## 50.5 RLOOConfig

`RLOOTrainer` (TRL 1.16.0) implements REINFORCE leave-one-out (Chapter 23): each sample's baseline is the mean reward of the *other* $k-1$ samples of the same prompt. It shares GRPOConfig's generation, vLLM and reward machinery. Differences in defaults: `num_generations` 2 (the minimum for a leave-one-out baseline), `beta` 0.05 (a KL penalty is on by default, folded into the reward), `normalize_advantages` False, and `reward_clip_range` (None) to clip rewards.

```
rargs = RLOOConfig(output_dir="/tmp/ref50r", use_cpu=True, bf16=False,
 ↪  per_device_train_batch_size=8, num_generations=4,
                  max_completion_length=24, max_steps=1, logging_steps=1,
 ↪  report_to="none", save_strategy="no",
                  disable_tqdm=True, gradient_checkpointing=False)
print(f"RLOO defaults: num_generations {RLOOConfig(output_dir='/tmp/x', use_cpu=True,
 ↪  bf16=False).num_generations}, "
      f"beta {rargs.beta}, normalize_advantages {rargs.normalize_advantages}, epsilon
       ↪  {rargs.epsilon}")
# beta > 0 needs a reference model, which the trainer loads from the model's path, so
 ↪  save the tiny model first
tiny_model().save_pretrained("/tmp/ref50/tiny"); tok.save_pretrained("/tmp/ref50/tiny")
rtr = RLOOTrainer(model="/tmp/ref50/tiny", processing_class=tok,
 ↪  reward_funcs=[length_reward], args=rargs, train_dataset=ds)
rtr.remove_callback(PrinterCallback)
rtr.train()
print({k: round(v, 4) for k, v in rtr.state.log_history[0].items() if k in ("loss",
 ↪  "reward", "kl", "entropy")})
```

**Why this, not that? Why beta = 0 by default in GRPO, not a KL penalty?**

**Reason.** For verifiable-reward reasoning RL, several 2025 studies (DAPO, Dr. GRPO, others; Chapter 26) found the KL term to the initial model mostly slowed learning; clipping already limits each step, and dropping the reference model saves memory and a forward pass. **Trade-off.** Nothing anchors the policy to the starting model, so drift (style changes, loss of general skills, reward hacking with learned reward models) is only limited by clipping and early stopping. **When the alternative is better.** Learned or noisy reward models (classic RLHF), long runs where general capabilities must be preserved, or safety-sensitive tuning: use a small `beta` (0.001–0.05) or RLOO's default 0.05.

**Why this, not that? Why vLLM colocate mode by default, not server mode?**

**Reason.** Colocation needs no extra GPUs or processes: every training GPU also generates, which suits single-node jobs. **Trade-off.** Generation and training compete for memory (`vllm_gpu_memory_utilization` 0.3 leaves the rest for training), and they cannot overlap. **When the alternative is better.** Multi-node jobs, large models, or long generations: a dedicated `trl vllm-serve` server on separate GPUs gives the engine more memory and lets you

**Notation on this page:** $k$: an index or count (see text)

size generation and training independently (Chapter 44).

## 50.6 Common mistakes

1. Reading `per_device_train_batch_size` as prompts (it counts completions).
2. A `generation_batch_size` not divisible by `num_generations` (configuration error) or groups too small to have variance.
3. Expecting clipping to matter with `num_iterations=1` and `steps_per_generation` equal to accumulation steps.
4. Forgetting `remove_unused_columns=False` semantics: GRPO already defaults to keeping columns; custom collators may still drop them.
5. Interpreting a near-zero loss as "nothing is happening".
6. Leaving `max_completion_length` at 512 for reasoning tasks (most completions truncated).

**Interview check**

**Q1. What does `per_device_train_batch_size` count in GRPOTrainer?** Completions; prompts per device = batch / `num_generations` per generation batch.
**Q2. What is the default `loss_type` and why?** `dapo`: normalising by total active tokens in the accumulated batch removes per-sequence length bias.
**Q3. What does `scale_rewards="none"` change?** Advantages are reward minus group mean without dividing by the group std, avoiding question-difficulty weighting.
**Q4. Why is the logged GRPO loss often zero?** Ratios are 1 at the first micro-step and normalised advantages sum to zero in each group; gradients are still non-zero.
**Q5. What do `vllm_importance_sampling_*` settings do?** Reweight or mask samples by the ratio between trainer and vLLM probabilities to correct numerical mismatch.
**Q6. How do RLOO's defaults differ from GRPO's?** $k = 2$ generations, a KL penalty of 0.05 in the reward, no advantage normalisation.

**Key takeaways**

- Geometry: completions per device; generation batch divisible by $G$; clipping matters only for re-used batches.
- Defaults: `beta=0`, `loss_type="dapo"`, `epsilon=0.2`, `scale_rewards="group"`, `max_completion_length=512`, vLLM off (colocate when on), IS correction on.
- Rewards: functions with dataset columns as kwargs, weights, GDPO-style aggregation; tools and environments for agents.
- No PPOTrainer in TRL 1.16; use OpenRLHF or veRL for critic-based PPO.

## 50.7 Parameters specific to GRPOConfig

*Generated from the installed source (transformers 5.20.2, peft 0.21.2, trl 1.16.0); 82 entries.*

- **`learning_rate`** = `1e-06`. The initial learning rate for AdamW. (overrides the TrainingArguments default 5e-05)

**Notation on this page:** $k$: an index or count (see text) • $G$: group size (samples per prompt)

- **bf16** = None. Whether to use bf16 (mixed) precision instead of 32-bit. Requires Ampere or higher NVIDIA architecture or Intel XPU or using CPU (use_cpu) or Ascend NPU. If not set, it defaults to True if fp16 is not set. (overrides the TrainingArguments default False)
- **gradient_checkpointing** = True. Enable gradient checkpointing to trade compute for memory. Reduces memory at the cost of ~20% slower training. (overrides the TrainingArguments default False)
- **logging_steps** = 10. Log every X updates steps. Should be an integer or a float in range [0,1). If smaller than 1, will be interpreted as ratio of total training steps. (overrides the TrainingArguments default 500)
- **remove_unused_columns** = False. Whether to only keep the column 'prompt' in the dataset. If you use a custom reward function that requires any column other than 'prompts' and 'completions', you should keep this to False. (overrides the TrainingArguments default True)
- **model_init_kwargs** = None. Keyword arguments for transformers. AutoModelForCausalLM.from_pretrained, used when the model argument of the GRPOTrainer is provided as a string. The revision value is also used when loading processing classes.
- **trust_remote_code** = False. Whether to allow loading models and tokenizers that ship custom Python code from the Hub. Forwarded to AutoModelForCausalLM.from_pretrained and AutoProcessor.from_pretrained.
- **router_aux_loss_coef** = None. Coefficient of the load-balancing auxiliary loss for Mixture-of-Experts (MoE) models, added to the training loss with this weight. When not set, the value declared by the model config is used.
- **disable_dropout** = False. Whether to disable dropout in the model. This is useful for training with a reference model, as it prevents the model from generating different logprobs for the same input.
- **cast_lm_head_to_fp32** = False. Whether to cast the language modeling head of the policy and reference models to float32. As recommended by the ScaleRL recipe. This flag is only supported when the model has untied word embedding and language modeling head layers i.e.
- **num_generations** = 8. Number of generations to sample. The effective batch size (num_processes * per_device_batch_size * gradient_accumulation_steps) must be evenly divisible by this value.
- **num_generations_eval** = None. Number of generations to sample during evaluation. This allows using fewer generations during evaluation to save computation. If None, uses the value of num_generations.
- **max_completion_length** = 512. Maximum length of the generated completion.
- **ds3_gather_for_generation** = True. This setting applies to DeepSpeed ZeRO-3. If enabled, the policy model weights are gathered for generation, improving generation speed. However, disabling this option allows training models that exceed the VRAM capacity of a single GPU, albeit at the cost of slower generation.
- **shuffle_dataset** = True. Whether to shuffle the training dataset.
- **pad_to_multiple_of** = 'auto'. If set, the prompts ids and completions ids will be padded to a multiple of this value. 'auto' pads to a multiple of 128 on Neuron (AWS Trainium), which compiles once per input shape, and does not pad elsewhere.

- **generation_batch_size** = None. Batch size to use for generation. If None, it defaults to the effective training batch size: per_device_train_batch_size * num_processes * steps_per_generation.
- **steps_per_generation** = None. Number of steps per generation. If None, it defaults to gradient_accumulation_steps.
- **temperature** = 1.0. Temperature for sampling. The higher the temperature, the more random the completions.
- **top_p** = 1.0. Float that controls the cumulative probability of the top tokens to consider. Must be in (0, 1]. Set to 1.0 to consider all tokens. ...)
- **top_k** = 0. Number of highest probability vocabulary tokens to keep for top-k-filtering. If 0, top-k-filtering is disabled and all tokens are considered.
- **min_p** = None. Minimum token probability, which will be scaled by the probability of the most likely token. It must be a value between 0.0 and 1.0. Typical values are in the 0.01-0.2 range.
- **generation_kwargs** = None. Additional keyword arguments to pass to GenerationConfig (if using transformers) or SamplingParams (if using vLLM) when sampling completions. This can be used to further customize the generation behavior, such as setting suppress_tokens, num_beams, etc.
- **chat_template_kwargs** = None. Additional keyword arguments to pass to the apply_chat_template function when generating completions.
- **repetition_penalty** = 1.0. Float that penalizes new tokens based on whether they appear in the prompt and the generated text so far. Values > 1.0 encourage the model to use new tokens, while values < 1.0 encourage the model to repeat tokens.
- **cache_implementation** = None. Implementation of the cache method for faster generation when use_vllm is set to False.
- **use_vllm** = False. Whether to use vLLM for generating completions. If set to True, the trainer will use vLLM for generation instead of the default model.generate(). Requires vllm to be installed.
- **vllm_mode** = 'colocate'. Mode to use for vLLM integration when use_vllm is set to True. Must be one of 'server' or 'colocate'. 'server': The trainer will send generation requests to a separate vLLM server.
- **vllm_model_impl** = 'vllm'. Model implementation to use for vLLM. Must be one of transformers or vllm. transformers: Use the transformers backend for model implementation. vllm: Use the vllm library for model implementation.
- **vllm_enable_sleep_mode** = False. Enable vLLM sleep mode to offload weights/cache during the optimizer step. Keeps GPU memory usage low, but waking the engine adds host–device transfer latency.
- **vllm_structured_outputs_regex** = None. Regex for vLLM structured outputs. If None (default), structured outputs is disabled.
- **vllm_server_base_url** = None. Base URL for the vLLM server (e.g., ' If provided, vllm_server_host and vllm_server_port are ignored. ...)
- **vllm_server_host** = '0.0.0.0'. Host of the vLLM server to connect to. Ignored if vllm_server_base_url is provided.
- **vllm_server_port** = 8000. Port of the vLLM server to connect to. Ignored if vllm_server_base_url is provided.

- **vllm_server_timeout** = 240.0. Total timeout duration in seconds to wait for the vLLM server to be up. If the server is not up after the timeout, a ConnectionError is raised.
- **vllm_group_port** = 51216. Port number for the weight update group. This is used to communicate with the vLLM server. Unless the port is occupied, there is no need to change it.
- **vllm_gpu_memory_utilization** = 0.3. Control the GPU memory utilization for vLLM. This setting only applies when vllm_mode is set to 'colocate'. If you are using vllm_mode= 'server', this parameter must be passed separately when launching the vLLM server via the –vllm_gpu_memory_utilization flag.
- **vllm_max_model_length** = None. Context window for vLLM. Set it to at least the maximum prompt length in the dataset plus max_completion_length; if omitted, it is inferred from the model config.
- **vllm_tensor_parallel_size** = 1. Control the tensor parallel size for vLLM. This setting only applies when vllm_mode is set to 'colocate'. If you are using vllm_mode='server', this parameter must be passed separately when launching the vLLM server via the –vllm_ tensor_parallel_size flag.
- **beta** = 0.0. KL coefficient. If 0.0 (default), the reference model is not loaded, reducing memory usage and improving training speed. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning use a value of 0.001.
- **num_iterations** = 1. Number of iterations per batch (denoted as $\mu$ in the algorithm).
- **epsilon** = 0.2. Epsilon value for clipping.
- **delta** = None. Enables the upper clipping bound in two-sided GRPO loss when set to a float. If None (default), standard GRPO clipping is used. Recommended to be greater than $1 + \varepsilon$ when enabled.
- **epsilon_high** = None. Upper-bound epsilon value for clipping. If not specified, it defaults to the same value as the lower-bound specified in argument epsilon. Paper DAPO recommends 0.28.
- **sapo_temperature_neg** = 1.05. Temperature for tokens with non-positive advantage scores used in the sapo loss function. This parameter is introduced in the Soft Adaptive Policy Optimization paper.
- **sapo_temperature_pos** = 1.0. Temperature for tokens with positive advantage scores used in the sapo loss function. This parameter is introduced in the Soft Adaptive Policy Optimization paper.
- **vespo_k_pos** = 2.0. k parameter for positive advantages, it is the power exponent in the VESPO loss. Controls how aggressively we down-weight samples with low importance weights (when the importance sampling ratio $< 1$).
- **vespo_lambda_pos** = 3.0. lambda parameter for positive advantages, it is the decay factor in the VESPO loss. Controls how aggressively we down-weight samples with high importance weights (when the importance sampling ratio $> 1$).
- **vespo_k_neg** = 3.0. k parameter for negative advantages, it is the power exponent in the VESPO loss. Controls how aggressively we down-weight samples with low importance weights (when the importance sampling ratio $< 1$).
- **vespo_lambda_neg** = 2.0. lambda parameter for negative advantages, it is the exponential decay factor in the VESPO loss. Controls how aggressively we down-weight samples with high importance weights (when the importance sampling ratio $> 1$).

- **importance_sampling_level** = `'token'`. Controls whether importance sampling ratios are computed at the 'token' or 'sequence' level. 'token' keeps the raw per-token log-probability ratios (one weight per token).
- **reward_weights** = `None`. Weights for each reward function. Must match the number of reward functions. If None, all rewards are weighted equally with weight 1.0.
- **multi_objective_aggregation** = `'sum_then_normalize'`. Method to aggregate multiple reward functions. Supported values are: 'sum_then_normalize' (default): First sums the weighted rewards from each reward function, then applies reward scaling/normalization as specified by scale_rewards (see scale_rewards for details).
- **scale_rewards** = `'group'`. Specifies the scaling strategy for rewards. Supported values are: True or group' (default): rewards are scaled by the standard deviation within each group, ensuring unit variance within a group.
- **loss_type** = `'dapo'`. Specifies the loss formulation to use. Supported values are 'grpo', 'dr_grpo', 'dapo', 'bnpo', 'cispo', 'sapo', 'luspo', and 'vespo'. 'grpo': Aggregates token-level losses by normalizing over sequence length.
- **mask_truncated_completions** = `False`. When enabled, truncated completions are excluded from the loss calculation, preventing them from being incorrectly penalized and introducing noise during training. According to the DAPO paper, this is a good practice for training stability.
- **sync_ref_model** = `False`. Whether to synchronize the reference model with the active model every ref_model_sync_steps steps, using the ref_model_mixup_alpha parameter.
- **ref_model_mixup_alpha** = `0.6`. $\alpha$ parameter from the TR-DPO paper, which controls the mix between the current policy and the previous reference policy during updates. The reference policy is updated according to the equation: $\pi_ref = \alpha * \pi_\theta + (1 - \alpha) * \pi_ref_prev$.
- **ref_model_sync_steps** = `512`. $\tau$ parameter from the TR-DPO paper, which determines how frequently the current policy is synchronized with the reference policy. To use this parameter, you must set sync_ref_model=True.
- **top_entropy_quantile** = `1.0`. $\rho$ parameter from Beyond the 80/20 Rule. Keeps in the policy loss term only the top-$\rho$ quantile of tokens by entropy of the probability distribution at each sequence position, improving results.
- **entropy_coef** = `0.0`. Coefficient of the entropy regularization term in the loss. A positive value adds an entropy bonus that encourages exploration. The bonus is always the mean per-token entropy regardless of loss_type (not rescaled to a loss type's policy normalization), so entropy_coef has the same meaning for every loss type.
- **use_adaptive_entropy** = `False`. Whether to use adaptive entropy control, introduced in Skywork-OR1 When enabled, entropy_coef is incremented by entropy_coef_delta when entropy is below entropy_target, and decremented otherwise.
- **entropy_coef_min** = `0.0`. Lower bound for the entropy coefficient when using adaptive entropy control.
- **entropy_coef_max** = `1.0`. Upper bound for the entropy coefficient when using adaptive entropy control.
- **entropy_coef_delta** = `0.005`. Step size for adjusting the entropy coefficient during adaptive entropy control.
- **entropy_target** = `0.2`. Target mean per-token entropy (nats) for adaptive entropy control. The coefficient is only applied when current entropy is at or below this value. Measured

over the same token set as the policy loss (all completion tokens, or the high-entropy subset when top_entropy_quantile < 1.0).

- **max_tool_calling_iterations** = None. Maximum number of tool-calling turns when training an agent. If None, there is no limit and generation stops when the model generates a response turn with no tool calls or when the total response length reaches max_model_length.
- **vllm_importance_sampling_correction** = True. Whether to apply Importance Sampling (IS) to correct for the mismatch between vLLM completion logprobs and recomputed training logprobs. If set to False, no IS is applied regardless of vllm_importance_sampling_mode.
- **vllm_importance_sampling_mode** = 'sequence_mask'. Specifies how Importance Sampling (IS) is performed when vllm_importance_sampling_correction=True.
- **vllm_importance_sampling_clip_max** = 3.0. Importance sampling upper bound C_max used by vllm_importance_sampling_mode. For '*_truncate' modes, ratios are clipped from above at C_max. For '*_mask' modes, ratios larger than C_max are set to zero.
- **vllm_importance_sampling_clip_min** = None. Importance sampling lower bound C_min used by vllm_importance_sampling_mode. For *_truncate modes, ratios are clipped from below at C_min. For *_mask modes, ratios below C_min are set to zero.
- **off_policy_mask_threshold** = None. Threshold for off-policy sequence masking. If None, off-policy sequence masking is disabled. When set, sequences with negative advantages and high KL divergence are masked out to stabilize training.
- **use_bias_correction_kl** = True. Whether to multiply the KL term by the importance sampling ratio, so that the KL gradient becomes the unbiased reverse-KL gradient, as described in the DeepSeek-V3.2 paper.
- **log_completions** = False. Whether to log a sample of (prompt, completion) pairs every logging_steps steps. If rich is installed, it prints the sample. If wandb logging is enabled, it logs it to wandb.
- **log_multimodal** = True. Whether to log multimodal content (images, videos, etc.) together with completions. Disable this to reduce log size when using high-resolution multimodal data.
- **num_completions_to_print** = None. Number of completions to print with rich. If None, all completions are logged.
- **log_unique_prompts** = False. Whether to log unique prompts. If True, only unique prompts are logged. If False, all prompts are logged.
- **log_completions_hub_repo** = None. Hugging Face Hub repository to save the completions. Should be a complete repository name like 'username/reponame' or 'orgname/reponame', or just 'reponame' in which case the repository will be created in the currently-logged-in Hugging Face user's namespace.
- **use_transformers_continuous_batching** = False. Whether to use transformers' continuous batching engine for generating completions. Requires transformers>=5.8.0.
- **transformers_continuous_batching_config** = None. Keyword arguments for transformers.generation.ContinuousBatchingConfig.
- **use_transformers_paged** = False. Deprecated. Use use_transformers_continuous_batching instead.
- **vllm_importance_sampling_cap** = None. Deprecated, use vllm_importance_sampling_

clip_max instead. Importance sampling cap C used by vllm_importance_sampling_mode. For '*_truncate' modes, ratios are clipped from above at C.

## 50.8 Parameters specific to RLOOConfig

*Generated from the installed source (transformers 5.20.2, peft 0.21.2, trl 1.16.0); 56 entries.*

- **`learning_rate`** = `1e-06`. The initial learning rate for AdamW. (overrides the TrainingArguments default 5e-05)
- **`bf16`** = `None`. Whether to use bf16 (mixed) precision instead of 32-bit. Requires Ampere or higher NVIDIA architecture or Intel XPU or using CPU (use_cpu) or Ascend NPU. If not set, it defaults to True if fp16 is not set. (overrides the TrainingArguments default False)
- **`gradient_checkpointing`** = `True`. Enable gradient checkpointing to trade compute for memory. Reduces memory at the cost of ~20% slower training. (overrides the TrainingArguments default False)
- **`logging_steps`** = `10`. Log every X updates steps. Should be an integer or a float in range [0,1). If smaller than 1, will be interpreted as ratio of total training steps. (overrides the TrainingArguments default 500)
- **`remove_unused_columns`** = `False`. Whether to only keep the column 'prompt' in the dataset. If you use a custom reward function that requires any column other than 'prompts' and 'completions', you should keep this to False. (overrides the TrainingArguments default True)
- **`model_init_kwargs`** = `None`. Keyword arguments for transformers. AutoModelForCausalLM.from_pretrained, used when the model argument of the RLOOTrainer is provided as a string. The revision value is also used when loading processing classes.
- **`trust_remote_code`** = `False`. Whether to allow loading models and tokenizers that ship custom Python code from the Hub. Forwarded to AutoModelForCausalLM.from_pretrained and AutoProcessor.from_pretrained.
- **`router_aux_loss_coef`** = `None`. Coefficient of the load-balancing auxiliary loss for Mixture-of-Experts (MoE) models, added to the training loss with this weight. When not set, the value declared by the model config is used.
- **`disable_dropout`** = `False`. Whether to disable dropout in the model. This is useful for training with a reference model, as it prevents the model from generating different logprobs for the same input.
- **`num_generations`** = `2`. Number of generations to sample. The effective batch size (num_processes * per_device_batch_size * gradient_accumulation_steps) must be evenly divisible by this value.
- **`num_generations_eval`** = `None`. Number of generations to sample during evaluation. This allows using fewer generations during evaluation to save computation. If None, uses the value of num_generations.
- **`max_completion_length`** = `512`. Maximum length of the generated completion.
- **`ds3_gather_for_generation`** = `True`. This setting applies to DeepSpeed ZeRO-3. If enabled, the policy model weights are gathered for generation, improving generation speed. However, disabling this option allows training models that exceed the VRAM capacity of a single

GPU, albeit at the cost of slower generation.

- **shuffle_dataset** = `True`. Whether to shuffle the training dataset.
- **pad_to_multiple_of** = `'auto'`. If set, the prompts ids and completions ids will be padded to a multiple of this value. 'auto' pads to a multiple of 128 on Neuron (AWS Trainium), which compiles once per input shape, and does not pad elsewhere.
- **generation_batch_size** = `None`. Batch size to use for generation. If None, it defaults to the effective training batch size: per_device_train_batch_size * num_processes * steps_per_generation.
- **steps_per_generation** = `None`. Number of steps per generation. If None, it defaults to gradient_accumulation_steps.
- **temperature** = `1.0`. Temperature for sampling. The higher the temperature, the more random the completions.
- **top_p** = `1.0`. Float that controls the cumulative probability of the top tokens to consider. Must be in (0, 1]. Set to 1.0 to consider all tokens. ...)
- **top_k** = `0`. Number of highest probability vocabulary tokens to keep for top-k-filtering. If 0, top-k-filtering is disabled and all tokens are considered.
- **min_p** = `None`. Minimum token probability, which will be scaled by the probability of the most likely token. It must be a value between 0.0 and 1.0. Typical values are in the 0.01-0.2 range.
- **generation_kwargs** = `None`. Additional keyword arguments to pass to GenerationConfig (if using transformers) or SamplingParams (if using vLLM) when sampling completions. This can be used to further customize the generation behavior, such as setting suppress_tokens, num_beams, etc.
- **chat_template_kwargs** = `None`. Additional keyword arguments to pass to the apply_chat_template function when generating completions.
- **repetition_penalty** = `1.0`. Float that penalizes new tokens based on whether they appear in the prompt and the generated text so far. Values > 1.0 encourage the model to use new tokens, while values < 1.0 encourage the model to repeat tokens.
- **cache_implementation** = `None`. Implementation of the cache method for faster generation when use_vllm is set to False.
- **use_vllm** = `False`. Whether to use vLLM for generating completions. If set to True, the trainer will use vLLM for generation instead of the default model.generate(). Requires vllm to be installed.
- **vllm_mode** = `'colocate'`. Mode to use for vLLM integration when use_vllm is set to True. Must be one of 'server' or 'colocate'. 'server': The trainer will send generation requests to a separate vLLM server.
- **vllm_model_impl** = `'vllm'`. Model implementation to use for vLLM. Must be one of transformers or vllm. transformers: Use the transformers backend for model implementation. vllm: Use the vllm library for model implementation.
- **vllm_enable_sleep_mode** = `False`. Enable vLLM sleep mode to offload weights/cache during the optimizer step. Keeps GPU memory usage low, but waking the engine adds host–device transfer latency.
- **vllm_structured_outputs_regex** = `None`. Regex for vLLM structured outputs. If None (default), structured outputs is disabled.
- **vllm_server_base_url** = `None`. Base URL for the vLLM server (e.g., ' If provided, vllm_

server_host and vllm_server_port are ignored. ...)

- **vllm_server_host** = '0.0.0.0'. Host of the vLLM server to connect to. Ignored if vllm_server_base_url is provided.
- **vllm_server_port** = 8000. Port of the vLLM server to connect to. Ignored if vllm_server_base_url is provided.
- **vllm_server_timeout** = 240.0. Total timeout duration in seconds to wait for the vLLM server to be up. If the server is not up after the timeout, a ConnectionError is raised.
- **vllm_group_port** = 51216. Port number for the weight update group. This is used to communicate with the vLLM server. Unless the port is occupied, there is no need to change it.
- **vllm_gpu_memory_utilization** = 0.3. Control the GPU memory utilization for vLLM. This setting only applies when vllm_mode is set to 'colocate'. If you are using vllm_mode='server', this parameter must be passed separately when launching the vLLM server via the –vllm_gpu_memory_utilization flag.
- **vllm_max_model_length** = None. Context window for vLLM. Set it to at least the maximum prompt length in the dataset plus max_completion_length; if omitted, it is inferred from the model config.
- **vllm_tensor_parallel_size** = 1. Control the tensor parallel size for vLLM. This setting only applies when vllm_mode is set to 'colocate'. If you are using vllm_mode='server', this parameter must be passed separately when launching the vLLM server via the –vllm_tensor_parallel_size flag.
- **beta** = 0.05. KL coefficient. If 0.0, the reference model is not loaded, reducing memory usage and improving training speed.
- **num_iterations** = 1. Number of iterations per batch (denoted as $\mu$ in the algorithm).
- **epsilon** = 0.2. Epsilon value for clipping.
- **epsilon_high** = None. Upper-bound epsilon value for clipping. If not specified, it defaults to the same value as the lower-bound specified in argument epsilon. Paper DAPO recommends 0.28.
- **reward_weights** = None. Weights for each reward function. Must match the number of reward functions. If None, all rewards are weighted equally with weight 1.0.
- **normalize_advantages** = False. Whether to normalize advantages. Normalization is done per generation batch to have mean 0.0 and standard deviation of 1.0.
- **reward_clip_range** = None. Clip range for rewards as (min, max). If None, no clipping is applied.
- **mask_truncated_completions** = False. When enabled, truncated completions are excluded from the loss calculation, preventing them from being incorrectly penalized and introducing noise during training. According to the DAPO paper, this is a good practice for training stability.
- **sync_ref_model** = False. Whether to synchronize the reference model with the active model every ref_model_sync_steps steps, using the ref_model_mixup_alpha parameter.
- **ref_model_mixup_alpha** = 0.6. $\alpha$ parameter from the TR-DPO paper, which controls the mix between the current policy and the previous reference policy during updates. The reference policy is updated according to the equation: $\pi$_ref = $\alpha$ * $\pi$_$\theta$ + (1 - $\alpha$) * $\pi$_ref_prev.
- **ref_model_sync_steps** = 512. $\tau$ parameter from the TR-DPO paper, which determines how frequently the current policy is synchronized with the reference policy. To use this param-

eter, you must set sync_ref_model=True.

- **log_completions** = `False`. Whether to log a sample of (prompt, completion) pairs every logging_steps steps. If rich is installed, it prints the sample. If wandb logging is enabled, it logs it to wandb.
- **log_multimodal** = `True`. Whether to log multimodal content (images, videos, etc.) together with completions. Disable this to reduce log size when using high-resolution multimodal data.
- **num_completions_to_print** = `None`. Number of completions to print with rich. If None, all completions are logged.
- **log_unique_prompts** = `False`. Whether to log unique prompts. If True, only unique prompts are logged. If False, all prompts are logged.
- **use_transformers_continuous_batching** = `False`. Whether to use transformers' continuous batching engine for generating completions. Requires transformers>=5.8.0.
- **transformers_continuous_batching_config** = `None`. Keyword arguments for transformers.generation.ContinuousBatchingConfig.
- **use_transformers_paged** = `False`. Deprecated. Use use_transformers_continuous_batching instead.

CHAPTER 51

# OpenRLHF and veRL

**Why this chapter matters.**
For multi-GPU RL with large models, and for PPO with a learned critic (which TRL 1.16 does not offer), the two most used open frameworks are **OpenRLHF** (version 0.11.2 here) and **veRL** (Volcano Engine RL, version 0.9.0, the open-source implementation of HybridFlow). Both use Ray to place workers, vLLM or SGLang for rollouts, and DeepSpeed, FSDP or Megatron for training. They name the same ideas differently, so this chapter starts with a translation table, then explains how each is configured and launched, and ends with generated parameter lists.

## 51.1 The same knobs under three names

| Concept | TRL `GRPOConfig` | OpenRLHF `train_ppo_ray` flag | veRL config key |
|---|---|---|---|
| Group size $G$ | `num_generations` (8) | `--rollout.n_samples_per_prompt` (1) | `actor_rollout_ref.rollout.n` (1) |
| Prompts per step | from `generation_batch_size` | `--rollout.batch_size` (1024) | `data.train_batch_size` (1024) |
| Optimizer mini-batch | per-device × accum × GPUs | `--train.batch_size` (128) | `actor_rollout_ref.actor.ppo_mini_batch_size` (256) |
| Epochs over a batch | `num_iterations` (1) | `--train.max_epochs` (1) | `actor_rollout_ref.actor.ppo_epochs` (1) |
| Response length | `max_completion_length` (512) | `--rollout.max_new_tokens` (None: fill `--data.max_len` 2048) | `data.max_response_length` (512) |
| Advantage estimator | GRPO (fixed) | `--algo.advantage.estimator` (gae) | `algorithm.adv_estimator` (gae) |
| Clip $\varepsilon$ | `epsilon` (0.2) | `--actor.eps_clip` (0.2) | `actor.clip_ratio` (0.2) |
| Clip-higher | `epsilon_high` | `--actor.eps_clip_low_high` | `actor.clip_ratio_low` / `_high` |

**Notation on this page:** $G$: group size (samples per prompt) • $\epsilon$: clip range for the probability ratio

| Concept | TRL `GRPOConfig` | OpenRLHF `train_ppo_ray` flag | veRL config key |
|---|---|---|---|
| KL in loss | `beta` (0.0) | `--algo.kl.use_loss` + `--algo.kl.init_coef` (0.01) | `actor.use_kl_loss` (false), `kl_loss_coef` (0.001) |
| KL in reward | (not offered) | default when not `use_loss` | `algorithm.use_kl_in_reward` (false) |
| KL estimator | k3-style | `--algo.kl.estimator` (`k1`) | `actor.kl_loss_type` (`low_var_kl` = k3) |
| Loss aggregation | `loss_type` (`dapo`) | `--actor.loss_agg_mode` (`token-mean`) | `actor.loss_agg_mode` (`token-mean`) |
| Actor learning rate | `learning_rate` (1e-6) | `--actor.adam.lr` (1e-6) | `actor.optim.lr` (1e-6) |
| Rollout engine | `use_vllm`, `vllm_mode` | `--vllm.num_engines`, `--vllm.tensor_parallel_size` | `rollout.name` (must set), `rollout.tensor_model_parallel_size` (2) |
| Engine memory share | `vllm_gpu_memory_utilization` (0.3) | `--vllm.gpu_memory_utilization` (0.95) | `rollout.gpu_memory_utilization` (0.5) |

Note the defaults that differ: OpenRLHF and veRL default to **PPO with GAE and one sample per prompt**, so a GRPO run must set the estimator *and* the group size. OpenRLHF's `group_norm` estimator and veRL's `grpo` estimator are GRPO's group-normalised advantage.

## 51.2 OpenRLHF

**Architecture.** A Ray driver creates groups of DeepSpeed workers for the actor, critic, reference and reward models, plus vLLM engines for generation. By default each model has its own GPUs (`--actor.num_gpus_per_node 8` and so on); `--train.colocate_actor_ref`, `--train.colocate_critic_reward` and `--train.colocate_all` share GPUs (with `--vllm.enable_sleep` and `--ds.enable_sleep` to free memory between phases). Weights go from DeepSpeed to vLLM over NCCL (`--vllm.sync_backend nccl`). `--train.async_enable` overlaps generation and training (queue size `--train.async_queue_size`, default 1), and `--train.partial_rollout_enable` lets long generations continue across weight updates.

**Training settings.** DeepSpeed ZeRO stage via `--ds.zero_stage` (2), precision `--ds.param_dtype` (`bf16`), optional `--ds.adam_offload`, LoRA via `--ds.lora.rank` (0 = off), packing via `--ds.packing_samples`, ring attention via `--ds.ring_attn_size`. Optimizer per model: `--actor.adam.lr` (1e-6), `--critic.adam.lr` (9e-6), betas (0.9, 0.95), scheduler `cosine_with_min_lr` with 3% warmup and a 10% floor; Muon is available via `--actor.optim muon`.

**Rewards.** Either a reward model (`--reward.model_name_or_path`), an HTTP endpoint, or a Python file passed to `--reward.remote_url`. The file must define `reward_func(queries, prompts, labels)`, where `queries` are the full decoded texts (prompt plus response); it re-

turns a dictionary with `"rewards"` (a tensor used for training), optionally `"scores"` (used for dynamic filtering and logging) and `"extra_logs"`. Shaping options include `--reward.clip_range` (−10, 10), `--reward.normalize_enable`, DAPO-style `--reward.overlong_buffer_len` with `--reward.overlong_penalty_factor`, and `--algo.dynamic_filtering_enable` with `--algo.dynamic_filtering_range` (drop prompts whose average score is outside the range).

**Agents.** `--train.agent_func_path` points to a Python file that defines `AgentExecutor`. For multi-turn environments, OpenRLHF provides a `MultiTurnAgentExecutor` driven by an `AgentInstanceBase` subclass with async `reset(states)` and `step(state_dict)`; `step` returns rewards, scores, environment feedback text, a `done` flag and optional per-turn sampling parameters.

```
import tempfile, importlib.util, torch, re, os, yaml
reward_file = os.path.join(tempfile.mkdtemp(), "math_reward.py")
with open(reward_file, "w") as f:
    f.write('''
import re, torch
def reward_func(queries, prompts, labels):
    rewards = []
    for q, p, label in zip(queries, prompts, labels):
        response = q[len(p):] if q.startswith(p) else q
        m = re.findall(r"\\\\boxed\\{([^}]*)\\}", response)
        rewards.append(1.0 if m and m[-1].strip() == str(label).strip() else 0.0)
    r = torch.tensor(rewards)
    return {"rewards": r, "scores": r, "extra_logs": {"boxed_rate":
 ↪  torch.tensor([float(bool(re.search(r"\\\\boxed", q))) for q in queries])}}
''')
spec = importlib.util.spec_from_file_location("reward_func", reward_file)   # how
 ↪  OpenRLHF loads it
mod = importlib.util.module_from_spec(spec); spec.loader.exec_module(mod)
out = mod.reward_func(["Q: 2+3? A: \\boxed{5}", "Q: 2+3? A: it is 6"], ["Q: 2+3? A:",
 ↪  "Q: 2+3? A:"], ["5", "5"])
print({k: (v.tolist() if hasattr(v, "tolist") else {kk: vv.tolist() for kk, vv in
 ↪  v.items()}) for k, v in out.items()})
```

A GRPO-style OpenRLHF launch (flags verified against 0.11.2; paths are placeholders):

```
ray start --head --num-gpus 8
python -m openrlhf.cli.train_ppo_ray \
  --actor.model_name_or_path Qwen/Qwen2.5-7B-Instruct \
  --actor.num_nodes 1 --actor.num_gpus_per_node 8 \
  --ref.num_nodes 1 --ref.num_gpus_per_node 8 \
  --vllm.num_engines 4 --vllm.tensor_parallel_size 2 --vllm.enable_prefix_caching \
  --train.colocate_all --vllm.enable_sleep --ds.enable_sleep
   ↪  --vllm.gpu_memory_utilization 0.6 \
  --algo.advantage.estimator group_norm --rollout.n_samples_per_prompt 8 \
  --algo.kl.use_loss --algo.kl.estimator k3 --algo.kl.init_coef 0.001 \
  --rollout.batch_size 128 --train.batch_size 1024 --rollout.max_new_tokens 4096
   ↪  --data.max_len 5120 \
  --data.prompt_dataset /data/math_prompts --data.input_key question --data.label_key
   ↪  answer --data.apply_chat_template \
```

```
  --reward.remote_url /code/math_reward.py --ds.zero_stage 3 --ds.param_dtype bf16 \
  --actor.adam.lr 1e-6 --actor.eps_clip_low_high 0.2 0.28 \
  --ckpt.output_dir /ckpt/run1 --ckpt.save_steps 20 --logger.logging_steps 1
```

## 51.3 veRL

**Architecture.** veRL follows HybridFlow (Chapter 44): a single Ray driver runs the algorithm (`RayPPOTrainer`), while worker groups run SPMD training (FSDP/FSDP2, Megatron, and others) and rollout (vLLM, SGLang). By default the actor, rollout engine and reference are colocated in one "hybrid engine" worker that switches between training and generation, resharding weights in place. The configuration is a Hydra/OmegaConf tree; you override keys on the command line as `key.subkey=value`. The reference YAML `verl/trainer/config/_generated_ppo_trainer.yaml` lists every key with its resolved default.

**Data and rewards.** Training data is a Parquet file whose rows contain `prompt` (a chat message list; key set by `data.prompt_key`), `data_source` (selects the scoring function; key set by `data.reward_fn_key`), `reward_model.ground_truth` and optional `extra_info`. The built-in `default_compute_score(data_source, solution_str, ground_truth, extra_info=None, ...)` covers datasets such as GSM8K and MATH; a custom function is supplied with `reward.custom_reward_function.path` and `.name` (default `compute_score`) and must accept the same arguments. Code rewards can call a sandbox service (`reward.sandbox_fusion`).

**Notable keys** (veRL 0.9.0 defaults): `algorithm.adv_estimator` (`gae`; options include `grpo`, `rloo`, `reinforce_plus_plus`, `remax`, `gpg`, `opo`, `grpo_passk`, `gdpo` and more); `algorithm.norm_adv_by_std_in_grpo` (true; false gives Dr. GRPO); `actor.policy_loss.loss_mode` (`vanilla`; also `gspo`, `sapo`, `cispo`, `clip_cov`, `kl_cov`, `gpg`, `geo_mean`, and others); `actor.loss_agg_mode` (`token-mean`; or `seq-mean-token-sum`, `seq-mean-token-mean`, `seq-mean-token-sum-norm`, `token-sum`); `actor.clip_ratio_c` (3.0, dual-clip bound for negative advantages); `algorithm.filter_groups.enable` (false; DAPO dynamic sampling); `algorithm.rollout_correction` (importance-sampling correction for train/infer mismatch, threshold 2.0); `rollout.multi_turn.enable` (false) with tool configs for agentic RL; `actor.ppo_max_token_len_per_gpu` (16384) with `use_dynamic_bsz` for token-budget micro-batches; `trainer.save_freq` and `trainer.test_freq` ($-1$ = off).

```
gen_yaml = "tools/pkgs/src/verl/verl/trainer/config/_generated_ppo_trainer.yaml"
if os.path.exists(gen_yaml):
    cfg = yaml.safe_load(open(gen_yaml))
    a, r, alg = cfg["actor_rollout_ref"]["actor"], cfg["actor_rollout_ref"]["rollout"],
 ↪  cfg["algorithm"]
    print("adv_estimator:", alg["adv_estimator"], "| rollout.n:", r["n"], "|
     ↪  rollout.name:", r["name"],
          "| ppo_mini_batch_size:", a["ppo_mini_batch_size"], "| clip_ratio:",
           ↪  a["clip_ratio"])
    print("use_kl_loss:", a["use_kl_loss"], a["kl_loss_coef"], a["kl_loss_type"], "|
     ↪  loss_agg_mode:", a["loss_agg_mode"],
          "| policy loss:", a["policy_loss"]["loss_mode"])
    print("rollout gpu_memory_utilization:", r["gpu_memory_utilization"], "|
     ↪  logprobs_mode:", r["logprobs_mode"],
```

```
          "| prefix caching:", r["enable_prefix_caching"], "| tp:",
           ↪  r["tensor_model_parallel_size"])
else:
    print("veRL source not unpacked; see tools/pkgs/README for the pip download command")
```

A GRPO launch with veRL (keys verified against 0.9.0; paths are placeholders):

```
python -m verl.trainer.main_ppo \
  algorithm.adv_estimator=grpo \
  data.train_files=/data/math/train.parquet data.val_files=/data/math/test.parquet \
  data.train_batch_size=256 data.max_prompt_length=1024 data.max_response_length=4096 \
  actor_rollout_ref.model.path=Qwen/Qwen2.5-7B-Instruct \
  actor_rollout_ref.actor.optim.lr=1e-6 actor_rollout_ref.actor.ppo_mini_batch_size=64 \
  actor_rollout_ref.actor.use_dynamic_bsz=true
 ↪  actor_rollout_ref.actor.ppo_max_token_len_per_gpu=24000 \
  actor_rollout_ref.actor.use_kl_loss=true actor_rollout_ref.actor.kl_loss_coef=0.001 \
  actor_rollout_ref.actor.kl_loss_type=low_var_kl
 ↪  actor_rollout_ref.actor.clip_ratio_high=0.28 \
  actor_rollout_ref.rollout.name=vllm actor_rollout_ref.rollout.n=8 \
  actor_rollout_ref.rollout.tensor_model_parallel_size=2
 ↪  actor_rollout_ref.rollout.gpu_memory_utilization=0.6 \
  reward.custom_reward_function.path=/code/math_reward.py \
  trainer.n_gpus_per_node=8 trainer.nnodes=1 trainer.total_epochs=5 trainer.save_freq=20
 ↪  trainer.test_freq=10 \
  trainer.logger='["console","wandb"]' trainer.project_name=grpo_math
 ↪  trainer.experiment_name=run1
```

**Why this, not that? Why OpenRLHF or veRL, not TRL, for large RL runs?**

**Reason.** They are built around Ray placement, separate or colocated vLLM/SGLang engines with NCCL weight sync, Megatron or FSDP2 backends, dynamic token-budget batching, asynchronous and partial rollouts, and PPO with a critic, which matter at tens to thousands of GPUs. **Trade-off.** Larger, faster-moving codebases with many interacting options; more operational work (Ray clusters, Hydra configs); defaults that assume PPO and must be changed for GRPO. **When the alternative is better.** Single-node experiments, LoRA runs, small models, teaching, or when you want the Hugging Face ecosystem's integration (datasets, Hub, PEFT): TRL is simpler and its defaults already follow 2025–26 practice.

**Why this, not that? Why veRL's colocated hybrid engine, not OpenRLHF's default separate placement?**

**Reason.** On a single node or small cluster, colocation keeps every GPU busy in every phase and makes weight sync a local reshard (Chapter 44). **Trade-off.** No overlap of generation and training; memory must be juggled with sleep/offload. **When the alternative is better.** Asynchronous training, heterogeneous hardware, or very large clusters: separate engines (OpenRLHF's original design, veRL's one-step-off and fully asynchronous trainer modes) scale better. Both frameworks now support both styles.

## 51.4 Common mistakes

1. Forgetting to change the estimator *and* the samples per prompt from their PPO defaults.
2. veRL: leaving `actor_rollout_ref.rollout.name` unset (`???` means required).
3. OpenRLHF: assuming `--rollout.max_new_tokens` defaults to a fixed number (it fills the remaining `--data.max_len` budget).
4. Mixing KL-in-reward and KL-in-loss by accident (both frameworks support both).
5. A reward function that returns Python floats where a tensor is expected (OpenRLHF) or a dictionary without a `score` field (veRL).

**Interview check**

**Q1. How do you run GRPO in OpenRLHF?** `--algo.advantage.estimator group_norm` with `--rollout.n_samples_per_prompt` greater than 1, usually `--algo.kl.use_loss` with a small coefficient.
**Q2. How do you run Dr. GRPO in veRL?** `algorithm.adv_estimator=grpo` with `algorithm.norm_adv_by_std_in_grpo=false` and a sequence-sum loss aggregation such as `seq-mean-token-sum-norm`.
**Q3. Where does veRL take the ground-truth answer from?** The `reward_model.ground_truth` field of each Parquet row, with `data_source` choosing the scorer.
**Q4. What does `low_var_kl` mean in veRL?** The k3 estimator $\frac{\pi_{\text{ref}}}{\pi_\theta} - \log \frac{\pi_{\text{ref}}}{\pi_\theta} - 1$, clipped for stability.
**Q5. What architecture does veRL implement?** HybridFlow: single-controller dataflow on Ray, SPMD workers per model, a hybrid engine resharding the actor between training and generation.

**Key takeaways**

- Translate concepts across TRL, OpenRLHF and veRL with the table above.
- OpenRLHF: Ray + DeepSpeed + vLLM, dotted CLI flags, Python-file rewards `reward_func(queries, prompts, labels)`.
- veRL: HybridFlow, Hydra keys, Parquet data with `data_source` and `reward_model.ground_truth`, `compute_score` rewards.
- Both default to PPO/GAE with one sample per prompt.

## 51.5 Generated reference: OpenRLHF `train_ppo_ray` flags

*Generated from the OpenRLHF 0.11.2 source (flags starting with `--`); 167 entries.*

- **`--ref.num_nodes`** = `1`. number of nodes for reference
- **`--ref.num_gpus_per_node`** = `8`. number of gpus per node for reference
- **`--reward.num_nodes`** = `1`. number of nodes for reward model
- **`--reward.num_gpus_per_node`** = `8`. number of gpus per node for reward model
- **`--train.colocate_actor_ref`** = `False`. whether to colocate reference and actor model, if true, they will share same gpus.
- **`--actor.num_nodes`** = `1`. number of nodes for actor

**Notation on this page:** $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights

- **--actor.num_gpus_per_node** = 8. number of gpus per node for actor
- **--critic.num_nodes** = 1. number of nodes for critic
- **--critic.num_gpus_per_node** = 8. number of gpus per node for critic
- **--train.colocate_critic_reward** = False. whether to colocate critic and reward model, if true, they will share same gpus.
- **--train.colocate_all** = False. whether to colocate all models (including vLLM engines), if true, they will share same gpus.
- **--vllm.num_engines** = None. number of vLLM Engines, set to 0 to disable vLLM
- **--vllm.tensor_parallel_size** = 1. tensor parallel size of vLLM Engine for multi-GPU inference
- **--vllm.sync_backend** = 'nccl'. DeepSpeed -> vLLM weight sync backend
- **--vllm.sync_with_ray** = False
- **--vllm.enable_prefix_caching** = False
- **--vllm.enforce_eager** = False. Disable CUDA graph in vLLM
- **--vllm.enable_sleep** = False. Enable sleep mode for vLLM when using –colocate_all_ models
- **--vllm.gpu_memory_utilization** = 0.95. vLLM gpu_memory_utilization
- **--algo.advantage.is_correction_level** = 'off'. Granularity of the gated statistic: off (correction disabled), token (each token), seq (per-sequence mean: geometric mean of the ratio, mean of a divergence; a rejection filter, mask only). Choices: ['off', 'token', 'seq'].
- **--algo.advantage.is_correction_mode** = 'mask'. Out-of-band treatment: mask drops the unit (zero gradient; survivors keep their per-token IS weight), clip clamps the token ratio into [low, high]. Choices: ['mask', 'clip'].
- **--algo.advantage.is_correction_gating** = 'ratio'. Statistic the gate reads: ratio (the IS weight pi_train/pi_rollout itself) or the sampled-token binary_kl / tv divergence between rollout and train policy (a trust region, upper bound only). Choices: ['ratio', 'binary_kl', 'tv'].
- **--algo.advantage.is_correction_threshold** = [0.5, 5.0]. LOW HIGH bounds on the gated statistic, or a single HIGH for an upper bound only (a trust-region delta).
- **--train.async_enable** = False. Enable async training
- **--train.async_queue_size** = 1. Queue size for async sampler<->trainer
- **--train.partial_rollout_enable** = False. Enable partial rollout in async mode. Uses vLLM pause/resume for weight sync instead of locking, allowing generation to overlap with training. In-flight samples may contain tokens from both old and new weights.
- **--eval.steps** = -1
- **--ckpt.save_steps** = -1
- **--logger.logging_steps** = 1
- **--ckpt.path** = './ckpt/checkpoints_ppo_ray'
- **--ckpt.save_hf** = False
- **--ckpt.disable_ds** = False
- **--ckpt.max_num** = 3
- **--ckpt.max_mem** = float('inf')
- **--ckpt.load_enable** = False
- **--ckpt.best_metric_key** = ''. Eval metric key for best checkpoint saving (e.g., eval_ default_pass1). Empty string auto-detects first pass1 metric. Set to 'none' to disable best

checkpoint saving.

- **--ds.use_universal_ckpt** = False. Use deepspeed universal checkpoint
- **--local_rank** = -1. local_rank for deepspeed
- **--ds.zero_stage** = 2. DeepSpeed ZeRO stage
- **--actor.gradient_checkpointing_enable** = False
- **--ds.deepcompile** = False
- **--ds.param_dtype** = 'bf16'. Model data type Choices: ['bf16', 'fp16'].
- **--train.enable_ema** = False. Enable EMA checkpoint for the model.
- **--train.ema_beta** = 0.992. EMA beta coefficient
- **--ds.zpg** = 1. ZeRO++ max partition size
- **--ds.adam_offload** = False. Offload Adam Optimizer
- **--ds.attn_implementation** = 'flash_attention_2'. Attention implementation (e.g., eager, flash_attention_2, flash_attention_3, kernels-community/vllm-flash-attn3)
- **--ds.experts_implementation** = None. MoE expert computation strategy passed to transformers from_pretrained (default: auto — transformers picks grouped_mm when supported, else eager) Choices: ['eager', 'batched_mm', 'grouped_mm', 'deepgemm'].
- **--ds.use_liger_kernel** = False. Enable Liger Kernel
- **--ds.grad_accum_dtype** = None. Adam grad accum data type
- **--ds.overlap_comm** = False
- **--actor.gradient_checkpointing_reentrant** = False
- **--data.disable_fast_tokenizer** = False
- **--data.dataloader_num_workers** = 0. Number of dataloader workers for IO (for Ray training, ensure sufficient CPU resources per actor)
- **--ds.enable_sleep** = False. Enable sleep mode for deepspeed when using –colocate_all_ models
- **--ds.tensor_parallel_size** = 1. DeepSpeed tensor parallel size
- **--ds.packing_samples** = False
- **--train.dynamic_batch_enable** = False
- **--rollout.max_tokens_per_gpu** = None
- **--train.max_tokens_per_gpu** = 16192
- **--ds.load_in_4bit** = False
- **--ds.lora.rank** = 0
- **--ds.lora.alpha** = 16
- **--ds.lora.target_modules** = 'all-linear'
- **--ds.lora.dropout** = 0
- **--ckpt.output_dir** = './ckpt'
- **--train.num_episodes** = 1
- **--rollout.batch_size** = 1024. Batch size for make experience
- **--rollout.vllm_generate_batch_size** = None. Batch size for vLLM generating samples
- **--rollout.micro_batch_size** = 1
- **--train.max_epochs** = 1
- **--data.max_len** = 2048. Max total sequence length (prompt + response)
- **--rollout.max_new_tokens** = None. Max tokens to generate per sample. If None, dynamically computed as max_len - prompt_len per sample.
- **--data.max_samples** = int(100000000.0). Max number of samples

- **--actor.eps_clip** = 0.2. PPO clip range
- **--actor.eps_clip_low_high** = None. PPO-clip low and high
- **--actor.dual_clip** = None. Dual-clip PPO
- **--critic.value_clip** = 0.5. PPO value clip range
- **--algo.advantage.lambd** = 1. PPO GAE lambd
- **--algo.advantage.gamma** = 1. PPO GAE gamma
- **--train.micro_batch_size** = 1. batch size per GPU
- **--train.batch_size** = 128. Global training batch size
- **--reward.normalize_enable** = False. Enable Reward Normalization
- **--rollout.top_p** = 1.0
- **--rollout.temperature** = 1.0
- **--train.seed** = 42
- **--train.full_determinism_enable** = False. Enable reproducible behavior during distributed training
- **--critic.freezing_steps** = -1. Freeze the actor for the first N steps to let the critic warm up
- **--rollout.n_samples_per_prompt** = 1. number of responses for each prompt in generation
- **--critic.save_value_network** = False. Save critic model
- **--algo.kl.target** = None
- **--algo.kl.horizon** = 10000
- **--algo.kl.init_coef** = 0.01. KL penalty in PPO
- **--actor.policy_loss_type** = 'ppo'. Choices: ['ppo', 'gspo'].
- **--actor.loss_agg_mode** = 'token-mean'. Policy-loss aggregation: token-mean (global token mean) or seq-mean-token-mean (every sequence weighs the same). Choices: ['token-mean', 'seq-mean-token-mean'].
- **--algo.kl.estimator** = 'k1'. In GRPO, k3 is utilized as the loss function, while k2, when used as the loss, is nearly equivalent to k1. Choices: ['k1', 'k2', 'k3'].
- **--algo.kl.unbiased_gradient** = False. When using KL as a loss, keep the chosen estimator's value but backprop the true reverse-KL gradient with an importance-sampling correction (w = pi_theta / pi_theta_old) for the off-policy inner loop.
- **--actor.aux_loss_coef** = 0. MoE balancing loss
- **--actor.entropy_coef** = None. Entropy loss coef, set to 0 means only enable entropy logs
- **--reward.clip_range** = (-10, 10). Reward clip range
- **--actor.optim** = 'adam'. Choices: ['adam', 'muon'].
- **--actor.muon.lr** = 0.02. LR for actor's Muon 2D-weight group
- **--actor.muon.momentum** = 0.95
- **--actor.muon.ns_steps** = 5. Newton-Schulz steps (placeholder, DS-ignored)
- **--actor.muon.nesterov** = True
- **--actor.muon.no_nesterov** = flag (sets False)
- **--actor.adam.lr** = 1e-06
- **--actor.adam.betas** = (0.9, 0.95)
- **--actor.adam.eps** = 1e-08
- **--actor.adam.weight_decay** = 0.0
- **--actor.lr_scheduler** = 'cosine_with_min_lr'
- **--actor.lr_warmup_ratio** = 0.03

- **--actor.min_lr_ratio** = 0.1
- **--actor.max_norm** = 1.0
- **--critic.optim** = 'adam'. Choices: ['adam', 'muon'].
- **--critic.muon.lr** = 0.02. LR for critic's Muon 2D-weight group
- **--critic.muon.momentum** = 0.95
- **--critic.muon.ns_steps** = 5. Newton-Schulz steps (placeholder, DS-ignored)
- **--critic.muon.nesterov** = True
- **--critic.muon.no_nesterov** = flag (sets False)
- **--critic.adam.lr** = 9e-06
- **--critic.adam.betas** = (0.9, 0.95)
- **--critic.adam.eps** = 1e-08
- **--critic.adam.weight_decay** = 0.0
- **--critic.lr_scheduler** = 'cosine_with_min_lr'
- **--critic.lr_warmup_ratio** = 0.03
- **--critic.min_lr_ratio** = 0.1
- **--critic.max_norm** = 1.0
- **--algo.advantage.estimator** = 'gae'. Choose advantage estimation method: gae, reinforce, rloo, reinforce_baseline, group_norm, dr_grpo, flash_reinforce (reward minus the rollout-batch mean, no group, no whitening: the n_samples_per_prompt=1 estimator; see examples/scripts/train_flash_reinforce_ray_agent_async.sh) Choices: ['gae', 'reinforce', 'rloo', 'reinforce_baseline', 'group_norm', 'dr_grpo', 'flash_reinforce'].
- **--algo.kl.use_loss** = False. whether to use KL loss from GRPO
- **--algo.advantage.no_std_norm** = False. disable dividing by std for advantages while keeping mean normalization
- **--reward.overlong_buffer_len** = None. reward with optional overlong penalty
- **--reward.overlong_penalty_factor** = 1. overlong penalty factor
- **--reward.stop_properly_penalty_coef** = None. Penalty for truncated samples (finish_reason='length'). If >= 0: multiplicative scaling [0,1]. If < 0: fixed reward override (e.g., -0.5).
- **--ds.ring_attn_size** = 1. Ring attention group size
- **--ds.ring_attn_head_stride** = 1. the number of heads to do ring attention each time. It should be a divisor of the number of heads. A larger value may results in faster training but will consume more memory.
- **--actor.model_name_or_path** = None. HF model name or path
- **--reward.model_name_or_path** = None. HF model name or path
- **--reward.remote_url** = None. remote RM API (HTTP)
- **--critic.model_name_or_path** = None. HF model name or path
- **--ds.value_head_prefix** = 'score'
- **--ref.offload** = False. Offload reference model to CPU
- **--reward.offload** = False. Offload reward model to CPU
- **--train.agent_func_path** = None. Agent script path
- **--data.prompt_dataset** = None. HF dataset name or path
- **--data.prompt_probs** = None. sampling probs for datasets
- **--data.prompt_split** = 'train'
- **--eval.dataset** = None. Path to the evaluation dataset

- **--eval.split** = 'train'
- **--eval.temperature** = 0.6. Temperature for evaluation
- **--eval.n_samples_per_prompt** = 4. Number of samples per prompt for evaluation
- **--data.input_key** = 'input'. JSON dataset key
- **--data.label_key** = None. JSON dataset key
- **--data.input_template** = None
- **--data.apply_chat_template** = False. Use HF tokenizer chat template
- **--logger.wandb.key** = None
- **--logger.wandb.org** = None
- **--logger.wandb.group** = None
- **--logger.wandb.project** = 'openrlhf_train_ppo'
- **--logger.wandb.run_name** = 'ppo_%s' % datetime.now().strftime('%m%dT%H...
- **--algo.dynamic_filtering_enable** = False. Enable dynamic filtering
- **--algo.dynamic_filtering_range** = (0, 1). Dynamic filtering rewards range
- **--data.image_key** = 'images'. Dataset key for image paths/URLs
- **--data.max_images_per_prompt** = 0. Max images per prompt for vLLM (0 = text-only)
- **--actor.freeze_visual_encoder** = False. Freeze vision encoder weights (only train language model). Reduces memory and weight sync overhead.
- **--logger.tensorboard_dir** = None. TensorBoard logging path
- **--use_ms** = False

## 51.6 Generated reference: veRL PPO trainer configuration

### 51.6.1 actor_rollout_ref.actor

*Generated from the veRL 0.9.0 source (`actor_rollout_ref.actor`); 78 entries.*

- **optim.optimizer** = AdamW. Optimizer class name (e.g., "AdamW", "AdamW8bit", "_AdamW", "Adam")
- **optim.optimizer_impl** = torch.optim. Module path to import optimizer Examples: "torch.optim", "torchao.optim", "bitsandbytes.optim"
- **optim.lr** = 1e-06. Learning rate
- **optim.lr_warmup_steps_ratio** = 0.0. Warmup steps ratio (used if lr_warmup_steps is 0 or negative)
- **optim.total_training_steps** = -1. Total training steps (must be overridden at runtime)
- **optim.weight_decay** = 0.01. Weight decay
- **optim.lr_warmup_steps** = -1. Prioritized. None, 0 or Negative values mean delegating to lr_warmup_steps_ratio.
- **optim.betas** = [0.9, 0.999]. Betas for Adam optimizer
- **optim.clip_grad** = 1.0. Clip gradient
- **optim.min_lr_ratio** = 0.0. Minimum LR ratio for cosine schedule
- **optim.num_cycles** = 0.5. Number of cosine cycles in LR schedule
- **optim.lr_scheduler_type** = constant. LR scheduler type: "constant" or "cosine"
- **optim.zero_indexed_step** = true. Whether the LR schedule uses 0-indexed steps
- **optim.warmup_style** = null. deprecated

- **optim.override_optimizer_config** = null. Additional optimizer-specific keyword arguments Example for torchao with bf16 stochastic rounding: optimizer_impl: torchao.optim optimizer: _AdamW override_optimizer_config: bf16_stochastic_round: true
- **fsdp_config.wrap_policy.min_num_params** = 0. Minimum number of parameters to trigger wrapping a layer with FSDP
- **fsdp_config.param_offload** = false. Whether to offload model parameters to CPU (trades speed for memory) Note that this differs from the offload_policy in FSDP
- **fsdp_config.optimizer_offload** = false. Whether to offload optimizer state to CPU Note that this differs from the offload_policy in FSDP
- **fsdp_config.offload_policy** = false. Only for FSDP2: offload param/grad/optimizer during train
- **fsdp_config.reshard_after_forward** = true. Reshard after forward pass to reduce memory footprint For FSDP1, false enables ShardingStrategy.SHARD_GRAD_OP
- **fsdp_config.fsdp_size** = -1. Number of GPUs in each FSDP shard group; -1 means auto
- **fsdp_config.forward_prefetch** = false. Only for FSDP1: FSDP1 configuration, prefetch the next forward-pass all-gather before the current forward computation.
- **fsdp_config.model_dtype** = fp32. model dtype of fsdp
- **fsdp_config.use_orig_params** = false. Whether to use original parameters in fsdp. Only avaiable in fsdp1
- **fsdp_config.seed** = 42. Random seed for reproducibility.
- **fsdp_config.full_determinism** = false. Whether to enable full determinism for distributed training, only for debugging.
- **fsdp_config.ulysses_sequence_parallel_size** = 1. ulysses sequence parallel size
- **fsdp_config.entropy_from_logits_with_chunking** = False (from actor_rollout_ref.actor.entropy.... Whether to use chunked entropy computation
- **fsdp_config.entropy_from_logits_chunk_size** = 2048 (from actor_rollout_ref.actor.entropy_.... Chunk size used by chunked entropy computation
- **fsdp_config.use_torch_compile** = true. Whether to use torch compile in fsdp.
- **fsdp_config.entropy_checkpointing** = False (from actor_rollout_ref.actor.entropy.... Whether to recompute entropy
- **fsdp_config.pad_to_length** = False (from actor_rollout_ref.actor.pad_to_.... Whether to pad packed micro-batches to a bucket-aligned length
- **fsdp_config.pad_to_length_bucket** = 1024. Padding granularity in tokens, on the global packed sequence (before the SP split). Rounded up internally to a multiple of ulysses_sequence_parallel_size.
- **fsdp_config.forward_only** = false. Whether to use forward only in fsdp.
- **fsdp_config.strategy** = fsdp. fsdp or fsdp2
- **fsdp_config.dtype** = bfloat16. Mixed precision training param dtype
- **rollout_n** = 1 (from actor_rollout_ref.rollout.n). Number of rollouts per update (mirrors actor rollout_n)
- **strategy** = fsdp. the abstract actor configs fsdp, fsdp2 or megatron. must be set.
- **ppo_mini_batch_size** = 256. Split each sample into sub-batches of this size for PPO
- **ppo_micro_batch_size** = null. [Deprecated] Global micro batch size
- **ppo_micro_batch_size_per_gpu** = null. Local per-GPU micro batch size
- **use_dynamic_bsz** = false. Whether to automatically adjust batch size at runtime

- **ppo_max_token_len_per_gpu** = 16384. Max tokens per GPU in one PPO batch; affects gradient accumulation Typically it should be: n * ${data.max_prompt_length} + ${data.max_response_length}
- **clip_ratio** = 0.2. PPO clip ratio
- **clip_ratio_low** = 0.2. Lower bound for asymmetric clipping (used in dual-clip PPO)
- **clip_ratio_high** = 0.2. Upper bound for asymmetric clipping (used in dual-clip PPO)
- **tau_pos** = 1.0. Positive and negative tau for smoothing function in SAPO default values used in the paper with Qwen3-30B-A3B-Base
- **tau_neg** = 1.05. negative tau for smoothing function in SAPO
- **freeze_vision_tower** = false. Whether to freeze vision model, if set true, it will be freeze vision model
- **policy_loss.loss_mode** = vanilla. Loss function mode: vanilla / clip-cov / kl-cov /gpg from
- **policy_loss.clip_cov_ratio** = 0.0002. Ratio of tokens to be clipped for clip-cov loss
- **policy_loss.clip_cov_lb** = 1.0. Lower bound for clip-cov loss
- **policy_loss.clip_cov_ub** = 5.0. Upper bound for clip-cov loss
- **policy_loss.kl_cov_ratio** = 0.0002. Ratio of tokens to be applied kl penalty for kl-cov loss
- **policy_loss.ppo_kl_coef** = 0.1. KL divergence penalty coefficient
- **policy_loss.dro_beta** = null. Quadratic log-ratio penalty coefficient for Direct Reward Optimization. Must be set to a positive value when policy_loss.loss_mode is "dro".
- **clip_ratio_c** = 3.0. Constant C in Dual-clip PPO; clips when advantage < 0 and ratio > C
- **loss_agg_mode** = token-mean. Loss aggregation mode: "token-mean", "token-sum", "seq-mean-token-sum", "seq-mean-token-mean", or "seq-mean-token-sum-norm"
- **loss_scale_factor** = null. Scale factor for "seq-mean-token-sum-norm" loss aggregation mode. If null, uses response_length. Set to a constant to ensure consistent normalization.
- **entropy_coeff** = 0. Entropy regularization coefficient in PPO loss
- **calculate_entropy** = false. When true, the actor forward will request entropy from the model
- **calculate_sum_pi_squared** = false. This computes $\Sigma\pi^2$ needed for the Logit-Gradient Norm proxy $W(\tau) = \Sigma_t[1 - 2\pi_t + \Sigma\pi^2]$ c.f.
- **use_kl_loss** = false. Whether to use KL loss instead of KL reward penalty. True for GRPO
- **use_prefix_grouper** = false. Whether to enable PrefixGrouper shared-prefix forward
- **use_torch_compile** = true. Whether to use torch.compile()
- **kl_loss_coef** = 0.001. KL loss coefficient when use_kl_loss is enabled. For GRPO
- **kl_loss_type** = low_var_kl. Type of KL divergence loss. Options: "kl"(k1), "abs", "mse"(k2), "low_var_kl"(k3), "full"
- **ppo_epochs** = 1. Number of PPO epochs per batch
- **shuffle** = false. Shuffle training data across PPO epochs
- **data_loader_seed** = 42. The seed used to construct mini-batch
- **use_fused_kernels** = false (from actor_rollout_ref.model.use_fus.... Whether to use custom fused kernels (e.g., FlashAttention, fused MLP)
- **grad_clip** = 1.0. Gradient clipping for actor updates, specific to the strategy.
- **ulysses_sequence_parallel_size** = 1. Sequence parallelism size for Ulysses-style model

parallelism [DEPRECATED] use fsdp_config.ulysses_sequence_parallel_size instead
- **entropy_from_logits_with_chunking** = false. calculate entropy with chunking to reduce memory peak
- **entropy_from_logits_chunk_size** = 2048. chunk size for entropy from logits with chunking
- **entropy_checkpointing** = false. recompute entropy
- **pad_to_length** = false. round every packed micro-batch up to a fsdp_config.pad_to_length_bucket multiple, so kernel shapes take only a few distinct values
- **use_remove_padding** = false (from actor_rollout_ref.model.use_rem.... Whether to remove padding tokens in inputs during training

### 51.6.2 actor_rollout_ref.rollout

*Generated from the veRL 0.9.0 source (actor_rollout_ref.rollout); 85 entries.*

- **name** = ???. actor_rollout_ref.rollout.name: hf/vllm/sglang/trtllm. The default value will be removed in the future
- **mode** = async. sync: LLM, async: AsyncLLM
- **nnodes** = 0. Number of nodes for standalone rollout server, must be > 0 in one-step-off/fully async training.
- **n_gpus_per_node** = 8 (from trainer.n_gpus_per_node). Number of GPUs per node for rollout server.
- **temperature** = 1.0. Sampling temperature for rollout.
- **top_k** = -1. Top-k sampling parameter. -1 for vLLM rollout, 0 for HF rollout.
- **top_p** = 1. Top-p sampling parameter. Default 1.0.
- **full_determinism** = false. Whether to enable full determinism for reproducibility in rollout. See for details.
- **seed** = 42. Random seed for rollout. Used as the seed for vLLM sampling and enable_full_determinism() when full_determinism is True.
- **prompt_length** = 512 (from data.max_prompt_length). typically the same as data max prompt length same as data.max_prompt_length if it exists
- **response_length** = 512 (from data.max_response_length). typically the same as data max response length same as data.max_response_length if it exists
- **dtype** = bfloat16. for vllm rollout Rollout model parameters type. Align with actor model's FSDP/Megatron type.
- **gpu_memory_utilization** = 0.5. Fraction of GPU memory used by vLLM/SGLang/TRTLLM for KV cache.
- **ignore_eos** = false. Whether to ignore EOS and continue generating after EOS is hit.
- **enforce_eager** = false. Whether to disable CUDA graph. Default False to best performance.
- **cudagraph_capture_sizes** = null. batch size of cudagraph to capture. Require enforce_eager: False to use this option Since cudagraph in inference engine can not be offloaded during update policy, you can use smaller batch size to save memory used in cuda graph, eg: [1,2, 4, 8, 16, 32] supported engines: vllm
- **free_cache_engine** = true. Whether to free engine KVCache after generation.
- **tensor_model_parallel_size** = 2. TP size for rollout. Not effective for hf

- **data_parallel_size** = 1. DP size for rollout
- **expert_parallel_size** = 1. EP size for rollout For MoE models in vllm, EP=1 refers to ETP parallel in fused_moe with TP*DP weight splits, EP>1 (should satisfy EP=TP*DP) refers to EP parallel in fused_moe
- **pipeline_model_parallel_size** = 1. PP size for rollout.
- **max_num_batched_tokens** = 8192. max number of tokens in a batch
- **max_model_len** = null. max length for rollout
- **max_num_seqs** = 256. max length of sequences
- **enable_chunked_prefill** = true. may get higher throughput when set to True. When activated, Please increase max_num_batched_tokens or decrease max_model_len.
- **enable_prefix_caching** = true. Prefix caching kv-cache blocks is a popular optimization in LLM inference to avoid redundant prompt computations.
- **logprobs_mode** = processed_logprobs. logprobs mode for rollout logprobs
- **scheduling_policy** = fcfs. scheduling policy for vllm rollout
- **load_format** = dummy. Which loader to use for rollout model weights: dummy, hf, megatron, etc. safetensors (for huge model, and set use_shm=True); dummy: randomly init model weight
- **layered_summon** = false. for huge model, layered summon can save memory (prevent OOM) but make it slower
- **log_prob_micro_batch_size** = null. [Will be deprecated, use log_prob_micro_batch_size_per_gpu] The batch size for one forward pass in the computation of log_prob. Global batch size.
- **log_prob_micro_batch_size_per_gpu** = null. The batch size for one forward pass in the computation of log_prob. Local batch size per GPU.
- **log_prob_use_dynamic_bsz** = false (from actor_rollout_ref.actor.use_dyn.... enable dynamic batch size (sequence packing) for log_prob computation same as actor_rollout_ref.actor.use_dynamic_bsz if it exists, otherwise false
- **log_prob_max_token_len_per_gpu** = 16384 (from actor_rollout_ref.actor.ppo_max... max token length for log_prob computation same as actor_rollout_ref.actor.ppo_max_token_len_per_gpu if it exists, otherwise 16384
- **disable_log_stats** = true. disable logging statistics
- **do_sample** = true. for hf rollout Whether to sample during training rollout. False uses greedy sampling.
- **n** = 1. number of responses (i.e. num sample times). > 1 for grpo
- **over_sample_rate** = 0. The over_sample_rate parameter controls the early termination threshold for training rollouts, where the system will abort remaining requests when (1 - over_sample_rate) * total_requests completions are reached.
- **multi_stage_wake_up** = false. Whether to wake up inference engine in multi-stage for SGLang to reduce peak memory during training-rollout transition. This is only effective for SGLang rollout.
- **engine_kwargs.vllm** = {}. vllm engine config
- **engine_kwargs.sglang** = {}. sglang engine config
- **engine_kwargs.trtllm** = {}. trtllm engine config
- **val_kwargs.top_k** = -1. sampling parameters for validation Top-k sampling parameter. -1 for vLLM rollout, 0 for HF rollout.

- **val_kwargs.top_p** = 1.0. Top-p sampling parameter. Default 1.0.
- **val_kwargs.temperature** = 0. Sampling temperature for rollout.
- **val_kwargs.n** = 1. whether to repeat n times for validation
- **val_kwargs.do_sample** = false. Whether to sample during training rollout. False uses greedy sampling.
- **multi_turn.enable** = false. set to True for multi-turn tool tasks; should set rollout.name to sglang as well
- **multi_turn.max_assistant_turns** = null. null for no limit (default max_length // 3)
- **multi_turn.tool_config_path** = null. YAML config for stateful tools that subclass BaseTool. Use this when a tool needs per-trajectory state, async create/release lifecycle hooks, or custom rewards. null disables this code path.
- **multi_turn.function_tool_path** = null. Path to a Python file with @function_tool-decorated functions, loaded once per AgentLoopWorker. Use this for stateless tools where the OpenAI-compatible JSON schema can be inferred from the function signature and Google-style docstring.
- **multi_turn.max_user_turns** = null. null for no limit (default max_length // 3)
- **multi_turn.max_parallel_calls** = 1. max parallel call for tools in single turn
- **multi_turn.max_tool_response_length** = 256. max length of tool response
- **multi_turn.tool_response_truncate_side** = middle. truncate side of tool response: left, middle, right
- **multi_turn.use_inference_chat_template** = false. - When set to True, the model's default chat template is used for multi-turn rollout, which typically matches production behavior. - When set to False, the token ids recorded for training are used instead; unlike the default chat template, these always include the model's full output, which may contain additional content such as reasoning content.
- **multi_turn.tokenization_sanity_check_mode** = strict. Tokenization is performed turn by turn and the resulting token ids are concatenated to form the full conversation. To ensure this matches the result of tokenizing the entire conversation at once, a sanity check is run at the end of each multi-turn rollout to compare the two sets of token ids.
- **multi_turn.format** = hermes. Format of the multi-turn rollout. Options: hermes, llama3_json,...
- **multi_turn.num_repeat_rollouts** = null. Number of repeat rollouts for each sample
- **calculate_log_probs** = true. support logging rollout prob for debugging purpose "Truncated importance sampling" requires rollout log probs, set to True when turning on Truncated importance sampling
- **agent.num_workers** = 8. Number of agent loop workers
- **agent.default_agent_loop** = single_turn_agent. default agent loop to use if agent_name not set in RL dataset
- **agent.agent_loop_config_path** = null. custom agent loop config path, which should contain list of configs to initialize AgentLoop instances. - name: react_agent _target_: recipe. langgraph_agent.react_agent_loop.ReactAgentLoop tools: ["get_current_temperature"] - name: math_expression _target_: recipe.langgraph_agent.example.math_expression. MathExpressionReactAgentLoop min_terms: 2 max_terms: 6
- **agent.custom_async_server.path** = null. Path to the custom async server implementation
- **agent.custom_async_server.name** = null. Class name of the custom async server class

(e.g. AsyncvLLMServer)

- **checkpoint_engine.backend** = naive. Backend for checkpoint engine: naive, nccl, nixl, hccl
- **checkpoint_engine.update_weights_bucket_megabytes** = 2048. Specifies the tensor bucket size (in megabytes) for batch weight updates during rollout operations. This parameter controls the maximum payload size for a single weight update request.
- **checkpoint_engine.engine_kwargs** = {}. Additional keyword arguments to pass to the checkpoint engine constructor
- **checkpoint_engine.custom_backend_module** = null. If set, this Python module is imported on every worker process before the backend is instantiated, allowing custom backends to register themselves in CheckpointEngineRegistry.
- **skip_tokenizer_init** = true. Whether to skip tokenizer initialization for rollout engine When enabled (True), the rollout assume token in token out for generation
- **enable_rollout_routing_replay** = false. Whether to enable rollout routing replay for MoE models When enabled (True), the rollout will record the routing decisions.
- **moe_load_balance_metrics_interval** = 0. Interval for logging rollout MoE load-balance metrics from routing replay data. Set to 0 to disable.
- **prometheus.enable** = false. whether enable prometheus on server mode rollout
- **prometheus.port** = 9090. Port number that Prometheus listens on, default is 9090
- **prometheus.file** = /tmp/ray/session_latest/metrics/prometheus/.... Path to Prometheus configuration file
- **prometheus.served_model_name** = null (from actor_rollout_ref.model.path). Specify served_model_name to avoid displaying overly long model paths in Grafana
- **quantization** = null. type of quantization in vllm, currently support fp8 and torchao
- **quantization_config_file** = null. extra quantization information serialized in a config file, e.g. torchao_config.json
- **disaggregation.enabled** = false. Master switch for PD disaggregation.
- **disaggregation.prefill_replicas** = 1. Number of prefill replicas (each uses tensor_model_parallel_size GPUs).
- **disaggregation.decode_replicas** = 1. Number of decode replicas.
- **disaggregation.decode_tensor_model_parallel_size** = null. Decode TP override; null = same as tensor_model_parallel_size.
- **disaggregation.transfer_backend** = nixl. KV cache transport: nixl | mooncake | ascend | mori | fake.
- **disaggregation.bootstrap_port** = null. Bootstrap server port on prefill side; null = auto-assign.
- **disaggregation.ib_device** = null. InfiniBand device (required for mooncake over RDMA).

### 51.6.3 actor_rollout_ref.model

*Generated from the veRL 0.9.0 source (actor_rollout_ref.model); 37 entries.*

- **path** = ~/models/deepseek-llm-7b-chat. path to the huggingface model
- **hf_config_path** = null. config to the huggingface config. In case it is not the same as path
- **tokenizer_path** = null. path to the huggingface tokenizer. In case it is not the same as path

- **use_shm** = false. whether to use shared memory for model loading
- **trust_remote_code** = false. whether to trust remote code.
- **custom_chat_template** = null. custom chat template for the model
- **external_lib** = null. whether to use external libs for the model
- **override_config** = {}. override hf config
- **enable_gradient_checkpointing** = true. whether to enable gradient checkpointing. Only valid when we use hf model definition
- **enable_activation_offload** = false. whether to enable activation offload. Only valid when we use hf model definition
- **use_remove_padding** = true. whether to use remove padding. Only valid when we use hf model definition
- **lora_rank** = 0. Set to positive value to enable LoRA (e.g., 32)
- **lora_alpha** = 16. LoRA scaling factor
- **target_modules** = all-linear. Target modules for LoRA adaptation
- **exclude_modules** = null. Exclude modules from LoRA adaptation
- **lora_adapter_path** = null. Path to pre-trained LoRA adapter to load for continued training
- **use_liger** = false. whether to use liger. Only valid when we use hf model definition
- **use_fused_kernels** = false. whether to use fused kernels.
- **fused_kernel_options.impl_backend** = torch. the implementation backend for fused kernels.
- **tiled_mlp.enabled** = false. whether to enable TiledMLP
- **tiled_mlp.num_shards** = 4. number of shards to split the input. Higher values reduce peak memory but may slightly impact performance.
- **lora.type** = lora. LoRA type: "lora", "vlm_lora", "canonical_lora", or "dora"
- **lora.merge** = false. whether to sync weights / refit by either merging LoRA adapters into the base model weights before transferring to vLLM (for better inference speed but more refit time and potential precision loss).
- **lora.rank** = 0. LoRA rank (Dimension of the low-rank projection space.). Set to 0 to disable LoRA
- **lora.alpha** = 32. Weighting factor for the low-rank projection. Defaults to 32
- **lora.dropout** = 0.0. Dropout rate for the low-rank projection. Defaults to 0.0
- **lora.target_modules** = ['linear_qkv', 'linear_proj', 'linear_fc1',.... A list of module names to apply LoRA to. For fused LoRA, Defaults to all linear layers ['linear_qkv', 'linear_proj', 'linear_fc1', 'linear_fc2'].
- **lora.exclude_modules** = []. A list of module names not to apply LoRa to. It will match all nn.Linear & nn.Linear-adjacent modules whose name does not match any string in exclude_modules. If used, will require target_modules to be empty list or None
- **lora.dropout_position** = pre. Position for applying dropout, can be 'pre' (before the low-rank projection) or 'post' (after). Defaults to 'pre'
- **lora.lora_A_init_method** = xavier. Initialization method for the low-rank matrix A. Defaults to "xavier".
- **lora.lora_B_init_method** = zero. Initialization method for the low-rank matrix B. Defaults to "zero".
- **lora.a2a_experimental** = false. Enables the experimental All-to-All (A2A) communication strategy. Defaults to False

- **lora.dtype** = null. Parameter data type for LoRA weights. Default to null, which will use model's dtype.
- **lora.adapter_path** = null. Path to pre-trained LoRA adapter weights (null to train from scratch)
- **lora.freeze_vision_model** = true. VLMLoRA additionally allows the user to specify whether the language or vision models should be frozen. For example, a common finetuning workload for multimodal models is to apply adapters to language model and fully finetune the vision model.
- **lora.freeze_vision_projection** = true
- **lora.freeze_language_model** = true

### 51.6.4 algorithm

*Generated from the veRL 0.9.0 source (algorithm); 24 entries.*

- **rollout_correction.rollout_is** = null. Rollout Correction: corrects off-policy distribution shifts See documentation: docs/algo/rollout_corr.md Use presets: RolloutCorrectionConfig.decoupled_seq_is(),.bypass_pg_is(), etc.
- **rollout_correction.rollout_is_threshold** = 2.0. Threshold for IS weighting: - float / float-like string: TIS upper bound - "lower_upper" string: IcePop bounds
- **rollout_correction.rollout_rs** = null. RS aggregation level: null (disabled), e.g. "token_k1", "seq_sum_k1", "seq_mean_k3"
- **rollout_correction.rollout_rs_threshold** = null. Threshold for rejection sampling (string or float; see code docs)
- **rollout_correction.bypass_mode** = false. Operating mode: false = Decoupled (3 policies), true = Bypass (2 policies)
- **rollout_correction.loss_type** = ppo_clip. Loss type in bypass mode (bypass_mode= true): - "ppo_clip": PPO clipped objective (IS handled by ratio, default) - "reinforce": REINFORCE with explicit IS weights (no PPO clipping)
- **rollout_correction.rollout_is_batch_normalize** = false. Batch normalize IS weights: false = raw weights, true = normalize to mean=1.0
- **gamma** = 1.0. Discount factor for future rewards
- **lam** = 1.0. Trade-off between bias and variance in the GAE estimator
- **adv_estimator** = gae. Advantage estimator type: "gae", "grpo", "reinforce_plus_plus", etc.
- **norm_adv_by_std_in_grpo** = true. Whether to normalize advantages by std (specific to GRPO)
- **filter_groups.enable** = false. Whether to enable group filtering
- **filter_groups.metric** = null. Metric used to identify groups with uniform rewards, acc / score / seq_reward / seq_final_reward /...
- **filter_groups.max_num_gen_batches** = 0. Maximum number of generation batches; non-positive values mean no upper limit
- **filter_groups.max_inflight_gen_batches** = 1. Maximum Sync DAPO prompt batches concurrently in flight, measured in train-batch units
- **use_kl_in_reward** = false. Whether to enable in-reward KL penalty
- **kl_penalty** = kl. How to estimate KL divergence: "kl", "abs", "mse", "low_var_kl", or

"full"
- **kl_ctrl.type** = fixed. KL control type: "fixed" or "adaptive"
- **kl_ctrl.kl_coef** = 0.001. Initial coefficient for KL penalty
- **kl_ctrl.horizon** = 10000. Horizon value for adaptive controller (if enabled)
- **kl_ctrl.target_kl** = 0.1. Target KL divergence (used for adaptive controller)
- **use_pf_ppo** = false. Whether to enable preference feedback PPO
- **pf_ppo.reweight_method** = pow. Method for reweighting samples: "pow", "max_min", or "max_random"
- **pf_ppo.weight_pow** = 2.0. Power used for weight scaling in "pow" method

### 51.6.5 data

*Generated from the veRL 0.9.0 source (data); 37 entries.*

- **tokenizer** = null. Tokenizer class or path. If null, it will be inferred from the model.
- **use_shm** = false. Whether to use shared memory for data loading.
- **train_files** = ~/data/rlhf/gsm8k/train.parquet. Training set parquet. Can be a list or a single file. The program will read all files into memory, so it can't be too large (< 100GB). The path can be either a local path or an HDFS path.
- **val_files** = ~/data/rlhf/gsm8k/test.parquet. Validation parquet. Can be a list or a single file.
- **train_max_samples** = -1. Maximum sample length to be used. Set to -1 to use full dataset, otherwise, randomly select the specified number of samples from train dataset
- **val_max_samples** = -1. Maximum sample length to be used. Set to -1 to use full dataset, otherwise, randomly select the specified number of samples from val dataset
- **prompt_key** = prompt. The field in the dataset where the prompt is located. Default is 'prompt'.
- **reward_fn_key** = data_source. The field used to select the reward function (if using different ones per example).
- **max_prompt_length** = 512. Maximum prompt length. All prompts will be left-padded to this length. An error will be reported if the length is too long.
- **max_response_length** = 512. Maximum response length. Rollout in RL algorithms (e.g. PPO) generates up to this length.
- **train_batch_size** = 1024. Batch size sampled for one training iteration of different RL algorithms.
- **gen_batch_size** = null. Number of prompts fetched from the dataloader per fetch.
- **val_batch_size** = null. Batch size used during validation. Can be null.
- **tool_config_path** = null (from actor_rollout_ref.rollout.multi_.... use tool config to calculate true prompt length
- **function_tool_path** = null (from actor_rollout_ref.rollout.multi_.... for @function_tool entries
- **return_raw_input_ids** = false. Whether to return the original input_ids without adding chat template. This is used when the reward model's chat template differs from the policy. If using a model-based RM with different templates, this should be True.
- **return_raw_chat** = true. Whether to return the original chat (prompt) without applying chat template.

- **return_full_prompt** = false. Whether to return the full prompt with chat template.
- **shuffle** = true. Whether to shuffle the data in the dataloader.
- **seed** = null. Seed to use when shuffling the data
- **dataloader_num_workers** = 8. num dataloader workers
- **image_patch_size** = 14. image patch size
- **validation_shuffle** = false. Whether to shuffle the validation set.
- **filter_overlong_prompts** = false. Whether to filter overlong prompts.
- **filter_overlong_prompts_workers** = 1. Number of workers for filtering overlong prompts. For large-scale datasets, filtering can be time-consuming. Use multiprocessing to speed up. Default is 1.
- **truncation** = error. Truncate the input_ids or prompt if they exceed max_prompt_length. Options: 'error', 'left', 'right', 'middle'. Default is 'error'.
- **image_key** = images. The field in the multi-modal dataset where the image is located. Default is 'images'.
- **video_key** = videos. The field in the multi-modal dataset where the video is located.
- **audio_key** = audios. The field in the multi-modal dataset where the audio is located.
- **trust_remote_code** = false. If the remote tokenizer has a Python file, this flag determines whether to allow using it.
- **custom_cls.path** = null. The path to the file containing your customized dataset class. If not specified, pre-implemented dataset will be used.
- **custom_cls.name** = null. The name of the dataset class within the specified file.
- **return_multi_modal_inputs** = true. Whether to return multi-modal inputs in the dataset. Set to False if rollout generates new multi-modal inputs.
- **apply_chat_template_kwargs** = {}. Additional kwargs when calling tokenizer.apply_chat_template
- **continuous_token.enable** = false. Enable Continuous Token for text-only multi-turn PPO rollout.
- **continuous_token.model_family** = auto. Model-family specific boundary handling. auto infers from model path/tokenizer name. Options: auto, default, qwen, qwen25, qwen3, qwen35, minimax, minimaxm2, minimaxm25, minimaxm27, glm47, glm5, gemma4, gptoss
- **mm_processor_kwargs** = {}. Additional kwargs passed to multimodal processors/backends.

### 51.6.6 critic

The critic's model, optim and fsdp_config subsections mirror the actor's and are omitted.

*Generated from the veRL 0.9.0 source (critic, without the model, fsdp_config, optim subsections); 40 entries.*

- **fsdp.wrap_policy.min_num_params** = 0
- **fsdp.param_offload** = false
- **fsdp.optimizer_offload** = false
- **fsdp.offload_policy** = false
- **fsdp.reshard_after_forward** = true
- **fsdp.fsdp_size** = -1

- **fsdp.forward_prefetch** = false
- **fsdp.model_dtype** = fp32
- **fsdp.use_orig_params** = false
- **fsdp.seed** = 42
- **fsdp.full_determinism** = false
- **fsdp.ulysses_sequence_parallel_size** = 1
- **fsdp.entropy_from_logits_with_chunking** = false
- **fsdp.entropy_from_logits_chunk_size** = 2048
- **fsdp.use_torch_compile** = true
- **fsdp.entropy_checkpointing** = false
- **fsdp.pad_to_length** = false
- **fsdp.pad_to_length_bucket** = 1024
- **fsdp.forward_only** = false
- **fsdp.strategy** = fsdp
- **fsdp.dtype** = bfloat16
- **rollout_n** = 1 (from actor_rollout_ref.rollout.n). Number of rollouts per update (mirrors actor rollout_n)
- **strategy** = fsdp. fsdp or fsdp2 strategy used for critic model training
- **enable** = null. whether to enable the critic worker. by default it is only enabled if advantage estimator is gae set it to True manually if you always want to enable critic worker
- **ppo_mini_batch_size** = 256 (from actor_rollout_ref.actor.ppo_mini_.... PPO mini-batch size per update
- **ppo_micro_batch_size** = null. [Deprecated] Global micro batch size
- **ppo_micro_batch_size_per_gpu** = null (from .ppo_micro_batch_size). Local per-GPU micro batch size
- **use_dynamic_bsz** = false (from actor_rollout_ref.actor.use_dyn.... Whether to automatically adjust batch size at runtime
- **ppo_max_token_len_per_gpu** = 32768. Max tokens per GPU in one PPO batch (doubled for critic)
- **forward_max_token_len_per_gpu** = ${.ppo_max_token_len_per_gpu}. Max token length per GPU in forward pass
- **ppo_epochs** = 1 (from actor_rollout_ref.actor.ppo_epochs). Number of PPO epochs per batch
- **shuffle** = false (from actor_rollout_ref.actor.shuffle). Shuffle training data across PPO epochs
- **data_loader_seed** = 42. The seed used to construct mini-batch
- **cliprange_value** = 0.5. PPO value function clipping range
- **loss_agg_mode** = token-mean (from actor_rollout_ref.actor.lo.... Loss aggregation mode: "token-mean", "seq-mean-token-sum", or "seq-mean-token-mean"
- **loss_scale_factor** = null (from actor_rollout_ref.actor.loss_sca.... Scale factor for "seq-mean-token-sum-norm" loss aggregation mode. If null, uses response_length. Set to a constant to ensure consistent normalization.
- **forward_micro_batch_size** = null (from .ppo_micro_batch_size). Forward-only batch size during inference (global)
- **forward_micro_batch_size_per_gpu** = null (from .ppo_micro_batch_size_per_gpu).

Forward-only batch size during inference (per GPU)
- **`ulysses_sequence_parallel_size`** = `1`. Sequence parallelism size for Ulysses-style model parallelism [DEPRECATED] use fsdp_config.ulysses_sequence_parallel_size instead
- **`grad_clip`** = `1.0`. Gradient clipping for critic updates

### 51.6.7 `reward`

*Generated from the veRL 0.9.0 source (`reward`); 57 entries.*

- **`num_workers`** = `8`. we launch num_workers reward managers to parallelize the reward computation
- **`custom_reward_function.path`** = `null`. The path to the file containing your customized reward function. If not specified, pre-implemented reward functions will be used.
- **`custom_reward_function.name`** = `compute_score`. The name of the reward function within the specified file. Default is 'compute_score'.
- **`reward_manager.source`** = `register`
- **`reward_manager.name`** = `naive`
- **`reward_manager.module.path`** = `null`
- **`reward_manager.module.name`** = `custom_reward_manager`
- **`reward_model.enable`** = `false`
- **`reward_model.enable_resource_pool`** = `false`. Whether to deploy the model to a separate resource pool. If true, n_gpus_per_node & nnodes will be used to determine the resource node.
- **`reward_model.n_gpus_per_node`** = `8`
- **`reward_model.nnodes`** = `0`
- **`reward_model.model_path`** = `null`
- **`reward_model.rollout.name`** = `???`
- **`reward_model.rollout.dtype`** = `bfloat16`
- **`reward_model.rollout.gpu_memory_utilization`** = `0.5`
- **`reward_model.rollout.enforce_eager`** = `true`
- **`reward_model.rollout.cudagraph_capture_sizes`** = `null`
- **`reward_model.rollout.free_cache_engine`** = `true`
- **`reward_model.rollout.data_parallel_size`** = `1`
- **`reward_model.rollout.expert_parallel_size`** = `1`
- **`reward_model.rollout.tensor_model_parallel_size`** = `2`
- **`reward_model.rollout.pipeline_model_parallel_size`** = `1`
- **`reward_model.rollout.max_num_batched_tokens`** = `8192`
- **`reward_model.rollout.max_model_len`** = `null`
- **`reward_model.rollout.max_num_seqs`** = `1024`
- **`reward_model.rollout.load_format`** = `auto`
- **`reward_model.rollout.layered_summon`** = `false`
- **`reward_model.rollout.scheduling_policy`** = `fcfs`
- **`reward_model.rollout.full_determinism`** = `false`
- **`reward_model.rollout.seed`** = `42`
- **`reward_model.rollout.multi_stage_wake_up`** = `false`
- **`reward_model.rollout.engine_kwargs`** = `{}`

- **reward_model.rollout.limit_images** = null
- **reward_model.rollout.enable_chunked_prefill** = true
- **reward_model.rollout.enable_prefix_caching** = true
- **reward_model.rollout.disable_log_stats** = true
- **reward_model.rollout.skip_tokenizer_init** = false
- **reward_model.rollout.ignore_eos** = false
- **reward_model.rollout.temperature** = 1.0
- **reward_model.rollout.top_k** = -1
- **reward_model.rollout.top_p** = 1
- **reward_model.rollout.do_sample** = true
- **reward_model.rollout.n** = 1
- **reward_model.rollout.prompt_length** = 2048
- **reward_model.rollout.response_length** = 2048
- **reward_model.rollout.quantization** = null
- **reward_model.rollout.quantization_config_file** = null
- **reward_model.rollout.disaggregation.enabled** = false
- **reward_model.rollout.disaggregation.prefill_replicas** = 1
- **reward_model.rollout.disaggregation.decode_replicas** = 1
- **reward_model.rollout.disaggregation.decode_tensor_model_parallel_size** = null
- **reward_model.rollout.disaggregation.transfer_backend** = nixl
- **reward_model.rollout.disaggregation.bootstrap_port** = null
- **reward_model.rollout.disaggregation.ib_device** = null
- **sandbox_fusion.url** = null. Cloud /local function URL for sandbox execution
- **sandbox_fusion.max_concurrent** = 64. Max concurrent requests allowed to sandbox
- **sandbox_fusion.memory_limit_mb** = 1024. Max memory limit for each sandbox process in MB

### 51.6.8 trainer

*Generated from the veRL 0.9.0 source (`trainer`); 38 entries.*

- **balance_batch** = true. Whether to balance batch sizes across distributed workers
- **total_epochs** = 30. Number of epochs in training
- **total_training_steps** = null. Total training steps (can be set explicitly or derived from epochs)
- **project_name** = verl_examples. Project name for experiment tracking (e.g., wandb)
- **experiment_name** = gsm8k. Experiment name for run identification in tracking tools
- **logger** = ['console', 'wandb']. Logging backends to use: "console", "wandb", etc.
- **log_val_generations** = 0. Number of generations to log during validation
- **rollout_data_dir** = null. Directory for logging rollout data; no dump if null
- **validation_data_dir** = null. Directory for logging validation data; no dump if null
- **nnodes** = 1. Number of nodes used in the training
- **n_gpus_per_node** = 8. Number of GPUs per node
- **save_freq** = -1. Save frequency (by iteration) for model checkpoints
- **esi_redundant_time** = 0. ESI refers to the elastic server instance used during training, similar to the training plan. For example, if you purchase 10 hours of computing power, the

ESI will automatically shut down after 10 hours of training.

- **resume_mode** = auto. Resume mode: "auto", "disable", or "resume_path" "auto": resume from last checkpoint if available "disable": start from scratch "resume_path": resume from a user-defined path
- **resume_from_path** = null. Path to resume training from (only used when resume_mode is "resume_path")
- **val_before_train** = true. Whether to run validation before training begins
- **val_only** = false. Whether to run validation only
- **test_freq** = -1. Validation frequency (in training iterations)
- **critic_warmup** = 0. Number of iterations to warm up the critic before updating policy
- **default_hdfs_dir** = null. Default path to distributed filesystem for saving checkpoints
- **del_local_ckpt_after_load** = false. Whether to delete local checkpoints after loading
- **default_local_dir** = checkpoints/${trainer.project_name}/${train.... Default local directory for saving checkpoints
- **max_actor_ckpt_to_keep** = null. Maximum number of actor checkpoints to keep
- **max_critic_ckpt_to_keep** = null. Maximum number of critic checkpoints to keep
- **ray_wait_register_center_timeout** = 300. Timeout (in seconds) for Ray worker to wait for registration
- **device** = cuda. Device to run training on (e.g., "cuda", "cpu")
- **use_v1** = true. Whether to use V1 trainer
- **v1.trainer_mode** = sync. Trainer mode: "sync", "colocate_async", "separate_async"
- **v1.sync** = {}. Synchronous PPO trainer
- **v1.colocate_async.num_warmup_batches** = 1. Number of warmup batches to add before training loop starts
- **v1.separate_async.num_warmup_batches** = 1. Number of warmup batches to add before training loop starts
- **v1.separate_async.parameter_sync_step** = 4. Frequency of parameter synchronization between trainer and rollout
- **v1.sampler.max_off_policy_threshold** = 8. Maximum number of model versions that trajectory can span
- **v1.sampler.max_off_policy_strategy** = drop. How to handle trajectory that exceeds the maximum number of model versions drop: drop the trajectory wait: dropless, wait all trajectories that reach threshold to finish
- **v1.sampler.sync_refill_failed_groups** = false. Whether sync trainers replace failed groups that have no trajectories
- **v1.sampler.custom_sampler.path** = null. Path to the custom sampler class
- **v1.sampler.custom_sampler.name** = null. Name of the custom sampler class
- **v1.sampler.sampler_kwargs** = {}. Additional kwargs for the custom sampler

CHAPTER 52

# Unsloth

**Why this chapter matters.**
Unsloth is a popular library for fast, memory-efficient fine-tuning on one GPU. It patches Hugging Face models with hand-written Triton kernels and a smarter gradient-checkpointing scheme, wraps PEFT and TRL, and can run GRPO with vLLM inside the same process (`fast_inference=True`). It is common in take-home tasks and small-company stacks, so you should know its entry points and defaults (Unsloth 2026.10.7, read from the source; it needs an NVIDIA, AMD or Intel GPU, so the code here is not executed).

## 52.1 The workflow

```
from unsloth import FastLanguageModel        # import before transformers/trl so patches
 ↪ apply
from trl import SFTConfig, SFTTrainer

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/Qwen2.5-7B-Instruct",
    max_seq_length=4096,
    load_in_4bit=True,          # default True: QLoRA-style 4-bit base weights
    fast_inference=False,       # True starts a vLLM engine in-process (for GRPO)
)
model = FastLanguageModel.get_peft_model(
    model, r=16, lora_alpha=16, lora_dropout=0.0,
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj",
 ↪ "down_proj"],
    use_gradient_checkpointing="unsloth",   # offloads checkpointed activations to CPU
 ↪ asynchronously
    random_state=3407,
)
trainer = SFTTrainer(model=model, processing_class=tokenizer, train_dataset=ds,
                     args=SFTConfig(per_device_train_batch_size=2,
 ↪ gradient_accumulation_steps=8,
                                    learning_rate=2e-4, max_steps=500, output_dir="out"))
trainer.train()
FastLanguageModel.for_inference(model)                        # switch to fast generation
 ↪ mode
model.save_pretrained_merged("merged", tokenizer, save_method="merged_16bit")
model.save_pretrained_gguf("gguf", tokenizer, quantization_method="q4_k_m")
```

## 52.2 The parameters that matter

**`FastLanguageModel.from_pretrained`** (defaults from the source):

- `model_name` (`"unsloth/Llama-3.2-1B-Instruct"`), `max_seq_length` (2048), `dtype` (None = automatic: bfloat16 where supported, else float16).
- Precision of the base weights: `load_in_4bit` (**True**), `load_in_8bit` (False), `load_in_16bit` (False), `load_in_fp8` (False), and `full_finetuning` (False; True trains all weights instead of LoRA).
- `use_gradient_checkpointing` (`"unsloth"`): Unsloth's variant offloads checkpointed activations to CPU memory with asynchronous copies, which allows longer contexts than standard checkpointing at a small speed cost.
- `fast_inference` (False): start a vLLM engine sharing the model's weights for fast generation, used by GRPO; with it, `gpu_memory_utilization` (0.5) sets vLLM's memory share and `max_lora_rank` (64) caps the LoRA rank vLLM can serve; `float8_kv_cache` (False) stores vLLM's KV cache in FP8.
- `random_state` (3407), `trust_remote_code` (False), `token`, `device_map`, `rope_scaling`, `fix_tokenizer` (True), `unsloth_tiled_mlp` (False; tiles MLP computation over the sequence to cut activation memory for very long contexts).

**`get_peft_model`**: `r` (16), `target_modules` (the seven attention and MLP projections), `lora_alpha` (16), `lora_dropout` (0.0), `bias` (`"none"`), `use_gradient_checkpointing` (`"unsloth"`), `random_state` (3407), `use_rslora` (False), `loftq_config` (`{}`), `modules_to_save`, `init_lora_weights` (True), `qat_scheme` (None; quantisation-aware training), `max_seq_length` (accepted only for backward compatibility; the source marks it "not used anymore"). Note that these defaults differ from PEFT's (`r=8`, `lora_alpha=8`, `target_modules=None`).

**Saving.** `save_pretrained_merged(save_method=...)`: `"lora"` (adapter only), `"merged_16bit"` (default; dequantise and merge into 16-bit weights, the right choice for vLLM serving), `"merged_4bit"`. `save_pretrained_gguf(quantization_method=...)` exports for llama.cpp; the default is `"fast_quantized"` (q8_0), with `"quantized"` (q4_k_m), `"not_quantized"`, `"f16"`, `"bf16"`, `"q8_0"`, `"q4_k_m"`, `"q5_k_m"` and others; the `iq*` types need an importance matrix file.

**RL.** Unsloth patches TRL's GRPO trainer so that it uses the in-process vLLM engine and its memory-efficient loss (the source also exports a `PatchFastRL` helper used by older recipes). Use TRL's `GRPOConfig` as in Chapter 50, with `use_vllm=True`.

```
import ast, os
src = "tools/pkgs/src/unsloth/unsloth/save.py"
if os.path.exists(src):
    tree = ast.parse(open(src).read())
    node = next(n for n in tree.body if isinstance(n, ast.Assign) and
 ↪  getattr(n.targets[0], "id", "") == "ALLOWED_QUANTS")
    quants = ast.literal_eval(node.value)
    print(len(quants), "GGUF quantization methods accepted without an importance
     ↪  matrix:")
    for k in ["not_quantized", "fast_quantized", "quantized", "q8_0", "q4_k_m",
     ↪  "q5_k_m"]:
```

```
        print(f"  {k:15s} {quants[k]}")
else:
    print("Unsloth source not unpacked")
```

**Why this, not that? Why Unsloth, not plain Transformers + PEFT + TRL?**

**Reason.** On a single GPU, its fused kernels and activation offloading make LoRA/QLoRA training substantially faster and fit longer sequences or bigger models; GRPO with in-process vLLM shares weights instead of holding two copies. **Trade-off.** It monkey-patches libraries (import order matters, and upgrades of Transformers or TRL can break patches), multi-GPU support has historically lagged the single-GPU path, and debugging inside patched code is harder. **When the alternative is better.** Multi-node training, custom architectures, research that needs exact control of the loss, or production pipelines that must track upstream library versions closely: use the plain Hugging Face stack, OpenRLHF or veRL.

## 52.3 Where Unsloth's savings come from

Unsloth's speed and memory claims come from a handful of ideas, each visible in the source (`unsloth/kernels/` and the companion package `unsloth_zoo`). None is magic; each is a standard trick implemented carefully.

1. **Fused linear cross-entropy.** The output layer produces a $B \times T \times |V|$ logits tensor. For a batch of 8 sequences of 4,096 tokens and Qwen's vocabulary of 151,936, that is about 5 billion numbers: 10 GB in bf16 and 20 GB in float32 (which the loss often upcasts to). A fused (or chunked) loss computes the logits for a slice of tokens, takes the log-sum-exp and the loss, and discards the slice, so the full tensor never exists (`unsloth_zoo/loss_utils.py` has `fused_linear_cross_entropy`; `unsloth/kernels/cross_entropy_loss.py` computes a stable log-sum-exp per row in Triton). TRL's distillation loss uses the same idea (Chapter 12).
2. **Offloaded gradient checkpointing.** With checkpointing, only each layer's input hidden state is kept for the backward pass. `Unsloth_Offloaded_Gradient_Checkpointer` copies that tensor to CPU memory with a non-blocking copy and brings it back during backward, overlapping the transfer with computation. GPU activation memory then no longer grows with the number of layers.
3. **Hand-written LoRA backward** (`fast_lora.py`): the gradients of the fused MLP and attention projections with LoRA are written out explicitly, so autograd does not store intermediate products it does not need.
4. **Fused small kernels**: RMSNorm, RoPE, SwiGLU and GeGLU in Triton, which cut memory traffic.

The first idea is easy to verify in plain PyTorch: computing the loss chunk by chunk gives the same value and the same gradients as the full computation.

```
import torch, torch.nn.functional as F
torch.manual_seed(0)
N, d, V = 512, 64, 5000                          # tokens, hidden size, vocabulary
h = torch.randn(N, d, requires_grad=True); W = (0.05 * torch.randn(V,
 ↪  d)).requires_grad_()
y = torch.randint(0, V, (N,))
full = F.cross_entropy(h @ W.T, y, reduction="sum") / N
g_full = torch.autograd.grad(full, (h, W))
def chunked_ce(h, W, y, chunk=64):
    total = 0.0
    for s in range(0, len(y), chunk):
        logits = h[s:s + chunk] @ W.T             # only chunk x V logits exist at once
        total = total + F.cross_entropy(logits, y[s:s + chunk], reduction="sum")
    return total / len(y)
part = chunked_ce(h, W, y)
g_part = torch.autograd.grad(part, (h, W))
print(f"loss full {full:.6f} | chunked {part:.6f} | max grad diff {max((a -
 ↪  b).abs().max().item() for a, b in zip(g_full, g_part)):.2e}")
print(f"peak logits held: full {N * V:,} numbers vs chunked {64 * V:,} ({N // 64}x
 ↪  fewer)")
B, T, Vq = 8, 4096, 151936
print(f"full logits for B={B}, T={T}, |V|={Vq:,}: {B * T * Vq * 2 / 1e9:.1f} GB bf16, {B
 ↪  * T * Vq * 4 / 1e9:.1f} GB fp32")
L, dm = 28, 3584                                 # layers and hidden size of a 7B-class model
print(f"checkpointed layer inputs for that batch: {L * B * T * dm * 2 / 1e9:.1f} GB bf16
 ↪  (moved to CPU by offloaded checkpointing)")
```

This toy loop still lets autograd keep each chunk's logits for the backward pass; a real fused kernel also recomputes them during backward (or computes the gradient in the forward pass), so the saving holds for training as well.

**Why this, not that? Why offload checkpointed activations to CPU, not just shrink the batch or sequence length?**

**Reason.** The batch and sequence length decide what the model learns from (long reasoning traces need long contexts); offloading keeps them as they are and pays only in PCIe traffic, which overlaps with computation when the copies are non-blocking. **Trade-off.** It uses pinned host memory (several GB per GPU for long contexts), adds a little time when transfers cannot be hidden, and helps only the activation part of memory, not weights or optimizer states. **When the alternative is better.** When the GPU has spare memory, or host memory and PCIe bandwidth are limited (many GPUs sharing one host), plain checkpointing or a smaller micro-batch with gradient accumulation is simpler and as fast.

## 52.4 A GRPO run with Unsloth

The pattern used by Unsloth's RL notebooks, with parameter names checked against Unsloth 2026.10.7 and TRL 1.16.0 (not executed here: it needs a GPU):

```
from unsloth import FastLanguageModel          # first import
from trl import GRPOConfig, GRPOTrainer

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/Qwen2.5-1.5B-Instruct", max_seq_length=2048,
    load_in_4bit=False,              # 16-bit base: no quantisation noise in rollouts or
 ↪  gradients
    fast_inference=True,             # in-process vLLM engine for rollouts
    max_lora_rank=32, gpu_memory_utilization=0.6,
)
model = FastLanguageModel.get_peft_model(model, r=32, lora_alpha=64,
                                         use_gradient_checkpointing="unsloth",
 ↪  random_state=3407)
args = GRPOConfig(
    output_dir="out/grpo", learning_rate=5e-6, num_generations=8,
    per_device_train_batch_size=8, gradient_accumulation_steps=2,
    max_completion_length=1024, use_vllm=True, temperature=1.0, max_steps=500,
 ↪  logging_steps=5,
)
trainer = GRPOTrainer(model=model, processing_class=tokenizer,
 ↪  reward_funcs=[correctness_reward, format_reward],
                      args=args, train_dataset=ds)
trainer.train()
```

Three checks specific to this setup: `max_lora_rank` must be at least the `r` you train, or the in-process vLLM engine cannot load the adapter; `gpu_memory_utilization` is the share of GPU memory the engine reserves, so training must fit in the rest; and rewards, KL and completion length should be watched exactly as in Chapter 50.

## 52.5 Common mistakes

1. Importing Unsloth after Transformers/TRL (patches may not apply; Unsloth warns).
2. Assuming PEFT defaults: Unsloth's LoRA defaults are `r=16`, `alpha=16`, seven target modules.
3. Forgetting that `load_in_4bit=True` is the default (you train QLoRA unless you change it).
4. Serving a LoRA-on-4-bit model without merging to 16-bit (`merged_16bit`) first.
5. Setting `fast_inference=True` with `gpu_memory_utilization` too high for training to fit.

- Training a LoRA rank larger than `max_lora_rank` with `fast_inference=True`.
- Expecting multi-GPU scaling from a single-GPU recipe; use the plain Hugging Face stack, OpenRLHF or veRL for multi-node work.

**Interview check**

**Q1. What does `use_gradient_checkpointing="unsloth"` do?** Checkpoints activations and offloads them to CPU asynchronously, cutting GPU memory for long sequences.
**Q2. What is `fast_inference=True` for?** Running vLLM inside the process, sharing weights with the trainer, for fast GRPO rollouts.

**Q3. Which save method should you use before serving with vLLM?** `merged_16bit`.
**Q4. What are Unsloth's default base-weight precision and LoRA rank?** 4-bit base weights and `r=16` with `lora_alpha=16`.
**Q5. How does a fused or chunked cross-entropy loss save memory?** It computes logits, log-sum-exp and loss for a slice of tokens at a time, so the full $B \times T \times |V|$ logits tensor never exists; the result and gradients are identical.
**Q6. What does offloaded gradient checkpointing store, and where?** Each layer's input hidden state, copied to pinned CPU memory with non-blocking transfers and brought back for recomputation in the backward pass.
**Q7. Why must `max_lora_rank` be at least the training rank in Unsloth GRPO?** The in-process vLLM engine serves the current adapter for rollouts and can only load adapters up to that rank.

**Key takeaways**

- `from_pretrained` (4-bit by default, `fast_inference` for vLLM) → `get_peft_model` (r 16, alpha 16, seven modules) → TRL trainer → `save_pretrained_merged` / `save_pretrained_gguf`.
- Unsloth's gradient checkpointing offloads activations; its defaults differ from PEFT's.
- Unsloth's savings: fused linear cross-entropy, CPU-offloaded checkpointing, hand-written LoRA backward and fused small kernels.
- For GRPO: `fast_inference=True`, `max_lora_rank` at least `r`, leave memory for training beyond `gpu_memory_utilization`.

## 52.6 Generated reference: `FastLanguageModel.from_pretrained`

*Generated from the Unsloth 2026.10.7 source (`FastLanguageModel.from_pretrained`); 30 entries.*

- **`model_name`** = `'unsloth/Llama-3.2-1B-Instruct'`
- **`max_seq_length`** = `2048`
- **`dtype`** = `None`
- **`load_in_4bit`** = `True`
- **`load_in_8bit`** = `False`
- **`load_in_16bit`** = `False`
- **`full_finetuning`** = `False`
- **`token`** = `None`
- **`device_map`** = `DEFAULT_DEVICE_MAP`
- **`device_map_planner_kwargs`** = `None`
- **`rope_scaling`** = `None`
- **`fix_tokenizer`** = `True`
- **`trust_remote_code`** = `False`
- **`use_gradient_checkpointing`** = `'unsloth'`
- **`resize_model_vocab`** = `None`
- **`revision`** = `None`
- **`use_exact_model_name`** = `False`

- **offload_embedding** = OFFLOAD_EMBEDDING_AUTO
- **float32_mixed_precision** = None
- **fast_inference** = False
- **gpu_memory_utilization** = 0.5
- **float8_kv_cache** = False
- **random_state** = 3407
- **max_lora_rank** = 64
- **disable_log_stats** = True
- **qat_scheme** = None
- **load_in_fp8** = False
- **unsloth_tiled_mlp** = False
- **text_only** = False
- **on_model_resolved** = None

## 52.7 Generated reference: `get_peft_model`

*Generated from the Unsloth 2026.10.7 source (`FastLlamaModel.get_peft_model`); 21 entries.*

- **r** = 16
- **target_modules** = _DEFAULT_TARGET_MODULES
- **lora_alpha** = 16
- **lora_dropout** = 0.0
- **bias** = 'none'
- **layers_to_transform** = None
- **layers_pattern** = None
- **finetune_last_n_layers** = None
- **use_gradient_checkpointing** = 'unsloth'
- **random_state** = 3407
- **max_seq_length** = 2048
- **use_rslora** = False
- **modules_to_save** = None
- **init_lora_weights** = True
- **loftq_config** = {}
- **temporary_location** = '_unsloth_temporary_saved_buffers'
- **qat_scheme** = None
- **target_parameters** = None
- **ensure_weight_tying** = None
- **offload_layers** = None
- **checkpoint_skip_layers** = 0

## 52.8 Generated reference: `save_pretrained_merged`

*Generated from the Unsloth 2026.10.7 source (`unsloth_save_pretrained_merged`); 19 entries.*

- **tokenizer** = None

- **save_method** = `'merged_16bit'`
- **push_to_hub** = `False`
- **token** = `None`
- **is_main_process** = `True`
- **state_dict** = `None`
- **save_function** = `torch.save`
- **max_shard_size** = `'5GB'`
- **safe_serialization** = `True`
- **variant** = `None`
- **save_peft_format** = `True`
- **tags** = `None`
- **temporary_location** = `'_unsloth_temporary_saved_buffers'`
- **maximum_memory_usage** = `0.75`
- **datasets** = `None`
- **calibration_dataset** = `None`
- **num_calibration_samples** = `512`
- **max_seq_length** = `2048`
- **install_missing_dependencies** = `True`

## 52.9 Generated reference: `save_pretrained_gguf`

*Generated from the Unsloth 2026.10.7 source (`unsloth_save_pretrained_gguf`); 19 entries.*

- **tokenizer** = `None`
- **quantization_method** = `'fast_quantized'`
- **first_conversion** = `None`
- **push_to_hub** = `False`
- **token** = `None`
- **private** = `None`
- **is_main_process** = `True`
- **state_dict** = `None`
- **save_function** = `torch.save`
- **max_shard_size** = `'5GB'`
- **safe_serialization** = `True`
- **variant** = `None`
- **save_peft_format** = `True`
- **tags** = `None`
- **temporary_location** = `'_unsloth_temporary_saved_buffers'`
- **maximum_memory_usage** = `0.85`
- **save_method** = `None`
- **imatrix_file** = `None`
- **merge_is_disposable** = `True`

CHAPTER 53

# DeepSpeed Configuration and vLLM Settings

**Why this chapter matters.**
Most large post-training runs pass a DeepSpeed JSON file to the trainer (Transformers, TRL, OpenRLHF) and drive rollouts through vLLM. A mis-set ZeRO stage, a forgotten `bf16` block or vLLM's 16-token default output length can waste a day of GPU time. This chapter covers the DeepSpeed 0.19.7 configuration keys that matter for post-training and the vLLM 0.32.0 sampling and engine settings used by RL frameworks, with generated lists at the end.

## 53.1 A DeepSpeed config for post-training

DeepSpeed reads one JSON (or Python dict) with top-level keys. With the Hugging Face Trainer, pass the path or dict as `TrainingArguments(deepspeed=...)`; values written as `"auto"` are filled in from `TrainingArguments` (batch sizes, learning rate, gradient clipping, precision), which avoids two sources of truth.

```
{
  "train_micro_batch_size_per_gpu": "auto",
  "gradient_accumulation_steps": "auto",
  "gradient_clipping": "auto",
  "bf16": {"enabled": true},
  "zero_optimization": {
    "stage": 3,
    "overlap_comm": true,
    "contiguous_gradients": true,
    "reduce_bucket_size": "auto",
    "stage3_prefetch_bucket_size": "auto",
    "stage3_param_persistence_threshold": "auto",
    "stage3_gather_16bit_weights_on_model_save": true,
    "offload_optimizer": {"device": "none"},
    "offload_param": {"device": "none"}
  }
}
```

**Top level** (defaults from `deepspeed/runtime/constants.py`): `train_batch_size` = `train_micro_batch_size_per_gpu` × `gradient_accumulation_steps` × number of GPUs (give any two; DeepSpeed derives the third); `gradient_clipping` (**1.0**; 0.0 disables); `steps_per_print`; `wall_clock_breakdown` (False); `communication_data_type` (None, i.e. the model's dtype; set `"fp32"` for more accurate gradient reduction in BF16 training).

**Precision.** `bf16.enabled` (False). In 0.19.7 the BF16 block also has `immediate_grad_update`

(False), `check_grad_overflow` (True), `bf16_master_weights_and_grads` (False) and `bf16_optimizer_states` (False): by default DeepSpeed keeps FP32 master weights and optimizer states, as Chapter 41 recommends. The `fp16` block has dynamic loss scaling by default (`loss_scale` 0, `initial_scale_power` 16, `loss_scale_window` 1000, `hysteresis` 2, `min_loss_scale` 1).

**ZeRO** (`zero_optimization`): `stage` (0; 1 shards optimizer states, 2 adds gradients, 3 adds parameters; Chapter 42), `overlap_comm` (None → True for stage 3 only), `contiguous_gradients` (True), `reduce_scatter` (True), `reduce_bucket_size` and `allgather_bucket_size` (5e8 elements), `sub_group_size` (1e9). Stage-3 keys keep their historical `stage3_` prefix in JSON: `stage3_prefetch_bucket_size` (5e7), `stage3_param_persistence_threshold` (1e5; parameters smaller than this stay unpartitioned), `stage3_max_live_parameters` (1e9), `stage3_max_reuse_distance` (1e9), `stage3_gather_16bit_weights_on_model_save` (False; set True to save a normal 16-bit checkpoint instead of shards). ZeRO++ options: `zero_hpz_partition_size` (1; secondary partition within a node), `zero_quantized_weights` and `zero_quantized_gradients` (False).

**Offload.** `offload_optimizer.device` and `offload_param.device`: `"none"` (default), `"cpu"` or `"nvme"` (with `nvme_path`); `pin_memory` (True), buffer counts and sizes. Offloading the optimizer to CPU (ZeRO-Offload) cuts GPU memory by the 12 bytes/parameter of optimizer states at a large speed cost; parameter offload (ZeRO-Infinity) goes further.

The configuration classes are pydantic models that **reject unknown keys**, which catches typos:

```
import warnings; warnings.filterwarnings("ignore")
from deepspeed.runtime.zero.config import DeepSpeedZeroConfig
from deepspeed.runtime.precision_config import DeepSpeedBF16Config
z2 = DeepSpeedZeroConfig(stage=2)
z3 = DeepSpeedZeroConfig(**{"stage": 3, "offload_optimizer": {"device": "cpu"},
 ↪  "stage3_prefetch_bucket_size": 5e7})
print("overlap_comm resolved: stage 2 ->", z2.overlap_comm, "| stage 3 ->",
 ↪  z3.overlap_comm)
print("stage-3 prefetch bucket:", z3.prefetch_bucket_size, "| optimizer offload:",
 ↪  z3.offload_optimizer.device,
      "pin_memory", z3.offload_optimizer.pin_memory)
print("bf16 block defaults:", DeepSpeedBF16Config(enabled=True))
try:
    DeepSpeedZeroConfig(stage=3, stage3_prefetch_bucket_sise=5e7)       # typo
except Exception as e:
    print("typo rejected:", type(e).__name__, "-", str(e).splitlines()[1])
```

**Why this, not that? Why ZeRO-3 (or FSDP) for 7B+ full fine-tuning, not ZeRO-2?**

**Reason.** ZeRO-2 still keeps a full 16-bit copy of the parameters on every GPU (2 bytes/parameter: 14 GB for 7B, 140 GB for 70B), which does not fit for large models; ZeRO-3 shards parameters too (Chapter 42). **Trade-off.** Stage 3 all-gathers parameters for every forward and backward pass (about 1.5× the communication of stage 2), and generation with sharded parameters is slow unless weights are gathered (TRL's `ds3_gather_for_generation`) or a separate engine is used. **When the alternative is better.** Models whose 16-bit weights fit comfortably on each GPU (≤ about 7B on 80 GB with room for activations), LoRA training, or slow inter-GPU links: ZeRO-2 is faster. OpenRLHF defaults to stage 2 for that reason.

## 53.2 vLLM settings for RL

**Sampling** (`SamplingParams`, per request): the RL-relevant fields were covered in Chapter 43 (`n`, `temperature`, `top_p`, `top_k`, `min_p`, `max_tokens` defaulting to **16**, `stop`, `stop_token_ids`, `logprobs`, `seed`, `skip_special_tokens`, `include_stop_str_in_output`). Others worth knowing: `presence_penalty`, `frequency_penalty`, `repetition_penalty` (keep neutral during RL training: they change the sampling distribution), `min_tokens`, `ignore_eos`, `structured_outputs` (constrain output to a JSON schema, regex or grammar), `logit_bias`, `allowed_token_ids`, `bad_words`, and `thinking_token_budget`.

**Engine** (`LLM(...)` or `vllm serve` arguments, gathered from the config classes): `tensor_parallel_size`, `pipeline_parallel_size`, `data_parallel_size`; `gpu_memory_utilization` (0.92); `max_model_len` (from the model config); `dtype` (`auto`); `kv_cache_dtype` (`auto`, or `fp8`); `enable_prefix_caching` (True); `max_num_seqs` and `max_num_batched_tokens` (chosen by hardware in `EngineArgs`; the `SchedulerConfig` class defaults of 128 and 2048 are fallbacks); `enforce_eager` (False); `max_logprobs` (20) and `logprobs_mode` (`raw_logprobs`); `enable_lora` (False), `max_loras` (1), `max_lora_rank` (16); `enable_sleep_mode` (False); `seed` (0); `worker_extension_cls` (lets RL frameworks add methods, such as weight updates, to vLLM workers); `WeightTransferConfig.backend` (`nccl`).

**Why this, not that? Why give vLLM its own memory fraction, not let it take the default 0.92?**

**Reason.** vLLM pre-allocates its KV cache up to `gpu_memory_utilization` of the GPU. In colocated RL the trainer needs most of the GPU between generations; at 0.92, training runs out of memory. **Trade-off.** A smaller fraction means a smaller KV cache, fewer concurrent sequences and lower generation throughput (Chapter 43's roofline). **When the alternative is better.** Dedicated rollout GPUs (OpenRLHF's separate engines, TRL's server mode), or colocation with sleep mode that releases the KV cache and weights during training: then a high fraction (0.8–0.95) is right.

## 53.3 Common mistakes

1. Different batch settings in the DeepSpeed JSON and `TrainingArguments` (use `"auto"`).
2. `stage3_gather_16bit_weights_on_model_save` left False, then wondering why the saved checkpoint is shards.
3. BF16 training without the `bf16` block (DeepSpeed then trains in FP32 or complains about mismatched dtypes).
4. CPU optimizer offload on a machine with slow CPU memory bandwidth or too little RAM (about 12 bytes per parameter needed).
5. vLLM `max_tokens` left at 16; penalties left on during RL; `max_logprobs` lower than the requested `logprobs`.

**Interview check**

**Q1. What does "auto" mean in a DeepSpeed config used with the HF Trainer?** The Trainer fills the value from `TrainingArguments`, keeping one source of truth.

**Q2. What is DeepSpeed's default gradient clipping?** 1.0 in 0.19.7 (0.0 disables it).
**Q3. What does `stage3_param_persistence_threshold` do?** Parameters smaller than the threshold are kept whole on every GPU to avoid many tiny all-gathers.
**Q4. Which vLLM setting decides how much GPU memory the KV cache can take?** `gpu_memory_utilization`, a fraction of total GPU memory for weights, activations and KV cache.
**Q5. Why should repetition and presence penalties be off during RL rollouts?** They change the sampling distribution, so the sampler's probabilities no longer match the policy the trainer optimises.

**Key takeaways**

- DeepSpeed JSON: batch triple, `gradient_clipping` (1.0), `bf16` block, `zero_optimization` (stage, buckets, `stage3_*`, offload); unknown keys are rejected.
- vLLM: per-request `SamplingParams` (set `max_tokens`), engine memory fraction, prefix caching, logprob mode, sleep mode, weight transfer.

## 53.4 Generated reference: DeepSpeed `zero_optimization`

JSON keys are shown (stage-3 settings under their `stage3_` aliases).

*Generated from the DeepSpeed 0.19.7 source (`DeepSpeedZeroConfig`); 44 entries.*

- **`stage`** = `0`. Chooses different stages of ZeRO Optimizer. Stage 0, 1, 2, and 3 refer to disabled, optimizer state partitioning, and optimizer+gradient state partitioning, and optimizer+gradient+parameter partitioning, respectively.
- **`contiguous_gradients`** = `True`. Copies the gradients to a contiguous buffer as they are produced. Avoids memory fragmentation during backward pass.
- **`reduce_scatter`** = `True`. Uses reduce or reduce scatter instead of allreduce to average gradients
- **`reduce_bucket_size`** = `500_000_000`. Number of elements reduced/allreduced at a time. Limits the memory required for the allgather for large model sizes
- **`use_multi_rank_bucket_allreduce`** = `True`. Combine the reduce buckets of the different ranks and do an All-Reduce instead of multiple Reduce ops. This feature is useful when the model is small and we want to scale it on too many GPUs which therefore reduces the message sizes of each packet.
- **`allgather_partitions`** = `True`. Chooses between allgather collective or a series of broadcast collectives to gather updated parameters from all the GPUs at the end of each step
- **`allgather_bucket_size`** = `500_000_000`. Number of elements allgathered at a time. Limits the memory required for the allgather for large model sizes
- **`overlap_comm`** = `None`. Attempts to overlap the reduction of the gradients with backward computation
- **`compute_grad_norm`** = `True`. Compute and retain the global gradient norm during ZeRO Stage 1/2 optimizer steps. Disable only when gradient clipping is off, the dedicated ZeRO-1 BF16 optimizer is not selected, and callers do not use get_global_grad_norm().
- **`load_from_fp32_weights`** = `True`. Boolean indicating whether to initialize fp32 master

weights from fp32 copies in checkpoint (no precision loss) or from model's fp16 copies (with precision loss). This can be used to initialize optimizer state even when checkpoint is missing optimizer state.

- **elastic_checkpoint** = False. Legacy elastic checkpoint support. ZeRO-3 elastic checkpointing is no longer supported; use Universal Checkpointing instead.
- **offload_param** = None. Enable offloading of model parameters to CPU or NVMe. This frees up GPU memory for larger models or batch sizes. Valid only with stage 3. Expects a dictionary containing values for:any:DeepSpeedZeroOffloadParamConfig.
- **offload_optimizer** = None. Enable offloading of optimizer state to CPU or NVMe, and optimizer computation to CPU. This frees up GPU memory for larger models or batch sizes. Valid for ZeRO stage 1, 2, 3.
- **zenflow** = None. Enable ZenFlow
- **sub_group_size** = 1_000_000_000. Tile size for parameter processing to fit massive models (with trillions of parameters). Used by ZeRO3-Offload and ZeRO-Infinity
- **cpu_offload_param** = None. Deprecated, please use offload_param
- **cpu_offload_use_pin_memory** = None. Deprecated, please use offload_param or offload_optimizer
- **cpu_offload** = None. Deprecated, please use offload_optimizer
- **stage3_prefetch_bucket_size** = 50_000_000. Maximum number of parameter elements to fetch ahead of use. Used by ZeRO3, ZeRO3-Offload, ZeRO-Infinity, and ZeRO-Inference.
- **stage3_param_persistence_threshold** = 100_000. Do not partition parameters smaller than this threshold. Smaller values use less memory, but can greatly increase communication (especially latency-bound messages).
- **stage3_model_persistence_threshold** = pp_int(sys.maxsize, 'sys.maxsize'). Maximum number of parameter elements that can be persisted in GPU and not partitioned. This imposes an upper bound on the number of unpartitioned parameters resulting from param_persistence_threshold setting.
- **stage3_max_live_parameters** = 1_000_000_000. The maximum number of parameters resident per GPU before releasing. Smaller values use less memory, but perform more communication.
- **stage3_max_reuse_distance** = 1_000_000_000. Do not release a parameter if it will be reused within this threshold of parameters. Smaller values use less memory, but perform more communication.
- **stage3_gather_16bit_weights_on_model_save** = False. Consolidate the weights before saving the model by save_16bit_model(). Since the weights are partitioned across GPUs, they aren't part of state_dict, so this function automatically gathers the weights when this option is enabled and then saves the fp16 model weights.
- **stage3_module_granularity_threshold** = 0. The granularity of a module is determined by the ratio of "parameter_count / (1 + descendant count)". ZeRO3 classifies modules with a granularity below the threshold as fine-grained, which are treated as integral units during parameter fetching.
- **stage3_use_all_reduce_for_fetch_params** = False. Use all_reduce op when fetching module parameters at stage3. This improves performance by reducing the overhead of concatenation and slicing on the host.
- **stage3_allgather_sequential** = False. Performs allgather on individual parameters se-

quentially, bypassing the standard parameter bucketing mechanism in stage3. This significantly reduces data copy overhead (eliminating copy-to-bucket operations) and lowers peak memory usage by avoiding the allocation of large temporary flattening buffers.

- **stage3_gather_fp16_weights_on_model_save** = `False`. Deprecated, please use gather_16bit_weights_on_model_save
- **ignore_unused_parameters** = `True`. Unused parameters in modules may be unexpected in static networks, but could be normal in dynamic networks. This controls whether or not training should terminate with an error message when unused parameters are detected.
- **legacy_stage1** = `False`. For backward-compatibility enable old ZeRO stage 1 implementation. Use at your own risk, will be deprecated soon.
- **round_robin_gradients** = `False`. Stage 1 and 2 optimization for CPU offloading that parallelizes gradient copying to CPU memory among ranks by fine-grained gradient partitioning. Performance benefit grows with gradient accumulation steps (more copying between optimizer steps) or GPU count (increased parallelism).
- **parameter_alignment** = `False`. Pad ZeRO Stage 1 and 2 flat buffers between parameters so each parameter starts at a 16-byte-aligned address. This is disabled by default because the padding increases flat-buffer and optimizer-state memory usage.
- **zero_hpz_partition_size** = `1`. Number of ranks in zero parameters partitioning secondary group
- **zero_quantized_weights** = `False`. Boolean indicating whether to quantize zero parameters (weights) for efficient all_gather comm
- **zero_quantized_nontrainable_weights** = `False`. Boolean indicating whether to quantize non-trainable zero parameters (weights) for efficient memory usage and communication. Different from zero_quantized_weights that stores the weights in original precision and only perform quantization during communication, this flag will store the weights in quantized precision.
- **zero_quantized_gradients** = `False`. Boolean indicating whether to use quantized zero gradients for efficient all_2_all_reduce comm
- **zeropp_loco_param** = `None`. This dictionary contains parameters for using LoCo-Zero++, with two key parameters: - err_beta: A coefficient for the moving average of quantization errors before and after gradient computation.
- **memory_efficient_linear** = `True`. Use memory efficient linear implementation, for Stage 3.
- **pipeline_loading_checkpoint** = `False`
- **override_module_apply** = `True`. Override nn.Module apply function, for Stage 3.
- **log_trace_cache_warnings** = `False`. Whether to log warnings from trace cache, such as invalidation events.
- **enable_sanity_checks** = `False`. Enable internal sanity checks, which could be useful for debugging
- **save_muon_momentum_buffer_in_memory** = `False`. When using the Muon optimizer with ZeRO Stage 3, keeps the Muon momentum buffer in GPU/CPU memory instead of swapping to NVMe with other optimizer states. Only relevant when using NVMe offloading.
- **leaf_module** = `DeepSpeedZeroLeafModuleConfig()`. Configuration for modules that should be treated as ZeRO3 leaf modules.

## 53.5 Generated reference: DeepSpeed offload settings

*Generated from the DeepSpeed 0.19.7 source (`DeepSpeedZeroOffloadParamConfig`); 6 entries.*

- **`device`** = `'none'`. Device memory to offload model parameters. Supported options are cpu and nvme.
- **`nvme_path`** = `None`. Filesystem path for NVMe device for parameter offloading.
- **`buffer_count`** = `5`. Number of buffers in buffer pool for parameter offloading to NVMe.
- **`buffer_size`** = `100_000_000`. Size of buffers in buffer pool for parameter offloading to NVMe.
- **`max_in_cpu`** = `1_000_000_000`. Number of parameter elements to maintain in CPU memory when offloading to NVMe is enabled.
- **`pin_memory`** = `True`. Offload to page-locked (pinned) CPU memory. Required for asynchronous, full-bandwidth GPU<->CPU transfers and for overlap of grad/param offload with compute. Defaults to True.

*Generated from the DeepSpeed 0.19.7 source (`DeepSpeedZeroOffloadOptimizerConfig`); 10 entries.*

- **`device`** = `'none'`. Device memory to offload optimizer state. Supported options are cpu and nvme. Optimizer computation is offload to CPU regardless of device option.
- **`nvme_path`** = `None`. Filesystem path for NVMe device for optimizer state offloading.
- **`buffer_count`** = `4`. Number of buffers in buffer pool for optimizer state offloading to NVMe. This should be at least the number of states maintained per parameter by the optimizer. For example, Adam optimizer has 4 states (parameter, gradient, momentum, and variance).
- **`pin_memory`** = `True`. Offload to page-locked (pinned) CPU memory. Required for asynchronous, full-bandwidth GPU<->CPU transfers and for overlap of grad/param offload with compute. Defaults to True.
- **`pipeline_read`** = `False`. For tile-based optimizer step processing, overlap read of next tile with computation of current tile. Used in ZeRO-Infinity.
- **`pipeline_write`** = `False`. For tile-based optimizer step processing, overlap write of previous tile with computation of current tile.
- **`fast_init`** = `False`. Enable fast optimizer initialization when offloading to NVMe.
- **`ratio`** = `1.0`. Percentage of offloaded optimizer states to CPU Adam. Only valid with ZeRO Stage 3.
- **`super_offload`** = `False`. Enable high performance CPU offloading for Superchips. Only valid with ZeRO Stage 3.
- **`cpuadam_cores_perc`** = `0.8`. Percentage of CPU cores to use for CPU Adam. Only valid with ZeRO Stage 3 and super_offload=True.

## 53.6 Generated reference: DeepSpeed `bf16` and `fp16`

*Generated from the DeepSpeed 0.19.7 source (`DeepSpeedBF16Config`); 5 entries.*

- **`enabled`** = `False`. Enable bfloat16 mixed-precision training/inference
- **`immediate_grad_update`** = `False`. Apply gradient updates immediately rather than delayed.

- **check_grad_overflow** = `True`. Detect gradient overflow/underflow before the optimizer step and skip the step when detected. Defaults to True to match the fp16 default. See issue #5242 and PR #6976 for context.
- **bf16_master_weights_and_grads** = `False`. Maintain master weights/gradients in bf16 precision for ZeRO optimizer.
- **bf16_optimizer_states** = `False`. Keep optimizer states in bf16 (only valid when bf16_master_weights_and_grads is enabled).

*Generated from the DeepSpeed 0.19.7 source (`DeepSpeedFP16Config`); 9 entries.*

- **enabled** = `False`. Enable fp16 mixed-precision training/inference
- **auto_cast** = `False`. Automatically cast inputs to fp16
- **loss_scale** = `0`. Loss scaling value. Default value of 0 means dynamic loss scaling instead of static loss scale.
- **initial_scale_power** = `16`. For dynamic loss scaling, set initial loss scale to 2^{initial_scale_power}.
- **loss_scale_window** = `1000`. Iteration intervals for raising/lowering dynamic loss scale value.
- **hysteresis** = `2`. Delay shift in dynamic loss scaling.
- **consecutive_hysteresis** = `False`. Refill hysteresis if iteration does not overflow/underflow.
- **min_loss_scale** = `1`. Minimum dynamic loss scale value.
- **fp16_master_weights_and_grads** = `False`. Maintain master weights in optimizer state as fp16 instead of fp32 (valid with DeepSpeedCPUAdam only).

## 53.7 Generated reference: vLLM `SamplingParams`

*Generated from the vLLM 0.32.0 source (`SamplingParams`); 39 entries.*

- **n** = `1`. Number of outputs to return for the given prompt request. The maximum allowed value is controlled by the VLLM_MAX_N_SEQUENCES environment variable (default: 16384). NOTE: AsyncLLM streams outputs by default.
- **presence_penalty** = `0.0`. Penalizes new tokens based on whether they appear in the generated text so far. Values $> 0$ encourage the model to use new tokens, while values $< 0$ encourage the model to repeat tokens.
- **frequency_penalty** = `0.0`. Penalizes new tokens based on their frequency in the generated text so far. Values $> 0$ encourage the model to use new tokens, while values $< 0$ encourage the model to repeat tokens.
- **repetition_penalty** = `1.0`. Penalizes new tokens based on whether they appear in the prompt and the generated text so far. Values $> 1$ encourage the model to use new tokens, while values $< 1$ encourage the model to repeat tokens.
- **temperature** = `1.0`. Controls the randomness of the sampling. Lower values make the model more deterministic, while higher values make the model more random. Zero means greedy sampling.
- **watermarking** = `None`. Whether to apply the engine's configured watermark to this request. None defaults to enabled.

- **top_p** = 1.0. Controls the cumulative probability of the top tokens to consider. Must be in (0, 1]. Set to 1 to consider all tokens. ...)
- **top_k** = 0. Controls the number of top tokens to consider. Set to 0 (or -1) to consider all tokens.
- **min_p** = 0.0. Represents the minimum probability for a token to be considered, relative to the probability of the most likely token. Must be in [0, 1]. Set to 0 to disable this.
- **seed** = None. Random seed to use for the generation.
- **stop** = None. String(s) that stop the generation when they are generated. The returned output will not contain the stop strings.
- **stop_token_ids** = None. Token IDs that stop the generation when they are generated. The returned output will contain the stop tokens unless the stop tokens are special tokens.
- **ignore_eos** = False. Whether to ignore the EOS token and continue generating tokens after the EOS token is generated.
- **max_tokens** = 16. Maximum number of tokens to generate per output sequence.
- **min_tokens** = 0. Minimum number of tokens to generate per output sequence before EOS or stop_token_ids can be generated
- **logprobs** = None. Number of log probabilities to return per output token. When set to None, no probability is returned. If set to a non-None value, the result includes the log probabilities of the specified number of most likely tokens, as well as the chosen tokens.
- **prompt_logprobs** = None. Number of log probabilities to return per prompt token. When set to -1, return all vocab_size log probabilities.
- **prompt_logprob_token_ids** = None. Token IDs to score per causal prompt row: an integer array of shape [num_rows, num_ids] or nested lists; -1 entries (padding) score -inf. Row i scores its IDs as predictions of prompt token prompt_logprob_start + i + 1, so the last prompt row is excluded.
- **prompt_logprob_start** = None. First causal prompt row to score; defaults to the first row.
- **logprob_token_ids** = None. Specific token IDs to return logprobs for. More efficient than logprobs=-1 when you only need logprobs for a small set of tokens. When set, logprobs for exactly these token IDs will be returned, in addition to the sampled token.
- **flat_logprobs** = False. Whether to return logprobs in flatten format (i.e. FlatLogprob) for better performance. NOTE: GC costs of FlatLogprobs is significantly smaller than list[dict[int, Logprob]].
- **detokenize** = True. Whether to detokenize the output.
- **skip_special_tokens** = True. Whether to skip special tokens in the output.
- **spaces_between_special_tokens** = True. Whether to add spaces between special tokens in the output.
- **include_stop_str_in_output** = False. Whether to include the stop strings in output text.
- **output_kind** = RequestOutputKind.CUMULATIVE
- **stream_interval** = None. Number of newly generated tokens to batch into each streamed RequestOutput. Raises the interval above the engine-level –stream-interval. Values below engine setting are clamped up to it.
- **skip_clone** = False. Internal flag indicating that this SamplingParams instance is safe to reuse without cloning. When True, clone() will return self without performing a deep copy. This should only be set when the params object is guaranteed to be dedicated to a single request and won't be modified in ways that would affect other uses.

- **output_text_buffer_length** = 0
- **structured_outputs** = None. Parameters for configuring structured outputs.
- **logit_bias** = None. If provided, the engine will construct a logits processor that applies these logit biases.
- **allowed_token_ids** = None. If provided, the engine will construct a logits processor which only retains scores for the given token ids.
- **extra_args** = None. Arbitrary additional args, that can be used by custom sampling implementations, plugins, etc. Not used by any in-tree sampling implementations.
- **bad_words** = None. Words that are not allowed to be generated. More precisely, only the last token of a corresponding token sequence is not allowed when the next generated token can complete the sequence.
- **skip_reading_prefix_cache** = None
- **thinking_token_budget** = None. Maximum number of tokens allowed for thinking operations.
- **repetition_detection** = None. Parameters for detecting repetitive N-gram patterns in output tokens. If such repetition is detected, generation will be ended early. LLMs can sometimes generate repetitive, unhelpful token patterns, stopping only when they hit the maximum output length (e.g. ...)
- **routed_experts_prompt_start** = 0. When enable_return_routed_experts is active, skip the first routed_experts_prompt_start prompt tokens from the returned routing data. In multi-turn agent scenarios, set this to the length of the already-returned prefix to avoid duplicating routing for prompt tokens covered by earlier turns.
- **trace_decode_token_ids** = None. If provided, forces the engine to emit this predetermined sequence of token IDs during decoding instead of sampling randomly. Real logprobs are still computed. Conflict checking is performed at the engine level.

## 53.8 Generated reference: vLLM engine configuration (selected classes)

### 53.8.1 CacheConfig

*Generated from the vLLM 0.32.0 source (`CacheConfig`); 35 entries.*

- **block_size** = None. Size of a contiguous cache block in number of tokens. Accepts None (meaning "use default"). After construction, always int.
- **user_specified_block_size** = False. Whether block_size was explicitly provided. Derived automatically.
- **kv_cache_layout** = None. Resolved physical KV cache layout name (a KVCacheLayout member). None means the layout has not been resolved yet.
- **prefix_match_unit** = None. The finest token boundary (in tokens) a prefix-cache hit can land on. Prefix-cache keys are computed every prefix_match_unit tokens.
- **gpu_memory_utilization** = 0.92. The fraction of GPU memory to be used for the model executor, which can range from 0 to 1. For example, a value of 0.5 would imply 50% GPU memory utilization. If unspecified, will use the default value of 0.92.
- **cache_dtype** = 'auto'. Data type for kv cache storage. If "auto", will use model data type. CUDA 11.8+ supports fp8 (=fp8_e4m3) and fp8_e5m2. ROCm (AMD GPU) supports fp8

(=fp8_e4m3). Intel Gaudi (HPU) supports fp8 (using fp8_inc).

- **is_attention_free** = False. Whether the model is attention-free. This is primarily set in ModelConfig and that value should be manually duplicated here.
- **num_gpu_blocks_override** = None. Number of GPU blocks to use. This overrides the profiled num_gpu_blocks if specified. Does nothing if None. Used for testing preemption.
- **sliding_window** = None. Sliding window size for the KV cache. This is primarily set in ModelConfig and that value should be manually duplicated here.
- **enable_prefix_caching** = True. Whether to enable prefix caching.
- **prefix_caching_hash_algo** = 'sha256'. Set the hash algorithm for prefix caching: - "sha256" uses Pickle for object serialization before hashing. This is the current default, as SHA256 is the most secure choice to avoid potential hash collisions. - "sha256_cbor" provides a reproducible, cross-language compatible hash.
- **prefix_cache_retention_interval** = 0. Token interval between retained sliding-window and Mamba prefix-cache checkpoints. 0 retains only semantic checkpoints, including the latest replay boundary and shared-prefix junctions.
- **kv_cache_dtype_skip_layers** = list(). Layer patterns to skip KV cache quantization. Accepts layer indices (e.g., '0', '2', '4') or attention type names (e.g., 'sliding_window').
- **mamba_page_size_padded** = None. Optional override for mamba page size; used by hybrid mamba/attention models to ensure exact alignment with attention page size.
- **skip_page_size_padded** = None. Optional override for the page size of layers skipped from KV cache quantization (–kv-cache-dtype-skip-layers); set during block-size alignment so unquantized skip layers pad up to the quantized primary's page.
- **mamba_block_size** = None. Size of a contiguous cache block in number of tokens for mamba cache. Can be set only when prefix caching is enabled. Value must be a multiple of 8 to align with causal_conv1d kernel.
- **mamba_cache_dtype** = 'auto'. The data type to use for the Mamba cache (both the conv as well as the ssm state). If set to 'auto', the data type will be inferred from the model config.
- **mamba_ssm_cache_dtype** = 'auto'. The data type to use for the Mamba cache (ssm state only, conv state will still be controlled by mamba_cache_dtype). If set to 'auto', the data type for the ssm state will be determined by mamba_cache_dtype.
- **mamba_cache_mode** = 'none'. The cache strategy for Mamba layers: - "none": set when prefix caching is disabled. - "align": only cache the mamba state of the last token of each scheduler step and when the token is at position i * block_size.
- **enable_mamba_shared_prefix_checkpoint** = False. Also register a Mamba "align" checkpoint at the shared-prefix junction – where an EAGLE/MTP sibling was observed to resume – instead of only at the prompt tail. Off by default; only takes effect with mamba_cache_mode "align", EAGLE on the Mamba group, and a prefix match unit smaller than the Mamba block size.
- **replayssm_buffer_len** = 16. ReplaySSM logical history length B for Mamba2. Triton uses B physical rows and FlashInfer uses B+T, where T is the target verification length. Kimi-K3 speculative decode does not use B.
- **use_replayssm** = False. Use the ReplaySSM Mamba2 decode kernel: cache recent SSM inputs and skip the per-step full-state store, writing the checkpoint back only on flush. Requires mamba_cache_mode 'none' or 'align' (prefix caching) and the Triton or FlashInfer mamba backend.

- **use_kda_recoverssm** = `False`. Whether Kimi-K3 KDA uses RecoverSSM speculative decode.
- **num_gpu_blocks** = `None`. The number of blocks to allocate for GPU memory.
- **num_cpu_blocks** = `None`. The number of blocks to allocate for CPU memory.
- **effective_attention_block_size** = `None`. Full-attention block size in tokens, including DCP, or None if unavailable.
- **kv_cache_size_tokens** = `None`. Per-DP-engine KV cache capacity in tokens (group-aware). Uses group-aware capacity since num_gpu_blocks * block_size can be wrong for hybrid models where requests occupy multiple KV cache groups.
- **kv_cache_max_concurrency** = `None`. Per-DP-engine maximum concurrency at max_model_len tokens.
- **hash_block_size** = `None`. Tokens per prefix-cache block hash.
- **cache_hit_alignment_tokens** = `None`. Token granularity at which prefix-cache hits land: hash_block_size with partial hash hits, otherwise the scheduler block size.
- **kv_sharing_fast_prefill** = `False`. In some KV sharing setups, e.g. YOCO some layers can skip tokens corresponding to prefill. This flag enables attention metadata for eligible layers to be overridden with metadata necessary for implementing this optimization in some models (e.g. ...)
- **swa_bounded_replay** = `True`. Keep the sliding-window KV of models that support it (DeepSeek-V4.1) out of prefix caching and rebuild it after a prefix hit by recomputing the hit's last window. The layers past the last KV-source layer then also run eager prefill steps on each request's trailing window only.
- **kv_cache_memory_bytes** = `None`. Size of KV Cache per GPU in bytes. By default, this is set to None and vllm can automatically infer the kv cache size based on gpu_memory_ utilization.
- **kv_offloading_size** = `None`. Size of the KV cache offloading buffer in GiB. When TP $> 1$, this is the total buffer size summed across all TP ranks. By default, this is set to None, which means no KV offloading is enabled.
- **kv_offloading_backend** = `'native'`. The backend to use for KV cache offloading. Supported backends include 'native' (vLLM native CPU offloading), 'lmcache'. KV offloading is only activated when kv_offloading_size is set.

### 53.8.2 SchedulerConfig

*Generated from the vLLM 0.32.0 source (`SchedulerConfig`); 24 entries.*

- **max_model_len** = `required`. Maximum length of a sequence (including prompt and generated text). Note: This is stored in the ModelConfig, and is used only here to provide fallbacks and validate other attributes.
- **is_encoder_decoder** = `required`. True if the model is an encoder-decoder model. Note: This is stored in the ModelConfig, and is used only here to disable chunked prefill and prefix caching for encoder-decoder models.
- **runner_type** = `'generate'`. The runner type to launch for the model.
- **max_num_batched_tokens** = `2048`. Maximum number of tokens that can be processed in a single iteration. The default value here is mainly for convenience when testing. In real usage, this should be set in EngineArgs.create_engine_config.

- **max_num_scheduled_tokens** = None. Maximum number of tokens that the scheduler may issue in a single iteration. This is usually equal to max_num_batched_tokens, but can be smaller in cases when the model might append tokens into the batch (such as speculative decoding).
- **max_num_seqs** = 128. Maximum number of sequences to be processed in a single iteration. The default value here is mainly for convenience when testing. In real usage, this should be set in EngineArgs.create_engine_config.
- **max_num_active_seqs** = None. Maximum number of requests the scheduler admits into RUNNING. max_num_seqs sizes the model runner (per-request buffers and CUDA graph capture) and is also the default admission limit.
- **long_prefill_token_threshold** = 0. For chunked prefill, a request is considered long if the prompt is longer than this number of tokens. 0 disables the cap (default). The cap is not applied when the request is the only one in the batch, since there is no other request for it to starve.
- **long_prefill_token_threshold_adaptive** = False. Floor the effective long prefill token threshold at a fair share of the token budget: max_num_batched_tokens divided by the number of queued and running requests. Only applies when long_prefill_token_threshold is nonzero.
- **max_num_queued_reqs** = None. Maximum number of requests that can be in-flight (waiting or running) at the same time, or None for no limit. When the limit is reached, new requests are rejected with HTTP 503 so the client can retry on another instance.
- **max_num_queued_tokens** = None. Maximum total prompt tokens of requests currently in the prefill phase, or None for no limit. When the limit is reached, new requests are rejected with HTTP 503. This is a TTFT QoS mechanism: by setting it to target_TTFT * prefill_throughput you reject requests when the prefill backlog would exceed the latency target.
- **enable_chunked_prefill** = True. If True, prefill requests can be chunked based on the remaining max_num_batched_tokens. The default value here is mainly for convenience when testing. In real usage, this should be set in EngineArgs.create_engine_config.
- **is_multimodal_model** = False. True if the model is multimodal.
- **max_num_encoder_input_tokens** = required. Multimodal encoder compute budget, only used in V1. NOTE: This is not currently configurable. It will be overridden by max_num_batched_tokens in case max multimodal embedding size is larger.
- **encoder_cache_size** = required. Multimodal encoder cache size, only used in V1. NOTE: This is not currently configurable. It will be overridden by max_num_batched_tokens in case max multimodal embedding size is larger.
- **policy** = 'fcfs'. The scheduling policy to use: - "fcfs" means first come first served, i.e. requests are handled in order of arrival. - "priority" means requests are handled based on given priority (lower value means earlier handling) and time of arrival deciding any ties).
- **disable_chunked_mm_input** = False. If set to true and chunked prefill is enabled, we do not want to partially schedule a multimodal item.
- **scheduler_cls** = None. The scheduler class to use. "vllm.v1.core.sched.scheduler.Scheduler" is the default scheduler. Can be a class directly or the path to a class of form "mod.custom_class".
- **disable_hybrid_kv_cache_manager** = None. If set to True, KV cache manager will allocate the same size of KV cache for all attention layers even if there are multiple type of attention

layers like full attention and sliding window attention.

- **`scheduler_reserve_full_isl`** = `True`. If True, the scheduler checks whether the full input sequence length fits in the KV cache before admitting a new request, rather than only checking the first chunk. Prevents over-admission and KV cache thrashing with chunked prefill.
- **`watermark`** = `0.0`. Fraction of total KV cache blocks to keep free (the watermark) when admitting waiting or preempted requests into the running queue. This headroom helps avoid frequent KV cache eviction and the resulting repeated preemption of requests when GPU memory is scarce.
- **`prefill_schedule_interval`** = `1`. For data-parallel deployments, only admit new prefill requests once every N engine steps, aligned across DP ranks, to better balance per-step forward-pass times.
- **`async_scheduling`** = `None`. If set to False, disable async scheduling. Async scheduling helps to avoid gaps in GPU utilization, leading to better latency and throughput.
- **`stream_interval`** = `1`. The interval (or buffer size) for streaming in terms of token length. A smaller value (1) makes streaming smoother by sending each token immediately, while a larger value (e.g., 10) reduces host overhead and may increase throughput by batching multiple tokens before sending.

### 53.8.3 `ModelConfig` (fields relevant to RL)

*Generated from the vLLM 0.32.0 source (`ModelConfig`); 15 entries.*

- **`model`** = `'Qwen/Qwen3-0.6B'`. Name or path of the Hugging Face model to use. It is also used as the content for model_name tag in metrics output when served_model_name is not specified.
- **`tokenizer`** = `None`. Name or path of the Hugging Face tokenizer to use. If unspecified, model name or path will be used.
- **`trust_remote_code`** = `False`. Trust remote code (e.g., from HuggingFace) when downloading the model and tokenizer.
- **`dtype`** = `'auto'`. Data type for model weights and activations: - "auto" will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models. - "half" for FP16.
- **`seed`** = `0`. Random seed for reproducibility. We must set the global seed because otherwise, different tensor parallel workers would sample different tokens, leading to inconsistent results.
- **`revision`** = `None`. The specific model version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.
- **`max_model_len`** = `None`. Model context length (prompt and output). If unspecified, will be automatically derived from the model config. When passing via –max-model-len, supports k/m/g/K/M/G in human-readable format.
- **`quantization`** = `None`. Method used to quantize the weights. If None, we first check the quantization_config attribute in the model config file. If that is None, we assume the model weights are not quantized and use dtype to determine the data type of the weights.
- **`enforce_eager`** = `False`. Whether to always use eager-mode PyTorch. If True, we will disable CUDA graph and always execute the model in eager mode. If False, we will use CUDA graph and eager execution in hybrid for maximal performance and flexibility.

- **max_logprobs** = 20. Maximum number of log probabilities to return when logprobs is specified in SamplingParams. The default value comes the default for the OpenAI Chat Completions API. -1 means no cap, i.e. all (output_length * vocab_size) logprobs are allowed to be returned and it may cause OOM.
- **logprobs_mode** = 'raw_logprobs'. Indicates the content returned in the logprobs and prompt_logprobs. Supported mode: 1) raw_logprobs, 2) processed_logprobs, 3) raw_logits, 4) processed_logits. Raw means the values before applying any logit processors, like bad words.
- **enable_prompt_embeds** = False. If True, enables passing text embeddings as inputs via the prompt_embeds key. WARNING: The vLLM engine may crash if incorrect shape of embeddings is passed. Only enable this flag for trusted users!
- **generation_config** = 'auto'. The folder path to the generation config. Defaults to "auto", the generation config will be loaded from model path. If set to "vllm", no generation config is loaded, vLLM defaults will be used.
- **override_generation_config** = dict(). Overrides or sets generation config. e.g. {"temperature": 0.5}. If used with –generation-config auto, the override parameters will be merged with the default config from the model.
- **enable_sleep_mode** = False. Enable sleep mode for the engine (only cuda and hip platforms are supported).

### 53.8.4 ParallelConfig (selected)

*Generated from the vLLM 0.32.0 source (ParallelConfig); 6 entries.*

- **pipeline_parallel_size** = 1. Number of pipeline parallel groups.
- **tensor_parallel_size** = 1. Number of tensor parallel groups.
- **data_parallel_size** = 1. Number of data parallel groups. MoE layers will be sharded according to the product of the tensor, prefill-context, and data parallel sizes.
- **enable_expert_parallel** = False. Use expert parallelism instead of tensor parallelism for MoE layers.
- **distributed_executor_backend** = None. Backend to use for distributed model workers, either "ray" or "mp" (multiprocessing). If the product of pipeline_parallel_size and tensor_parallel_size is less than or equal to the number of GPUs available, "mp" will be used to keep processing on a single host.
- **worker_extension_cls** = ''. The full name of the worker extension class to use. The worker extension class is dynamically inherited by the worker class. This is used to inject new attributes and methods to the worker class for use in collective_rpc calls.

### 53.8.5 LoRAConfig and WeightTransferConfig

*Generated from the vLLM 0.32.0 source (LoRAConfig); 12 entries.*

- **max_lora_rank** = 16. Max LoRA rank.
- **max_loras** = 1. Max number of LoRAs in a single batch.
- **fully_sharded_loras** = False. By default, only half of the LoRA computation is sharded with tensor parallelism. Enabling this will use the fully sharded layers. At high sequence

length, max rank or tensor parallel size, this is likely faster.

- **`max_cpu_loras`** = `None`. Maximum number of LoRAs to store in CPU memory. Must be >= than max_loras.
- **`lora_dtype`** = `'auto'`. Data type for LoRA. If auto, will default to base model dtype.
- **`target_modules`** = `None`. Restrict LoRA to specific module suffixes (e.g., ["o_proj", "qkv_proj"]). If None, all supported LoRA modules are used. This allows deployment-time control over which modules have LoRA applied, useful for performance tuning.
- **`default_mm_loras`** = `None`. Dictionary mapping specific modalities to LoRA model paths; this field is only applicable to multimodal models and should be leveraged when a model always expects a LoRA to be active when a given modality is present.
- **`enable_tower_connector_lora`** = `False`. If True, LoRA support for the tower (vision encoder) and connector of multimodal models will be enabled. This is an experimental feature and currently only supports some MM models such as the Qwen VL series.
- **`specialize_active_lora`** = `False`. Whether to construct lora kernel grid by the number of active LoRA adapters. When set to True, separate cuda graphs will be captured for different counts of active LoRAs (powers of 2 up to max_loras), which can improve performance for variable LoRA usage patterns at the cost of increased startup time and memory usage.
- **`enable_mixed_moe_lora_format`** = `False`. If True, force the engine to use the universal 2D MoE LoRA wrapper (FusedMoEWithLoRA) regardless of the model's is_3d_moe_weight flag, so that 2D-format and 3D-format MoE LoRA adapters can be served in the same deployment.
- **`enable_moe_shared_loras`** = `False`. If True, load MoE expert adapters in the "shared-outer" layout, where the gate/up (w1/w3) lora_A and the down (w2) lora_B are shared across all experts (stored once with expert-dim 1) instead of per-expert.
- **`max_lora_cls_labels`** = `None`. Maximum output size for LoRA classification heads. Defaults to the base classification head size.

*Generated from the vLLM 0.32.0 source (`WeightTransferConfig`); 1 entries.*

- **`backend`** = `'nccl'`. The backend to use for weight transfer. Validated against the WeightTransferEngineFactory registry at engine creation time.

PART XI

# Projects

CHAPTER 54

# Projects I: SFT, Rewards and Preferences

**Why this part matters.**
Interviewers trust what you have built more than what you have read. The ten projects in this part (five here and five in the next chapter) are chosen so that each one produces a result you can explain in two minutes, a plot you can show, and a failure you can describe. Every project has the same parts: a **goal**, the **skills** it proves, **steps**, **starter code** (tested on CPU with toy data; the real run swaps in a real model), **metrics**, the **results you should expect**, the **compute** needed, and **stretch goals**. Do them in order: later projects reuse earlier ones.

## 54.1 How to run a project well

A project is not "I ran a script". It is a small experiment. Write down, before you start: the question, the baseline, the one thing you will change, the metric that decides, and what result would surprise you. Keep a run log (date, git commit, config, metric). Fix the seeds and run at least two seeds for any claim. Keep an evaluation set you never train on and never tune on more than a few times. At the end write one page: question, setup, result table, one plot, one failure, what you would do next. That page is what you talk through in an interview.

**Why this, not that? Why small models and many runs, not one big run?**

**Reason.** Post-training skill is mostly judgment about what to change and how to tell whether it helped. You learn that from many cheap, controlled runs (a 0.5B to 1.5B model trains in minutes to hours), not from one expensive run you cannot repeat. **Trade-off.** Some effects only appear at scale: emergent long reasoning under RL, some reward-hacking behaviours, systems bottlenecks across nodes. Small-model results can mislead on those. **When the alternative is better.** When the question itself is about scale (throughput across nodes, memory of a 70B model, behaviour that needs a strong base model), rent the bigger setup for a short, well-planned run after the small runs have fixed the bugs.

## 54.2 Project 1: Instruction-tune a small model with correct loss masking

**Goal.** Turn a base model (for example Qwen2.5-0.5B) into a chat model with SFT and LoRA, and show that you handled the chat template, the loss mask and packing correctly.

**Skills.** Chat templates and special tokens (Chapter 6), dataset formats and masking (Chapters 8, 9, 47), LoRA (Chapters 10, 46), evaluation of instruction following.

**Steps.**

1. Pick about 10k–50k conversations from an open instruction dataset with a permissive licence. Deduplicate and remove items that overlap your evaluation set (Chapter 39).
2. Render each conversation with the model's chat template. Look at the raw text of five examples. Check the end-of-turn token is present after every assistant message, and that no default system prompt was added that you do not want (the Qwen2.5 template inserts one when no system message is given).
3. Build the loss mask: only assistant tokens (including the end-of-turn token) count. Verify on one example by printing the tokens with mask 1.
4. Train with LoRA on all linear layers (rank 16, alpha 32, learning rate 2e-4, 1–3 epochs). Run a second configuration with loss on all tokens.
5. Evaluate both on a held-out set: exact-match on short-answer items, the rate of responses that stop properly, and a pairwise judge comparison on open prompts (Chapter 37).

**Starter code.** The function below builds the loss mask by rendering the conversation turn by turn and checks it. It works with any tokenizer whose template renders a prefix of a conversation as a prefix of the full text (true for ChatML templates such as Qwen's; check yours).

```
import os; os.environ["HF_HUB_OFFLINE"] = "1"
import warnings; warnings.filterwarnings("ignore")
from transformers import AutoTokenizer
tok = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B-Instruct")

def build_example(messages, tok):
    """Return input_ids and labels; labels are -100 everywhere except assistant turns."""
    ids, labels = [], []
    for i in range(len(messages)):
        prefix = tok.apply_chat_template(messages[: i + 1], tokenize=False)
        prev = tok.apply_chat_template(messages[:i], tokenize=False) if i else ""
        assert prefix.startswith(prev), "template is not prefix-stable; use a
         ↪ generation-marked template"
        piece = tok(prefix[len(prev):], add_special_tokens=False)["input_ids"]
        if messages[i]["role"] == "assistant":
            # mask the header '<|im_start|>assistant\n' (3 tokens), train on content +
             ↪ <|im_end|> + newline
            header = len(tok("<|im_start|>assistant\n",
 ↪ add_special_tokens=False)["input_ids"])
            lab = [-100] * header + piece[header:]
        else:
            lab = [-100] * len(piece)
        ids += piece; labels += lab
    return ids, labels

msgs = [{"role": "system", "content": "Be brief."},
        {"role": "user", "content": "What is 2+3?"},
        {"role": "assistant", "content": "5"},
        {"role": "user", "content": "And 4+4?"},
        {"role": "assistant", "content": "8"}]
ids, labels = build_example(msgs, tok)
full = tok.apply_chat_template(msgs, tokenize=True)
```

```
full = full["input_ids"] if isinstance(full, dict) or hasattr(full, "keys") else full
print("matches full render:", ids == list(full))
print("trained tokens:", [tok.decode([t]) for t, l in zip(ids, labels) if l != -100])
print("fraction of tokens trained: %.2f" % (sum(l != -100 for l in labels) /
 ↪  len(labels)))
```

The trained tokens are the two answers plus their <|im_end|> and newline. If <|im_end|> were missing from the trained set, the model would never learn to stop.

```
# Real run (one 24 GB GPU, about 1 hour for 20k conversations of ~600 tokens)
from trl import SFTConfig, SFTTrainer
from peft import LoraConfig
cfg = SFTConfig(output_dir="out/p1", num_train_epochs=2, per_device_train_batch_size=8,
                gradient_accumulation_steps=4, learning_rate=2e-4,
 ↪  lr_scheduler_type="cosine",
                warmup_steps=0.03, max_length=2048, packing=True,
 ↪  assistant_only_loss=True,
                logging_steps=10, bf16=True)
trainer = SFTTrainer(model="Qwen/Qwen2.5-0.5B", args=cfg, train_dataset=train_ds,
                     peft_config=LoraConfig(r=16, lora_alpha=32,
 ↪  target_modules="all-linear"))
trainer.train()
```

`assistant_only_loss=True` needs a chat template with generation markers (Chapter 47); with a base model you set a template that has them, or use the manual masking above and pass `input_ids/labels` directly.

**Metrics.** Training and validation loss on assistant tokens; stop rate (fraction of responses that end with the end-of-turn token before the length limit); exact-match on short-answer items; judge win-rate of masked vs unmasked training.

**Expected results.** Both runs reach similar validation loss on assistant tokens. The all-token run spends capacity learning to predict user turns and often produces responses that continue into a fake "user" turn; its stop rate is lower. Masked training usually wins the judge comparison, by a small margin on clean data.

**Compute.** One 16–24 GB GPU; 1–3 hours. A free notebook GPU works with a 0.5B model and LoRA.

**Stretch goals.** Compare packing with and without position-id reset (cross-contamination, Chapter 9); compare NEFTune noise; try a DFT loss (Chapter 47); measure the effect of the default system prompt being present in training but not in evaluation.

**Why this, not that? Why mask the prompt, not train on everything?**

**Reason.** The goal is $p(\text{response} \mid \text{prompt})$. Loss on prompt tokens trains the model to imitate users, adds gradient noise from text you do not want it to produce, and lets long prompts dominate the loss. **Trade-off.** Masking discards signal; with little data, prompt tokens carry useful language-modelling signal and can regularise. **When the alternative is better.** Continued pre-training on domain text, very small datasets where extra tokens

help, or datasets where prompts are high quality and short (the effect is then small). Some reports find loss on prompts helps when responses are very short.

## 54.3 Project 2: Train and audit a reward model

**Goal.** Train a Bradley–Terry reward model on preference pairs, then audit it: accuracy, calibration, and sensitivity to length and other spurious features.

**Skills.** Bradley–Terry loss (Chapter 19), RewardTrainer (Chapter 48), calibration, reward-hacking risks (Chapter 21).

**Steps.**

1. Take a public preference dataset with chosen/rejected pairs; split train/validation/test by prompt (not by pair) so the same prompt never appears in two splits.
2. Train a reward model from an SFT model (0.5B–1.5B) for one epoch with learning rate about 1e-5 (full fine-tune) or 1e-4 (LoRA).
3. Report pairwise accuracy on the test set, overall and by bucket (length difference, topic, and margin of agreement if available).
4. **Length audit.** Measure the correlation between reward and response length, and the accuracy on pairs where the chosen response is shorter.
5. **Calibration.** Treat $\sigma(r_w - r_l)$ as the predicted probability that the chosen response wins. Bin predictions and compare with the observed win rate (expected calibration error, ECE).
6. **Perturbation test.** Append irrelevant but plausible filler ("I hope this helps!") to responses and measure how much the reward rises.

**Starter code.** A simulation in which the human labels depend on hidden quality, and the data has a mild length bias. The reward model sees quality-correlated features and length. It learns the length bias, which the audit then detects.

```
import torch, math
torch.manual_seed(0)
N, D = 4000, 8
def make_pairs(n):
    qa, qb = torch.randn(n), torch.randn(n)                 # hidden quality
    la, lb = torch.rand(n) * 2, torch.rand(n) * 2           # length (in 100s of tokens)
    xa = torch.cat([qa[:, None] + 0.5 * torch.randn(n, D - 1), la[:, None]], 1)  # noisy
 ↪  features + length
    xb = torch.cat([qb[:, None] + 0.5 * torch.randn(n, D - 1), lb[:, None]], 1)
    p = torch.sigmoid(2.0 * (qa - qb) + 0.6 * (la - lb))      # labellers like quality,
 ↪  and a bit of length
    a_wins = torch.rand(n) < p
    xw = torch.where(a_wins[:, None], xa, xb); xl = torch.where(a_wins[:, None], xb, xa)
    return xw, xl
xw, xl = make_pairs(N); tw, tl = make_pairs(2000)
rm = torch.nn.Linear(D, 1)
opt = torch.optim.Adam(rm.parameters(), lr=0.05)
```

**Notation on this page:** $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $r(x, y)$: the reward for response $y$ to prompt $x$

```
for step in range(300):
    loss = -torch.nn.functional.logsigmoid(rm(xw) - rm(xl)).mean()
    opt.zero_grad(); loss.backward(); opt.step()
with torch.no_grad():
    m = (rm(tw) - rm(tl)).squeeze(1)
    acc = (m > 0).float().mean().item()
    shorter = tw[:, -1] < tl[:, -1]
    print(f"test accuracy {acc:.3f} | on pairs where chosen is shorter {(m[shorter] >
     ↪  0).float().mean():.3f}"
          f" | longer {(m[~shorter] > 0).float().mean():.3f}")
    print(f"weight on length feature {rm.weight[0, -1]:.2f} (labellers' true length
     ↪  weight 0.6)")
    # Expected calibration error (10 bins)
    p = torch.sigmoid(m); y = torch.ones_like(p)              # chosen always "won";
 ↪  symmetrise:
    p = torch.cat([p, 1 - p]); y = torch.cat([y, torch.zeros_like(y)])
    ece = 0.0
    for lo in torch.linspace(0, 0.9, 10):
        b = (p >= lo) & (p < lo + 0.1)
        if b.any(): ece += b.float().mean() * abs(p[b].mean() - y[b].mean())
    print(f"ECE {ece:.3f}")
    filler = tw.clone(); filler[:, -1] += 0.5               # +50 tokens of filler
    print(f"mean reward gain from 50 filler tokens: {(rm(filler) - rm(tw)).mean():.2f} "
          f"(std of rewards {rm(tw).std():.2f})")
```

The reward model is well calibrated and accurate overall, and it learned the labellers' length preference (weight about 0.5 against the true 0.6). That is the problem: it is less accurate when the chosen answer is the shorter one, and 50 tokens of pure filler raise the reward by about a sixth of a standard deviation with no change in quality. A policy optimised against it has a cheap way to gain reward: pad (Chapter 21). Good calibration does not mean the reward is safe to optimise.

**Metrics.** Test accuracy (overall and by bucket), ECE, reward–length correlation, filler gain in units of the reward standard deviation.

**Expected results.** On real data, 0.5B–1.5B reward models trained on general preference data typically reach about 65–75% test accuracy; agreement between human labellers caps it (often 70–80%). Length correlation is usually clearly positive.

**Compute.** One 24 GB GPU, a few hours for 50k pairs at 1k tokens.

**Stretch goals.** Add the reward-centring term (`center_rewards_coefficient`) and show it removes the arbitrary offset; train an ensemble of three and use disagreement as uncertainty; train with a length-debiasing term (regress out length, as in ODIN-style approaches) and repeat the audit.

## 54.4 Project 3: A DPO-family ablation

**Goal.** Starting from the Project 1 SFT model, compare DPO (sigmoid), IPO and length-normalised DPO on the same data, with matched compute, and explain the differences through their loss shapes.

**Skills.** DPO derivation and variants (Chapters 22, 23), the TRL DPO configuration (Chapter 49), evaluation with judges and length control (Chapter 37).

**Steps.**

1. Use 10k–30k preference pairs whose prompts are in the SFT model's domain. Ideally, generate the pairs on-policy: sample two responses from the SFT model and label them with a strong judge or a reward model (Chapter 24).
2. Train `loss_type="sigmoid"` with $\beta \in \{0.05, 0.1, 0.3\}$; then `"ipo"` and `"sigmoid_norm"` at their best $\beta$. Keep the learning rate low (5e-7 to 1e-6 for full fine-tuning; about 5e-6 to 2e-5 for LoRA).
3. Log implicit reward margins, chosen and rejected log-probabilities, and response length every few steps.
4. Evaluate: judge win-rate against the SFT model with and without length control, average length, and a capability check (a small set of maths or knowledge questions) to catch regressions.

**Starter code.** The losses written from log-probabilities, and their "push" (the negative gradient with respect to the margin), so you can predict behaviour before training.

```
import torch
def dpo_losses(pi_w, pi_l, ref_w, ref_l, len_w, len_l, beta=0.1, simpo_gamma=0.5):
    """Inputs: summed log-probs of chosen/rejected under policy and reference, and their
     ↪ lengths."""
    u = beta * ((pi_w - ref_w) - (pi_l - ref_l))                    # DPO margin
    h = (pi_w - ref_w) / len_w - (pi_l - ref_l) / len_l             # per-token
 ↪ log-ratio difference
    return {
        "sigmoid": -torch.nn.functional.logsigmoid(u),
        "ipo": (h - 1 / (2 * beta)) ** 2,                           # TRL: per-token
         ↪ averages
        "sigmoid_norm": -torch.nn.functional.logsigmoid(beta * h),
        "simpo": -torch.nn.functional.logsigmoid(beta * (pi_w / len_w - pi_l / len_l) -
         ↪ simpo_gamma),  # no reference
    }
# Push = -dloss/dmargin for the sigmoid loss: sigma(-u). Strong when the pair is wrong,
 ↪ fades when right.
u = torch.tensor([-2.0, 0.0, 2.0, 6.0], requires_grad=True)
(-torch.nn.functional.logsigmoid(u)).sum().backward()
print("sigmoid push at u =", u.tolist(), "->", [round(x, 3) for x in (-u.grad).tolist()])
# IPO keeps pushing toward a target, then pulls back: overshoot is penalised
h = torch.tensor([0.0, 2.5, 5.0, 10.0], requires_grad=True); beta = 0.1
((h - 1 / (2 * beta)) ** 2).sum().backward()
print("ipo push at h =", h.tolist(), "->", [round(x, 2) for x in (-h.grad).tolist()])
# Length bias check: same per-token quality, longer chosen response gets a larger
 ↪ sigmoid margin
t = lambda v: torch.tensor([v])
long_ = dpo_losses(t(-40.), t(-60.), t(-50.), t(-60.), t(100.), t(100.))
short_ = dpo_losses(t(-4.), t(-6.), t(-5.), t(-6.), t(10.), t(10.))
print("sigmoid loss long %.3f vs short %.3f | sigmoid_norm long %.3f vs short %.3f" % (
```

```
    long_["sigmoid"].item(), short_["sigmoid"].item(), long_["sigmoid_norm"].item(),
↪   short_["sigmoid_norm"].item()))
```

Summed log-probabilities make the margin grow with length, so plain DPO is "more satisfied" by long responses with the same per-token improvement; the normalised loss treats them the same.

**Metrics.** Raw and length-controlled win-rate vs SFT; mean length; chosen and rejected log-probabilities over training (both falling is a common DPO pattern, Chapter 22); capability regression.

**Expected results.** DPO raises the raw win-rate clearly; part of the gain is length. With $\beta = 0.05$ the length growth and log-probability drop are largest. IPO and the normalised loss grow length less, often at a slightly lower raw win-rate but a similar length-controlled one. If chosen log-probabilities fall a lot, add an SFT term (`loss_type=["sigmoid","sft"]` with `loss_weights`).

**Compute.** One 24–48 GB GPU (policy and reference, or `precompute_ref_log_probs=True` to free memory); a few hours per run for a 0.5B–1.5B model.

**Stretch goals.** Online/iterative DPO (regenerate pairs from the current policy every epoch); compare to KTO with unpaired labels; plot the win-rate against KL from the SFT model for every run (the fair comparison).

**Why this, not that? Why compare methods at matched KL, not at matched steps?**

**Reason.** Every method trades reward against distance from the reference. A method that moved further can look better simply because it moved further. Plotting win-rate against KL (or against length) shows the frontier each method reaches. **Trade-off.** Measuring KL needs reference log-probabilities on samples from the policy (extra compute), and you need several $\beta$ values per method to draw a frontier. **When the alternative is better.** When the deployment budget is fixed in steps or tokens and you only care which recipe is best at that budget, compare at matched compute, but report the KL too.

## 54.5 Project 4: Distil a reasoning teacher into a small student

**Goal.** Compare sequence-level distillation (SFT on teacher outputs) with on-policy distillation (student samples, teacher scores every token) on maths word problems.

**Skills.** Forward vs reverse KL (Chapters 3, 12), on-policy training, TRL's `DistillationTrainer`.

**Steps.**

1. Teacher: a 7B instruct model with good maths; student: a 0.5B model of the same tokenizer family (token-level distillation needs a shared vocabulary).
2. **Sequence-level.** Sample 4 teacher solutions per training problem, keep only correct ones (rejection sampling), and SFT the student on them.
3. **On-policy.** Train with `DistillationTrainer`: the student generates, and the loss is a divergence between student and teacher next-token distributions on the student's own tokens. Its `beta` interpolates the generalised JSD: 0.0 is forward KL, 0.5 JSD, 1.0 reverse KL (the

default).

4. Evaluate accuracy on a held-out maths set at temperature 0 and at temperature 0.7 with 8 samples (pass@1 and pass@8, Project 10).

**Starter code.** A teacher with two modes and a student that can only fit one shows the two KL directions: forward KL spreads the student over both modes (mass where the teacher has little), reverse KL locks onto one mode.

```
import torch
x = torch.linspace(-6, 6, 241)
def normal(mu, s): return torch.exp(-0.5 * ((x - mu) / s) ** 2)
teacher = normal(-2.5, 0.6) + normal(2.5, 0.6); teacher = teacher / teacher.sum()
def student(mu, log_s):
    p = torch.exp(-0.5 * ((x - mu) / log_s.exp()) ** 2); return p / p.sum()
def fit(direction, mu0):
    mu, ls = torch.tensor(mu0, requires_grad=True), torch.tensor(-0.5,
 ↪  requires_grad=True)
    opt = torch.optim.Adam([mu, ls], lr=0.05)
    for _ in range(1500):
        q = student(mu, ls) + 1e-12
        loss = (teacher * (teacher.clamp_min(1e-12).log() - q.log())).sum() if direction
 ↪  == "forward" \
            else (q * (q.log() - teacher.clamp_min(1e-12).log())).sum()
        opt.zero_grad(); loss.backward(); opt.step()
    q = student(mu, ls).detach()
    return mu.item(), ls.exp().item(), q[(x > -1) & (x < 1)].sum().item()
for d in ["forward", "reverse"]:
    mu, s, mid = fit(d, 1.0)                          # both start slightly right of centre
    print(f"{d:8s} KL: mean {mu:+.2f}, width {s:.2f}, student mass between the modes
     ↪  {mid:.2f}")
print(f"teacher mass between the modes {teacher[(x > -1) & (x < 1)].sum():.3f}")
```

For language models, "mass between the modes" is the student's probability on token sequences the teacher would never write. Forward KL (sequence-level distillation is a sampled version of it) produces a student that hedges; reverse KL (on-policy) produces a student that commits to what it can do well, at the cost of diversity.

**Metrics.** Pass@1 and pass@8; average output length; teacher log-probability per token of student outputs (how "teacher-like" they are); compute in GPU-hours.

**Expected results.** Sequence-level distillation gives a large jump over the base student cheaply. On-policy distillation on top of it usually adds more at a similar or smaller token budget, because it trains on the student's own mistakes (Chapter 12 cites the reports). Pass@8 gains less than pass@1 under reverse KL.

**Compute.** Teacher sampling: a few GPU-hours with vLLM. On-policy distillation: teacher and student on the same 40–80 GB GPU (0.5B + 7B in bf16 is about 15 GB of weights).

**Stretch goals.** Vary `beta` (0, 0.5, 1); mix on-policy and teacher-generated batches; distil from a teacher with a different tokenizer by sequence-level only and compare.

## 54.6 Project 5: QLoRA on a single GPU, with a memory budget you can defend

**Goal.** Fine-tune a 7B–8B model with QLoRA on one 24 GB GPU, predicting the memory before you run and checking the prediction with the PyTorch memory profiler.

**Skills.** Memory accounting (Chapter 40), quantisation (Chapter 11), LoRA and bitsandbytes settings (Chapter 46), gradient checkpointing.

**Steps.**

1. Write the memory budget (starter code): weights, LoRA parameters, their gradients and Adam states, activations (with and without checkpointing), CUDA workspace.
2. Load the model with `BitsAndBytesConfig(load_in_4bit=True, bnb_4bit_quant_type="nf4", bnb_4bit_use_double_quant=True, bnb_4bit_compute_dtype=torch.bfloat16)`; note that the default quant type is `fp4` and the default compute dtype is float32, both worse choices here.
3. Train LoRA rank 16 on all linear layers with gradient checkpointing; sequence length 2048, micro-batch 1–2.
4. Record `torch.cuda.max_memory_allocated()` and compare with the prediction. Explain the gap.
5. Compare against LoRA on the 16-bit model (if it fits) for quality: same data, same steps.

**Starter code.**

```
def lora_params(d_model, d_ff, n_layers, r, n_kv_heads, head_dim, n_heads):
    q = r * (d_model + n_heads * head_dim); o = r * (n_heads * head_dim + d_model)
    kv = 2 * r * (d_model + n_kv_heads * head_dim)
    mlp = 3 * r * (d_model + d_ff)                        # gate, up, down
    return n_layers * (q + o + kv + mlp)

def budget_gb(n_params, n_embed, lora, seq, micro_bs, d_model, n_layers, bits=4,
 ↪  ckpt=True):
    # bitsandbytes quantises the linear layers; embeddings and lm_head stay in bf16
    weights = (n_params - n_embed) * (bits / 8) * 1.03 + n_embed * 2   # +3%: NF4 block
 ↪  constants
    adapters = lora * (2 + 2 + 8 + 4)          # bf16 weights + bf16 grads + fp32 Adam m,
 ↪  v + fp32 master
    per_tok_layer = 34 * d_model                # bytes per token per layer, 16-bit,
 ↪  FlashAttention (Ch. 40)
    tokens = micro_bs * seq
    act = tokens * (n_layers * 2 * d_model + per_tok_layer) if ckpt else tokens *
 ↪  n_layers * per_tok_layer
    logits = tokens * 128256 * 4                # float32 logits for the loss
    return {k: round(v / 1e9, 2) for k, v in dict(weights=weights,
     ↪  adapters_and_optim=adapters,
                                                  activations=act, logits=logits).items()}

# A Llama-3-8B-like shape (8.03B parameters, 128,256-token vocabulary, untied
 ↪  embeddings)
```

```
n_lora = lora_params(d_model=4096, d_ff=14336, n_layers=32, r=16, n_kv_heads=8,
 ↪  head_dim=128, n_heads=32)
n_embed = 2 * 128256 * 4096
print("LoRA parameters: %.1f M" % (n_lora / 1e6))
for ck in (True, False):
    b = budget_gb(8.03e9, n_embed, n_lora, seq=2048, micro_bs=1, d_model=4096,
 ↪  n_layers=32, ckpt=ck)
    print("checkpointing" if ck else "no checkpointing", b, "total ~%.1f GB (+1-2 GB
     ↪  CUDA context)" % sum(b.values()))
print("full fine-tune with Adam, mixed precision: ~%.0f GB before activations" % (8.0e9
 ↪  * 16 / 1e9))
```

The activation term uses Chapter 40's estimate of about 34 *sbh* bytes per layer (with FlashAttention), and about 2 *sbh* per layer with checkpointing plus one layer being recomputed. The float32 logits for a 128k vocabulary are about 1 GB at 2048 tokens and often more in practice (a copy for the softmax gradient). Reconcile your measurement with these terms one at a time.

**Metrics.** Predicted vs measured peak memory; tokens per second; validation loss; downstream accuracy vs 16-bit LoRA.

**Expected results.** About 5–6 GB for the weights of an 8B model (4-bit linear layers plus 16-bit embeddings), well under 1 GB for adapters and their optimizer states, about 1 GB of logits, and activations that depend on the sequence length: about 0.8 GB with checkpointing and about 9 GB without at 2048 tokens. So QLoRA fits in 24 GB with room to spare at micro-batch 1. Without checkpointing, doubling the sequence length or the micro-batch pushes past 24 GB, which is why checkpointing is the default in QLoRA recipes. Quality is usually within noise of 16-bit LoRA on instruction data; throughput is lower (dequantisation cost).

**Compute.** One 24 GB GPU; a few hours.

**Stretch goals.** Sweep rank (8, 16, 64, 256) and learning rate together and plot loss vs rank (Chapter 10); try paged 8-bit AdamW; measure merge-and-quantise quality loss when exporting to GGUF with Unsloth (Chapter 52).

**Why this, not that? Why measure memory against a written prediction, not just reduce batch size until it fits?**

**Reason.** A prediction tells you which term dominates, so you know which lever to pull (checkpointing, sequence length, offload, chunked loss) and you can answer the interview question "how much memory does X need?" from first principles. **Trade-off.** It takes an hour and the activation model is approximate; the gap between prediction and measurement (fragmentation, temporary buffers) needs investigation. **When the alternative is better.** A one-off run on familiar hardware with a known recipe: trial and error is faster.

**Talking about these projects**

**Q1. "Tell me about a fine-tuning project." How should the answer be structured?** Question, baseline, the change, the metric, the result with numbers and seeds, one failure and how you found it, what you would do next. For Project 1: "masking assistant tokens raised

the stop rate from X to Y; I found the bug by printing trained tokens."

**Q2. What was the hardest bug in Project 1, typically?** Template issues: a missing end-of-turn token in the trained span, a default system prompt added silently, or a tokenizer that merges the boundary between turns, so the prefix-rendered and full-rendered token ids differ.

**Q3. Your reward model has 72% accuracy. Is that good?** Compare with inter-annotator agreement on the same data and with a length-only baseline. If length alone gives 60%, much of the 72% may be length.

**Q4. In Project 3, both chosen and rejected log-probs fell. Is the run broken?** Not necessarily; DPO only controls the margin. It becomes a problem if chosen probabilities fall so far that generation quality drops; add an SFT term or lower the learning rate.

**Q5. Why did reverse-KL distillation reduce pass@8 gains?** Reverse KL is mode-seeking: the student concentrates on fewer solution paths, so extra samples add less diversity.

**Q6. Where did the memory go in your QLoRA run?** Mostly activations and the logits, not weights: 4-bit weights of 8B are about 5 GB, the adapters and optimizer states well under 1 GB.

**Key takeaways**

- Every project: goal, baseline, one change, decisive metric, two seeds, one-page write-up.
- P1 masking and templates; P2 reward-model audit (accuracy, ECE, length); P3 DPO family at matched KL; P4 forward vs reverse KL distillation; P5 predicted vs measured memory.

CHAPTER 55

# Projects II: RL, Agents, Systems and Evaluation

**Why this chapter matters.**
These five projects cover what post-training interviews in 2026 focus on: RL with verifiable rewards, multi-turn tool use, reward hacking, the systems side of RL, and evaluation you can trust. Each one again has goal, skills, steps, tested starter code, metrics, expected results, compute and stretch goals.

## 55.1 Project 6: GRPO on grade-school maths with a verifier

**Goal.** Improve a small instruct model's maths accuracy with GRPO and a rule-based verifier, and explain every curve on the dashboard.

**Skills.** RLVR (Chapter 26), GRPO and its variants (Chapters 27, 28), reward functions in TRL (Chapter 50), RL debugging (Chapter 31).

**Steps.**

1. Model: Qwen2.5-0.5B-Instruct or 1.5B-Instruct. Data: a grade-school maths training set with numeric answers; hold out a test set.
2. Prompt the model to reason and put the final answer in `\boxed{}`. Measure the baseline accuracy with 8 samples per test problem at temperature 1.0 and with greedy decoding.
3. Write two reward functions: correctness (1 if the extracted answer equals the reference numerically, else 0) and format (small reward, for example 0.1, for exactly one `\boxed{}`). Unit-test them on 20 hand-written completions including tricky ones.
4. Filter the training set: drop problems the base model solves 8/8 or 0/8 (no learning signal in a group with equal rewards).
5. Train GRPO: `num_generations=8`, `max_completion_length=512`, `temperature=1.0`, `beta=0.0`, `loss_type="dapo"` (the TRL 1.16 defaults), learning rate 1e-6 (full) or 1e-5 (LoRA), `use_vllm=True` with `vllm_mode="colocate"`.
6. Watch: mean reward, fraction of groups with zero reward variance, completion length, clipped-ratio, entropy, and test accuracy every 50 steps.

**Starter code.** A verifier that is strict enough not to be hacked and loose enough not to reject correct answers, plus group advantages.

```
import re, torch
from fractions import Fraction

def extract_boxed(text):
```

```
    """Return the content of the last \\boxed{...}, handling nested braces; None if
     ↪  absent."""
    start = text.rfind("\\boxed{")
    if start < 0: return None
    i, depth, out = start + len("\\boxed{"), 1, []
    while i < len(text):
        c = text[i]; depth += (c == "{") - (c == "}")
        if depth == 0: return "".join(out)
        out.append(c); i += 1
    return None                                          # unbalanced: treat as no answer

def to_number(s):
    s = s.replace(",", "").replace("$", "").replace("\\%", "").strip()
    s = re.sub(r"^\\text\{(.*)\}$", r"\1", s)
    m = re.fullmatch(r"\\frac\{(-?\d+)\}\{(-?\d+)\}", s) or
 ↪  re.fullmatch(r"(-?\d+)/(-?\d+)", s)
    try:
        return Fraction(int(m.group(1)), int(m.group(2))) if m else Fraction(s)
    except (ValueError, ZeroDivisionError):
        return None

def correctness_reward(completions, answer, **kwargs):
    out = []
    for c, a in zip(completions, answer):
        text = c[0]["content"] if isinstance(c, list) else c
        boxed = text.count("\\boxed{")
        pred = extract_boxed(text)
        ok = boxed == 1 and pred is not None and to_number(pred) is not None and
 ↪  to_number(pred) == to_number(a)
        out.append(1.0 if ok else 0.0)                       # several boxed answers =
 ↪  hedging = no reward
    return out

def format_reward(completions, **kwargs):
    return [0.1 if (c[0]["content"] if isinstance(c, list) else c).count("\\boxed{") ==
     ↪  1 else 0.0
            for c in completions]

tests = [("so the answer is \\boxed{42}", "42", 1), ("\\boxed{42.0}", "42", 1),
 ↪  ("\\boxed{1,000}", "1000", 1),
         ("\\boxed{\\frac{3}{4}}", "0.75", 1), ("\\boxed{41} or \\boxed{42}", "42", 0),
 ↪  ("42", "42", 0),
         ("\\boxed{\\text{42}}", "42", 1), ("\\boxed{42", "42", 0), ("\\boxed{x=42}",
 ↪  "42", 0)]
got = correctness_reward([t for t, _, _ in tests], [a for _, a, _ in tests])
print("verifier unit tests pass:", got == [float(e) for _, _, e in tests])

def group_advantages(rewards, G, scale=True):
    r = torch.tensor(rewards, dtype=torch.float32).view(-1, G)
    a = r - r.mean(1, keepdim=True)
```

```
    if scale: a = a / (r.std(1, keepdim=True) + 1e-4)
    return a.view(-1)
r = [1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1]                    # 3 prompts x G=4
A = group_advantages(r, 4)
print("advantages:", [round(x, 2) for x in A.tolist()])
print("groups with no signal:", int((torch.tensor(r, dtype=torch.float32).view(-1,
 ↪  4).std(1) == 0).sum()), "of 3")
```

The "x=42" case is a judgement call: rejecting it is strict, and the model learns the format quickly; accepting it needs a parser and opens new hacks. Write such decisions down.

```
# Real run: one 80 GB GPU (1.5B model, colocated vLLM), roughly 6-12 hours for 500 steps
from trl import GRPOConfig, GRPOTrainer
cfg = GRPOConfig(output_dir="out/p6", learning_rate=1e-6, num_generations=8,
 ↪  per_device_train_batch_size=8,
                 gradient_accumulation_steps=8, max_completion_length=512,
 ↪  temperature=1.0, beta=0.0,
                 loss_type="dapo", use_vllm=True, vllm_mode="colocate",
 ↪  vllm_gpu_memory_utilization=0.3,
                 reward_weights=[1.0, 1.0], logging_steps=1, log_completions=True,
 ↪  bf16=True)
trainer = GRPOTrainer(model="Qwen/Qwen2.5-1.5B-Instruct", args=cfg,
 ↪  train_dataset=train_ds,
                      reward_funcs=[correctness_reward, format_reward])
trainer.train()
```

The dataset needs a `prompt` column (conversational) and an `answer` column; extra columns are passed to reward functions as keyword arguments.

**Metrics.** Test accuracy (greedy and mean over 8 samples); pass@8; mean completion length; zero-variance group fraction; entropy; the fraction of completions truncated at the length limit.

**Expected results.** Format reward saturates within tens of steps. Accuracy on grade-school maths rises by several to over ten points for a 0.5B–1.5B instruct model within a few hundred steps; most of the gain is in sampled accuracy (pass@1 at T=1), less in pass@8 (Chapter 30 discusses whether RL creates or sharpens abilities). Length may first drop, then grow slowly. If reward rises but test accuracy does not, read 20 completions: look for answer-format hacks.

**Compute.** 1× 80 GB GPU (or 2× 24 GB with server-mode vLLM on one); 6–12 hours.

**Stretch goals.** Compare `loss_type` "grpo", "dr_grpo" and "dapo"; `scale_rewards` "group" vs "none" (Dr. GRPO); asymmetric clipping with `epsilon_high=0.28`; dynamic sampling (drop zero-variance groups); a curriculum by base-model pass rate (Chapter 36).

**Why this, not that? Why filter out problems the model always or never solves, not train on all of them?**

**Reason.** GRPO's advantage is zero for every member of a group with equal rewards, so those prompts cost generation time and give no gradient (with `loss_type="dapo"` they also dilute the token normalisation). **Trade-off.** Problems that are 0/8 now may become solvable later, and 8/8 problems guard against forgetting. Filtering once at the start loses both. **When**

**the alternative is better.** Re-filter periodically (a moving curriculum), or keep a small fraction of solved problems as a regulariser; with a large G (32+) even hard problems give some signal.

## 55.2 Project 7: A multi-turn tool-use environment

**Goal.** Build an environment in which the model solves arithmetic word problems by calling a calculator tool over several turns, with correct masking of tool outputs, a turn limit, and an honest reward.

**Skills.** Environment design (Chapters 32, 33), multi-turn and tool-use RL (Chapter 34), sandboxing (Chapter 35), reward design (Chapter 36).

**Steps.**

1. Define the protocol: the model writes `<tool>calc: EXPR</tool>` to call the calculator, or `<answer>X</answer>` to finish. The environment returns `<result>...</result>` as a new message.
2. Implement `reset(task)` and `step(model_text)` returning observation, reward, terminated, truncated, info (Gymnasium semantics: *terminated* when the episode ends by itself, *truncated* when a limit cuts it).
3. Make the tool safe: parse expressions with Python's `ast` module and allow only numbers and arithmetic operators. Never `eval` model text.
4. Reward: 1 for a correct final answer, 0 otherwise; small penalty per malformed call; nothing for calling the tool (otherwise the model learns to call it needlessly).
5. During training, tool-result tokens are part of the context but excluded from the loss (they were not generated by the policy).
6. Connect to a trainer: TRL's GRPOTrainer accepts `tools` and an `environment_factory` (Chapter 50); veRL and OpenRLHF have agent loops (Chapter 51).

**Starter code.**

```
import ast, operator, re
OPS = {ast.Add: operator.add, ast.Sub: operator.sub, ast.Mult: operator.mul, ast.Div:
 ↪  operator.truediv,
       ast.USub: operator.neg, ast.Pow: operator.pow}
def safe_calc(expr):
    def ev(n):
        if isinstance(n, ast.Constant) and isinstance(n.value, (int, float)): return
         ↪  n.value
        if isinstance(n, ast.BinOp) and type(n.op) in OPS:
            if isinstance(n.op, ast.Pow) and abs(ev(n.right)) > 10: raise
             ↪  ValueError("exponent too large")
            return OPS[type(n.op)](ev(n.left), ev(n.right))
        if isinstance(n, ast.UnaryOp) and type(n.op) in OPS: return
         ↪  OPS[type(n.op)](ev(n.operand))
        raise ValueError("disallowed expression")
```

```
    return ev(ast.parse(expr, mode="eval").body)

class CalcEnv:
    def __init__(self, max_turns=4, bad_call_penalty=0.1):
        self.max_turns, self.pen = max_turns, bad_call_penalty
    def reset(self, task):
        self.task, self.turn, self.penalty = task, 0, 0.0
        return [{"role": "user", "content": task["question"]}], {}
    def step(self, text):
        self.turn += 1
        ans = re.search(r"<answer>(.*?)</answer>", text, re.S)
        if ans:
            try: ok = abs(float(ans.group(1)) - self.task["answer"]) < 1e-6
            except ValueError: ok = False
            return None, float(ok) - self.penalty, True, False, {"turns": self.turn}
        call = re.search(r"<tool>calc:(.*?)</tool>", text, re.S)
        if call:
            try: result = str(safe_calc(call.group(1).strip()))
            except (ValueError, SyntaxError, ZeroDivisionError) as e:
                result, self.penalty = f"error: {e}", self.penalty + self.pen
        else:
            result, self.penalty = "error: expected <tool> or <answer>", self.penalty +
 ↪  self.pen
        truncated = self.turn >= self.max_turns
        obs = {"role": "tool", "content": f"<result>{result}</result>"}
        return obs, (-self.penalty if truncated else 0.0), False, truncated, {"turns":
         ↪  self.turn}

def rollout(env, task, policy):
    msgs, _ = env.reset(task); total, loss_mask = 0.0, []
    while True:
        text = policy(msgs)
        msgs.append({"role": "assistant", "content": text}); loss_mask.append(1)
        obs, r, term, trunc, info = env.step(text); total += r
        if term or trunc: return total, info["turns"], loss_mask
        msgs.append(obs); loss_mask.append(0)                    # tool output: in
 ↪  context, not in loss

task = {"question": "A box holds 17 rows of 23 apples. 45 are eaten. How many are
 ↪  left?", "answer": 346}
def good_policy(msgs):
    last = msgs[-1]["content"]
    if "<result>" not in last: return "<tool>calc: 17*23 - 45</tool>"
    return f"<answer>{last[8:-9]}</answer>"
def looping_policy(msgs): return "<tool>calc: __import__('os').system('ls')</tool>"
for name, pol in [("good", good_policy), ("looping/unsafe", looping_policy)]:
    R, T, mask = rollout(CalcEnv(), task, pol)
    print(f"{name:15s} reward {R:+.2f} turns {T} loss mask over messages {mask}")
print("safe_calc('2**3*5') =", safe_calc("2**3*5"))
```

The unsafe call is rejected by the parser, the episode is truncated after four turns, and the accu-

mulated penalty makes the return negative.

**Metrics.** Success rate; mean turns; malformed-call rate; tool-call rate on problems that do not need the tool; truncation rate.

**Expected results.** A 1.5B–3B instruct model learns the protocol quickly (malformed calls fall to near zero); success rises when problems need arithmetic the model gets wrong in its head. Without a turn limit and a penalty for errors, policies that loop appear.

**Compute.** Same as Project 6; multi-turn rollouts are slower because each turn is a new generation call (prefix caching helps: Chapter 43).

**Stretch goals.** Replace the calculator with a Python sandbox in a container with time and memory limits (Chapter 35); add a "search" tool over a small document set; compare outcome reward with per-turn rewards; measure how the loss mask changes results by deliberately training on tool tokens.

**Why this, not that? Why parse tool input with an allow-list, not run it in eval with restricted globals?**

**Reason.** Restricted `eval` is famously escapable (attribute access to `__class__`, `__subclasses__` and so on). An RL policy optimises against your environment millions of times and will find escapes if any reward follows. **Trade-off.** An allow-list parser supports only what you wrote; richer tools need more code. **When the alternative is better.** When the tool must run general code: then use a real sandbox (a container or micro-VM with no network, limits and a fresh filesystem per episode), never in-process eval.

## 55.3 Project 8: Reward over-optimisation, measured

**Goal.** Reproduce the over-optimisation curve: as a policy is optimised against a proxy reward, the true ("gold") reward first rises, then falls. Show how a KL penalty and a better proxy change the curve.

**Skills.** Reward hacking and Goodhart's law (Chapter 21), KL-regularised objectives (Chapters 20, 22), measurement at matched KL.

**Steps.**

1. Real version: use a large reward model as "gold"; train a small proxy reward model on gold-labelled pairs (Project 2); run RL against the proxy; score samples with gold every N steps; plot gold and proxy against $\sqrt{\mathrm{KL}}$ from the start policy (the axis used by Gao et al., 2022).
2. Repeat with KL penalties of different strength, and with a proxy trained on more data.
3. Toy version (starter code): best-of-$n$ sampling against a proxy. Each response has a hidden quality and a "length"; the gold reward likes quality and dislikes very long answers; the proxy (a learned reward model) likes quality and length. Picking the best of $n$ samples by the proxy is the simplest optimiser, and its KL from the base policy is known exactly: $\mathrm{KL}_{\mathrm{BoN}} = \log n - (n-1)/n$.

**Starter code.**

**Notation on this page:** $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $n$: number of samples

```
import torch, math
torch.manual_seed(0)
def sample(m):
    q = torch.randn(m); l = torch.distributions.Exponential(1.0).sample((m,))   #
 ↪  quality, "length"
    gold = q - 0.15 * l ** 2              # the true reward dislikes very long answers
    proxy = q + 0.5 * l                   # the learned reward model likes length
    return gold, proxy
trials = 2000
for n in [1, 4, 16, 64, 256]:
    g, p = sample(trials * n); g, p = g.view(trials, n), p.view(trials, n)
    gold_bon = g[torch.arange(trials), p.argmax(1)].mean().item()   # pick by proxy,
 ↪  score by gold
    gold_oracle = g.max(1).values.mean().item()                     # pick by gold (the
 ↪  ceiling)
    kl = math.log(n) - (n - 1) / n
    print(f"n={n:4d} sqrt(KL)={math.sqrt(kl):4.2f} proxy={p.max(1).values.mean():5.2f} "
          f"gold(best-of-n by proxy)={gold_bon:+.2f} gold(oracle
           ↪  best-of-n)={gold_oracle:+.2f}")
```

The proxy score rises steadily with $n$. The gold score of the proxy's pick rises at first (quality and proxy agree on most samples), peaks between $n = 4$ and $n = 16$, then falls below where it started: at large $n$ the proxy's favourite is almost always an extremely long answer. The oracle column shows that the base policy had much better answers available all along. RL against the proxy follows the same shape along its own KL path; a KL penalty stops it earlier on the curve, which bounds both the damage and the gain.

**Metrics.** Gold and proxy reward vs $\sqrt{\mathrm{KL}}$; the KL at which gold peaks; length; the fraction of samples a human (you) judges as hacked.

**Expected results.** The gap between proxy and gold widens with KL; larger proxy models and more proxy training data move the gold peak to higher KL (Gao et al., 2022). A KL penalty bounds the damage but also the gain. Ensembles of reward models (taking the minimum or mean minus spread) delay the peak.

**Compute.** Toy: seconds. Real: a few GPU-days (gold scoring is the expensive part).

**Stretch goals.** Reward-model ensembles; length-penalised rewards; best-of-n sampling against the proxy (over-optimisation without training, at $\mathrm{KL} \approx \log n - (n-1)/n$).

## 55.4 Project 9: A mini asynchronous RL system

**Goal.** Build a small producer–consumer RL loop in which rollout workers generate with slightly stale weights, measure the staleness, and show how importance-weight truncation corrects the off-policy error.

**Skills.** RL system design (Chapter 44), rollout engines and logprob mismatch (Chapter 43), off-policy correction (Chapters 17, 28).

**Steps.**

1. Implement a learner (policy weights with a version number) and $W$ rollout worker threads. Workers copy the weights when they start a batch, sample, and push (samples, behaviour log-probabilities, version) into a bounded queue.
2. The learner pops batches, computes the policy-gradient loss with the importance ratio $\rho = \pi_\theta / \mu$ between the current policy and the behaviour policy, truncates $\rho$ at $C$ (truncated importance sampling, TIS), and steps.
3. Measure the staleness (learner version minus sample version) distribution as you vary the queue size and the number of workers.
4. Compare reward curves for: no correction, TIS with $C = 2$, and dropping samples older than one version.

**Starter code.** A bandit-sized version with real threads.

```
import threading, queue, torch, random
torch.manual_seed(0); random.seed(0)
K = 10
true_reward = torch.linspace(0, 1, K)                          # arm 9 is best
class Learner:
    def __init__(self):
        self.logits = torch.zeros(K, requires_grad=True); self.version = 0; self.lock =
        ↪  threading.Lock()
        self.opt = torch.optim.SGD([self.logits], lr=0.5)
    def snapshot(self):
        with self.lock: return self.logits.detach().clone(), self.version
def worker(learner, q, stop, batch=32):
    while not stop.is_set():
        logits, ver = learner.snapshot()
        p = torch.softmax(logits, 0)
        a = torch.multinomial(p, batch, replacement=True)
        r = true_reward[a] + 0.1 * torch.randn(batch)
        try: q.put((a, r, p[a].log(), ver), timeout=0.1)
        except queue.Full: pass
def train(n_workers, qsize, clip_c, steps=150):
    learner, q, stop = Learner(), queue.Queue(maxsize=qsize), threading.Event()
    ths = [threading.Thread(target=worker, args=(learner, q, stop), daemon=True) for _
 ↪  in range(n_workers)]
    for t in ths: t.start()
    stale = []
    for _ in range(steps):
        a, r, logp_mu, ver = q.get()
        with learner.lock:
            stale.append(learner.version - ver)
            logp = torch.log_softmax(learner.logits, 0)[a]
            rho = (logp.detach() - logp_mu).exp()
            w = rho.clamp(max=clip_c) if clip_c else torch.ones_like(rho)
            adv = r - r.mean()
            loss = -(w * adv * logp).mean()
            learner.opt.zero_grad(); loss.backward(); learner.opt.step();
 ↪  learner.version += 1
    stop.set(); [t.join() for t in ths]
```

**Notation on this page:** $W$: a weight matrix • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\rho$: importance ratio $\pi_\theta / \pi_{\theta_{\text{old}}}$ for a token • $\mu$: a mean

```
    p = torch.softmax(learner.logits.detach(), 0)
    return (p * true_reward).sum().item(), sum(stale) / len(stale), max(stale)
for nw, qs, c in [(1, 1, None), (4, 16, None), (4, 16, 2.0)]:
    rew, mean_st, max_st = train(nw, qs, c)
    print(f"workers={nw} queue={qs:2d} TIS C={c}: expected reward {rew:.3f}, staleness
     ↪  mean {mean_st:.1f} max {max_st}")
```

The exact numbers vary run to run because thread timing is not deterministic; the pattern to look for is that a larger queue and more workers increase staleness, and that the correction keeps learning stable when staleness is high. On a bandit this simple, stale gradients still point the right way, so the effect on final reward is small; on LLMs with large learning rates and long staleness it is not (Chapter 44).

**Metrics.** Staleness distribution; throughput (samples/s); reward vs wall-clock time (not vs steps: async wins on wall-clock); fraction of truncated importance weights.

**Expected results.** More workers and a bigger queue give higher throughput and higher staleness. Without correction, learning can become unstable at high staleness; TIS or a one-version staleness limit restores stability at a small cost in sample efficiency. In LLM RL the behaviour log-probabilities should come from the inference engine, so the same correction also handles engine–trainer numerical mismatch (Chapter 43).

**Compute.** CPU. Real version: wire it to vLLM workers and a small model on 2 GPUs.

**Stretch goals.** Replace threads with processes and a weight-broadcast step; partial rollouts (interrupt long generations at weight updates); measure the trainer's idle time against Chapter 44's simulation.

**Why this, not that? Why use the inference engine's log-probabilities as the behaviour policy, not recompute them in the trainer?**

**Reason.** The samples were drawn from the engine's distribution (its kernels, precision and sampling settings). The correct importance ratio divides by the probability the sample actually had. **Trade-off.** Engine log-probabilities can be noisier or computed after processing (temperature, penalties), so you must request the right mode (vLLM's `logprobs_mode`) and transfer them with the samples. **When the alternative is better.** Fully synchronous on-policy training where the trainer's recomputed "old" log-probabilities define the PPO ratio; many frameworks recompute them and add a separate engine-vs-trainer correction (as TRL's vLLM importance-sampling options do).

## 55.5 Project 10: An evaluation harness you can trust

**Goal.** Build a small harness that reports pass@k with confidence intervals, compares two models with a paired test, and checks for contamination.

**Skills.** Evaluation statistics (Chapter 37), data hygiene (Chapter 39).

**Steps.**

1. For each problem, sample $n$ completions (for example $n = 16$) and count the correct ones

**Notation on this page:** $n$: number of samples

$c$.

2. Report pass@k with the unbiased estimator $1 - \binom{n-c}{k} / \binom{n}{k}$ (Chen et al., 2021), averaged over problems.
3. Report a 95% confidence interval by bootstrapping over problems (not over samples: problems are the unit of variation).
4. Compare two models on the same problems with a paired bootstrap of the per-problem difference.
5. Contamination: compute the fraction of each test item's 13-grams that appear in the training data; flag items above a threshold; report scores with flagged items removed.

**Starter code.**

```
import math, random
from collections import Counter
random.seed(0)
def pass_at_k(n, c, k):
    if n - c < k: return 1.0
    return 1.0 - math.comb(n - c, k) / math.comb(n, k)
print("pass@1 n=16 c=4: %.3f (= c/n)" % pass_at_k(16, 4, 1), "| pass@8: %.3f" %
 ↪  pass_at_k(16, 4, 8))
# naive estimate of pass@8 from only 8 samples is noisy; the unbiased estimator uses all
 ↪  16
def bootstrap_ci(values, B=2000):
    means = sorted(sum(random.choices(values, k=len(values))) / len(values) for _ in
 ↪  range(B))
    return means[int(0.025 * B)], means[int(0.975 * B)]
# Two models on 200 problems: per-problem correct counts out of 16; B is truly better on
 ↪  some problems
p_true = [random.betavariate(0.7, 0.9) for _ in range(200)]
cA = [sum(random.random() < p for _ in range(16)) for p in p_true]
cB = [sum(random.random() < min(1, p + 0.04) for _ in range(16)) for p in p_true]
a1 = [pass_at_k(16, c, 1) for c in cA]; b1 = [pass_at_k(16, c, 1) for c in cB]
print("A pass@1 %.3f CI (%.3f, %.3f)" % (sum(a1) / 200, *bootstrap_ci(a1)))
print("B pass@1 %.3f CI (%.3f, %.3f)" % (sum(b1) / 200, *bootstrap_ci(b1)))
diff = [b - a for a, b in zip(a1, b1)]
lo, hi = bootstrap_ci(diff)
print("paired difference %.3f CI (%.3f, %.3f) -> %s" % (sum(diff) / 200, lo, hi,
      "significant" if lo > 0 or hi < 0 else "not significant"))
def ngrams(text, n=13):
    w = text.lower().split(); return {tuple(w[i:i + n]) for i in range(len(w) - n + 1)}
train_text = ("the train left the station at nine and travelled at sixty miles per hour "
              "for three hours before stopping at a small town to take on water")
test_items = ["a train left the station at nine and travelled at sixty miles per hour "
              "for three hours before stopping how far did it go",
              "a farmer has twelve cows and buys five more how many cows does the farmer
               ↪  have now in total today"]
tg = ngrams(train_text)
for t in test_items:
    g = ngrams(t); frac = len(g & tg) / max(1, len(g))
    print("overlap %.2f %s" % (frac, "FLAG" if frac > 0.3 else "ok"), "|", t[:45] + "...")
```

The separate confidence intervals of A and B overlap, yet the paired comparison is significant, because both models find the same problems hard and the pairing removes that shared variation.

**Metrics.** pass@1, pass@k with CIs; paired differences; contamination rate and the score on clean items only.

**Expected results.** With 200 problems, the 95% interval on pass@1 is several points wide; differences of 1–2 points are rarely significant without pairing and many samples. Contaminated items often score higher than clean ones; if removing them changes the ranking, say so.

**Compute.** Dominated by sampling: 200 problems × 16 samples with vLLM takes minutes for a small model.

**Stretch goals.** Add a judge-based metric with position swapping and length control (Chapter 37); add a canary string check; estimate how many problems you need to detect a 2-point difference with 80% power by simulation.

**Why this, not that? Why bootstrap over problems, not treat every sample as independent?**

**Reason.** Samples for the same problem are strongly correlated (an easy problem is solved by most samples). Treating 3,200 samples as independent understates the uncertainty several-fold. **Trade-off.** Bootstrapping over problems gives wide intervals on small benchmarks, which is uncomfortable but honest. **When the alternative is better.** When the question is about one fixed problem set (for example "how reliable is the model on these exact 200 tasks?"), sampling variance per problem is what matters, and a per-sample analysis answers that narrower question.

**Talking about these projects**

**Q1. Your GRPO reward went up but test accuracy did not. What do you check first?** Read completions with high reward for hacks (format tricks, multiple answers), then check that train and test prompts share format, then check truncation and the zero-variance group fraction.

**Q2. In Project 7, why exclude tool outputs from the loss?** They were not sampled by the policy; including them trains the model to predict tool outputs (hallucinate results) and makes the policy-gradient estimate wrong.

**Q3. In Project 8, why plot against $\sqrt{\mathrm{KL}}$?** Gao et al. found gold reward is close to a simple function of $d = \sqrt{\mathrm{KL}}$ (for best-of-n, $d(\alpha - \beta d)$), so curves are comparable across methods and scales on that axis.

**Q4. What does truncated importance sampling trade?** Truncating $\rho$ at $C$ bounds the variance of the update at the cost of bias toward the behaviour policy.

**Q5. A model improved pass@1 by 1.5 points on 500 problems. Is it real?** Run a paired bootstrap over problems; with several samples per problem and pairing, 1.5 points can be significant; without, usually not.

**Q6. How do you find contamination without the training data?** Membership-inference signals (unusually low perplexity on test items compared with paraphrases), completion of a test item from its first half, and performance drop on rewritten or newer versions of the benchmark.

**Notation on this page:** $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $d$: hidden size of the model • $\beta$: strength of the KL penalty (how close to $\pi_{\mathrm{ref}}$ the policy must stay) • $\alpha$: a weighting coefficient (defined in the text) • $\rho$: importance ratio $\pi_\theta/\pi_{\theta_{\mathrm{old}}}$ for a token

**Key takeaways**

- P6 GRPO with a unit-tested verifier and filtered prompts; P7 multi-turn tool env with safe tools and loss masks; P8 gold vs proxy vs $\sqrt{\text{KL}}$; P9 staleness and TIS; P10 pass@k, bootstrap over problems, paired tests, n-gram contamination.

**Notation on this page:** $D_{\text{KL}}(p\|q)$: KL divergence of $q$ from $p$

PART XII

# Interview Preparation

CHAPTER 56

# Interview Questions I: Concepts and Mathematics

**How to use this part.**
Chapters 56 and 57 hold 160 new questions with model answers, on top of the questions at the end of every earlier chapter. This chapter has conceptual questions (Q1–Q55) and mathematical questions (Q56–Q95); the next has coding (C1–C30), system design (S1–S20) and behavioural questions (B1–B15). Practise aloud: give the one-sentence answer first, then the reason, then a trade-off or an example. The chapter given in brackets is where the topic is taught in depth. An interviewer cares less about the exact words than about whether you can explain *why* and say *when it fails*.

## 56.1 Answering well

A strong answer has three layers. **The claim** in one sentence ("DPO optimises the same KL-regularised objective as RLHF, without sampling"). **The mechanism** in two or three sentences (the closed-form optimal policy, the reward written as a log-ratio, the Bradley–Terry likelihood). **The judgement**: a trade-off, a failure mode, or a number ("it is offline, so it cannot fix errors it never sees; online variants close part of the gap"). If you do not know, say what you would check and how. Never invent a paper, a number or a parameter name; interviewers in this field often wrote the paper.

**Why this, not that? Why practise short answers with a trade-off, not long complete ones?**

**Reason.** Interviews are conversations: the interviewer probes where they want depth. A short claim plus a trade-off invites the follow-up you are prepared for, and shows judgement, which is what they are hiring for. **Trade-off.** A short answer can sound shallow if you never get the follow-up; keep a "second layer" (derivation or number) ready and offer it. **When the alternative is better.** Written take-home tasks and research presentations, where completeness is expected and there is no back-and-forth.

## 56.2 Conceptual questions: the post-training pipeline

**Interview check**

**Q1. What is post-training, and why does a pre-trained model need it?** Everything after next-token pre-training: SFT, preference optimisation, RL, distillation, safety training. A pre-trained model continues text; it does not reliably follow instructions, stop, refuse harmful requests or reason at length. Post-training changes behaviour more than knowledge (Chapter 1).

**Q2. Describe a typical 2026 post-training pipeline for a reasoning chat model.** SFT on curated instruction and long chain-of-thought data (often distilled), then RL with verifiable rewards on maths, code and tool use, then preference optimisation or RLHF for helpfulness and style with safety data mixed in, then evaluation and red-teaming; several rounds, with rejection sampling producing new SFT data between rounds (Chapters 1, 26, 30).

**Q3. What does SFT teach that pre-training did not?** The response format (chat template, turns, stopping), the assistant persona, the task distribution, and which of the many behaviours learned in pre-training to express. The "superficial alignment" hypothesis says it mostly selects formats; it is only partly true (reasoning SFT teaches new long-form skills).

**Q4. Why do we need preference optimisation after SFT?** SFT imitates demonstrations, so it cannot learn from comparisons, cannot exceed the demonstrator, and does not learn what *not* to do. Preferences give a relative signal that is easier to collect than ideal answers and penalise bad behaviour directly.

**Q5. What is the difference between RLHF and RLVR?** RLHF optimises a learned reward model trained on human preferences; RLVR optimises a programmatic verifier (unit tests, exact answer match). RLVR's reward is harder to hack and cheaper, but only exists where correctness can be checked (Chapters 20, 26).

**Q6. When would you choose DPO over PPO-based RLHF?** When you have a fixed preference dataset, limited compute and engineering time, and the task is style or helpfulness. Choose online RL when the reward is verifiable, when the policy must explore beyond the data, or when you need to fix errors the dataset does not contain (Chapters 22, 24).

**Q7. What is a chat template, and what goes wrong with it?** The rule for turning a list of messages into one token sequence with role markers and end-of-turn tokens. Mismatches between training and inference templates, missing end-of-turn tokens in the loss, double BOS tokens, and silently added system prompts are common bugs (Chapter 6).

**Q8. Why mask the prompt in SFT?** To train only $p(\text{response} \mid \text{prompt})$: loss on prompts teaches the model to imitate users and lets long prompts dominate the gradient (Chapter 9).

**Q9. What is packing, and what is its risk?** Concatenating several examples into one sequence to avoid padding waste. Without resetting positions and blocking attention across examples, tokens attend to unrelated examples (cross-contamination). Padding-free packing with per-example position ids and variable-length attention avoids it (Chapters 9, 47).

**Q10. LoRA versus full fine-tuning: when is LoRA enough?** For most SFT and RL on small to medium datasets, LoRA with enough rank on all linear layers matches full fine-tuning; RL needs very little capacity because each episode carries few bits. Full fine-tuning wins for

**Notation on this page:** $p$: a probability distribution (often the target or data)

large knowledge-heavy datasets or continued pre-training (Chapter 10).
**Q11. What does QLoRA change, and what does it cost?** The frozen base weights are stored in 4-bit NF4 and de-quantised on the fly; LoRA adapters train in 16-bit. It cuts weight memory about 4× at a throughput cost and small quality loss (Chapters 11, 46).
**Q12. What is knowledge distillation for LLMs, and which kinds exist?** Training a student to imitate a teacher: sequence-level (SFT on teacher samples), token-level (match the teacher's next-token distributions on fixed text) and on-policy (match the teacher on the student's own samples) (Chapter 12).
**Q13. Why is on-policy distillation effective?** It trains on the states the student actually reaches, including its mistakes, so errors do not compound (no exposure bias), and the teacher gives a dense per-token signal, unlike a sparse RL reward.

## 56.3 Conceptual questions: reinforcement learning

**Interview check**

**Q14. Map LLM generation onto an MDP.** State: prompt plus tokens so far; action: next token; transition: append the token (deterministic); reward: usually only at the end of the sequence; episode ends at end-of-sequence or a length limit. Multi-turn agents add environment responses to the transition (Chapter 13).
**Q15. Why is RL for LLMs often treated as a bandit?** With a single terminal reward and deterministic transitions, the whole response can be viewed as one action; GRPO and RLOO estimate one advantage per response, which matches this view.
**Q16. What is the difference between on-policy and off-policy learning?** On-policy methods learn from samples of the current policy (REINFORCE, PPO within its clip region); off-policy methods learn from samples of another policy (Q-learning, DPO on a fixed dataset), needing importance weights or a different objective (Chapter 18).
**Q17. Why do we need a baseline in policy gradients?** Subtracting a baseline that does not depend on the action leaves the gradient unbiased and reduces its variance; the best simple baseline is near the expected reward (Chapter 15).
**Q18. What does the critic do in PPO, and why do GRPO and RLOO drop it?** It estimates $V(s)$ to compute advantages per token with GAE. For LLMs it costs a model-sized network and is hard to train with sparse terminal rewards; group sampling gives a good baseline from other responses to the same prompt instead (Chapters 16, 27).
**Q19. What does PPO's clipping do?** It removes the incentive to move a token's probability ratio outside $[1-\epsilon, 1+\epsilon]$ in the direction that improves the objective, approximating a trust region so several gradient steps can reuse one batch (Chapter 17).
**Q20. What is GAE, and what do $\gamma$ and $\lambda$ control?** Generalised advantage estimation averages $n$-step advantage estimates with weights $\lambda^n$; $\lambda$ trades bias (low $\lambda$ trusts the critic) for variance (high $\lambda$ trusts returns); $\gamma$ discounts future rewards. LLM RL usually sets $\gamma = 1$ (Chapter 16).
**Q21. Why is there a KL penalty in RLHF?** To keep the policy close to the reference model, which limits reward hacking and preserves fluency and capabilities; it also defines the

**Notation on this page:** $V(s)$: state value (expected future return from $s$) • $s$: a state • $V$: vocabulary size • $\epsilon$: clip range for the probability ratio • $\gamma$: discount factor (0 to 1) • $\lambda$: GAE parameter (0 to 1), trades bias for variance • $n$: number of samples

closed-form optimal policy that DPO uses (Chapter 20).

**Q22. Many 2025–26 RLVR recipes set the KL coefficient to zero. Why?** With a verifier the reward is hard to hack, and the KL term slows the long-distance policy changes that reasoning RL needs; clipping and a small learning rate provide enough stability. The risk is entropy collapse and drift on untrained skills (Chapters 27, 28).

**Q23. What is entropy collapse, and how do you fight it?** The policy's token entropy falls quickly, exploration stops, and reward plateaus. Remedies: higher sampling temperature, asymmetric clipping (clip-higher), entropy bonuses with care, removing KL-to-reference only where safe, and training on prompts of the right difficulty (Chapter 28).

**Q24. What is the difference between reward hacking and over-optimisation?** Over-optimisation is the general effect of optimising an imperfect proxy past the point where it tracks the true goal; reward hacking is a specific exploit the policy finds (padding, format tricks, test tampering) (Chapter 21).

**Q25. What is a process reward model, and when is it worth it?** A model that scores each reasoning step rather than only the outcome. It gives denser credit and can guide search, but step labels are expensive and PRMs can be hacked; outcome rewards with many samples have mostly won for RL training, while PRMs remain useful for search and verification (Chapter 29).

**Q26. What does "RL sharpens rather than teaches" mean, and is it true?** Evidence that RLVR raises pass@1 but not pass@k for large $k$ suggests it reweights solutions the base model could already produce. It is partly true for short runs; long runs, curriculum and better exploration show some genuinely new capability (Chapter 30).

**Q27. Why does response length often grow during reasoning RL?** Longer reasoning helps on hard problems and the reward is per sequence; some loss normalisations also bias length (per-sequence averaging under-penalises long wrong answers). Length growth is useful up to a point, then becomes waste (Chapters 27, 28).

**Q28. What is the "length bias" in GRPO's original normalisation, and how was it fixed?** Averaging the loss per sequence before averaging over sequences gives each token of a long response less weight, so long wrong answers are penalised less. Dr. GRPO and DAPO normalise by total tokens (or a constant) instead (Chapter 28).

## 56.4 Conceptual questions: rewards, preferences and data

**Interview check**

**Q29. How is a reward model trained?** A language model with a scalar head, trained on chosen/rejected pairs with the Bradley–Terry loss $-\log \sigma(r_w - r_l)$ (Chapter 19).

**Q30. Why is a reward model's absolute value meaningless?** The Bradley–Terry loss depends only on reward differences, so any constant can be added; centring or normalising rewards fixes the scale for RL (Chapters 19, 48).

**Q31. What makes a good preference dataset?** Prompts that match the deployment distribution, responses from models close to the policy being trained (on-policy), clear labelling guidelines, measured agreement, balanced lengths, and deduplication against evaluations

**Notation on this page:** $k$: an index or count (see text) • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • $r(x, y)$: the reward for response $y$ to prompt $x$

(Chapter 39).

**Q32. What is RLAIF and Constitutional AI?** Replacing human preference labels with labels from an AI judge guided by written principles; Constitutional AI adds a self-critique and revision stage for SFT data, then RL from AI feedback (Chapter 25).

**Q33. What biases do LLM judges have?** Position bias (preferring the first or second answer), length bias, self-preference for their own style, and sensitivity to formatting. Mitigations: swap positions, length-controlled metrics, multiple judges, rubric-based prompts and calibration against humans (Chapter 37).

**Q34. What is a generative reward model?** A judge that writes reasoning before giving a verdict (or a score), instead of a scalar head. It generalises better and can be trained with RL itself, at higher inference cost (Chapter 19).

**Q35. How would you design a reward for an open-ended task such as writing?** Rubric-based judges with specific criteria, pairwise comparisons against a reference, penalties for length and known hacks, an ensemble to reduce variance, and regular human checks of top-reward samples (Chapter 36).

**Q36. Why does data quality matter more than quantity in SFT?** Small, diverse, high-quality sets (thousands of examples) often match much larger noisy sets for instruction following, because SFT mostly sets format and behaviour; noisy data teaches errors and inconsistent styles (Chapter 8).

**Q37. What is rejection sampling fine-tuning?** Sample several responses per prompt, keep those that pass a verifier or score highest under a reward model, and SFT on them; it is a simple, stable form of policy improvement (Chapters 8, 26).

**Q38. How do you detect benchmark contamination?** N-gram overlap with training data, canary strings, perplexity comparisons between test items and paraphrases, and performance gaps between old and new versions of a benchmark (Chapter 39, Project 10).

## 56.5 Conceptual questions: environments, agents and systems

**Interview check**

**Q39. What makes an RL environment for LLMs good?** Verifiable rewards that cannot be gamed, a difficulty range matched to the model, determinism or seeded randomness, fast resets, isolation (sandboxing), and logging of full trajectories (Chapters 33, 36).

**Q40. What is the difference between terminated and truncated in Gymnasium?** Terminated means the episode reached a terminal state (bootstrapped value is zero); truncated means a time or length limit stopped it (the value should still be bootstrapped). Confusing them biases value estimates (Chapter 32).

**Q41. How do you mask tokens in multi-turn RL?** Train only on tokens the policy generated; environment and tool outputs are in the context but excluded from the loss and from the KL (Chapter 34).

**Q42. Why do agentic RL runs need sandboxes?** The policy runs code and tools millions of times and will find any exploit that increases reward, including escaping or tampering with tests; isolation protects the infrastructure and the reward's integrity (Chapter 35).

**Q43. What is curriculum learning in RL post-training?** Ordering or weighting tasks by difficulty so most groups have mixed outcomes (some successes, some failures), giving useful gradients; often done by tracking per-task pass rates (Chapter 36).
**Q44. Where does time go in an RL step for reasoning models?** Mostly generation (often 60–80% of the step), because decoding is memory-bound and long responses dominate; then training forward/backward and reference/old log-probability passes (Chapter 44).
**Q45. Why is vLLM used for rollouts instead of Hugging Face generate?** Continuous batching, paged KV cache and optimised kernels give many times higher throughput (Chapter 43).
**Q46. What is the "logprob mismatch" problem?** The inference engine and the trainer compute slightly different probabilities for the same tokens (kernels, precision, sampling settings), so the on-policy ratio is not exactly one; at scale this destabilises training. Fixes: importance-sampling corrections, matched precision, deterministic kernels (Chapter 43).
**Q47. Colocated versus disaggregated rollout and training: trade-offs?** Colocated shares GPUs and avoids idle time with fast weight sync, but needs memory juggling and cannot overlap generation with training. Disaggregated uses separate GPUs, allows asynchrony and different parallelism, but needs weight transfer and can leave GPUs idle (Chapter 44).
**Q48. What does asynchronous RL buy, and what does it cost?** Overlapping generation with training raises throughput; samples become slightly off-policy (stale), needing corrections or staleness limits (Chapter 44, Project 9).
**Q49. What are ZeRO stages 1, 2 and 3?** Sharding optimizer states (1), plus gradients (2), plus parameters (3) across data-parallel GPUs; each step saves memory and adds communication (Chapters 42, 53).
**Q50. Tensor parallelism versus pipeline parallelism?** Tensor parallelism splits each matrix across GPUs and needs fast interconnect (within a node); pipeline parallelism splits layers into stages, communicates less but has pipeline bubbles (Chapter 42).

## 56.6 Conceptual questions: evaluation and safety

**Interview check**

**Q51. How would you evaluate a post-trained chat model?** Capability benchmarks (maths, code, knowledge), instruction-following checks, judge-based win-rates with length control, safety and refusal sets (both harmful and over-refusal), multilingual and long-context checks, and human review, with confidence intervals and contamination checks (Chapter 37).
**Q52. What is pass@k and why use the unbiased estimator?** The probability that at least one of $k$ samples is correct. Estimating it from exactly $k$ samples is very noisy; drawing $n > k$ samples and using $1 - \binom{n-c}{k} / \binom{n}{k}$ uses all of them (Chapter 37).
**Q53. What is over-refusal and how do you measure it?** Refusing harmless requests that look superficially risky; measured with sets of benign prompts containing trigger words, tracked alongside harmful-request refusal (Chapter 38).

**Notation on this page:** $k$: an index or count (see text) • $n$: number of samples

**Q54. What is the "alignment tax"?** Capability loss caused by alignment training; mitigated by mixing pre-training or capability data, KL regularisation and model merging (Chapter 38).
**Q55. How do you red-team a model?** Combine human experts, automated attack generation (other models searching for jailbreaks), known attack templates, and multi-turn attacks; measure attack success rate and feed failures back as training data (Chapter 38).

## 56.7 Mathematical questions

**Interview check**

**Q56. Derive the gradient of the log-softmax with respect to the logits.** $\log p_i = z_i - \log \sum_j e^{z_j}$, so $\partial \log p_i / \partial z_k = \mathbb{1}[i = k] - p_k$. For cross-entropy with target $c$ the gradient is $p - e_c$ (Chapters 2, 9).
**Q57. Why is cross-entropy loss the same as maximum likelihood?** Minimising $-\frac{1}{N}\sum \log p_\theta(y_i)$ maximises the product of probabilities of the data; the expected value equals $H(p_{\text{data}}, p_\theta) = H(p_{\text{data}}) + D_{\text{KL}}(p_{\text{data}} \| p_\theta)$, so it also minimises forward KL (Chapter 3).
**Q58. Show that KL divergence is non-negative.** By Jensen's inequality, $-D_{\text{KL}}(p\|q) = \mathbb{E}_p[\log(q/p)] \le \log \mathbb{E}_p[q/p] = \log 1 = 0$, with equality only when $p = q$ (Chapter 3).
**Q59. Forward versus reverse KL: which is mode-seeking and why?** Reverse $D_{\text{KL}}(q\|p) = \mathbb{E}_q[\log q - \log p]$ is infinite where $q > 0$ and $p = 0$, so $q$ avoids regions $p$ does not cover and settles on one mode. Forward KL is infinite where $p > 0$ and $q = 0$, so $q$ covers all modes (Chapter 3, Project 4).
**Q60. Derive the policy gradient theorem for a single-step bandit.** $\nabla_\theta \mathbb{E}_{a\sim\pi_\theta}[R(a)] = \sum_a R(a)\nabla_\theta \pi_\theta(a) = \sum_a R(a)\pi_\theta(a)\nabla_\theta \log \pi_\theta(a) = \mathbb{E}[R(a)\nabla_\theta \log \pi_\theta(a)]$ (the log-derivative trick) (Chapter 15).
**Q61. Prove that a constant baseline does not bias the policy gradient.** $\mathbb{E}_{a\sim\pi}[b\nabla \log \pi(a)] = b\sum_a \nabla\pi(a) = b\nabla \sum_a \pi(a) = b\nabla 1 = 0$.
**Q62. Write the PPO clipped objective and explain each part.** $\mathcal{L} = -\mathbb{E}_t[\min(\rho_t \hat{A}_t,\ \text{clip}(\rho_t, 1-\epsilon, 1+\epsilon)\hat{A}_t)]$ with $\rho_t = \pi_\theta(a_t|s_t)/\pi_{\theta_{\text{old}}}(a_t|s_t)$. The min takes the pessimistic bound, so the objective gains nothing from moving $\rho$ outside the clip range in the improving direction (Chapter 17).
**Q63. When is the PPO gradient zero for a token?** When $\hat{A} > 0$ and $\rho > 1 + \epsilon$, or $\hat{A} < 0$ and $\rho < 1 - \epsilon$; then the clipped branch is active and does not depend on $\theta$.
**Q64. Write the GRPO advantage and loss.** For $G$ responses to one prompt with rewards $r_i$: $\hat{A}_i = (r_i - \text{mean}(r))/\text{std}(r)$, the same for every token of response $i$; the loss is PPO's clipped surrogate with this advantage, averaged over tokens (with a normalisation that depends on the variant), optionally plus $\beta$ times a KL estimate to the reference (Chapter 27).
**Q65. What is the RLOO baseline and why is it unbiased?** For response $i$, the baseline is the mean reward of the other $G - 1$ responses; it does not depend on response $i$'s action, so the gradient stays unbiased. The leave-one-out advantage equals $\frac{G}{G-1}(r_i - \bar{r})$ (Chapter 27).
**Q66. Is GRPO's standard-deviation scaling unbiased?** No: dividing by the group's standard deviation (which depends on the samples) reweights prompts, giving more weight to

**Notation on this page:** $z$: logits (raw scores before softmax) • $p$: a probability distribution (often the target or data) • $\mathbb{1}[\cdot]$: 1 if the condition is true, else 0 • $k$: an index or count (see text) • $\theta$: the model's trainable weights • $y$: the response (output tokens); $y_t$ is token $t$ • $N$: number of examples • $D_{\text{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $H$: entropy (uncertainty) of a distribution • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $q$: a second distribution (often the model or proposal) • *and 15 more: see the Notation table*

prompts with low variance (very easy or very hard). Dr. GRPO removes it (Chapter 28).

**Q67. Derive the optimal policy of the KL-regularised objective.** Maximise $\mathbb{E}_{y\sim\pi}[r(x,y)] - \beta D_{\mathrm{KL}}(\pi\|\pi_{\mathrm{ref}})$. Rewriting as $-\beta D_{\mathrm{KL}}(\pi\|\pi_{\mathrm{ref}}e^{r/\beta}/Z) + \beta\log Z$ shows the maximiser is $\pi^*(y|x) = \pi_{\mathrm{ref}}(y|x)e^{r(x,y)/\beta}/Z(x)$ (Chapter 22).

**Q68. Derive the DPO loss from that optimal policy.** Invert: $r(x,y) = \beta\log\frac{\pi^*(y|x)}{\pi_{\mathrm{ref}}(y|x)} + \beta\log Z(x)$. Substitute into Bradley–Terry; $Z$ cancels in the difference, giving $-\log\sigma\left(\beta\log\frac{\pi_\theta(y_w)}{\pi_{\mathrm{ref}}(y_w)} - \beta\log\frac{\pi_\theta(y_l)}{\pi_{\mathrm{ref}}(y_l)}\right)$ (Chapter 22).

**Q69. What is the gradient of the DPO loss, and what does it mean?** $-\beta\,\sigma(-u)\,[\nabla\log\pi_\theta(y_w) - \nabla\log\pi_\theta(y_l)]$ with $u$ the margin: raise the chosen, lower the rejected, weighted by how wrong the implicit reward currently is.

**Q70. Why does IPO use a squared loss?** With deterministic preferences, DPO's optimum pushes the margin to infinity (overfitting); IPO regresses the margin to a finite target $1/(2\beta)$, so it stops (Chapter 23).

**Q71. What is the Bradley–Terry model?** $P(y_1 \succ y_2) = \sigma(r(y_1) - r(y_2))$: each item has a latent score and the preference probability is logistic in the difference (Chapter 19).

**Q72. Give the three common KL estimators used in RL for LLMs.** With $\rho = \pi_{\mathrm{ref}}/\pi_\theta$ for a sampled token: $k_1 = -\log\rho$ (unbiased, high variance, can be negative), $k_2 = \frac{1}{2}(\log\rho)^2$ (biased, low variance), $k_3 = \rho - 1 - \log\rho$ (unbiased for $D_{\mathrm{KL}}(\pi_\theta\|\pi_{\mathrm{ref}})$, always non-negative). They come from Schulman's note *Approximating KL Divergence* (Chapters 3, 20).

**Q73. Is $k_3$ used as a loss term an unbiased gradient of KL?** No. Differentiating $k_3$ through the formula gives $(1-\rho)\nabla\log\pi_\theta$, whose expectation under $\pi_\theta$ is $-\sum_y \pi_{\mathrm{ref}}(y)\nabla\log\pi_\theta(y)$: the gradient of the *forward* KL $D_{\mathrm{KL}}(\pi_{\mathrm{ref}}\|\pi_\theta)$, not the reverse KL it estimates. $k_2$ in the loss gives the reverse-KL gradient. Tang and Munos (arXiv 2506.09477) analyse this; the weight $1-\rho$ is also unbounded when the policy gives a token much less probability than the reference (Chapter 3).

**Q74. Show that the per-token KL of a sequence adds up to the sequence KL.** For autoregressive models, $\log\frac{\pi(y)}{\pi_{\mathrm{ref}}(y)} = \sum_t \log\frac{\pi(y_t|y_{<t})}{\pi_{\mathrm{ref}}(y_t|y_{<t})}$; taking expectations under $\pi$ gives the sequence KL as an expected sum of per-token terms (chain rule of KL).

**Q75. Write the GAE recursion.** $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$, $\hat{A}_t = \delta_t + \gamma\lambda\hat{A}_{t+1}$, computed backwards from the end of the episode (Chapter 16).

**Q76. What are the limits of GAE when $\lambda = 0$ and $\lambda = 1$?** $\lambda = 0$: $\hat{A}_t = \delta_t$ (one-step TD, low variance, biased by the critic). $\lambda = 1$: $\hat{A}_t = \sum_k \gamma^k r_{t+k} - V(s_t)$ (Monte Carlo return minus baseline, unbiased, high variance).

**Q77. What is the Bellman equation for $V^\pi$?** $V^\pi(s) = \mathbb{E}_{a\sim\pi}\left[r(s,a) + \gamma\,\mathbb{E}_{s'}[V^\pi(s')]\right]$ (Chapter 14).

**Q78. Why does Adam need bias correction?** Its moment estimates start at zero, so early averages are biased toward zero; dividing by $1-\beta_1^t$ and $1-\beta_2^t$ corrects them (Chapter 4).

**Q79. How many bytes per parameter does mixed-precision Adam training need?** About 16: 2 (bf16 weights) + 2 (bf16 gradients) + 4 (fp32 master weights) + 4 + 4 (Adam moments). A 7B model needs about 112 GB before activations (Chapter 40).

**Q80. How many parameters does a LoRA adapter add to a $d_{out} \times d_{in}$ matrix at rank $r$?** $r(d_{in} + d_{out})$; for 4096 × 4096 at $r = 16$, 131,072 (Chapter 10).

**Notation on this page:** $\pi_{\mathrm{ref}}$: the frozen reference model • $\mathbb{E}$: expected value (average over the distribution in the subscript) • $D_{\mathrm{KL}}(p\|q)$: KL divergence of $q$ from $p$ • $\beta$: strength of the KL penalty (how close to $\pi_{\mathrm{ref}}$ the policy must stay) • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $r(x,y)$: the reward for response $y$ to prompt $x$ • $\pi^*$: the optimal policy • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\sigma(z)$: the logistic (sigmoid) function $1/(1+e^{-z})$ • *and 16 more: see the Notation table*

**Q81. Why is LoRA's $B$ initialised to zero?** So the adapted model equals the base model at the start; with $B = 0$ the gradient of $A$ is zero at the first step, so $B$ learns first (Chapter 10).
**Q82. What is the scale factor in LoRA and why does rsLoRA change it?** The update is $\frac{\alpha}{r}BA$; with $\alpha/r$ the update magnitude shrinks as rank grows, so high ranks learn slowly. rsLoRA uses $\alpha/\sqrt{r}$ to keep it stable across ranks (Chapter 10).
**Q83. What is the arithmetic intensity of decoding, and why does it matter?** FLOPs per byte moved. Decoding one token per sequence does about 2 FLOPs per parameter per token while reading every parameter, so intensity is about the batch size; at small batches the GPU is memory-bandwidth bound, which is why batching more sequences increases throughput almost for free (Chapter 43).
**Q84. How large is the KV cache for one sequence?** $2\times n_{\text{layers}}\times n_{\text{kv heads}}\times d_{\text{head}}\times$seq len$\times$bytes. For 32 layers, 8 KV heads of 128, 8k tokens in bf16: about 1.07 GB (Chapter 43).
**Q85. What does temperature do mathematically?** Divides the logits by $T$ before the softmax: $p_i \propto e^{z_i/T}$. $T \to 0$ is greedy, $T = 1$ the model's distribution, $T > 1$ flatter. Log-probabilities used for RL must be computed at the same temperature as sampling (Chapter 7).
**Q86. Why does entropy fall during policy-gradient RL?** For a softmax policy updated with natural-gradient-like steps, the change in entropy at a state is approximately $-\text{Cov}_{a\sim\pi}(\log\pi(a), A(a))$ (*The Entropy Mechanism of Reinforcement Learning for Reasoning Language Models*, arXiv 2505.22617). Raising already-likely tokens that had positive advantage (the common case) makes the covariance positive, so entropy drops; that paper's Clip-Cov and KL-Cov limit updates on the highest-covariance tokens (Chapter 28).
**Q87. Derive the expected KL of best-of-$n$ sampling.** Under the standard analysis, choosing the best of $n$ i.i.d. samples gives $\text{KL}(\pi_{\text{BoN}}\|\pi) = \log n - \frac{n-1}{n}$ (an upper bound in general; exact for continuous rewards without ties) (Chapter 21).
**Q88. Why does importance sampling increase variance?** The weights $\pi/\mu$ can be large where $\mu$ is small; the variance of the weighted estimate grows with $\mathbb{E}_\mu[(\pi/\mu)^2]$, which explodes when the policies differ. Clipping or truncating weights trades bias for variance (Chapter 18).
**Q89. What is the Elo or Bradley–Terry rating used in Chatbot-Arena-style leaderboards?** A Bradley–Terry model fitted to pairwise votes: each model gets a score and win probabilities are logistic in score differences; confidence intervals come from bootstrapping the votes (Chapter 37).
**Q90. What is the bias of the per-sequence mean in GRPO's loss with respect to length?** With per-sequence averaging, each token of a length-$L$ response has weight $1/L$; for a negative advantage, longer wrong answers receive less penalty per token, which encourages length on failed attempts (Chapter 28).
**Q91. For a 7B model, how many training FLOPs does one token cost?** About $6N$: $2N$ forward and $4N$ backward, so about $4.2\times10^{10}$ FLOPs per token; with a reference pass add $2N$ (Chapter 40).
**Q92. Why does the Bradley–Terry loss saturate, and what does that imply?** $-\log\sigma(u)$ has gradient $-\sigma(-u)$, which vanishes as the margin grows; well-separated pairs stop contributing, so training focuses on hard or mislabelled pairs (a reason label noise hurts).

**Notation on this page:** $B$: batch size • $\alpha$: a weighting coefficient (defined in the text) • $r(x, y)$: the reward for response $y$ to prompt $x$ • $d$: hidden size of the model • $n$: number of samples • $T$: number of steps or tokens in a sequence • $z$: logits (raw scores before softmax) • $p$: a probability distribution (often the target or data) • $\pi_\theta$: the policy (the model), with weights $\theta$ • $a$: an action • $D_{\text{KL}}(p\|q)$: KL divergence of $q$ from $p$ • *and 5 more: see the Notation table*

**Q93. What is the variance of a Bernoulli reward, and how does it affect GRPO?** $p(1-p)$, maximal at $p = 0.5$. Groups with $p$ near 0 or 1 often have zero variance and no signal; that is why difficulty filtering matters (Chapter 27).

**Q94. Explain the reparameterisation of the DPO implicit reward as a token-level quantity.** $\beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)} = \sum_t \beta \log \frac{\pi_\theta(y_t|x,y_{<t})}{\pi_{\text{ref}}(y_t|x,y_{<t})}$, so DPO implicitly assigns a per-token reward; this is why DPO models can serve as dense token-level reward signals (Chapter 22).

**Q95. Why does dividing the loss by the number of tokens in the batch, rather than per sequence, matter with gradient accumulation?** If each micro-batch is normalised by its own token count and then averaged, micro-batches with few tokens are over-weighted; the correct global mean divides the summed loss by the total token count across all accumulated micro-batches (Chapters 9, 45).

**Key takeaways**

- Answer in three layers: claim, mechanism, judgement.
- Know cold: log-softmax gradient, KL facts, policy-gradient and baseline proof, PPO clip, GAE, GRPO/RLOO advantages, DPO derivation, KL estimators, memory per parameter, KV-cache size.

**Notation on this page:** $p$: a probability distribution (often the target or data) • $\pi_{\text{ref}}$: the frozen reference model • $\pi_\theta$: the policy (the model), with weights $\theta$ • $\theta$: the model's trainable weights • $\beta$: strength of the KL penalty (how close to $\pi_{\text{ref}}$ the policy must stay) • $y_{<t}$: the tokens before position $t$ • $x$: the prompt (input) • $y$: the response (output tokens); $y_t$ is token $t$ • $t$: time step or token position

CHAPTER 57

# Interview Questions II: Coding, System Design and Behaviour

**How to use this chapter.**
Coding rounds for post-training roles ask you to write the core pieces from memory in PyTorch: a masked loss, sequence log-probabilities, sampling, attention with a KV cache, LoRA, and the RL and preference losses. Each coding question below (C1–C30) has a tested solution that checks itself against a reference. Write your own version first, then compare. The system design questions (S1–S20) and behavioural questions (B1–B15) follow.

**Why this, not that? Why write every solution with a self-check, not just the function?**

**Reason.** Interviewers watch how you verify code. Comparing against a library function, a brute-force version or a known identity (for example "pass@1 equals c/n") catches the sign, shift and masking errors that are the most common bugs in this field. **Trade-off.** The check takes a few minutes of a 45-minute interview. **When the alternative is better.** When the interviewer explicitly asks only for pseudocode or for the idea; then say how you would test it in one sentence.

## 57.1 Coding questions

The blocks share one namespace and run in order.

**C1. Write a numerically stable log-softmax.** Subtract the maximum before exponentiating so that `exp` never overflows.

```
import torch, math
import torch.nn.functional as F
torch.manual_seed(0)
def log_softmax(z, dim=-1):
    m = z.max(dim, keepdim=True).values
    return z - m - torch.log(torch.exp(z - m).sum(dim, keepdim=True))
z = torch.randn(4, 50) * 30 + 1000                    # huge logits: naive exp would
 ↪  overflow
print("C1 matches F.log_softmax:", torch.allclose(log_softmax(z), F.log_softmax(z, -1),
 ↪  atol=1e-4))
```

**C2. Cross-entropy with an ignore index, averaged over non-ignored tokens.**

```
def masked_ce(logits, labels, ignore_index=-100):
    lp = log_softmax(logits)
    mask = labels != ignore_index
    picked = lp.gather(-1, labels.clamp_min(0).unsqueeze(-1)).squeeze(-1)
    return -(picked * mask).sum() / mask.sum()
logits = torch.randn(2, 7, 11); labels = torch.randint(0, 11, (2, 7)); labels[0, :3] =
↪  -100
print("C2 matches F.cross_entropy:", torch.allclose(masked_ce(logits, labels),
      F.cross_entropy(logits.view(-1, 11), labels.view(-1), ignore_index=-100)))
```

**C3. Per-sequence log-probability of a completion from causal-LM logits.** Logits at position $t$ predict token $t+1$, so shift by one; sum only over completion tokens.

```
def sequence_logprob(logits, input_ids, completion_mask):
    """logits [B,T,V] from a causal LM on input_ids [B,T]; completion_mask [B,T] marks
    ↪  completion tokens."""
    lp = log_softmax(logits[:, :-1])                          # predicts tokens 1..T-1
    tok = input_ids[:, 1:]
    per_tok = lp.gather(-1, tok.unsqueeze(-1)).squeeze(-1)
    return (per_tok * completion_mask[:, 1:]).sum(-1)
B, T, V = 2, 6, 9
ids = torch.randint(0, V, (B, T)); lg = torch.randn(B, T, V)
cm = torch.tensor([[0, 0, 0, 1, 1, 1], [0, 0, 1, 1, 1, 0]])
brute = torch.stack([sum(F.log_softmax(lg[b, t - 1], -1)[ids[b, t]] for t in range(1, T)
↪  if cm[b, t])
                     for b in range(B)])
print("C3 matches brute force:", torch.allclose(sequence_logprob(lg, ids, cm), brute))
```

**C4. Sample with temperature, top-k and top-p (nucleus).** Apply temperature, then top-k, then keep the smallest set of tokens whose probability reaches $p$ (always keeping the most likely token).

```
def sample_next(logits, temperature=1.0, top_k=0, top_p=1.0, generator=None):
    if temperature == 0: return logits.argmax(-1)
    z = logits / temperature
    if top_k > 0:
        kth = z.topk(top_k, -1).values[..., -1, None]
        z = z.masked_fill(z < kth, float("-inf"))
    if top_p < 1.0:
        sz, idx = z.sort(-1, descending=True)
        cum = sz.softmax(-1).cumsum(-1)
        remove = cum - sz.softmax(-1) >= top_p                # mass before this token
↪  already >= p
        sz = sz.masked_fill(remove, float("-inf"))
        z = torch.full_like(z, float("-inf")).scatter(-1, idx, sz)
    return torch.multinomial(z.softmax(-1), 1, generator=generator).squeeze(-1)
lg = torch.log(torch.tensor([[0.5, 0.3, 0.15, 0.05]]))
g = torch.Generator().manual_seed(0)
draws = torch.stack([sample_next(lg, top_p=0.8, generator=g) for _ in
↪  range(4000)]).flatten()
```

```
print("C4 top-p=0.8 keeps tokens", sorted(set(draws.tolist())), "freq of token 0: %.2f
↪  (expect 0.5/0.8=0.62)"
      % (draws == 0).float().mean())
```

### C5. Causal multi-head attention, checked against PyTorch's fused version.

```
def mha(x, Wq, Wk, Wv, Wo, n_heads):
    B, T, D = x.shape; h = D // n_heads
    q, k, v = [(x @ W).view(B, T, n_heads, h).transpose(1, 2) for W in (Wq, Wk, Wv)]
    att = (q @ k.transpose(-1, -2)) / math.sqrt(h)
    att = att.masked_fill(torch.triu(torch.ones(T, T, dtype=torch.bool), 1),
↪  float("-inf")).softmax(-1)
    return (att @ v).transpose(1, 2).reshape(B, T, D) @ Wo
D, H = 32, 4
Ws = [torch.randn(D, D) / math.sqrt(D) for _ in range(4)]
x = torch.randn(2, 10, D)
q, k, v = [(x @ W).view(2, 10, H, D // H).transpose(1, 2) for W in Ws[:3]]
ref = F.scaled_dot_product_attention(q, k, v, is_causal=True).transpose(1, 2).reshape(2,
↪  10, D) @ Ws[3]
print("C5 matches SDPA:", torch.allclose(mha(x, *Ws, H), ref, atol=1e-5))
```

### C6. Add a KV cache and show incremental decoding equals the full forward pass.

```
def mha_step(x_t, cache, Wq, Wk, Wv, Wo, n_heads):
    """x_t: [B,1,D]; cache: dict with 'k','v' of shape [B,H,t,h] (or empty)."""
    B, _, D = x_t.shape; h = D // n_heads
    q, k, v = [(x_t @ W).view(B, 1, n_heads, h).transpose(1, 2) for W in (Wq, Wk, Wv)]
    cache["k"] = torch.cat([cache["k"], k], 2) if "k" in cache else k
    cache["v"] = torch.cat([cache["v"], v], 2) if "v" in cache else v
    att = (q @ cache["k"].transpose(-1, -2) / math.sqrt(h)).softmax(-1)   # no mask
↪  needed: only past keys
    return (att @ cache["v"]).transpose(1, 2).reshape(B, 1, D) @ Wo
cache = {}
steps = torch.cat([mha_step(x[:, t:t + 1], cache, *Ws, H) for t in range(10)], 1)
print("C6 cached decode matches full:", torch.allclose(steps, mha(x, *Ws, H),
↪  atol=1e-5),
      "| cache shape", tuple(cache["k"].shape))
```

### C7. RMSNorm.

```
class RMSNorm(torch.nn.Module):
    def __init__(self, d, eps=1e-6):
        super().__init__(); self.w = torch.nn.Parameter(torch.ones(d)); self.eps = eps
    def forward(self, x):
        return self.w * x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)
n = RMSNorm(32); y = n(x)
print("C7 RMS of output ~1:", round(y.pow(2).mean(-1).sqrt().mean().item(), 4),
      "| matches F.rms_norm:", torch.allclose(y, F.rms_norm(x, (32,), n.w, 1e-6),
       ↪  atol=1e-6))
```

**C8. Rotary position embedding (RoPE), and a check that attention scores depend only on relative position.**

```
def rope(x, pos, base=10000.0):
    """x [..., T, d] with d even; pos [T]. Rotates pairs (x[2i], x[2i+1]) by angle pos *
     ↪  base^(-2i/d)."""
    d = x.shape[-1]
    freqs = base ** (-torch.arange(0, d, 2, dtype=torch.float32) / d)
    ang = pos[:, None].float() * freqs[None]
    cos, sin = ang.cos(), ang.sin()
    x1, x2 = x[..., 0::2], x[..., 1::2]
    out = torch.empty_like(x)
    out[..., 0::2] = x1 * cos - x2 * sin; out[..., 1::2] = x1 * sin + x2 * cos
    return out
qv, kv = torch.randn(1, 16), torch.randn(1, 16)
s1 = (rope(qv, torch.tensor([7])) * rope(kv, torch.tensor([3]))).sum()
s2 = (rope(qv, torch.tensor([104])) * rope(kv, torch.tensor([100]))).sum()
print("C8 score at positions (7,3) equals (104,100):", torch.allclose(s1, s2,
 ↪  atol=1e-4))
```

**C9. A LoRA linear layer with merge.**

```
class LoRALinear(torch.nn.Module):
    def __init__(self, base: torch.nn.Linear, r=8, alpha=16):
        super().__init__()
        self.base = base.requires_grad_(False)
        self.A = torch.nn.Parameter(torch.randn(r, base.in_features) /
         ↪  math.sqrt(base.in_features))
        self.B = torch.nn.Parameter(torch.zeros(base.out_features, r))
        self.s = alpha / r
    def forward(self, x): return self.base(x) + self.s * (x @ self.A.T) @ self.B.T
    def merged(self):
        m = torch.nn.Linear(self.base.in_features, self.base.out_features)
        m.weight.data = self.base.weight + self.s * self.B @ self.A; m.bias.data =
 ↪  self.base.bias.clone()
        return m
lin = LoRALinear(torch.nn.Linear(32, 16)); xx = torch.randn(5, 32)
same_at_init = torch.allclose(lin(xx), lin.base(xx))
with torch.no_grad(): lin.B.normal_()
print("C9 identical at init:", same_at_init, "| merge matches:", torch.allclose(lin(xx),
 ↪  lin.merged()(xx), atol=1e-5),
      "| trainable:", sum(p.numel() for p in lin.parameters() if p.requires_grad), "=
       ↪  r(d_in+d_out) =", 8 * (32 + 16))
```

**C10. Bradley–Terry reward-model loss and accuracy.**

```
def bt_loss(r_chosen, r_rejected, margin=None):
    d = r_chosen - r_rejected - (margin if margin is not None else 0)
    return -F.logsigmoid(d).mean(), (r_chosen > r_rejected).float().mean()
rc, rr = torch.tensor([2.0, 0.5, -1.0]), torch.tensor([1.0, 1.0, -3.0])
```

```
loss, acc = bt_loss(rc, rr)
print("C10 loss %.4f (manual %.4f) acc %.2f" % (loss, -sum(math.log(1 / (1 +
 ↪  math.exp(-d))) for d in [1, -0.5, 2]) / 3, acc))
```

**C11. DPO loss from policy and reference sequence log-probabilities, with implicit rewards.**

```
def dpo_loss(pw, pl, rw, rl, beta=0.1):
    cr, rjr = beta * (pw - rw), beta * (pl - rl)              # implicit rewards
    return -F.logsigmoid(cr - rjr).mean(), cr.detach(), rjr.detach()
pw = torch.tensor([-10.0, -20.0], requires_grad=True); pl = torch.tensor([-12.0, -18.0],
 ↪  requires_grad=True)
rw, rl = torch.tensor([-11.0, -20.0]), torch.tensor([-11.0, -19.0])
loss, cr, rjr = dpo_loss(pw, pl, rw, rl); loss.backward()
print("C11 loss %.4f | margins %s | grad on chosen %s (negative = loss falls as chosen
 ↪  logprob rises)"
      % (loss.item(), [round(v, 3) for v in (cr - rjr).tolist()], [round(v, 4) for v in
       ↪  pw.grad.tolist()]))
```

**C12. GRPO: group advantages and the clipped token-level loss with a completion mask.** Use the DAPO/TRL-1.16-default normalisation: divide by the total number of completion tokens in the batch.

```
def grpo_loss(logp, old_logp, rewards, mask, G, eps_low=0.2, eps_high=0.2, scale=True):
    """logp, old_logp, mask: [B*G, T]; rewards: [B*G]."""
    r = rewards.view(-1, G)
    adv = r - r.mean(1, keepdim=True)
    if scale: adv = adv / (r.std(1, keepdim=True) + 1e-4)
    adv = adv.view(-1, 1)                                     # same advantage for every
 ↪  token
    ratio = (logp - old_logp).exp()
    obj = torch.min(ratio * adv, ratio.clamp(1 - eps_low, 1 + eps_high) * adv)
    return -(obj * mask).sum() / mask.sum()
BG, T = 8, 5
old = -torch.rand(BG, T); cur = (old + 0.05 * torch.randn(BG, T)).requires_grad_(True)
mask = (torch.arange(T)[None] < torch.randint(2, T + 1, (BG, 1))).float()
rew = torch.tensor([1., 0, 0, 1, 1, 1, 1, 1])
L = grpo_loss(cur, old, rew, mask, G=4); L.backward()
print("C12 loss %.4f | grad zero on the all-correct group: %s | grad zero on padding:
 ↪  %s" % (
    L.item(), bool(cur.grad[4:].abs().sum() == 0), bool((cur.grad * (1 -
 ↪  mask)).abs().sum() == 0)))
```

**C13. PPO clipping: show the gradient vanishes outside the clip range in the improving direction.**

```
def ppo_token_obj(ratio, adv, eps=0.2):
    return torch.min(ratio * adv, ratio.clamp(1 - eps, 1 + eps) * adv)
for r0, a in [(1.1, 1.0), (1.3, 1.0), (0.7, -1.0), (0.7, 1.0)]:
```

```
    ratio = torch.tensor(r0, requires_grad=True); ppo_token_obj(ratio,
↪   torch.tensor(a)).backward()
    print(f"C13 ratio {r0} adv {a:+.0f}: d obj/d ratio = {ratio.grad.item():+.1f}")
```

The last case ($\rho = 0.7, \hat{A} > 0$) still has a gradient: PPO only removes the incentive to move *further* in the improving direction.

**C14. Generalised advantage estimation, checked against the $\lambda = 1$ Monte Carlo limit.**

```
def gae(rewards, values, gamma=1.0, lam=0.95):
    """rewards [T], values [T+1] (last = bootstrap value, 0 if terminal)."""
    T = len(rewards); adv = torch.zeros(T); last = 0.0
    for t in reversed(range(T)):
        delta = rewards[t] + gamma * values[t + 1] - values[t]
        last = delta + gamma * lam * last; adv[t] = last
    return adv, adv + values[:-1]                              # advantages, returns
↪   (value targets)
rw = torch.tensor([0., 0., 0., 1.]); vals = torch.tensor([0.2, 0.4, 0.5, 0.7, 0.0])
a1, _ = gae(rw, vals, lam=1.0)
mc = torch.tensor([rw[t:].sum() for t in range(4)]) - vals[:-1]
print("C14 lambda=1 equals MC return - V:", torch.allclose(a1, mc), "| lambda=0.95:",
↪   [round(v, 3) for v in gae(rw, vals)[0].tolist()])
```

**C15. The three KL estimators against the exact KL.**

```
p_logits, q_logits = torch.randn(50), torch.randn(50)          # policy pi and reference
pi, ref = p_logits.softmax(0), q_logits.softmax(0)
exact = (pi * (pi.log() - ref.log())).sum()
y = torch.multinomial(pi, 200000, replacement=True)
lr = ref.log()[y] - pi.log()[y]                                # log(ref/pi) on samples
↪   from pi
k1, k2, k3 = -lr, 0.5 * lr ** 2, lr.exp() - 1 - lr
print("C15 exact %.4f | k1 %.4f (std %.2f) | k2 %.4f (biased) | k3 %.4f (std %.2f, min
↪   %.3f)" % (
    exact, k1.mean(), k1.std(), k2.mean(), k3.mean(), k3.std(), k3.min()))
```

**C16. RLOO advantages, and the identity with the scaled mean-centred reward.**

```
def rloo_adv(rewards, G):
    r = rewards.view(-1, G)
    return (r - (r.sum(1, keepdim=True) - r) / (G - 1)).view(-1)
rwd = torch.tensor([1., 0, 0, 1, 0.3, 0.9, 0.1, 0.5])
lhs = rloo_adv(rwd, 4); r4 = rwd.view(-1, 4)
rhs = (4 / 3 * (r4 - r4.mean(1, keepdim=True))).view(-1)
print("C16 RLOO = G/(G-1)(r - mean):", torch.allclose(lhs, rhs))
```

**C17. Unbiased pass@k, numerically stable for large $n$.**

**Notation on this page:** $\rho$: importance ratio $\pi_\theta/\pi_{\theta_{\text{old}}}$ for a token • $A$, $\hat{A}$: advantage (how much better an action is than average); hat = estimate • $\lambda$: GAE parameter (0 to 1), trades bias for variance • $n$: number of samples

```
def pass_at_k(n, c, k):
    if n - c < k: return 1.0
    return 1.0 - math.prod(1 - k / i for i in range(n - c + 1, n + 1))   # = 1 -
     ↪ C(n-c,k)/C(n,k)
print("C17", round(pass_at_k(16, 4, 1), 4), "(=c/n)", round(pass_at_k(16, 4, 8), 4),
      "| exact form:", round(1 - math.comb(12, 8) / math.comb(16, 8), 4), "| large n:",
       ↪ round(pass_at_k(10000, 30, 100), 4))
```

**C18. Pack variable-length examples into fixed-capacity rows (first-fit decreasing) with position ids that reset per example.**

```
def pack(lengths, capacity):
    rows = []                                                    # each row: list of example
 ↪ indices
    for i in sorted(range(len(lengths)), key=lambda i: -lengths[i]):
        for row in rows:
            if sum(lengths[j] for j in row) + lengths[i] <= capacity: row.append(i); break
        else: rows.append([i])
    pos = [[p for j in row for p in range(lengths[j])] for row in rows]
    return rows, pos
lens = [7, 3, 5, 2, 8, 1, 4]
rows, pos = pack(lens, 10)
print("C18 rows", rows, "| fill", [sum(lens[j] for j in r) for r in rows], "| positions
 ↪ of row 0", pos[0])
```

**C19. Build an assistant-only loss mask from role-tagged segments.**

```
def assistant_labels(segments):
    """segments: list of (role, token_ids). Returns input_ids and labels (-100 outside
     ↪ assistant content)."""
    ids, labels = [], []
    for role, toks in segments:
        ids += toks; labels += toks if role == "assistant" else [-100] * len(toks)
    return ids, labels
ids, labs = assistant_labels([("system", [1, 2]), ("user", [3, 4, 5]), ("assistant", [6,
 ↪ 7, 99]), ("user", [8]), ("assistant", [9, 99])])
print("C19 trained ids:", [i for i, l in zip(ids, labs) if l != -100], "(99 =
 ↪ end-of-turn, must be trained)")
```

**C20. Gradient accumulation with correct token normalisation.** Dividing each micro-batch by its own token count and averaging gives the wrong gradient when token counts differ.

```
torch.manual_seed(1)
w = torch.randn(4, requires_grad=True)
micro = [torch.randn(n, 4) for n in (3, 17)]                    # two micro-batches, 3 and
 ↪ 17 tokens
full = torch.cat(micro); (full @ w).pow(2).mean().backward(); g_ref = w.grad.clone();
 ↪ w.grad = None
total = sum(m.shape[0] for m in micro)
```

```
for m in micro: ((m @ w).pow(2).sum() / total).backward()
g_ok = w.grad.clone(); w.grad = None
for m in micro: ((m @ w).pow(2).mean() / len(micro)).backward()
print("C20 global-token normalisation matches:", torch.allclose(g_ok, g_ref),
      "| per-micro-batch mean matches:", torch.allclose(w.grad, g_ref))
```

### C21. Masked mean of per-token entropy from logits.

```
def masked_entropy(logits, mask):
    lp = log_softmax(logits); ent = -(lp.exp() * lp).sum(-1)
    return (ent * mask).sum() / mask.sum()
lg = torch.zeros(2, 3, 8); m = torch.tensor([[1., 1, 0], [1, 0, 0]])
print("C21 uniform over 8 tokens: %.4f = ln 8 = %.4f" % (masked_entropy(lg, m),
 ↪  math.log(8)))
```

### C22. Greedy decoding with stop strings and a token budget, on a toy model.

```
def generate(step_fn, prompt_ids, max_new_tokens, eos_id, decode, stop_strings=()):
    ids = list(prompt_ids); out = []
    for _ in range(max_new_tokens):
        nxt = int(step_fn(ids).argmax()); ids.append(nxt); out.append(nxt)
        if nxt == eos_id: return out, "eos"
        if any(s in decode(out) for s in stop_strings): return out, "stop_string"
    return out, "length"
vocab = list("abc</>") + ["<eos>"]
seq = [0, 1, 3, 4, 5, 2, 6]                                  # "ab</>c<eos>"
step = lambda ids: F.one_hot(torch.tensor(seq[len(ids) - 1]), len(vocab)).float()
dec = lambda toks: "".join(vocab[t] for t in toks)
for stops, budget in [((), 10), (("</>",), 10), ((), 3)]:
    toks, why = generate(step, [0], budget, 6, dec, stops)
    print(f"C22 stops={stops} budget={budget}: '{dec(toks)}' finished by {why}")
```

### C23. A repetition penalty for reward shaping: fraction of repeated n-grams.

```
def repeated_ngram_fraction(tokens, n=3):
    grams = [tuple(tokens[i:i + n]) for i in range(len(tokens) - n + 1)]
    return 0.0 if not grams else 1 - len(set(grams)) / len(grams)
print("C23", repeated_ngram_fraction("the cat sat on the mat".split()),
      round(repeated_ngram_fraction(("so so so so " * 5).split()), 3))
```

### C24. Normalise and compare numeric answers robustly.

```
from fractions import Fraction
import re
def same_number(a, b):
    def norm(s):
        s = s.strip().replace(",", "").rstrip(".").replace("\\$", "").replace("$", "")
        m = re.fullmatch(r"\\frac\{(-?\d+)\}\{(-?\d+)\}", s)
        try: return Fraction(int(m[1]), int(m[2])) if m else Fraction(s)
```

```
        except (ValueError, ZeroDivisionError): return None
    x, y = norm(a), norm(b)
    return x is not None and x == y
print("C24", [same_number(a, b) for a, b in [("1,000", "1000"), ("0.5", "\\frac{1}{2}"),
 ↪  ("3.", "3"), ("abc", "abc"), ("1/0", "1")]])
```

### C25. A conservative reward from an ensemble of reward models.

```
def conservative_reward(scores, k=1.0):
    """scores [n_models, batch] -> mean - k * std (penalises disagreement, a common
     ↪  anti-hacking trick)."""
    return scores.mean(0) - k * scores.std(0)
sc = torch.tensor([[1.0, 2.0], [1.1, -1.0], [0.9, 3.5]])
print("C25 mean", sc.mean(0).tolist(), "| conservative", [round(v, 3) for v in
 ↪  conservative_reward(sc).tolist()])
```

### C26. A minimal Gymnasium environment that passes the official checker.

```
import gymnasium as gym
from gymnasium import spaces
from gymnasium.utils.env_checker import check_env
class GuessNumber(gym.Env):
    """Guess a hidden integer in [0, 15]; observation = last hint (0 none, 1 higher, 2
     ↪  lower, 3 correct)."""
    def __init__(self, max_steps=6):
        self.observation_space, self.action_space = spaces.Discrete(4),
         ↪  spaces.Discrete(16)
        self.max_steps = max_steps
    def reset(self, seed=None, options=None):
        super().reset(seed=seed)
        self.target, self.t = int(self.np_random.integers(16)), 0
        return 0, {}
    def step(self, action):
        self.t += 1
        hint = 3 if action == self.target else (1 if action < self.target else 2)
        terminated = hint == 3; truncated = (not terminated) and self.t >= self.max_steps
        return hint, (1.0 if terminated else 0.0), terminated, truncated, {}
check_env(GuessNumber(), skip_render_check=True)
env = GuessNumber(); obs, _ = env.reset(seed=3); lo, hi, steps = 0, 15, 0
while True:
    a = (lo + hi) // 2; obs, r, term, trunc, _ = env.step(a); steps += 1
    if term or trunc: break
    lo, hi = (a + 1, hi) if obs == 1 else (lo, a - 1)
print("C26 checker passed; binary search solved it:", term, "in", steps, "steps")
```

### C27. Running reward normalisation with Welford's algorithm.

```
class RunningStats:
    def __init__(self): self.n, self.mean, self.m2 = 0, 0.0, 0.0
    def update(self, xs):
```

```
        for x in xs:
            self.n += 1; d = x - self.mean; self.mean += d / self.n; self.m2 += d * (x -
             ↪ self.mean)
    @property
    def std(self): return math.sqrt(self.m2 / max(1, self.n - 1))
data = torch.randn(1000) * 3 + 5; rs = RunningStats()
for chunk in data.split(64): rs.update(chunk.tolist())
print("C27 running mean/std %.4f %.4f vs torch %.4f %.4f" % (rs.mean, rs.std,
 ↪ data.mean(), data.std()))
```

**C28. Truncated importance sampling for the engine–trainer mismatch.** The sampler's log-probabilities differ slightly from the trainer's; weight each token by $\min(\pi_{\text{train}}/\pi_{\text{engine}}, C)$.

```
def tis_weights(train_logp, engine_logp, C=2.0):
    return (train_logp - engine_logp).exp().clamp(max=C).detach()
tl = torch.tensor([-1.0, -2.0, -0.5]); el = torch.tensor([-1.02, -3.5, -0.49])
print("C28 weights", [round(v, 3) for v in tis_weights(tl, el).tolist()], "(middle token
 ↪ truncated at C=2)")
```

**C29. Best-of-$n$ selection with a reward model, and its KL from the base policy.**

```
def best_of_n(candidates, reward_fn):
    scores = [reward_fn(c) for c in candidates]
    i = max(range(len(candidates)), key=scores.__getitem__)
    return candidates[i], math.log(len(candidates)) - (len(candidates) - 1) /
     ↪ len(candidates)
best, kl = best_of_n(["short", "a medium answer", "a much longer answer here"],
 ↪ reward_fn=lambda s: -abs(len(s) - 14))
print(f"C29 picked '{best}', KL bound {kl:.3f} nats")
```

**C30. Compute the DPO "rewards/accuracies" and "margins" metrics that TRL logs, from a batch.**

```
def dpo_metrics(pw, pl, rw, rl, beta=0.1):
    cr, rjr = beta * (pw - rw), beta * (pl - rl)
    return {"rewards/chosen": cr.mean().item(), "rewards/rejected": rjr.mean().item(),
            "rewards/accuracies": (cr > rjr).float().mean().item(), "rewards/margins":
             ↪ (cr - rjr).mean().item()}
b_pw, b_pl = torch.tensor([-10.0, -20.0, -15.0]), torch.tensor([-12.0, -18.0, -30.0])
 ↪ # policy chosen/rejected
b_rw, b_rl = torch.tensor([-11.0, -20.0, -16.0]), torch.tensor([-11.0, -19.0, -25.0])
 ↪ # reference
print("C30", {k: round(v, 3) for k, v in dpo_metrics(b_pw, b_pl, b_rw, b_rl).items()})
```

## 57.2 System design questions

**Interview check**

**S1. Design the RL post-training system for a 70B reasoning model on 256 GPUs.** Start from the step-time model: generation dominates, so disaggregate: about 60–70% of GPUs as vLLM rollout engines (TP 4–8 per engine), the rest as an FSDP or Megatron trainer. Use one-step-off asynchronous training with importance-sampling correction, a reward service with sandboxed verifiers scaled separately, NCCL weight broadcast after each step (bucketed, about 140 GB in bf16), partial rollouts for long-tail generations, and checkpointing of trainer, data position and RNG. Monitor reward, entropy, length, staleness, logprob mismatch and GPU idle time (Chapter 44).

**S2. Generation takes 80% of your step time. What do you do?** Raise rollout throughput (larger batches, prefix caching for shared prompts, FP8 KV cache, speculative decoding), cut the long tail (length limits, partial rollouts, over-sampling then dropping slow requests), move to asynchronous training, or shift GPUs from training to generation (Chapters 43, 44).

**S3. Design a reward service for code RL.** A queue-backed pool of sandboxed workers (containers or micro-VMs, no network, CPU and memory limits, timeouts) that run tests on submitted code; idempotent requests keyed by sample id; result caching; per-test timeouts; logging of stdout/stderr; protection of the test files from the solution; autoscaling on queue depth (Chapter 35).

**S4. How would you build an evaluation pipeline that runs on every checkpoint?** A job triggered on checkpoint save; a fixed, versioned suite (capabilities, instruction following, safety, over-refusal, judge win-rate with length control); vLLM serving; fixed sampling settings and seeds; confidence intervals; results to a dashboard with comparisons against baseline; contamination checks when data changes (Chapter 37).

**S5. Design a data pipeline for SFT data from multiple sources.** Ingestion with licence and source tags; normalisation to one message schema; exact and near-duplicate removal (MinHash); decontamination against all evaluation sets; quality filtering (heuristics plus a classifier or judge); PII scrubbing; mixture weighting by source and task; versioned output with statistics and samples for human review (Chapter 39).

**S6. Your RL run's reward suddenly spikes and then accuracy collapses. How do you debug it?** Look at samples around the spike for a hack; check gradient norm, clip fraction, entropy and KL; check the reward function for a bug or exploited edge case; check data (a bad shard); check the logprob mismatch metric; roll back to the last good checkpoint and rerun with the fix (Chapter 31).

**S7. How do you serve many LoRA adapters efficiently?** One base model in memory with batched multi-LoRA kernels (vLLM `enable_lora`, `max_loras`, `max_lora_rank`); adapters loaded on demand with an LRU cache in GPU and CPU memory; requests routed by adapter id; batch requests across adapters (Chapter 53).

**S8. Design an online preference-learning loop with human raters.** Sample pairs from the current policy for prompts from the live distribution; send to raters with clear guidelines and gold items for quality control; train a reward model or run online DPO periodically; evaluate before deployment; track rater agreement and drift (Chapter 24).

**S9. How would you train a model to use a web-search tool with RL?** Build an environment with a cached search index (for determinism and cost), a tool protocol, multi-turn rollouts with tool outputs masked out of the loss, rewards from answer correctness on questions with verifiable answers plus small penalties for wasted calls, and a curriculum by difficulty (Chapter 34).

**S10. Training and inference produce different log-probabilities. How do you design around it?** Request log-probabilities from the engine with the same temperature and processing; use the same precision where possible; add truncated importance-sampling or masking corrections; monitor the per-token KL between engine and trainer; for MoE models consider routing replay (Chapter 43).

**S11. Design checkpointing for a multi-day RL run.** Save the trainer state (sharded weights, optimizer, scheduler), dataloader position, RNG states, reward-model or verifier versions and the rollout buffer (or design so it can be discarded); save every N steps asynchronously; keep the last few plus milestone checkpoints; test resume regularly (Chapter 44).

**S12. How would you reduce the memory of a 7B full fine-tune to fit on 8×40 GB GPUs?** ZeRO-3 or FSDP (16 bytes/parameter divided by 8 is 14 GB per GPU), gradient checkpointing, FlashAttention, a chunked or fused cross-entropy loss, and optional CPU optimizer offload if still short (Chapters 40, 42).

**S13. Design a safety-training stage that does not increase over-refusal.** Mix harmful prompts with refusals and borderline-benign prompts with helpful answers; train preference pairs that penalise both unsafe compliance and unnecessary refusal; measure both rates on held-out sets every checkpoint; use a rule-based or rubric judge for consistency (Chapter 38).

**S14. How do you make an RL experiment reproducible enough to compare two loss variants?** Fixed seeds for data order, sampling and initialisation; the same prompts and evaluation; multiple seeds per variant; logging of configs and commits; reporting curves with variance across seeds, at matched compute and KL (Chapter 31).

**S15. Design distillation of a 400B teacher into an 8B student for maths.** Generate many teacher solutions on a broad problem set with high-throughput serving; filter by correctness; SFT the student (sequence-level); then on-policy distillation with teacher log-probabilities served by a scoring endpoint (teacher prefill only, which is cheaper than generation); evaluate pass@1 and pass@k; consider RLVR after distillation (Chapter 12).

**S16. How would you choose between TRL, OpenRLHF and veRL for a new project?** TRL for single-node and small multi-node work with Hugging Face integration and fast iteration; OpenRLHF for Ray-based PPO/GRPO with separate vLLM engines at medium scale; veRL for large-scale, flexible placement and Megatron or FSDP backends with many recipes. Check the specific features you need in the installed version (Chapters 50, 51).

**S17. Design a multi-turn RL environment for customer-support agents.** Simulated users (an LLM with a persona and a hidden goal), a mock backend with tools (database queries, refunds) and policies, rewards from task completion checked against the backend state plus policy-compliance checks, turn limits, and logging; validate the simulator against real transcripts (Chapters 33, 34).

**S18. Your judge-based win-rate rose 8 points, but users do not prefer the new model. Why?** Length or style bias in the judge, distribution mismatch between evaluation prompts

and real use, judge self-preference, or contamination. Check length-controlled metrics, human evaluation on real prompts, and per-category breakdowns (Chapter 37).
**S19. How do you scale GRPO to very long responses (32k tokens)?** Sequence or context parallelism for training, dynamic batching by tokens, padding-free packing, long-tail mitigation in generation (partial rollouts, async), token-level loss normalisation, and overlong reward shaping or filtering for truncated responses (Chapters 28, 44).
**S20. Design monitoring for an RL run.** Per-step: rewards (mean and by source), advantage statistics, zero-variance group fraction, entropy, KL to reference, clip fractions, gradient norm, response length and truncation rate, logprob mismatch, staleness, throughput and GPU utilisation by phase; per-checkpoint: evaluation suite; plus sampled completions for human reading (Chapter 44).

## 57.3 Behavioural questions

Answer with a short story: the situation, what you did, the result with a number, and what you learned. Prepare stories from your own projects (Chapters 54 and 55) so that each one shows a different strength.

**Interview check**

**B1. Tell me about a time an experiment failed. What did you learn?** Pick a real failure with a clear cause (for example a masking bug that made SFT look fine on loss but hurt stopping). Explain how you found it (reading samples), what you changed in your process (a unit test for masks), and the result after the fix.
**B2. Describe a time you had to choose between speed and rigour.** Show that you matched rigour to the decision's stakes: a quick single-seed check to kill a bad idea, then multi-seed runs before claiming a result.
**B3. How do you decide what to work on next?** Expected impact times probability of success divided by cost; start from the biggest gap in evaluation results; prefer experiments that answer a question either way.
**B4. Tell me about a time you disagreed with a teammate on a technical decision.** Describe how you turned the disagreement into a cheap experiment or a shared metric, and how you committed to the decision once made.
**B5. How do you stay current in a fast-moving field?** A routine: read new papers in your area weekly, reproduce one result a month, follow framework release notes, and keep notes on claims you have verified versus only read.
**B6. Tell me about something you built end to end.** Choose the project with the clearest measured outcome; cover design choices and one trade-off you would revisit.
**B7. How do you handle ambiguous goals, such as "make the model more helpful"?** Turn them into measurable proxies with known weaknesses (a judge win-rate with length control plus human spot checks), agree on them with stakeholders, and revisit them as you learn.
**B8. Describe a time you found a bug in someone else's code or in a library.** Show how you isolated it with a minimal reproduction, reported or fixed it, and protected your work against it (pinning a version, adding a test).

**B9. What would you do in your first 90 days on a post-training team?** Reproduce the team's current baseline end to end, learn the evaluation suite and its weaknesses, ship one small improvement or tool, and then propose a larger experiment based on what you learned.
**B10. How do you communicate negative results?** Plainly and early, with the evidence and what it rules out, so others do not repeat the work; negative results with clear setups are valuable.
**B11. Tell me about a time you had too much to do.** How you prioritised (by impact and deadlines), what you dropped or delegated, and how you told the people affected.
**B12. Why post-training and RL?** A specific answer: the behaviour of a model is mostly shaped after pre-training; RL with verifiable rewards and agents is where capability gains are happening; you enjoy the mix of maths, systems and careful measurement. Connect it to your experience as an AI engineer.
**B13. How do you make sure your results are trustworthy?** Held-out evaluation, multiple seeds, confidence intervals, contamination checks, reading samples, and comparing against strong baselines at matched compute.
**B14. Describe a time you had to learn something quickly.** A concrete example with the learning method (build a minimal version from scratch, then read the reference implementation) and the result.
**B15. What is a research result in post-training you find over-hyped, and why?** Pick one with a reasoned critique, for example claims of large gains from tiny RL datasets evaluated on small, noisy benchmarks; explain what evidence would convince you (multiple seeds, larger evaluation sets, pass@k analysis).

**Key takeaways**

- Coding: write the function, then a self-check against a library, a brute force or an identity.
- System design: start from the step-time and memory model, then placement, asynchrony, reward service, monitoring and recovery.
- Behavioural: situation, action, measured result, lesson; one story per strength.

CHAPTER 58

# Cheat Sheet

**How to use this sheet.**
The formulas, numbers and defaults you are most likely to need in an interview, on a few pages. Defaults are for the versions used in this book (Transformers 5.20.2, TRL 1.16.0, PEFT, DeepSpeed 0.19.7, vLLM 0.32.0, veRL 0.9.0, OpenRLHF 0.11.2); check the installed version before relying on them.

## 58.1 Core formulas

| Quantity | Formula |
|---|---|
| Softmax, log-softmax | $p_i = e^{z_i}/\sum_j e^{z_j}$, $\partial \log p_i/\partial z_k = \mathbb{1}[i=k] - p_k$ |
| Cross-entropy gradient | $\nabla_z \ell = p - e_c$ |
| Temperature | $p_i \propto e^{z_i/T}$ |
| KL | $D_{\mathrm{KL}}(p\|q) = \mathbb{E}_p[\log p - \log q] \geq 0$; reverse KL is mode-seeking |
| KL estimators ($\rho = \pi_{\mathrm{ref}}/\pi_\theta$) | $k_1 = -\log\rho$, $k_2 = \frac{1}{2}(\log\rho)^2$, $k_3 = \rho - 1 - \log\rho$ |
| Policy gradient | $\nabla J = \mathbb{E}[(R-b)\nabla \log \pi_\theta(a \mid s)]$ |
| PPO | $-\mathbb{E}[\min(\rho\hat{A},\ \mathrm{clip}(\rho, 1-\epsilon, 1+\epsilon)\hat{A})]$ |
| GAE | $\delta_t = r_t + \gamma V_{t+1} - V_t$, $\hat{A}_t = \delta_t + \gamma\lambda\hat{A}_{t+1}$ |
| GRPO advantage | $\hat{A}_i = (r_i - \mathrm{mean}(r))/\mathrm{std}(r)$ per group (Dr. GRPO: no std) |
| RLOO advantage | $r_i - \frac{1}{G-1}\sum_{j\neq i} r_j = \frac{G}{G-1}(r_i - \bar{r})$ |
| KL-regularised optimum | $\pi^*(y \mid x) \propto \pi_{\mathrm{ref}}(y \mid x)\, e^{r(x,y)/\beta}$ |
| Bradley–Terry | $P(y_w \succ y_l) = \sigma(r_w - r_l)$, loss $-\log\sigma(r_w - r_l)$ |
| DPO | $-\log\sigma(\beta[\log\frac{\pi_\theta(y_w)}{\pi_{\mathrm{ref}}(y_w)} - \log\frac{\pi_\theta(y_l)}{\pi_{\mathrm{ref}}(y_l)}])$ |
| IPO (TRL, per-token averages $h$) | $(h - \frac{1}{2\beta})^2$ |
| SimPO | $-\log\sigma(\frac{\beta}{\lvert y_w\rvert}\log\pi_\theta(y_w) - \frac{\beta}{\lvert y_l\rvert}\log\pi_\theta(y_l) - \gamma)$ |
| pass@k (unbiased) | $1 - \binom{n-c}{k}/\binom{n}{k}$ |
| Best-of-$n$ KL | $\leq \log n - (n-1)/n$ |
| LoRA | $W = W_0 + \frac{\alpha}{r}BA$, $B = 0$ at init; params $r(d_{in} + d_{out})$; rsLoRA $\alpha/\sqrt{r}$ |

## 58.2 Numbers to know

| Item | Value |
|---|---|
| Mixed-precision Adam | ~16 bytes/parameter (2 weights + 2 grads + 4 master + 8 moments); 7B → ~112 GB |
| Inference weights | 2 bytes/param bf16, 1 byte FP8/INT8, ~0.5 byte 4-bit (+ block constants) |
| Training FLOPs | ~$6N$ per token (forward $2N$, backward $4N$) |
| Activations per layer | ~$34\,sbh$ bytes (FlashAttention), ~$2\,sbh$ with checkpointing |
| KV cache | $2\,n_L\,n_{kv}\,d_h\,s$×bytes; 32 layers, 8×128, 8k tokens, bf16 ≈ 1.07 GB |
| Decode arithmetic intensity | ≈ batch size (FLOPs per byte of weights); small batches are memory-bound |
| RL step time | generation is often 60–80% for reasoning RL |
| 70B weights in bf16 | ~140 GB to broadcast per weight sync |

## 58.3 Default hyperparameters (TRL 1.16 unless noted)

| Setting | Default | Typical choice |
|---|---|---|
| `SFTConfig` `learning_rate` | 2e-5 | 1e-5–2e-5 full; 1e-4–2e-4 LoRA |
| `SFTConfig` `max_length`, `packing` | 1024, False | 2048–8192, True with `bfd` |
| `DPOConfig` `beta`, `loss_type`, `learning_rate` | 0.1, `['sigmoid']`, 1e-6 | β 0.05–0.5; LR 5e-7–1e-6 full |
| `GRPOConfig` `num_generations` | 8 | 8–16 (64 in large runs) |
| `GRPOConfig` `beta`, `loss_type`, `scale_rewards` | 0.0, `dapo`, `group` | as default; `none` for Dr. GRPO |
| `GRPOConfig` `max_completion_length`, `temperature` | 512, 1.0 | 1k–32k for reasoning |
| `GRPOConfig` `epsilon`, `epsilon_high` | 0.2, None | `epsilon_high` 0.28 (clip-higher) |
| `RLOOConfig` `beta`, `num_generations` | 0.05, 2 | 4–8 |
| `LoraConfig` `r`, `lora_alpha`, `lora_dropout` | 8, 8, 0.0 | 16–64, 2r, 0–0.05; `target_modules="all-linear"` |
| `BitsAndBytesConfig` 4-bit quant type, compute dtype | `fp4`, float32 | `nf4`, bfloat16, double quant |

| Setting | Default | Typical choice |
|---|---|---|
| DeepSpeed `gradient_clipping`, ZeRO `stage` | 1.0, 0 | 1.0; 2 or 3 |
| vLLM `SamplingParams.max_tokens` | 16 | always set it |
| vLLM `gpu_memory_utilization` | 0.92 | 0.3–0.6 colocated; 0.9 dedicated |

## 58.4 Debugging RL: symptom → first checks

| Symptom | First checks |
|---|---|
| Reward flat at start | Reward function on hand examples; prompt format; zero-variance groups; learning rate |
| Reward up, eval flat | Read top-reward samples for hacks; train/test format mismatch; truncation |
| Entropy collapses | Temperature, clip-higher, difficulty filtering, entropy/KL settings |
| Length explodes | Loss normalisation; overlong penalty; truncation handling; length-biased reward |
| Loss spikes, grad norm spikes | Logprob mismatch metric; stale samples; bad data shard; LR too high |
| KL grows fast | LR, $\beta$, number of epochs per batch, clip range |
| OOM in RL | vLLM memory fraction, sleep mode, micro-batch, checkpointing, logits chunking |

## 58.5 Which method when

| Situation | Method |
|---|---|
| New behaviour or format, demonstrations available | SFT (LoRA first) |
| Fixed preference pairs, style/helpfulness | DPO (or IPO/normalised DPO if length grows) |
| Unpaired thumbs-up/down data | KTO |
| Verifiable task (maths, code, tests) | GRPO / RLOO / DAPO-style RLVR |
| Need dense credit over long agent episodes | PPO with critic, or turn-level advantages |
| Small model, strong teacher available | Distillation (sequence-level, then on-policy) |
| Limited GPU memory | QLoRA, ZeRO-3/FSDP, checkpointing, offload |

**Why this, not that? Why memorise a few numbers, not look everything up?**

**Reason.** Design questions are about orders of magnitude: whether something fits, what dominates, where to spend GPUs. A handful of anchors (16 bytes/parameter, $6N$ FLOPs, KV-cache size, generation share) lets you reason live. **Trade-off.** Defaults change between versions; memorised defaults go stale. **When the alternative is better.** For exact parameter names and defaults in real work, always read the installed source or docs (Part X), never memory.

CHAPTER 59

# Glossary

**How to use the glossary.**
Short definitions of the terms used in this book, in alphabetical order. The chapter where a term is explained in depth is given in brackets.

**A**

**Accelerate.** Hugging Face library that runs the same training script on one GPU, many GPUs or many nodes, and wraps DeepSpeed and FSDP [42].

**Action.** What the agent chooses at each step; for an LLM, the next token (or a whole response in the bandit view) [13].

**Activation.** An intermediate output of a layer, stored in the forward pass for the backward pass [40].

**Actor.** The policy network in actor–critic methods; in RLHF, the LLM being trained [16].

**Adam, AdamW.** Optimisers that keep running averages of the gradient and its square; AdamW applies weight decay directly to the weights instead of through the gradient [4].

**Adapter.** A small set of trainable weights added to a frozen model, such as a LoRA module [10].

**Adaptive KL controller.** Adjusts the KL coefficient during RLHF to keep the KL near a target [20].

**Advantage.** How much better an action is than the average action in that state: $A(s,a) = Q(s,a) - V(s)$ [14].

**Agent.** A model that acts over several steps, usually calling tools and reading their outputs [34].

**Agentic RL.** RL on multi-step tasks with tools and environments, rewarded on the final outcome or per turn [34].

**Alignment.** Training a model to behave as intended: helpful, honest and harmless [1].

**Alignment tax.** Capability lost through alignment training [38].

**All-gather.** Collective operation that gives every GPU the concatenation of all GPUs' shards [42].

**all-linear.** PEFT shortcut that targets every linear layer except the output head [46].

**All-reduce.** Collective operation that sums (or averages) a tensor across GPUs and gives every GPU the result [42].

**AMP.** Automatic mixed precision: running selected operations in 16-bit automatically [41].

**Answer extraction.** Parsing the final answer out of a response, for example from `\boxed{}` [26].

**Arithmetic intensity.** FLOPs per byte of memory moved; decides whether a kernel is compute-bound or memory-bound [43].

**Assistant-only loss.** SFT loss computed only on the assistant's tokens [47].

**Asynchronous RL.** Generation and training run at the same time on different GPUs, so samples are slightly off-policy [44].

**Attention.** Mechanism where each token computes a weighted sum of other tokens' value vectors, with weights from query–key similarity [5].

**Attention head.** One of several parallel attention computations in a layer, each with its own projections [5].

**Autoregressive model.** A model that generates one token at a time, each conditioned on all previous tokens [5].

**B**

**Backpropagation.** Computing gradients layer by layer with the chain rule [4].

**Bandit.** An RL problem with one decision per episode; LLM RL with a single terminal reward is often treated as one [13].

**Baseline.** A value subtracted from the reward in policy gradients to reduce variance without bias [15].

**Batch size (effective).** Per-device batch × gradient-accumulation steps × number of devices [45].

**Beam search.** Decoding that keeps the $k$ most likely partial sequences; rarely used for chat models [7].

**Behaviour policy.** The policy that generated the data; in off-policy learning it differs from the policy being trained [18].

**Bellman equation.** Recursive definition of a value function in terms of the immediate reward and the next state's value [14].

**Benchmark.** A fixed dataset and scoring rule used to compare models [37].

**Best-of-$n$.** Sample $n$ responses and keep the one a reward model scores highest [21].

**BF16 (bfloat16).** 16-bit floating-point format with float32's exponent range and 8 bits of precision; the standard training format [41].

**BFD packing.** Best-fit-decreasing packing of examples into fixed-length rows; TRL's default packing strategy [47].

**Bias (statistical).** Difference between an estimator's expected value and the true value [2].

**Bias–variance trade-off.** Lowering an estimator's variance often adds bias, and the reverse [16].

**Binary cross-entropy.** Cross-entropy for two classes; the Bradley–Terry loss is one [19].

**bitsandbytes.** Library for 8-bit and 4-bit quantisation and 8-bit optimisers [11].

**Bootstrapping (RL).** Using an estimated value of the next state in a target, as in TD learning [14].

**Bootstrapping (statistics).** Resampling data with replacement to estimate uncertainty [37].

**BOS / EOS.** Beginning- and end-of-sequence tokens [6].

**BPE.** Byte-pair encoding: a tokenizer that repeatedly merges the most frequent adjacent symbol pairs [6].

**Bradley–Terry model.** Preference model in which $P(a \succ b) = \sigma(r_a - r_b)$ [19].

**Bubble (pipeline).** Idle time in pipeline parallelism while stages wait for each other [42].

**Byte-level BPE.** BPE over bytes, so any text can be tokenised without unknown tokens [6].

**C**

**Calibration.** How well predicted probabilities match observed frequencies [37].

**Canary string.** A unique string placed in evaluation data to detect whether it leaked into training data [55].

**Catastrophic forgetting.** Loss of earlier abilities when training on new data [8].

**Causal mask.** Attention mask that stops a token from attending to later tokens [5].

**Chain of thought (CoT).** Step-by-step reasoning written before the answer [30].

**Chat template.** Rule (usually Jinja) that turns a list of messages into one token sequence with role markers [6].

**Checkpointing (activation / gradient).** Storing only some activations in the forward pass and recomputing the rest in the backward pass to save memory [40].

**Chinchilla scaling.** Compute-optimal pre-training uses about 20 tokens per parameter [5].

**Chunked prefill.** Splitting a long prompt's prefill into chunks so it can be batched with decoding [43].

**Clip fraction.** Fraction of tokens whose PPO ratio fell outside the clip range [17].

**Clip-higher.** Asymmetric PPO clipping with a larger upper bound, to keep low-probability tokens able to grow (DAPO) [28].

**Collective communication.** Operations involving all GPUs in a group, such as all-reduce [42].

**Colocation.** Running rollout and training on the same GPUs, switching between them [44].

**Completion.** The model-generated part of a sequence, after the prompt [47].

**Completion mask.** Mask marking completion tokens, used to restrict losses [27].

**Compute-bound.** Limited by arithmetic throughput rather than memory bandwidth [43].

**Confidence interval.** A range that contains the true value with a stated probability under repeated sampling [37].

**Constitutional AI.** Training with AI feedback guided by written principles, including self-critique and revision [25].

**Contamination.** Evaluation data appearing in training data [39].

**Context length.** Maximum number of tokens a model can process at once [5].

**Continuous batching.** Adding and removing sequences from a running batch at every decoding step [43].

**Control variate.** A zero-mean quantity added to an estimator to reduce variance [3].

**Cosine schedule.** Learning-rate schedule that decays along a half cosine after warm-up [4].

**Credit assignment.** Working out which actions caused a reward [14].

**Critic.** A network that estimates the value function [16].

**Cross-entropy.** $H(p, q) = -\mathbb{E}_p[\log q]$; the standard language-model loss [3].

**Curriculum.** Ordering or weighting training tasks by difficulty [36].

**D**

**DAPO.** Decoupled clip and dynamic sampling policy optimisation: GRPO with clip-higher, dynamic sampling, token-level loss and overlong shaping [28].

**Data mixture.** The proportions of different sources or tasks in training data [39].

**Data parallelism (DP).** Each GPU holds a model copy and processes different data; gradients are averaged [42].

**Decoding.** Generating tokens one at a time after the prefill [43].

**Decontamination.** Removing training examples that overlap with evaluation data [39].

**Deduplication.** Removing exact and near-duplicate examples (for example with MinHash) [39].

**DeepSpeed.** Microsoft's library for large-scale training: ZeRO, offload, pipeline parallelism [53].

**Dense reward.** Reward given at many steps rather than only at the end [36].

**DFT.** Dynamic fine-tuning: an SFT loss variant that reweights tokens by their probability [47].

**Direct alignment algorithm.** A preference method that trains the policy directly on preferences without RL, such as DPO [22].

**Discount factor ($\gamma$).** Weight on future rewards; LLM RL usually uses 1 [13].

**Distillation.** Training a student model to imitate a teacher [12].

**Double quantisation.** Quantising the quantisation constants themselves, saving about 0.4 bits per parameter in QLoRA [10, 11].

**DPO.** Direct preference optimisation: a classification loss on preference pairs equivalent to KL-regularised reward maximisation [22].

**Draft model.** The small model that proposes tokens in speculative decoding [43].

**Dr. GRPO.** GRPO without per-group std scaling and per-sequence length normalisation [28].

**Dropout.** Randomly zeroing activations in training; usually disabled in LLM post-training [5].

**Dynamic loss scaling.** Raising and lowering the FP16 loss scale automatically on overflows [41].

**Dynamic sampling.** Dropping groups whose rewards are all equal and resampling until the batch is full [28].

**E**

**ECE.** Expected calibration error: weighted average gap between predicted confidence and accuracy across bins [37].

**Embedding.** The vector representing a token at the input of the model [5].

**Entropy.** $H(p) = -\sum p \log p$; for a policy, how random its choices are [3].

**Entropy collapse.** Rapid loss of policy entropy during RL, ending exploration [28].

**Environment.** The program that receives actions and returns observations and rewards [32].

**Episode.** One run of an agent in an environment from reset to the end [32].

**Epoch.** One pass over the training data [4].

**Evaluation harness.** Software that runs benchmarks with fixed prompts, sampling and scoring [37].

**Exact match.** Metric: 1 if the normalised answer equals the reference [37].

**Expected value.** The probability-weighted average of a random quantity [2].

**Expert parallelism.** Placing different experts of a mixture-of-experts layer on different GPUs [42].

**Exploration.** Trying actions whose value is uncertain to discover better ones [13].

**Exposure bias.** Mismatch between training on ground-truth prefixes and generating from one's own prefixes [12].

**F**

**Few-shot prompting.** Putting examples of the task in the prompt [1].

**Fine-tuning.** Further training of a pre-trained model on specific data [8].

**FlashAttention.** An exact attention kernel that avoids storing the full attention matrix [40].

**FLOPs.** Floating-point operations; training costs about $6N$ per token [40].

**Format reward.** Reward for following a required output format [26].

**Forward KL.** $D_{\mathrm{KL}}(p\|q)$ with the target $p$ first; mode-covering when fitting $q$ [3].

**FP8.** 8-bit floating-point formats (E4M3, E5M2) for faster matrix multiplications and smaller caches [41].

**FSDP.** PyTorch's fully sharded data parallelism, similar to ZeRO-3 [42].

**Full fine-tuning.** Updating all model weights [10].

**Fused kernel.** One GPU kernel that does several operations to save memory traffic [40].

**G**

**GAE.** Generalised advantage estimation: an exponentially weighted average of $n$-step advantage estimates [16].

**Gating (router).** The part of an MoE layer that chooses which experts process each token [5].

**Generative reward model.** A judge that writes reasoning and then a verdict, rather than a scalar head [19].

**GGUF.** File format used by llama.cpp for quantised models [52].

**GKD.** Generalised knowledge distillation: on-policy distillation with a generalised JSD loss [12].

**Gold reward.** In over-optimisation studies, the reference reward the proxy is meant to approximate [21].

**Goodhart's law.** When a measure becomes a target, it stops being a good measure [21].

**GQA.** Grouped-query attention: several query heads share one key/value head, shrinking the KV cache [5].

**Gradient.** The vector of partial derivatives of the loss with respect to the parameters [4].

**Gradient accumulation.** Summing gradients over several micro-batches before one optimiser step [45].

**Gradient clipping.** Scaling the gradient down when its norm exceeds a threshold [4].

**Gradient norm.** Length of the gradient vector; a key health metric [31].

**Greedy decoding.** Always picking the most likely next token [7].

**Group (GRPO).** The $G$ responses sampled for one prompt [27].

**GRPO.** Group relative policy optimisation: PPO-style loss with advantages from reward normalisation within a group, no critic [27].

**Gymnasium.** The standard Python API for RL environments (`reset`, `step`) [32].

**H**

**Hallucination.** Fluent but false output [38].

**Helpfulness / harmlessness.** The two main axes of preference data for assistants [19].

**Hybrid engine.** A system that switches one set of GPUs between training and generation modes [44].

**HybridFlow.** The design behind veRL: single-controller dataflow with multi-controller workers [44].

**Hyperparameter.** A setting chosen before training, such as the learning rate [4].

**I**

**Implicit reward.** In DPO, $\beta \log(\pi_\theta / \pi_{\text{ref}})$ of a response [22].

**Importance ratio.** $\pi_\theta(a)/\pi_{\text{old}}(a)$ [17].

**Importance sampling.** Re-weighting samples from one distribution to estimate expectations under another [18].

**In-context learning.** Learning a task from examples in the prompt without weight updates [1].

**Inference engine.** Optimised serving software such as vLLM or SGLang [43].

**Instruction tuning.** SFT on instruction–response pairs [8].

**Inter-annotator agreement.** How often different labellers agree; caps achievable reward-model accuracy [19].

**IPO.** Identity preference optimisation: squared-loss alternative to DPO that does not push margins to infinity [23].

**J**

**Jailbreak.** A prompt that makes a model ignore its safety training [38].

**Jinja.** The template language used for chat templates [6].

**JSD.** Jensen–Shannon divergence: symmetric average of KL divergences to the mixture [3].

**Judge (LLM-as-a-judge).** A model used to score or compare responses [37].

**K**

**k1, k2, k3.** The three per-sample KL estimators from Schulman's note [3].

**KL divergence.** $D_{\mathrm{KL}}(p\|q) = \mathbb{E}_p[\log p/q]$ [3].

**KL penalty.** A term $\beta D_{\mathrm{KL}}(\pi\|\pi_{\mathrm{ref}})$ keeping the policy close to the reference [20].

**KTO.** Kahneman–Tversky optimisation: preference learning from unpaired good/bad labels [23].

**KV cache.** Stored keys and values of past tokens, reused during decoding [43].

**L**

**Label smoothing.** Training against a softened target; in DPO, a model of label noise (robust DPO) [49].

**LayerNorm.** Normalisation that subtracts the mean and divides by the standard deviation of features [5].

**Learning rate.** Step size of the optimiser [4].

**Length bias.** Preference of judges or reward models for longer responses [21].

**Length-controlled win-rate.** Win-rate adjusted to remove the effect of length differences [37].

**Length normalisation.** Dividing log-probabilities or losses by length [23].

**Log-derivative trick.** $\nabla p = p\nabla \log p$, the step behind the policy gradient [15].

**Logit.** Unnormalised score before the softmax [2].

**Logprob mismatch.** Difference between the inference engine's and the trainer's log-probabilities for the same tokens [43].

**Long context.** Inputs of tens to hundreds of thousands of tokens [5].

**LoRA.** Low-rank adaptation: training a low-rank update $BA$ added to frozen weights [10].

**Loss masking.** Excluding tokens from the loss by setting labels to $-100$ [9].

**Loss scaling.** Multiplying the loss in FP16 training to avoid gradient underflow [41].

**Loss spike.** A sudden jump in loss, often from bad data, high learning rate or numerical issues [31].

**M**

**Majority voting.** Answering with the most common final answer among samples (self-consistency) [30].

**Margin (preference).** Difference between implicit or explicit rewards of chosen and rejected responses [22].

**Markov decision process (MDP).** States, actions, transitions, rewards and a discount [13].

**Master weights.** FP32 copy of weights kept by the optimiser in mixed-precision training [41].

**Maximum likelihood.** Choosing parameters that make the data most probable [2].

**Megatron-LM.** NVIDIA's framework for tensor, pipeline and sequence parallel training [42].

**Memory-bound.** Limited by memory bandwidth rather than compute [43].

**Merging (model).** Combining weights of several fine-tuned models, or merging LoRA into the base [10].

**Micro-batch.** The batch processed in one forward/backward pass on one device [45].

**MinHash.** Hashing technique for fast near-duplicate detection [39].

**Mixed precision.** Computing in 16-bit while keeping some states in 32-bit [41].

**Mixture of experts (MoE).** Layers with many expert MLPs of which a router activates a few per token [5].

**Mode collapse.** Loss of output diversity [28].

**Mode-seeking.** Concentrating on one mode of a target distribution (reverse KL) [3].

**Monte Carlo return.** The observed sum of rewards from a time step to the episode's end [14].

**Multi-head attention.** Attention computed in several heads whose outputs are concatenated [5].

**Multi-turn RL.** RL over conversations with several model and environment turns [34].

**N**

**NCCL.** NVIDIA's library for GPU collective communication [42].

**Near-duplicate.** Two texts that differ only slightly; found with MinHash or embeddings [39].

**NEFTune.** Adding noise to embeddings during SFT [9].

**NF4.** 4-bit NormalFloat: a 4-bit format with levels at quantiles of a normal distribution [11].

**Nucleus sampling.** See top-p.

**num_generations.** Group size $G$ in TRL's GRPO and RLOO [50].

**O**

**Observation.** What the agent sees of the environment's state [32].

**Offload.** Moving optimiser states or parameters to CPU memory or NVMe [53].

**Off-policy.** Learning from data produced by a different policy [18].

**Online DPO.** DPO with pairs sampled from the current policy and labelled by a judge or reward model [24].

**On-policy.** Learning from data produced by the current policy [18].

**On-policy distillation.** The student generates and the teacher scores its tokens [12].

**OpenRLHF.** Ray-based RLHF framework with vLLM rollouts and DeepSpeed training [51].

**ORM.** Outcome reward model: scores only the final answer [29].

**ORPO.** Odds-ratio preference optimisation: SFT loss plus an odds-ratio preference term, without a reference model [23].

**Overlong shaping.** Penalising responses close to or over the length limit (DAPO) [28].

**Over-optimisation.** Optimising a proxy reward past the point where it tracks the true goal [21].

**Over-refusal.** Refusing harmless requests [38].

**P**

**Packing.** Concatenating examples into fixed-length sequences [9].

**Padding-free.** Training on flattened variable-length sequences without padding tokens [47].

**PagedAttention.** vLLM's paged KV-cache management, like virtual memory pages [43].

**Paged optimiser.** Optimiser whose states can page between GPU and CPU memory (bitsandbytes) [10].

**Pairwise comparison.** Judging which of two responses is better [37].

**Parquet.** Columnar file format used for veRL datasets [51].

**Partial rollout.** Pausing unfinished generations at a weight update and resuming them later [44].

**Pass@k.** Probability that at least one of $k$ samples is correct [37].

**PEFT.** Parameter-efficient fine-tuning; also the Hugging Face library [10].

**Perplexity.** $e^{\text{cross-entropy}}$; lower is better [3].

**Pipeline parallelism (PP).** Splitting layers into stages on different GPUs [42].

**Policy.** A mapping from states to action probabilities; for us, the LLM [13].

**Policy gradient.** Gradient of expected reward with respect to policy parameters [15].

**Position ids.** The positions given to tokens for positional encoding; reset per example when packing [9].

**PPO.** Proximal policy optimisation: clipped surrogate objective with a critic and GAE [17].

**Preference data.** Prompts with chosen and rejected responses [19].

**Prefill.** Processing the prompt in one parallel forward pass before decoding [43].

**Prefix caching.** Reusing KV-cache blocks for shared prompt prefixes [43].

**PRM.** Process reward model: scores intermediate reasoning steps [29].

**Prompt.** The input the model conditions on [7].

**Proxy reward.** The learned or approximate reward actually optimised [21].

**Q**

**Q-function.** $Q(s, a)$: expected return after taking $a$ in $s$ [14].

**QLoRA.** LoRA on a 4-bit quantised frozen base model [11].

**Quantisation.** Storing weights or activations with fewer bits [11].

**R**

**Rank (LoRA).** Inner dimension $r$ of the update $BA$ [10].

**Ray.** Distributed Python framework used by OpenRLHF and veRL [51].

**Reasoning model.** A model trained to write long chains of thought before answering [30].

**Red teaming.** Searching for inputs that make a model misbehave [38].

**Reference-free.** A preference loss with no reference model, such as SimPO or ORPO [23].

**Reference model.** The frozen model (usually the SFT model) the policy is regularised toward [20].

**Refusal.** A response declining to help [38].

**Regularisation.** Any term or technique that limits overfitting or drift, such as weight decay or a KL penalty [4].

**REINFORCE.** The basic Monte Carlo policy-gradient algorithm [15].

**Rejection sampling.** Keeping only samples that pass a filter or score highest [26].

**Residual stream.** The running hidden state that each layer reads from and adds to [5].

**Return.** Sum of (discounted) rewards from a time step [13].

**Reverse KL.** $D_{\mathrm{KL}}(q\|p)$ with the model first; mode-seeking [3].

**Reward.** Scalar feedback from the environment or a reward model [13].

**Reward centring.** Penalising the mean reward so outputs are centred near zero [19, 48].

**Reward ensemble.** Several reward models combined, often conservatively, to reduce hacking [21].

**Reward hacking.** The policy exploiting flaws in the reward [21].

**Reward model (RM).** A model that scores responses, trained on preferences [19].

**Reward shaping.** Adding terms to the reward to guide learning [36].

**RLAIF.** RL from AI feedback [25].

**RLHF.** RL from human feedback: reward model plus RL [20].

**RLOO.** REINFORCE leave-one-out: baseline from the other samples of the same prompt [27].

**RLVR.** RL with verifiable rewards [26].

**RMSNorm.** Normalisation by root-mean-square without mean subtraction [5].

**Rollout.** Generating responses (trajectories) from the policy [43].

**RoPE.** Rotary position embedding: rotates query and key pairs by position-dependent angles [5].

**Rubric.** A written list of criteria a judge scores against [36].

**Rule-based reward.** A reward computed by code (regex, parsers, tests) rather than a model [26].

**S**

**Safety training.** Post-training aimed at refusing harmful requests and reducing harms [38].

**Sample efficiency.** Performance gained per sample [18].

**Sampling parameters.** Temperature, top-p, top-k, maximum tokens and similar generation settings [7].

**Sandbox.** An isolated environment for running untrusted code [35].

**Scaling laws.** Power-law relations between loss and model size, data and compute [1].

**Scheduler (LR).** Rule for changing the learning rate over training [4].

**Self-play.** Training against copies of oneself [24].

**Sequence-level distillation.** SFT of the student on teacher-generated responses [12].

**Sequence parallelism.** Splitting the sequence dimension across GPUs [42].

**SFT.** Supervised fine-tuning on demonstrations [8].

**SGLang.** An inference engine with RadixAttention prefix caching [43].

**Sharding.** Splitting tensors across devices [42].

**SimPO.** Reference-free, length-normalised preference loss with a target margin [23].

**Single controller.** One process orchestrates the whole RL dataflow (veRL's driver) [44].

**Sleep mode (vLLM).** Releasing the KV cache and weights from GPU memory while training runs [53].

**Softmax.** Function turning logits into probabilities [2].

**Sparse reward.** Reward only at the end of an episode [36].

**Speculative decoding.** A small draft model proposes tokens that the large model verifies in parallel [43].

**Staleness.** How many policy versions old a sample is [44].

**State.** Full description of the situation at a time step [13].

**Stop rate.** Fraction of responses that end properly before the length limit [54].

**SwiGLU.** The gated MLP used in most modern LLMs [5].

**Synthetic data.** Training data generated by models rather than people [39].

**System prompt.** The initial instruction message setting behaviour [6].

**T**

**TD error.** $\delta = r + \gamma V(s') - V(s)$ [14].

**Teacher forcing.** Training on ground-truth prefixes [9].

**Temperature.** Divides logits before softmax; controls randomness [7].

**Tensor parallelism (TP).** Splitting individual weight matrices across GPUs [42].

**Test-time compute.** Spending more inference compute (longer reasoning, more samples) for better answers [30].

**TIS.** Truncated importance sampling: importance weights capped at a constant [43].

**Token.** A unit of text in the model's vocabulary [6].

**Tokenizer.** Converts text to token ids and back [6].

**Tool use.** The model calling external functions and reading their results [34].

**Top-k sampling.** Sampling only from the $k$ most likely tokens [7].

**Top-p sampling.** Sampling from the smallest set of tokens whose probability reaches $p$ [7].

**Trajectory.** The sequence of states, actions and rewards in an episode [13].

**Transformer.** The attention-based architecture of modern LLMs [5].

**TRL.** Hugging Face's Transformer Reinforcement Learning library: SFT, DPO, GRPO, RLOO and more [47].

**Truncation (episode).** Ending an episode by a time or length limit rather than a terminal state [32].

**Trust region.** A limit on how far a policy may move in one update [17].

**U**

**Unbiased estimator.** One whose expected value equals the true quantity [2].

**Unit-test reward.** Reward from the fraction of tests a generated program passes [26].

**Unsloth.** Library with optimised kernels for memory-efficient fine-tuning and RL on few GPUs [52].

**V**

**Value function.** $V(s)$: expected return from state $s$ [14].

**Value head.** Scalar output layer added to an LLM to predict values or rewards [19].

**Variance.** Spread of an estimator around its mean [2].

**Verifier.** Program that checks whether an answer is correct [26].

**veRL.** ByteDance's HybridFlow-based RL framework for LLMs [51].

**vLLM.** High-throughput inference engine with PagedAttention and continuous batching [43].

**Vocabulary.** The set of tokens a model knows [6].

**W**

**Warm-up.** Raising the learning rate gradually at the start of training [4].

**Weight decay.** Shrinking weights toward zero each step [4].

**Weight sync.** Copying updated trainer weights to the inference engine [44].

**Weight tying.** Sharing the input embedding matrix with the output head [5].

**Win-rate.** Fraction of pairwise comparisons a model wins [37].

**Z**

**ZeRO.** Zero redundancy optimiser: shards optimiser states (stage 1), gradients (2) and parameters (3) across data-parallel GPUs [42].

**Zero-shot.** Performing a task with no examples in the prompt [1].

**Zero-variance group.** A GRPO group whose responses all got the same reward, giving no gradient [27].

CHAPTER A

# Study Plans

**Why this appendix matters.**
A book this long is easier to use with a plan. Below are three paths: a two-week fast track before an interview, an eight-week path that covers everything with projects, and a reference path for working engineers.

## A.1 Two-week fast track

| Day | Read | Do |
|---|---|---|
| 1 | Ch. 1, 5 (skim), 6 | Render chat templates for two models; find their end-of-turn tokens |
| 2 | Ch. 8, 9 | Coding C1–C3, C19, C20 |
| 3 | Ch. 10, 11 | C9; the LoRA/QLoRA memory numbers in the cheat sheet |
| 4 | Ch. 13–15 | Q56–Q61; C13 |
| 5 | Ch. 16, 17 | Q62–Q63, Q75–Q76; C14 |
| 6 | Ch. 19–21 | Q29–Q30, Q71; C10, C25 |
| 7 | Ch. 22, 23 | Q67–Q70; C11, C30 |
| 8 | Ch. 26, 27 | C12, C16; Project 6 starter code |
| 9 | Ch. 28, 30 | Q22–Q28, Q86, Q90 |
| 10 | Ch. 33, 34 | Project 7 starter code; C26 |
| 11 | Ch. 37 | C17; Project 10 starter code |
| 12 | Ch. 40, 42, 43 | Q79–Q84; S12 |
| 13 | Ch. 44 | S1–S6, S20 |
| 14 | Ch. 56–58 | Mock interview: 10 random questions aloud; behavioural stories B1–B15 |

## A.2 Eight-week full path

| Week | Parts | Project |
|---|---|---|
| 1 | Part I (foundations) | Re-derive every boxed formula on paper |
| 2 | Part II (SFT, PEFT, quantisation, distillation) | Project 1 |
| 3 | Part III (RL from the ground up) | Re-implement REINFORCE, GAE and PPO on a bandit |
| 4 | Parts IV–V (RLHF, preference optimisation) | Projects 2 and 3 |
| 5 | Part VI (reasoning RL) | Project 6 |
| 6 | Parts VII–VIII (environments, evaluation, safety, data) | Projects 7 and 10 |
| 7 | Parts IX–X (systems, library reference) | Projects 5 and 9 |
| 8 | Parts XI–XII | Projects 4 and 8; full mock interviews |

## A.3 Reference path

Read Chapter 1, then go straight to the chapter you need. Part X gives exact parameters with version labels; Chapter 58 gives the numbers; the glossary gives definitions with chapter pointers.

**Why this, not that? Why projects interleaved with reading, not reading first?**

**Reason.** Building the piece right after reading about it fixes the idea and exposes misunderstandings while the chapter is fresh. **Trade-off.** Progress through the text is slower. **When the alternative is better.** With an interview in days, read and practise answers first (the fast track), and build only the starter code.

CHAPTER B

# Versions and Verification

**Why this appendix matters.**
Library parameters change between releases. This appendix records exactly which versions the book was checked against and how, so you can repeat the checks against newer versions.

## B.1 Versions used

| Library | Version | How it was checked |
|---|---|---|
| PyTorch | 2.14.1 (CPU build) | Installed; all snippets run |
| Transformers | 5.20.2 | Installed; parameter lists generated from the dataclasses |
| TRL | 1.16.0 | Installed; configs generated, tiny trainers run on CPU |
| PEFT | 0.21.2 | Installed; LoraConfig generated, LoRA snippets run |
| Datasets / Accelerate | 5.1.0 / 1.15.0 | Installed |
| DeepSpeed | 0.19.7 | Installed; config classes validated on CPU |
| Gymnasium | 1.4.0 | Installed; environment checker run |
| vLLM | 0.32.0 | Source inspection of the release wheel (GPU-only install) |
| veRL | 0.9.0 | Source inspection (configuration YAML and Python source) |
| OpenRLHF | 0.11.2 | Source inspection of the wheel (argument parser) |
| Unsloth | 2026.10.7 | Source inspection (needs a GPU to import) |

The generated lists in Part X were produced by two scripts in the book's source: one imports the installed dataclasses and reads field names, defaults and help strings; the other parses library

source files with Python's `ast` module (class attributes, function signatures and argparse flags) and veRL's generated YAML configuration. Nothing in those lists was typed by hand.

## B.2 Checking a parameter yourself

```
# A dataclass config (TRL, Transformers, PEFT): list fields and defaults
python -c "import dataclasses, trl; print([(f.name, f.default) for f in
 ↪  dataclasses.fields(trl.GRPOConfig)][:10])"
# A function signature
python -c "import inspect, trl; print(inspect.signature(trl.GRPOTrainer.__init__))"
# A library that needs a GPU to import: download the wheel and read the source
pip download vllm==0.32.0 --no-deps -d /tmp/wheels && cd /tmp/wheels && unzip -q
 ↪  vllm-*.whl -d vllm_src
grep -n "max_tokens" vllm_src/vllm/sampling_params.py | head
```

**Why this, not that? Why generate parameter lists from source, not copy them from documentation pages?**

**Reason.** Documentation pages lag behind releases and sometimes describe a different version; the installed source is what actually runs. **Trade-off.** Source docstrings can be terse, and a source-inspected library (not imported) may compute some defaults at run time, which static parsing shows as an expression. **When the alternative is better.** For usage guidance, examples and recommended settings, the official documentation and recipes are better than raw source.

CHAPTER C

# References

**About this list.**
Papers cited in the book, by arXiv identifier, with titles checked against arXiv. Blog posts and documentation cited in the text (for example Schulman's note on KL approximation and the "LoRA Without Regret" post) are named where they are used.


- arXiv 1312.5602: *Playing Atari with Deep Reinforcement Learning*
- arXiv 1502.05477: *Trust Region Policy Optimization*
- arXiv 1503.02531: *Distilling the Knowledge in a Neural Network*
- arXiv 1506.02438: *High-Dimensional Continuous Control Using Generalized Advantage Estimation*
- arXiv 1602.01783: *Asynchronous Methods for Deep Reinforcement Learning*
- arXiv 1604.06174: *Training Deep Nets with Sublinear Memory Cost*
- arXiv 1706.03741: *Deep reinforcement learning from human preferences*
- arXiv 1707.06347: *Proximal Policy Optimization Algorithms*
- arXiv 1710.03740: *Mixed Precision Training*
- arXiv 1801.01290: *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*
- arXiv 1802.01561: *IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures*
- arXiv 1806.03377: *PipeDream: Fast and Efficient Pipeline Parallel DNN Training*
- arXiv 1811.06965: *GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism*
- arXiv 1902.00751: *Parameter-Efficient Transfer Learning for NLP*
- arXiv 1909.08053: *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*
- arXiv 1909.08593: *Fine-Tuning Language Models from Human Preferences*
- arXiv 1910.02054: *ZeRO: Memory Optimizations Toward Training Trillion Parameter Models*
- arXiv 2005.12729: *Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO*
- arXiv 2006.05990: *What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study*
- arXiv 2006.16668: *GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding*
- arXiv 2009.01325: *Learning to summarize from human feedback*
- arXiv 2009.03300: *Measuring Massive Multitask Language Understanding*
- arXiv 2009.10897: *Revisiting Design Choices in Proximal Policy Optimization*
- arXiv 2101.00190: *Prefix-Tuning: Optimizing Continuous Prompts for Generation*
- arXiv 2101.03961: *Switch Transformers: Scaling to Trillion Parameter Models with Simple and*

*Efficient Sparsity*
- arXiv 2101.06840: *ZeRO-Offload: Democratizing Billion-Scale Model Training*
- arXiv 2103.03874: *Measuring Mathematical Problem Solving With the MATH Dataset*
- arXiv 2104.07857: *ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning*
- arXiv 2104.08691: *The Power of Scale for Parameter-Efficient Prompt Tuning*
- arXiv 2106.09685: *LoRA: Low-Rank Adaptation of Large Language Models*
- arXiv 2107.03374: *Evaluating Large Language Models Trained on Code*
- arXiv 2107.06499: *Deduplicating Training Data Makes Language Models Better*
- arXiv 2110.14168: *Training Verifiers to Solve Math Word Problems*
- arXiv 2203.02155: *Training language models to follow instructions with human feedback*
- arXiv 2203.11171: *Self-Consistency Improves Chain of Thought Reasoning in Language Models*
- arXiv 2203.14465: *STaR: Bootstrapping Reasoning With Reasoning*
- arXiv 2204.05862: *Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback*
- arXiv 2205.05198: *Reducing Activation Recomputation in Large Transformer Models*
- arXiv 2205.05638: *Few-Shot Parameter-Efficient Fine-Tuning is Better and Cheaper than In-Context Learning*
- arXiv 2205.14135: *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*
- arXiv 2208.07339: *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*
- arXiv 2209.05433: *FP8 Formats for Deep Learning*
- arXiv 2210.10760: *Scaling Laws for Reward Model Overoptimization*
- arXiv 2210.17323: *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*
- arXiv 2211.10438: *SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models*
- arXiv 2211.14275: *Solving math word problems with process- and outcome-based feedback*
- arXiv 2211.17192: *Fast Inference from Transformers via Speculative Decoding*
- arXiv 2212.08073: *Constitutional AI: Harmlessness from AI Feedback*
- arXiv 2212.10560: *Self-Instruct: Aligning Language Models with Self-Generated Instructions*
- arXiv 2304.06767: *RAFT: Reward rAnked FineTuning for Generative Foundation Model Alignment*
- arXiv 2304.11277: *PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel*
- arXiv 2304.12244: *WizardLM: Empowering large pre-trained language models to follow complex instructions*
- arXiv 2305.10425: *SLiC-HF: Sequence Likelihood Calibration with Human Feedback*
- arXiv 2305.11206: *LIMA: Less Is More for Alignment*
- arXiv 2305.14314: *QLoRA: Efficient Finetuning of Quantized LLMs*
- arXiv 2305.18290: *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*
- arXiv 2305.20050: *Let's Verify Step by Step*
- arXiv 2306.00978: *AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*
- arXiv 2306.05685: *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*
- arXiv 2306.08543: *MiniLLM: On-Policy Distillation of Large Language Models*

- arXiv 2306.10209: *ZeRO++: Extremely Efficient Collective Communication for Giant Model Training*
- arXiv 2306.13649: *On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes*
- arXiv 2307.08691: *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*
- arXiv 2307.09288: *Llama 2: Open Foundation and Fine-Tuned Chat Models*
- arXiv 2307.13854: *WebArena: A Realistic Web Environment for Building Autonomous Agents*
- arXiv 2307.15043: *Universal and Transferable Adversarial Attacks on Aligned Language Models*
- arXiv 2308.01263: *XSTest: A Test Suite for Identifying Exaggerated Safety Behaviours in Large Language Models*
- arXiv 2308.01320: *DeepSpeed-Chat: Easy, Fast and Affordable RLHF Training of ChatGPT-like Models at All Scales*
- arXiv 2308.03825: *"Do Anything Now": Characterizing and Evaluating In-The-Wild Jailbreak Prompts on Large Language Models*
- arXiv 2308.08998: *Reinforced Self-Training (ReST) for Language Modeling*
- arXiv 2309.00267: *RLAIF vs. RLHF: Scaling Reinforcement Learning from Human Feedback with AI Feedback*
- arXiv 2309.06180: *Efficient Memory Management for Large Language Model Serving with PagedAttention*
- arXiv 2309.14509: *DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models*
- arXiv 2310.01889: *Ring Attention with Blockwise Transformers for Near-Infinite Context*
- arXiv 2310.03693: *Fine-tuning Aligned Language Models Compromises Safety, Even When Users Do Not Intend To!*
- arXiv 2310.03716: *A Long Way to Go: Investigating Length Correlations in RLHF*
- arXiv 2310.05914: *NEFTune: Noisy Embeddings Improve Instruction Finetuning*
- arXiv 2310.06452: *Understanding the Effects of RLHF on LLM Generalisation and Diversity*
- arXiv 2310.06770: *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?*
- arXiv 2310.08419: *Jailbreaking Black Box Large Language Models in Twenty Queries*
- arXiv 2310.10505: *ReMax: A Simple, Effective, and Efficient Reinforcement Learning Method for Aligning Large Language Models*
- arXiv 2310.12036: *A General Theoretical Paradigm to Understand Learning from Human Preferences*
- arXiv 2310.12773: *Safe RLHF: Safe Reinforcement Learning from Human Feedback*
- arXiv 2310.13548: *Towards Understanding Sycophancy in Language Models*
- arXiv 2311.04850: *Rethinking Benchmark and Contamination for Language Models with Rephrased Samples*
- arXiv 2311.07911: *Instruction-Following Evaluation for Large Language Models*
- arXiv 2311.12022: *GPQA: A Graduate-Level Google-Proof Q&A Benchmark*
- arXiv 2312.00886: *Nash Learning from Human Feedback*
- arXiv 2312.01552: *The Unlocking Spell on Base LLMs: Rethinking Alignment via In-Context Learning*
- arXiv 2312.03732: *A Rank Stabilization Scaling Factor for Fine-Tuning with LoRA*
- arXiv 2312.06585: *Beyond Human Data: Scaling Self-Training for Problem-Solving with Language*

*Models*
- arXiv 2312.07104: *SGLang: Efficient Execution of Structured Language Model Programs*
- arXiv 2312.08935: *Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations*
- arXiv 2312.09244: *Helping or Herding? Reward Model Ensembles Mitigate but do not Eliminate Reward Hacking*
- arXiv 2312.15685: *What Makes Good Data for Alignment? A Comprehensive Study of Automatic Data Selection in Instruction Tuning*
- arXiv 2401.05566: *Sleeper Agents: Training Deceptive LLMs that Persist Through Safety Training*
- arXiv 2401.08417: *Contrastive Preference Optimization: Pushing the Boundaries of LLM Performance in Machine Translation*
- arXiv 2401.10020: *Self-Rewarding Language Models*
- arXiv 2402.01306: *KTO: Model Alignment as Prospect Theoretic Optimization*
- arXiv 2402.03300: *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*
- arXiv 2402.04249: *HarmBench: A Standardized Evaluation Framework for Automated Red Teaming and Robust Refusal*
- arXiv 2402.04333: *LESS: Selecting Influential Data for Targeted Instruction Tuning*
- arXiv 2402.04792: *Direct Language Model Alignment from Online AI Feedback*
- arXiv 2402.05749: *Generalized Preference Optimization: A Unified Approach to Offline Alignment*
- arXiv 2402.07319: *ODIN: Disentangled Reward Mitigates Hacking in RLHF*
- arXiv 2402.09353: *DoRA: Weight-Decomposed Low-Rank Adaptation*
- arXiv 2402.12354: *LoRA+: Efficient Low Rank Adaptation of Large Models*
- arXiv 2402.14740: *Back to Basics: Revisiting REINFORCE Style Optimization for Learning from Human Feedback in LLMs*
- arXiv 2402.19446: *ArCHer: Training Language Model Agents via Hierarchical Multi-Turn RL*
- arXiv 2403.00409: *Provably Robust DPO: Aligning Language Models with Noisy Feedback*
- arXiv 2403.04132: *Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference*
- arXiv 2403.07691: *ORPO: Monolithic Preference Optimization without Reference Model*
- arXiv 2403.07974: *LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code*
- arXiv 2403.13787: *RewardBench: Evaluating Reward Models for Language Modeling*
- arXiv 2403.17031: *The N+ Implementation Details of RLHF with PPO: A Case Study on TL;DR Summarization*
- arXiv 2403.19159: *Disentangling Length from Quality in Direct Preference Optimization*
- arXiv 2404.01318: *JailbreakBench: An Open Robustness Benchmark for Jailbreaking Large Language Models*
- arXiv 2404.02948: *PiSSA: Principal Singular Values and Singular Vectors Adaptation of Large Language Models*
- arXiv 2404.03715: *Direct Nash Optimization: Teaching Language Models to Self-Improve with General Preferences*
- arXiv 2404.04475: *Length-Controlled AlpacaEval: A Simple Way to Debias Automatic Evaluators*
- arXiv 2404.04656: *Binary Classifier Optimization for Large Language Model Alignment*
- arXiv 2404.07972: *OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments*

- arXiv 2404.10719: *Is DPO Superior to PPO for LLM Alignment? A Comprehensive Study*
- arXiv 2404.12358: *From r to Q*: Your Language Model is Secretly a Q-Function*
- arXiv 2404.18796: *Replacing Judges with Juries: Evaluating LLM Generations with a Panel of Diverse Models*
- arXiv 2404.19733: *Iterative Reasoning Preference Optimization*
- arXiv 2405.00332: *A Careful Examination of Large Language Model Performance on Grade School Arithmetic*
- arXiv 2405.00675: *Self-Play Preference Optimization for Language Model Alignment*
- arXiv 2405.07863: *RLHF Workflow: From Reward Modeling to Online RLHF*
- arXiv 2405.09673: *LoRA Learns Less and Forgets Less*
- arXiv 2405.11143: *OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework*
- arXiv 2405.14734: *SimPO: Simple Preference Optimization with a Reference-Free Reward*
- arXiv 2406.01574: *MMLU-Pro: A More Robust and Challenging Multi-Task Language Understanding Benchmark*
- arXiv 2406.08464: *Magpie: Alignment Data Synthesis from Scratch by Prompting Aligned LLMs with Nothing*
- arXiv 2406.11717: *Refusal in Language Models Is Mediated by a Single Direction*
- arXiv 2406.11827: *WPO: Enhancing RLHF with Weighted Preference Optimization*
- arXiv 2406.12045: *$\tau$-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains*
- arXiv 2406.12845: *Interpretable Preferences via Multi-Objective Reward Modeling and Mixture-of-Experts*
- arXiv 2406.18510: *WildTeaming at Scale: From In-the-Wild Jailbreaks to (Adversarially) Safer Language Models*
- arXiv 2407.08608: *FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision*
- arXiv 2407.14503: *Catastrophic Goodhart: regularizing RLHF with KL divergence does not mitigate heavy-tailed reward misspecification*
- arXiv 2407.19594: *Meta-Rewarding Language Models: Self-Improving Alignment with LLM-as-a-Meta-Judge*
- arXiv 2407.21783: *The Llama 3 Herd of Models*
- arXiv 2408.03314: *Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters*
- arXiv 2408.06266: *Anchored Preference Optimization and Contrastive Revisions: Addressing Underspecification in Alignment*
- arXiv 2408.15240: *Generative Verifiers: Reward Modeling as Next-Token Prediction*
- arXiv 2409.19256: *HybridFlow: A Flexible and Efficient RLHF Framework*
- arXiv 2410.01679: *VinePPO: Refining Credit Assignment in RL Training of LLMs*
- arXiv 2410.10989: *Liger Kernel: Efficient Triton Kernels for LLM Training*
- arXiv 2410.18252: *Asynchronous RLHF: Faster and More Efficient Off-Policy RL for Language Models*
- arXiv 2410.18451: *Skywork-Reward: Bag of Tricks for Reward Modeling in LLMs*
- arXiv 2411.01111: *Rule Based Rewards for Language Model Safety*
- arXiv 2411.15124: *Tulu 3: Pushing Frontiers in Open Language Model Post-Training*
- arXiv 2412.06559: *ProcessBench: Identifying Process Errors in Mathematical Reasoning*
- arXiv 2412.14093: *Alignment faking in large language models*

- arXiv 2412.16339: *Deliberative Alignment: Reasoning Enables Safer Language Models*
- arXiv 2412.21139: *Training Software Engineering Agents and Verifiers with SWE-Gym*
- arXiv 2501.03262: *REINFORCE++: Stabilizing Critic-Free Policy Optimization with Global Advantage Normalization*
- arXiv 2501.04519: *rStar-Math: Small LLMs Can Master Math Reasoning with Self-Evolved Deep Thinking*
- arXiv 2501.07301: *The Lessons of Developing Process Reward Models in Mathematical Reasoning*
- arXiv 2501.12599: *Kimi k1.5: Scaling Reinforcement Learning with LLMs*
- arXiv 2501.12948: *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*
- arXiv 2501.14249: *Humanity's Last Exam*
- arXiv 2501.18837: *Constitutional Classifiers: Defending against Universal Jailbreaks across Thousands of Hours of Red Teaming*
- arXiv 2501.19393: *s1: Simple test-time scaling*
- arXiv 2502.03373: *Demystifying Long Chain-of-Thought Reasoning in LLMs*
- arXiv 2502.03387: *LIMO: Less is More for Reasoning*
- arXiv 2502.17424: *Emergent Misalignment: Narrow finetuning can produce broadly misaligned LLMs*
- arXiv 2502.18449: *SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution*
- arXiv 2503.01491: *What's Behind PPO's Collapse in Long-CoT? Value Optimization Holds the Secret*
- arXiv 2503.09516: *Search-R1: Training LLMs to Reason and Leverage Search Engines with Reinforcement Learning*
- arXiv 2503.11926: *Monitoring Reasoning Models for Misbehavior and the Risks of Promoting Obfuscation*
- arXiv 2503.14476: *DAPO: An Open-Source LLM Reinforcement Learning System at Scale*
- arXiv 2503.18892: *SimpleRL-Zoo: Investigating and Taming Zero Reinforcement Learning for Open Base Models in the Wild*
- arXiv 2503.20783: *Understanding R1-Zero-Like Training: A Critical Perspective*
- arXiv 2503.24290: *Open-Reasoner-Zero: An Open Source Approach to Scaling Up Reinforcement Learning on the Base Model*
- arXiv 2504.05118: *VAPO: Efficient and Reliable Reinforcement Learning for Advanced Reasoning Tasks*
- arXiv 2504.11536: *ReTool: Reinforcement Learning for Strategic Tool Use in LLMs*
- arXiv 2504.13837: *Does Reinforcement Learning Really Incentivize Reasoning Capacity in LLMs Beyond the Base Model?*
- arXiv 2504.16828: *Process Reward Models That Think*
- arXiv 2504.20073: *RAGEN: Understanding Self-Evolution in LLM Agents via Multi-Turn Reinforcement Learning*
- arXiv 2504.20571: *Reinforcement Learning for Reasoning in Large Language Models with One Training Example*
- arXiv 2505.03335: *Absolute Zero: Reinforced Self-play Reasoning with Zero Data*
- arXiv 2505.07291: *INTELLECT-2: A Reasoning Model Trained Through Globally Decentralized Reinforcement Learning*

- arXiv 2505.09388: *Qwen3 Technical Report*
- arXiv 2505.22312: *Skywork Open Reasoner 1 Technical Report*
- arXiv 2505.22617: *The Entropy Mechanism of Reinforcement Learning for Reasoning Language Models*
- arXiv 2505.24298: *AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning*
- arXiv 2505.24864: *ProRL: Prolonged Reinforcement Learning Expands Reasoning Boundaries in Large Language Models*
- arXiv 2506.01937: *RewardBench 2: Advancing Reward Model Evaluation*
- arXiv 2506.01939: *Beyond the 80/20 Rule: High-Entropy Minority Tokens Drive Effective Reinforcement Learning for LLM Reasoning*
- arXiv 2506.04178: *OpenThoughts: Data Recipes for Reasoning Models*
- arXiv 2506.09477: *On a few pitfalls in KL divergence gradient estimation for RL*
- arXiv 2506.10947: *Spurious Rewards: Rethinking Training Signals in RLVR*
- arXiv 2506.13585: *MiniMax-M1: Scaling Test-Time Compute Efficiently with Lightning Attention*
- arXiv 2506.14245: *Reinforcement Learning with Verifiable Rewards Implicitly Incentivizes Correct Reasoning in Base LLMs*
- arXiv 2507.17746: *Rubrics as Rewards: Reinforcement Learning Beyond Verifiable Domains*
- arXiv 2507.18071: *Group Sequence Policy Optimization*
- arXiv 2508.05629: *On the Generalization of SFT: A Reinforcement Learning Perspective with Reward Rectification*
- arXiv 2510.11370: *Stabilizing MoE Reinforcement Learning by Aligning Training and Inference Routers*
- arXiv 2510.13786: *The Art of Scaling Reinforcement Learning Compute for LLMs*
- arXiv 2510.26788: *Defeating the Training-Inference Mismatch via FP16*
- arXiv 2511.20347: *Soft Adaptive Policy Optimization*
- arXiv 2512.02556: *DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models*
- arXiv 2602.05261: *Length-Unbiased Sequence Policy Optimization: Revealing and Controlling Response Length Variation in RLVR*
- arXiv 2602.10693: *VESPO: Variational Sequence-Level Soft Policy Optimization for Stable Off-Policy LLM Training*