Generative Models in Finance

Week 2: Reinforcement Learning Training of LLMs

Cristopher Salvi
Imperial College London

Spring 2026

Overview

1. From Pre-training to Fine-Tuning
2. Reinforcement Learning Foundations for LLMs
3. Proximal Policy Optimisation (PPO)
4. Group Relative Policy Optimisation (GRPO) and Direct Preference Optimisation (DPO)
5. The Reinforcement Learning from Human Feedback (RLHF) Pipeline and Frontiers
6. RL for Mathematical Reasoning

Reference: R. Patel, *Understanding Reinforcement Learning for Model Training, and Future Directions with GRAPE*, `references/llm_training.pdf`, 2025.

Part 1: From Pre-training to Fine-Tuning

- Recall the standard LLM training pipeline:
  1. Pre-training: next-token prediction on a large text corpus, yielding a base model π_{base}
  2. Supervised Fine-Tuning (SFT): adapt the base model on curated (prompt, response) pairs to produce π_{SFT}
  3. RLHF / Preference Alignment: further optimise π_{SFT} using human (or AI) preference feedback
- We will cover all three stages: pre-training, fine-tuning, and alignment via reinforcement learning
- Throughout, we denote the policy (i.e. the language model) by π_θ, parameterised by $\theta \in \mathbb{R}^d$

What is Pre-training?

- Pre-training is the first and most expensive stage of LLM development
- The model learns from a massive corpus of text in a self-supervised fashion: no human labels are required
- The learning signal comes from the data itself — specifically, from the task of next-token prediction:

$$\text{Given context } (x_1, \ldots, x_{t-1}), \quad \text{predict } x_t$$

- The resulting model π_{base} is a base model: it models $P(x_t \mid x_{<t})$ and can sample continuations, but it has not been trained to condition on instructions or produce structured responses
- Pre-training determines the support of the learned distribution; it encodes the statistical regularities of the training corpus
- Scale: modern base models are trained on $\sim 10^{13}$ tokens using $\sim 10^4$ GPUs for $\sim 10^{24}$ FLOPs

The Pre-training Objective

- The pre-training loss (Radford et al., 2018) is the cross-entropy (equivalently, negative log-likelihood) over the training corpus $\mathcal{C} = (x_1, x_2, \ldots, x_N)$:

$$\mathcal{L}_{\text{PT}}(\theta) = -\frac{1}{N} \sum_{t=1}^{N} \log \pi_\theta(x_t \mid x_1, \ldots, x_{t-1}) \tag{2.1}$$

- This is equivalent to maximum likelihood estimation (MLE): we seek θ that maximises the probability of the observed corpus under the model
- Connection to information theory: minimising (2.1) is equivalent to minimising the KL divergence $D_{\text{KL}}(P_{\text{data}} \| \pi_\theta)$. Indeed:

$$D_{\text{KL}}(P_{\text{data}} \| \pi_\theta) = \mathbb{E}_{P_{\text{data}}}[\log P_{\text{data}}(x_t | x_{<t}) - \log \pi_\theta(x_t | x_{<t})] = \underbrace{H(P_{\text{data}})}_{\text{const. in } \theta} + \mathcal{L}_{\text{PT}}(\theta)$$

  Since $H(P_{\text{data}})$ does not depend on θ, $\arg\min_\theta D_{\text{KL}} = \arg\min_\theta \mathcal{L}_{\text{PT}}$

- Teacher forcing and causal masking: at each position t, the model is conditioned on the *true* preceding tokens $(x_1, \ldots, x_{t-1})$, not on its own predictions. Because the ground-truth tokens are known at training time, the causal attention mask $M_{ij} = \mathbf{1}[j \leq i]$ allows the Transformer to compute $\pi_\theta(x_t \mid x_{<t})$ for all $t = 1, \ldots, N$ in parallel, yielding $O(N)$ loss terms from a single $O(N^2 d)$ forward pass

Radford, A., Narasimhan, K., Salimans, T. & Sutskever, I. (2018). Improving Language Understanding by Generative Pre-Training.

Pre-training: Data and Scale

- Pre-training corpora are drawn from diverse web-scale sources:
  - Common Crawl: petabytes of raw web text (requires heavy filtering)
  - Wikipedia, books, code repositories (GitHub), scientific papers (arXiv)
  - Proprietary data for commercial models
- Data quality pipeline: raw text $\rightarrow$ language filtering $\rightarrow$ deduplication $\rightarrow$ quality scoring $\rightarrow$ toxicity filtering
- Tokenisation: recall from Week 1 that Byte Pair Encoding (BPE) converts raw text into subword tokens with $|\mathcal{V}| \approx 32{,}000$–$128{,}000$
- Scaling laws (Kaplan et al., 2020; Hoffmann et al., 2022): the pre-training loss decreases predictably as a power law in:
  - Model size (number of parameters)
  - Dataset size (number of tokens)
  - Compute budget (FLOPs)
- Chinchilla scaling (Hoffmann et al., 2022): for compute-optimal training, the number of tokens D should scale linearly with the number of parameters N, i.e. $D \propto N$

From Base Model to Assistant

- A pre-trained base model π_{base} is a text completion engine: given a prefix, it generates a plausible continuation
- Problem: base models do not naturally follow instructions
  - Input: "What is the capital of France?"
  - Base model output: "What is the capital of Germany? What is the capital of Spain? . . . " (continues the pattern of questions)
- An assistant model should instead respond: "The capital of France is Paris."
- The gap between base model behaviour and desired assistant behaviour motivates fine-tuning:
  1. Supervised Fine-Tuning (SFT): teach the model the format and style of helpful responses using demonstration data
  2. Reinforcement Learning from Human Feedback (RLHF): teach the model to distinguish good from bad responses using preference feedback
- The base model already *has* the knowledge (from pre-training); fine-tuning teaches it *when and how* to use that knowledge

The SFT Objective

- Let $\mathcal{D}_{\text{SFT}} = \{(x_q, y_q)\}_{q=1}^{Q}$ be a dataset of Q prompt-response pairs, where each prompt $x_q = (x_{q,1}, \ldots, x_{q,S_q})$ is a token sequence of length S_q and each response $y_q = (y_{q,1}, \ldots, y_{q,T_q})$ is a token sequence of length T_q
- The SFT loss is the conditional negative log-likelihood over *response* tokens only (with teacher forcing as in pre-training):

$$\mathcal{L}_{\text{SFT}}(\theta) = -\frac{1}{Q}\sum_{q=1}^{Q}\frac{1}{T_q}\sum_{t=1}^{T_q}\log \pi_\theta(y_{q,t} \mid x_q, y_{q,<t}) \tag{2.2}$$

  where $y_{q,<t} = (y_{q,1}, \ldots, y_{q,t-1})$ is the ground-truth prefix. The prompt tokens x_q appear in the conditioning but are *not* included in the loss (loss masking)
- Data quality: SFT performance is highly sensitive to the quality of (x_q, y_q) pairs
  - Diversity: prompts should cover a wide range of tasks (QA, summarisation, coding, maths, etc.)
  - Quality: responses should be expert-written, accurate, and well-formatted
  - Quantity: a relatively small number of high-quality examples ($Q \sim 10^3$–10^5) can be effective (Zhou et al., 2023)

Low-Rank Structure of Fine-Tuning Updates

- Let $W_0 \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ be a pre-trained weight matrix and W_{ft} the same matrix after full fine-tuning. Define the update $\Delta W = W_{\text{ft}} - W_0$
- The singular value decomposition (SVD) of ΔW is:

$$\Delta W = U \Sigma V^\top = \sum_{i=1}^{\min(d_{\text{out}}, d_{\text{in}})} \sigma_i \, \mathbf{u}_i \mathbf{v}_i^\top$$

  where $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ are the singular values
- Aghajanyan et al. (2021) observed that for fine-tuning on downstream tasks, the singular values σ_i decay rapidly. The effective rank

$$r_{\text{eff}}(\Delta W) = \frac{\left(\sum_i \sigma_i\right)^2}{\sum_i \sigma_i^2} = \frac{\|\Delta W\|_*^2}{\|\Delta W\|_F^2}$$

  satisfies $r_{\text{eff}} \ll \min(d_{\text{out}}, d_{\text{in}})$. For GPT-3 175B, $r_{\text{eff}} \leq 10$ for most weight matrices
- So the best rank-r approximation $\Delta W_r = U_r \Sigma_r V_r^\top$ (by Eckart–Young) captures most of the fine-tuning signal for small r
- This motivates directly *parametrising* ΔW as a rank-r matrix during training $\Rightarrow$ LoRA

Aghajanyan, A. et al. (2021). Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning. *ACL*.

LoRA: Formulation

Definition 2.1 (Low-Rank Adaptation, LoRA (Hu et al., 2022))

Given a pre-trained weight matrix $W_0 \in \mathbb{R}^{d_{out} \times d_{in}}$ and input $x \in \mathbb{R}^{d_{in}}$, the adapted forward pass is:

$$h = W_0 x + \frac{\alpha}{r} BAx \tag{2.3}$$

where $B \in \mathbb{R}^{d_{out} \times r}$, $A \in \mathbb{R}^{r \times d_{in}}$, $r \ll \min(d_{out}, d_{in})$, and $\alpha > 0$ is a fixed scaling hyperparameter.

- W_0 is frozen; only (B, A) receive gradients. The effective update is $\Delta W = \frac{\alpha}{r} BA \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ with $\text{rank}(\Delta W) \leq r$
- Trainable parameters per matrix: $r(d_{\text{out}} + d_{\text{in}})$ instead of $d_{\text{out}} \cdot d_{\text{in}}$. The compression ratio is:

$$\frac{d_{\text{out}} \cdot d_{\text{in}}}{r(d_{\text{out}} + d_{\text{in}})} = \frac{d}{2r} \quad (\text{when } d_{\text{out}} = d_{\text{in}} = d)$$

For $d = 4096$, $r = 16$: ratio $= 128\times$

Hu, E. J. et al. (2022). LoRA: Low-Rank Adaptation of Large Language Models. *ICLR*.

LoRA: Scaling and Expressiveness

- Scaling factor α/r: if B_{ik}, A_{kj} are independent mean-0, variance-σ^2, then:

$$\mathbb{E}[\|BA\|_F^2] = \sum_{i,j} \sum_{k=1}^{r} \mathbb{E}[B_{ik}^2]\,\mathbb{E}[A_{kj}^2] = d_{\text{out}}\, d_{\text{in}}\, r\, \sigma^4$$

  so $\|BA\|_F = \Theta(\sqrt{r})$ in r. Hence $\frac{\alpha}{r}\|BA\|_F = O(\alpha/\sqrt{r})$; setting $\alpha = r$ gives $O(\sqrt{r})$, $\alpha = \sqrt{r}$ gives $O(1)$

- When $r = \min(d_{\text{out}}, d_{\text{in}})$, every $\Delta W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ satisfies $\text{rank}(\Delta W) \leq r$, so writing $\Delta W = U\Sigma V^\top$ and setting $B = U\Sigma$, $A = V^\top$ recovers $\Delta W = BA$. LoRA then spans the full space and reduces to unconstrained fine-tuning

LoRA: Initialisation, Gradients, and Memory

- Initialisation: $A_{ij} \sim \mathcal{N}(0, \sigma^2)$, $B = 0$, so $\Delta W\big|_{t=0} = BA = 0$. Training begins from the pre-trained model exactly; the first gradient step on B is:

$$\nabla_B \mathcal{L} = \frac{\alpha}{r} \nabla_h \mathcal{L} \cdot (Ax)^\top, \qquad \nabla_A \mathcal{L} = \frac{\alpha}{r} B^\top \nabla_h \mathcal{L} \cdot x^\top$$

  Since $B = 0$ initially, $\nabla_A \mathcal{L}\big|_{t=0} = 0$: only B is updated in the first step

- Training memory: for each adapted matrix, the optimizer (Adam) stores first and second moment estimates for B and A only. Per matrix:

$$\text{Full fine-tuning: } 2 \cdot d_{\text{out}} \cdot d_{\text{in}} \quad \text{(Adam states)}$$
$$\text{LoRA: } 2 \cdot r(d_{\text{out}} + d_{\text{in}}) \quad \text{(Adam states)}$$

  For a 70B-parameter model with $r = 16$, this reduces optimizer memory from $\sim$560 GB to $\sim$4 GB (in fp32)

- Which matrices to adapt: applying LoRA to all linear projections $(W_Q, W_K, W_V, W_O, W_1, W_2)$ in each of L layers yields $6L$ adapter pairs. Total trainable parameters: $6L \cdot r(d + d) = 12Lrd$

- Multi-task serving: given frozen W_0, different tasks use adapters (B_i, A_i). Swapping requires loading $12Lrd$ parameters; for $L = 80$, $d = 8192$, $r = 16$: $\sim$126M parameters ($< 0.2\%$ of 70B)

Why Not Just SFT?

- SFT teaches the model to *imitate* a fixed dataset of expert responses
- But imitation has fundamental limits:
  - Ceiling effect: the model can be *at most* as good as the demonstrations it was trained on
  - No self-improvement: SFT cannot discover response strategies that are *better* than anything in the training data
  - Quality is expensive: writing thousands of expert-quality responses requires significant human effort
- A better approach: instead of showing the model what a good answer looks like, *let the model try many answers and tell it which ones are better*
- This is the core idea of reinforcement learning (RL): the model learns from *trial and error*, guided by a reward signal
- Think of it this way: SFT is like learning to cook by following recipes exactly, while RL is like experimenting in the kitchen and having a food critic rate your dishes — you can eventually surpass any recipe book

Part 2: Reinforcement Learning Foundations

Goal: build the mathematical framework of reinforcement learning (RL) from scratch and specialise it to LLMs. No prior RL knowledge is assumed.

- What is reinforcement learning?
- Markov Decision Processes (MDPs)
- Policies, value functions, and the advantage function
- The policy gradient theorem and REINFORCE
- Generalised Advantage Estimation (GAE)
- Specialisation to LLMs: the KL-constrained objective

What is Reinforcement Learning? — The Idea

- Imagine training a dog. You cannot show the dog a "correct walk" to imitate (that would be supervised learning). Instead, you let the dog try different behaviours and give it a treat when it does something good. Over time, the dog learns which behaviours lead to treats
- In the LLM setting: the model generates a response (a sequence of actions), and then receives a score (reward) indicating how good the response was. Over many trials, it learns to generate higher-scoring responses
- The key elements:
  1. An agent (the model π_θ) takes actions (generates tokens) in an environment
  2. After completing a sequence of actions, the agent receives a scalar reward $R \in \mathbb{R}$
  3. The goal is to find parameters θ that maximise the expected cumulative reward:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=1}^{T} r_t\right]$$

  where $\tau = (s_1, a_1, r_1, \ldots, s_T, a_T, r_T)$ is a trajectory (a full episode of interaction)
- Unlike SFT, where the loss compares the model's output directly to a target y_q, in RL there is *no target* — just a scalar reward that says "how well did you do overall?" The model must *explore* different actions to discover which ones lead to high reward

Markov Decision Processes

Definition 2.2 (Markov Decision Process (MDP))

An MDP is a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma, T)$ *where:*

- $\mathcal{S}$ *is the state space*
- $\mathcal{A}$ *is the action space*
- $P(s' \mid s, a)$ *is the transition kernel: probability of moving to state* s' *given state* s *and action* a
- $R(s, a) \in \mathbb{R}$ *is the reward function*
- $\gamma \in [0, 1]$ *is the discount factor*
- T *is the horizon (episode length)*

A policy $\pi(a \mid s)$ *is a conditional distribution over actions given states. The agent's goal is to find a policy* π^* *that maximises* $J(\pi) = \mathbb{E}_\pi\left[\sum_{t=1}^{T} \gamma^{t-1} r_t\right]$.

- In plain English: the agent is in some situation (state), picks an action, receives a reward, and moves to a new situation. The policy is its decision-making rule. The discount factor γ controls how much the agent cares about future vs. immediate rewards ($\gamma = 1$: equal weight; $\gamma \to 0$: myopic)
- The Markov property: $P(s_{t+1} \mid s_1, a_1, \ldots, s_t, a_t) = P(s_{t+1} \mid s_t, a_t)$ — the future depends on the present state and action only, not on the full history

LLM Text Generation as an MDP

- Let us specialise the MDP framework to autoregressive text generation:
  - State at time t: $s_t = (x, y_1, \ldots, y_{t-1}) \in \mathcal{V}^*$ (prompt x concatenated with tokens generated so far)
  - Action at time t: $a_t = y_t \in \mathcal{V}$ (next token chosen from the vocabulary)
  - Policy: $\pi_\theta(a_t \mid s_t) = \pi_\theta(y_t \mid x, y_{<t})$ (the language model's conditional distribution)
  - Transition: deterministic concatenation; $s_{t+1} = (s_t, a_t) = (x, y_1, \ldots, y_t)$
  - Reward: typically sparse and terminal; $r_t = 0$ for $t < T$ and $r_T = R(x, y)$ where R is a reward model scoring the complete response $y = (y_1, \ldots, y_T)$
  - Discount: $\gamma = 1$ (undiscounted, since episodes are finite)
- The horizon T is the response length; the episode terminates when $y_T =$ <eos>
- Notice that the transition is deterministic and the state grows by one token per step — all stochasticity comes from the policy π_θ itself. This is a much simpler MDP than typical RL environments (robotics, games)

LLM as MDP: A Concrete Walkthrough

- Prompt: "What is 2+3?" (tokens: $x = [\text{What}, \text{is}, 2, +, 3, ?]$)
- Step-by-step episode:
  1. $s_1 = [\text{What}, \text{is}, 2, +, 3, ?]$, policy picks $a_1 =$ "The" $(r_1 = 0)$
  2. $s_2 = [\ldots, ?, \text{The}]$, policy picks $a_2 =$ "answer" $(r_2 = 0)$
  3. $s_3 = [\ldots, \text{The}, \text{answer}]$, policy picks $a_3 =$ "is" $(r_3 = 0)$
  4. $s_4 = [\ldots, \text{is}]$, policy picks $a_4 =$ "5" $(r_4 = 0)$
  5. $s_5 = [\ldots, 5]$, policy picks $a_5 =$ <eos> $(r_5 = R(x, y) = +1$ correct!$)$
- Key observations:
  - Reward is sparse: $r_t = 0$ for all intermediate tokens; only the final token triggers a score
  - The state simply grows by one token at each step (deterministic transitions)
  - The challenge: the model must learn that choosing "5" at step 4 (rather than "4" or "6") is what made the response correct — this is the credit assignment problem

Reward Hacking and the Need for Regularisation

- We have framed LLM generation as an MDP, and the natural objective is to maximise expected reward. But a naive approach fails
- Consider the unconstrained objective $\max_\theta \mathbb{E}_{x\sim\mathcal{D}}\,\mathbb{E}_{y\sim\pi_\theta(\cdot|x)}[R(x,y)]$
- Since R is a learned approximation $R_\psi \approx R^*$, the optimal policy $\pi^*_\psi = \arg\max_\pi \mathbb{E}[R_\psi(x,y)]$ may differ substantially from $\arg\max_\pi \mathbb{E}[R^*(x,y)]$
- In particular, π^*_ψ concentrates mass on regions where R_ψ overestimates R^* — this is reward hacking
- This is an instance of Goodhart's law (Goodhart, 1975): optimising a proxy measure R_ψ causes it to diverge from the quantity of interest R^*
- Solution: constrain the policy to remain close to a reference π_{ref} (typically π_{SFT}), so that π_θ cannot move into regions where R_ψ is unreliable

Goodhart, C. A. E. (1975). Problems of Monetary Management: The U.K. Experience. *Papers in Monetary Economics*, Reserve Bank of Australia.

The KL-Constrained RL Objective

- We want to maximise reward but *not stray too far* from the SFT model π_{ref}. The KL divergence $D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})$ measures how different the current policy is from the reference, so we add it as a penalty
- The KL-regularised objective adds a divergence penalty to the reward:

$$\max_\theta \; \mathbb{E}_{x\sim\mathcal{D}} \, \mathbb{E}_{y\sim\pi_\theta(\cdot|x)} \Big[R(x,y) - \beta \, D_{\text{KL}}\big(\pi_\theta(\cdot|x) \,\|\, \pi_{\text{ref}}(\cdot|x)\big) \Big] \tag{2.4}$$

- The coefficient $\beta > 0$ controls the regularisation strength: large β keeps $\pi_\theta \approx \pi_{\text{ref}}$; small β allows larger deviations
- Per-token decomposition: both π_θ and π_{ref} factorise autoregressively: $\pi(y|x) = \prod_{t=1}^{T} \pi(y_t|s_t)$. Therefore $\log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)} = \sum_{t=1}^{T} \log \frac{\pi_\theta(y_t|s_t)}{\pi_{\text{ref}}(y_t|s_t)}$, and taking expectations:

$$D_{\text{KL}}\big(\pi_\theta(\cdot|x) \,\|\, \pi_{\text{ref}}(\cdot|x)\big) = \mathbb{E}_{y\sim\pi_\theta}\left[\log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}\right] = \mathbb{E}_{y\sim\pi_\theta}\left[\sum_{t=1}^{T} \log \frac{\pi_\theta(y_t|s_t)}{\pi_{\text{ref}}(y_t|s_t)}\right]$$

- This per-token form defines an effective per-token reward:

$$\tilde{r}_t = -\beta \log \frac{\pi_\theta(y_t|s_t)}{\pi_{\text{ref}}(y_t|s_t)}, \quad t < T; \qquad \tilde{r}_T = R(x,y) - \beta \log \frac{\pi_\theta(y_T|s_T)}{\pi_{\text{ref}}(y_T|s_T)}$$

so the KL-regularised problem reduces to a standard RL problem with shaped rewards $\tilde{r}_t$

The Optimisation Challenge

- We want to maximise the expected return:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=1}^{T} r_t\right] = \sum_{\tau} p_\theta(\tau)\, R(\tau)$$

  where $\tau = (s_1, a_1, r_1, \ldots, s_T, a_T, r_T)$ is a trajectory sampled by rolling out π_θ, and $R(\tau) = \sum_{t=1}^{T} r_t$ is its total reward

- The trajectory probability is $p_\theta(\tau) = \prod_{t=1}^{T} \pi_\theta(a_t \mid s_t) \cdot P(s_{t+1} \mid s_t, a_t)$. In the LLM setting, the transitions are deterministic ($s_{t+1} = (s_t, a_t)$), so $p_\theta(\tau) = \prod_{t=1}^{T} \pi_\theta(a_t \mid s_t)$
- Problem: $J(\theta)$ is an expectation over *discrete* sequences $\tau \in \mathcal{V}^T$. We cannot compute $\nabla_\theta J(\theta)$ by backpropagating through the sampling operation (sampling is not differentiable)
- In SFT, the loss is a smooth function of the model's output logits, so standard backpropagation works. In RL, the reward depends on the *sampled tokens*, and the sampling operation ("pick token y_t from the distribution $\pi_\theta(\cdot|s_t)$") is not differentiable — we cannot compute $\partial y_t / \partial \theta$
- The policy gradient theorem (Williams, 1992) gets around this: it computes $\nabla_\theta J(\theta)$ from sampled trajectories without differentiating through the sampling step

The Log-Derivative Trick

- The key identity is the log-derivative trick (also called the score function estimator). For any differentiable $p_\theta(\tau) > 0$:

$$\nabla_\theta p_\theta(\tau) = p_\theta(\tau) \cdot \nabla_\theta \log p_\theta(\tau)$$

  This follows from $\nabla_\theta \log p_\theta(\tau) = \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)}$

- Applying this to $J(\theta)$:

$$\begin{aligned} \nabla_\theta J(\theta) &= \sum_\tau \nabla_\theta p_\theta(\tau)\, R(\tau) = \sum_\tau p_\theta(\tau)\, \nabla_\theta \log p_\theta(\tau)\, R(\tau) \\ &= \mathbb{E}_{\tau \sim p_\theta}[\nabla_\theta \log p_\theta(\tau)\, R(\tau)] \end{aligned} \tag{2.5}$$

- The gradient is now an *expectation* under p_θ — we can estimate it by sampling trajectories from π_θ
- So to compute the gradient: (1) generate several responses from the model, (2) score each one, (3) for each response, compute $\nabla_\theta \log p_\theta(\tau)$ (which *is* differentiable — it just involves the model's log-probabilities), and (4) weight it by the reward. We never differentiate through the sampling step itself
- $R(\tau)$ is treated as a scalar weight; we never differentiate through R or through the sampling of τ. The only gradient is $\nabla_\theta \log p_\theta(\tau)$, which is differentiable since $\log p_\theta(\tau) = \sum_t \log \pi_\theta(a_t \mid s_t)$

Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8, 229–256.

The Policy Gradient Theorem

- Substituting $\log p_\theta(\tau) = \sum_{t=1}^{T} \log \pi_\theta(a_t \mid s_t)$ into (2.5):

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta}\left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right) R(\tau)\right]$$

- Causality argument: the term $\nabla_\theta \log \pi_\theta(a_t \mid s_t)$ at time t is multiplied by the *full* return $R(\tau) = \sum_{t'=1}^{T} r_{t'}$. But for $t' < t$, the reward $r_{t'} = R(s_{t'}, a_{t'})$ is determined before a_t is sampled, so $r_{t'}$ is constant with respect to a_t given s_t. Therefore:

$$\mathbb{E}_{a_t \sim \pi_\theta(\cdot|s_t)}\big[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot r_{t'}\big] = r_{t'} \underbrace{\nabla_\theta \sum_{a_t} \pi_\theta(a_t|s_t)}_{=1} = 0$$

- Dropping these zero-expectation terms yields the REINFORCE gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, G_t\right] \tag{2.6}$$

  where $G_t = \sum_{t'=t}^{T} r_{t'}$ is the return-to-go from step t

- The intuition is simple: increase the log-probability of action a_t in proportion to how much future reward G_t followed
  - If a token was followed by high total reward ⇒ make that token *more likely* next time
  - If a token was followed by low total reward ⇒ make that token *less likely* next time
  - This is trial-and-error learning: actions that led to good outcomes are reinforced

The REINFORCE Algorithm

- REINFORCE (Williams, 1992) is the simplest policy gradient algorithm. It estimates (2.6) via Monte Carlo sampling:
  1. Sample a trajectory $\tau = (s_1, a_1, r_1, \ldots, s_T, a_T, r_T)$ by rolling out π_θ
  2. Compute the return-to-go $G_t = \sum_{t'=t}^{T} r_{t'}$ for each $t = 1, \ldots, T$
  3. Compute the gradient estimate: $\hat{g} = \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, G_t$
  4. Update parameters: $\theta \leftarrow \theta + \alpha\, \hat{g}$
- Unbiasedness: $\mathbb{E}[\hat{g}] = \nabla_\theta J(\theta)$ by construction from (2.6)
- High variance: the estimator $\hat{g}$ has variance $\mathrm{Var}(\hat{g}) = \mathrm{Var}\big[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)\, G_t\big]$, which grows with the horizon T and the stochasticity of π_θ. For LLMs with $T \sim 10^2$–10^3 tokens and $|\mathcal{V}| \sim 10^5$, this variance is prohibitively large
- If the gradient estimate fluctuates wildly from sample to sample, the parameter updates "jump around" rather than moving steadily toward a good policy. Training becomes slow and unstable
- This motivates (i) variance reduction via baselines and advantage estimation, and (ii) constrained updates via PPO (Part 3)

Variance Reduction with Baselines

- REINFORCE weights each token's gradient by the total future reward G_t. But G_t can be large even for "average" actions — what matters is whether G_t is *above or below* what we would typically expect from state s_t. Subtracting a baseline ("what we normally get from this state") centres the signal and reduces noise
- Formally: we subtract $b(s_t)$ from the return:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\,\big(G_t - b(s_t)\big)\right] \tag{2.7}$$

- Unbiasedness: for any $b(s_t)$ depending only on s_t (not on a_t), the subtraction does not introduce bias. Proof:

$$\mathbb{E}_{a_t\sim\pi_\theta(\cdot|s_t)}\big[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\, b(s_t)\big] = b(s_t)\,\nabla_\theta \underbrace{\sum_{a_t} \pi_\theta(a_t \mid s_t)}_{=1} = 0$$

- Optimal baseline: write $g_t = \nabla_\theta \log \pi_\theta(a_t|s_t)$. Minimising the second moment $\mathbb{E}_{a_t}[\|g_t\|^2(G_t - b)^2]$ w.r.t. b:

$$\frac{\partial}{\partial b}\,\mathbb{E}_{a_t}\big[\|g_t\|^2(G_t - b)^2\big] = -2\,\mathbb{E}_{a_t}\big[\|g_t\|^2(G_t - b)\big] \overset{!}{=} 0 \quad\Longrightarrow\quad b^*(s_t) = \frac{\mathbb{E}_{a_t}[\|g_t\|^2 G_t]}{\mathbb{E}_{a_t}[\|g_t\|^2]}$$

When $\|g_t\|^2$ is approximately constant across actions, $b^*(s_t) \approx \mathbb{E}_\pi[G_t \mid s_t] = V^\pi(s_t)$

Value Functions and the Advantage

Definition 2.3 (Value, Action-Value, and Advantage Functions)

For a policy π:

- *State-value:* $V^\pi(s) = \mathbb{E}_\pi\left[\sum_{t'=t}^{T} r_{t'} \mid s_t = s\right]$ *("how good is this state on average?")*
- *Action-value:* $Q^\pi(s,a) = \mathbb{E}_\pi\left[\sum_{t'=t}^{T} r_{t'} \mid s_t = s, a_t = a\right]$ *("how good is taking action a in state s?")*
- *Advantage:* $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$ *("how much better is action a compared to the average action in state s?")*

- Relationships: by definition $V^\pi(s) = \mathbb{E}_{a\sim\pi(\cdot|s)}[Q^\pi(s,a)]$, so $\mathbb{E}_{a\sim\pi(\cdot|s)}[A^\pi(s,a)] = \mathbb{E}_{a\sim\pi}[Q^\pi(s,a)] - V^\pi(s) = V^\pi(s) - V^\pi(s) = 0$
- Advantage form of the policy gradient: substituting $b(s_t) = V^\pi(s_t)$ into (2.7) gives:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, A^{\pi_\theta}(s_t, a_t)\right]$$

since $\mathbb{E}[G_t \mid s_t, a_t] = Q^\pi(s_t, a_t)$ by definition of Q^π, so $\mathbb{E}[G_t - V^\pi(s_t) \mid s_t, a_t] = A^\pi(s_t, a_t)$

- The gradient is now weighted by A^π: tokens with $A^\pi > 0$ (better than average under π) have their probability increased; tokens with $A^\pi < 0$ have their probability decreased. Tokens with $A^\pi \approx 0$ contribute negligible gradient — this is the variance reduction mechanism

Estimating the Advantage: Temporal Difference Residual

- Computing $A^\pi(s_t, a_t)$ requires knowing $V^\pi(s_t)$ — the expected total reward from state s_t — but this is unknown
- We learn a parametric approximation $V_\phi(s) \approx V^\pi(s)$ (the critic, trained by regression on observed returns)
- The temporal difference (TD) residual provides a one-step estimate of the advantage:

$$\delta_t = r_t + \gamma\, V_\phi(s_{t+1}) - V_\phi(s_t) \tag{2.8}$$

  where $\gamma \in [0, 1]$ is the discount factor ($\gamma = 1$ in the undiscounted LLM setting)
- To see why δ_t estimates A^π, recall the *Bellman equation*, obtained by splitting the sum defining V^π:

$$V^\pi(s_t) = \mathbb{E}_\pi\Big[\sum_{t' \geq t} \gamma^{t'-t} r_{t'} \mid s_t\Big] = \mathbb{E}_\pi\big[r_t + \gamma\, V^\pi(s_{t+1}) \mid s_t\big]$$

  Therefore:

$$A^\pi(s_t, a_t) = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$$

  Replacing V^π with V_ϕ gives $\delta_t = A^\pi(s_t, a_t) + \gamma\,(V_\phi(s_{t+1}) - V^\pi(s_{t+1})) - (V_\phi(s_t) - V^\pi(s_t))$
- The approximation error depends on $V_\phi - V^\pi$. This gives a bias–variance trade-off:
  - δ_t has low variance (one-step bootstrap, no sum over future randomness)
  - δ_t has bias proportional to $\|V_\phi - V^\pi\|$

Generalised Advantage Estimation (GAE)

- We need to estimate $A^{\pi}(s_t, a_t)$, but we face two extreme options: (1) use only one step of lookahead (low variance but biased), or (2) use the entire remaining trajectory (unbiased but very noisy). GAE gives a smooth dial between these extremes
- Schulman et al. (2016) introduced GAE to interpolate between bias and variance in advantage estimation
- The GAE estimator is defined as:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{\ell=0}^{T-t} (\gamma\lambda)^{\ell}\, \delta_{t+\ell} \tag{2.9}$$

  where $\lambda \in [0,1]$ and $\delta_{t+\ell}$ is the TD residual (2.8)

- Expanding (2.9):

$$\hat{A}_t^{\mathrm{GAE}} = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \cdots$$

  This is an exponentially-weighted average of multi-step advantage estimates

- The parameter λ controls the bias–variance trade-off:
  - $\lambda = 0$: $\hat{A}_t = \delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$ (low variance, high bias)
  - $\lambda = 1$: $\hat{A}_t = \sum_\ell \delta_{t+\ell} = \sum_\ell r_{t+\ell} + \underbrace{V_\phi(s_{T+1})}_{=0} - V_\phi(s_t) = G_t - V_\phi(s_t)$ (telescoping, γ=1) (high variance, low bias)

Schulman, J., Moritz, P., Levine, S., Jordan, M. & Abbeel, P. (2016). High-Dimensional Continuous Control Using Generalized Advantage Estimation. *ICLR*.

Bias–Variance Trade-off in Advantage Estimation

- GAE interpolates between two limiting cases. Setting $\gamma = 1$ (undiscounted, as in LLM-RLHF):

	$\lambda = 0$ (TD)	$\lambda = 1$ (MC)	$0 < \lambda < 1$ (GAE)
Estimator	$\delta_t = r_t + V_\phi(s_{t+1}) - V_\phi(s_t)$	$G_t - V_\phi(s_t)$	$\sum_\ell \lambda^\ell \delta_{t+\ell}$
Bias	$O(\|V_\phi - V^\pi\|)$	0 (unbiased)	$O(\lambda\|V_\phi - V^\pi\|)$
Variance	$O(\mathrm{Var}(r_t))$	$O(\sum_t \mathrm{Var}(r_t))$	Interpolated

- $\lambda = 0$: $\hat{A}_t = \delta_t$. Depends on V_ϕ for bootstrapping — low variance (single-step) but biased if V_ϕ is inaccurate
- $\lambda = 1$: $\hat{A}_t = \sum_{\ell \geq 0} \delta_{t+\ell} = G_t - V_\phi(s_t)$. Uses the full empirical return — unbiased (up to baseline) but variance grows with T
- LLM-RLHF: $\lambda \in [0.95, 0.99]$, $\gamma = 1$. High λ is needed because LLM episodes are long ($T \sim 10^2$–10^3) and the critic V_ϕ may be inaccurate early in training, so low bias is preferred over low variance

Putting It Together: The LLM RL Objective

- So far we have (1) framed text generation as an MDP, (2) derived the policy gradient (REINFORCE), (3) introduced advantage estimation (GAE) for variance reduction. Now we combine everything
- Recall from the KL-constrained objective that the effective per-token reward is:

$$\tilde{r}_t = \begin{cases} -\beta \log \frac{\pi_\theta(y_t|s_t)}{\pi_{\text{ref}}(y_t|s_t)} & t < T \\ R(x,y) - \beta \log \frac{\pi_\theta(y_T|s_T)}{\pi_{\text{ref}}(y_T|s_T)} & t = T \end{cases}$$

- Substituting into the advantage-based policy gradient with GAE:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{x,y\sim\pi_\theta}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(y_t|s_t)\, \hat{A}_t^{\text{GAE}}\right]$$

  where $\hat{A}_t^{\text{GAE}} = \sum_{\ell=0}^{T-t} \lambda^\ell \tilde{\delta}_{t+\ell}$ and $\tilde{\delta}_t = \tilde{r}_t + V_\phi(s_{t+1}) - V_\phi(s_t)$
- This is a complete specification of the policy gradient for LLM-RLHF. However, using this gradient directly (as in REINFORCE) leads to high variance and unstable updates
- Part 3: PPO addresses stability via *clipped* importance-weighted updates. Part 4: GRPO eliminates the critic V_ϕ via group-normalised advantages

Part 3: Proximal Policy Optimisation (PPO)

- We now have the tools to train LLMs with RL (policy gradient + GAE), but there is a practical problem: training instability
- PPO (Schulman et al., 2017) is the principal algorithm used for RLHF in models such as ChatGPT (Ouyang et al., 2022)
- Core idea: when updating the model, *don't change too much at once*. PPO constrains each update by clipping the probability ratio between the new and old policies
- Like tuning a radio — if you turn the dial too aggressively, you overshoot the station. PPO ensures you only turn the dial a little at each step
- PPO avoids the computational complexity of Trust Region Policy Optimisation (TRPO) while achieving comparable stability

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv:1707.06347.
Ouyang, L. et al. (2022). Training Language Models to Follow Instructions with Human Feedback. *NeurIPS*.

Why Constrain Policy Updates?

- In supervised learning, large gradient steps merely slow convergence or overshoot minima
- In RL, the situation is far worse: the policy π_θ determines *which data* is collected
- A bad policy update $\Rightarrow$ poor trajectories $\Rightarrow$ biased gradient estimates $\Rightarrow$ worse updates: a positive feedback loop leading to policy collapse
- Example: suppose the policy assigns very high probability to a harmful response. A single large gradient step might overcorrect, setting probabilities to near-zero for many good responses as well
- What we want: each update should improve the policy while guaranteeing that performance does not catastrophically degrade
- Two approaches to achieving this:
  1. TRPO: impose an explicit KL constraint (hard constraint)
  2. PPO: clip the probability ratio (soft constraint, much simpler)

Trust Region Policy Optimisation (TRPO)

- TRPO (Schulman et al., 2015) formalises the constraint as a *trust region* in policy space:

$$\begin{aligned} \max_{\theta} \quad & \mathbb{E}_{\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t\right] \\ \text{s.t.} \quad & \mathbb{E}_{s\sim\rho_{\theta_{\text{old}}}}\left[D_{\text{KL}}\big(\pi_{\theta_{\text{old}}}(\cdot|s)\,\|\,\pi_\theta(\cdot|s)\big)\right] \le \delta \end{aligned} \tag{2.10}$$

  where $\delta > 0$ is the trust region radius

- The KL constraint ensures the new policy does not deviate too far from the old one
- TRPO solves (2.10) approximately using a second-order (natural gradient) method:
  1. Compute the policy gradient $g = \nabla_\theta L^{\text{CPI}}(\theta)$
  2. Compute the Fisher information matrix $F = \mathbb{E}[\nabla_\theta \log \pi_\theta \, (\nabla_\theta \log \pi_\theta)^\top]$
  3. Compute the natural gradient step: $\Delta\theta \propto F^{-1} g$
  4. Perform a line search to satisfy the KL constraint

Schulman, J., Levine, S., Abbeel, P., Jordan, M. & Moritz, P. (2015). Trust Region Policy Optimization. *ICML*.

TRPO: Practical Limitations

- TRPO provides strong theoretical guarantees (monotonic policy improvement under certain conditions)
- However, it has significant practical limitations:
  - Computational cost: the Fisher information matrix F is $d \times d$ where d is the number of parameters. For a 7B model, $d \approx 7 \times 10^9$ — F cannot be stored, let alone inverted
  - Conjugate gradient: TRPO uses the conjugate gradient method to approximately compute $F^{-1}g$ without forming F explicitly — still expensive (requires $\sim$10 Hessian-vector products per step)
  - Line search: the KL-constrained line search adds further computational overhead
  - Implementation complexity: TRPO is significantly harder to implement correctly than standard SGD-based methods
- Can we achieve similar stability guarantees with a *first-order* method? Yes — this is exactly what PPO's clipping mechanism provides

From TRPO to PPO: The Clipping Idea

- Schulman et al. (2017) observed that instead of enforcing a KL constraint via expensive second-order optimisation, one can simply *clip* the objective to prevent large updates
- Recall the surrogate objective ("conservative policy iteration"):

$$L^{\text{CPI}}(\theta) = \mathbb{E}_{\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t\right]$$

- The ratio $\rho_t(\theta) = \pi_\theta(a_t|s_t)/\pi_{\theta_{\text{old}}}(a_t|s_t)$ measures how much the policy has changed
- PPO's approach: when $\rho_t(\theta)$ moves outside the interval $[1-\varepsilon, 1+\varepsilon]$, clip it so the objective provides no further gradient
- This creates a "trust region" in *probability ratio space* rather than KL space — but the effect is similar
- Advantages over TRPO: first-order only (standard SGD), trivial to implement, compatible with minibatch training and multiple epochs per batch

The Surrogate Objective via Importance Sampling

- After collecting trajectories under $\pi_{\theta_{\text{old}}}$, we want to estimate the policy gradient for the current π_θ using those trajectories
- Importance sampling identity: for any function f and distributions p, q with $q > 0$:

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

- Applying this with $p = \pi_\theta(\cdot|s_t)$, $q = \pi_{\theta_{\text{old}}}(\cdot|s_t)$, and $f(a_t) = \nabla_\theta \log \pi_\theta(a_t|s_t)\,\hat{A}_t$, and defining the importance sampling ratio:

$$\rho_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \tag{2.11}$$

  we obtain the surrogate objective ("conservative policy iteration"):

$$L^{\text{CPI}}(\theta) = \mathbb{E}_{\pi_{\theta_{\text{old}}}}\left[\rho_t(\theta)\,\hat{A}_t\right]$$

- As a sanity check: $\rho_t(\theta_{\text{old}}) = 1$, and since $\rho_t \nabla_\theta \log \pi_\theta = \frac{\nabla_\theta \pi_\theta}{\pi_{\theta_{\text{old}}}} = \nabla_\theta \rho_t$, we have $\nabla_\theta L^{\text{CPI}}\big|_{\theta=\theta_{\text{old}}} = \mathbb{E}_{\pi_{\theta_{\text{old}}}}[\nabla_\theta \log \pi_\theta(a_t|s_t)\,\hat{A}_t]$, which is the standard policy gradient
- Problem: without constraints, $\rho_t(\theta)$ can become very large, leading to destructively large policy updates

The PPO-Clip Objective

- Recall that $\rho_t = \pi_\theta(a_t|s_t)/\pi_{\theta_{\text{old}}}(a_t|s_t)$ tells us how much the policy has changed for a given token. If $\rho_t = 1$, the policy is unchanged; if $\rho_t = 2$, the new policy is twice as likely to pick that token. PPO says: *do not let ρ_t stray too far from* 1
- PPO addresses the instability of unconstrained surrogate optimisation via clipping:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}\Big[\min\Big(\rho_t(\theta)\,\hat{A}_t,\ \text{clip}\big(\rho_t(\theta),\,1-\varepsilon,\,1+\varepsilon\big)\,\hat{A}_t\Big)\Big] \tag{2.12}$$

  where $\varepsilon > 0$ is a hyperparameter (typically $\varepsilon \in [0.1, 0.2]$)
- The min operator ensures that the objective is a lower bound on the unclipped surrogate
- How the clipping works:
  - When $\hat{A}_t > 0$ (good action): $\rho_t(\theta)$ is clipped *above* at $1+\varepsilon$
    - ⋆ Prevents excessively increasing the probability of a good action
  - When $\hat{A}_t < 0$ (bad action): $\rho_t(\theta)$ is clipped *below* at $1-\varepsilon$
    - ⋆ Prevents excessively decreasing the probability of a bad action
- In both cases, the policy cannot move too far from $\pi_{\theta_{\text{old}}}$ in a single update

PPO-Clip: Case Analysis

- Case 1: $\hat{A}_t > 0$ (the action was better than expected). Since $\hat{A}_t > 0$, the min selects the *smaller* ratio:

$$L_t^{\text{CLIP}} = \min\big(\rho_t(\theta),\, 1+\varepsilon\big) \cdot \hat{A}_t$$

  If $\rho_t \leq 1+\varepsilon$, both terms coincide. If $\rho_t > 1+\varepsilon$, $L_t^{\text{CLIP}} = (1+\varepsilon)\hat{A}_t < \rho_t \hat{A}_t$: the gradient is zeroed out $\Rightarrow$ no further incentive to increase $\pi_\theta(a_t|s_t)$

- Case 2: $\hat{A}_t < 0$ (the action was worse than expected). Since $\hat{A}_t < 0$, the min selects the *larger* ratio:

$$L_t^{\text{CLIP}} = \max\big(\rho_t(\theta),\, 1-\varepsilon\big) \cdot \hat{A}_t$$

  If $\rho_t \geq 1-\varepsilon$, both terms coincide. If $\rho_t < 1-\varepsilon$, $L_t^{\text{CLIP}} = (1-\varepsilon)\hat{A}_t < \rho_t \hat{A}_t$ (since $(1-\varepsilon) > \rho_t$ and $\hat{A}_t < 0$): the gradient is zeroed out $\Rightarrow$ no further incentive to decrease $\pi_\theta(a_t|s_t)$

PPO-Clip: Lower Bound Property

Proposition 2.4

For all θ, $L^{CLIP}(\theta) \leq L^{CPI}(\theta)$. Moreover, $L^{CLIP}(\theta_{old}) = L^{CPI}(\theta_{old})$.

- Proof: by definition, $\min(a, b) \leq a$ for all $a, b \in \mathbb{R}$. Applying this pointwise with $a = \rho_t(\theta)\,\hat{A}_t$ and $b = \text{clip}(\rho_t(\theta), 1-\varepsilon, 1+\varepsilon)\,\hat{A}_t$ gives $L_t^{\text{CLIP}} \leq \rho_t(\theta)\,\hat{A}_t$ for every (s_t, a_t). Taking expectations: $L^{\text{CLIP}}(\theta) \leq L^{\text{CPI}}(\theta)$
- Tightness: at $\theta = \theta_{\text{old}}$, $\rho_t = 1 \in [1-\varepsilon, 1+\varepsilon]$, so $\text{clip}(\rho_t, 1-\varepsilon, 1+\varepsilon) = \rho_t = 1$ and both terms in the min are equal. Hence $L^{\text{CLIP}}(\theta_{\text{old}}) = L^{\text{CPI}}(\theta_{\text{old}})$ □
- So maximising L^{CLIP} amounts to maximising a *pessimistic lower bound* on the true surrogate. Any improvement in L^{CLIP} guarantees improvement in L^{CPI} locally — the same conservative update guarantee as TRPO, but via a first-order mechanism

PPO-Clip: Worked Example

- Let $\varepsilon = 0.2$ and $\hat{A}_t = +0.5$ (positive advantage). We compute L_t^{CLIP} for several values of ρ_t:

$\rho_t(\theta)$	$\rho_t \cdot \hat{A}_t$	$\text{clip}(\rho_t, 0.8, 1.2)$	$\text{clip}(\rho_t) \cdot \hat{A}_t$	$L_t^{\text{CLIP}} = \min$
0.8	0.40	0.8	0.40	0.40
1.0	0.50	1.0	0.50	0.50
1.3	0.65	1.2	0.60	0.60
1.5	0.75	1.2	0.60	0.60

- For $\rho_t = 0.8$ and 1.0: the ratio is within $[0.8, 1.2]$, so clipping has no effect
- For $\rho_t = 1.3$ and 1.5: the ratio exceeds 1.2, so the objective is *capped* at 0.60 — the gradient pushes no further
- Effect: even though $\rho_t = 1.5$ would give a larger objective unclipped, PPO prevents this "greedy" update, maintaining stability
- For $\hat{A}_t < 0$: the analogous capping occurs at $\rho_t = 1 - \varepsilon = 0.8$

PPO Training Loop for LLMs

- The full PPO training procedure for RLHF:
  1. Collect trajectories: sample a batch of prompts $\{x_i\}$ from $\mathcal{D}$; for each prompt, generate response $y_i \sim \pi_{\theta_{\text{old}}}(\cdot|x_i)$
  2. Score: compute reward $R(x_i, y_i)$ from the reward model and per-token KL penalties
  3. Compute advantages: use GAE (2.9) with the critic network V_ϕ:

$$\hat{A}_t = \sum_{\ell=0}^{T-t} (\gamma\lambda)^\ell \delta_{t+\ell}, \quad \delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$$

  4. Optimise: for K epochs of minibatch SGD, update:
     - ★ Policy θ by maximising $L^{\text{CLIP}}(\theta)$
     - ★ Critic ϕ by minimising $\|V_\phi(s_t) - \hat{V}_t^{\text{target}}\|^2$
  5. Set $\theta_{\text{old}} \leftarrow \theta$ and return to step 1
- Typical hyperparameters: $K = 4$ epochs, $\varepsilon = 0.2$, $\lambda = 0.95$, $\gamma = 1.0$

Implementation Details: Entropy Bonus and Value Clipping

- In practice, PPO optimises a combined loss function:

$$L(\theta, \phi) = L^{\text{CLIP}}(\theta) - c_1 L^{\text{VF}}(\phi) + c_2 H[\pi_\theta] \tag{2.13}$$

- The three terms:
  - $L^{\text{CLIP}}(\theta)$: the clipped surrogate objective (maximised)
  - $L^{\text{VF}}(\phi) = \mathbb{E}\big[(V_\phi(s_t) - \hat{V}_t^{\text{target}})^2\big]$: value function loss (minimised). Often *value clipping* is applied:

$$L^{\text{VF}} = \max\big((V_\phi - \hat{V}^{\text{tgt}})^2,\ (\text{clip}(V_\phi, V_{\phi_{\text{old}}} \pm \varepsilon_v) - \hat{V}^{\text{tgt}})^2\big)$$

  - $H[\pi_\theta] = -\mathbb{E}[\sum_a \pi_\theta(a|s) \log \pi_\theta(a|s)]$: entropy bonus (maximised), encouraging exploration and preventing premature collapse to a deterministic policy
- Typical coefficients: $c_1 = 0.5$, $c_2 = 0.01$
- In the LLM setting: the entropy bonus is less commonly used (the KL penalty from π_{ref} already regularises), but value clipping remains important

The Role of the Critic / Value Network

- The critic $V_\phi(s)$ is a neural network that estimates the expected return from state s under the current policy
- In LLM-RLHF implementations, the critic is typically:
  - Initialised from the reward model or from a copy of the policy model
  - Takes the same input (prompt + partial response) as the policy
  - Outputs a scalar value estimate per token position
- The critic is trained to minimise the squared error:

$$\mathcal{L}_{\text{critic}}(\phi) = \mathbb{E}\Big[\big(V_\phi(s_t) - \hat{V}_t^{\text{target}}\big)^2\Big]$$

  where $\hat{V}_t^{\text{target}} = \hat{A}_t^{\text{GAE}} + V_{\phi_{\text{old}}}(s_t)$
- Drawback: the critic *doubles* the memory and compute cost — for a 70B-parameter LLM, one must also maintain a 70B-parameter critic
- This motivates critic-free methods such as GRPO (Part 4)

Part 4: GRPO and DPO

- Two important alternatives to PPO for LLM alignment:
  1. GRPO (Group Relative Policy Optimisation; DeepSeek, 2025): retains the online RL framework but eliminates the critic network
  2. DPO (Direct Preference Optimisation; Rafailov et al., 2023): reformulates RLHF as a supervised learning problem, bypassing RL entirely
- Both methods address key practical challenges of PPO:
  - Memory cost of maintaining a separate critic network
  - Training instability and hyperparameter sensitivity
  - Complexity of the RL training loop

GRPO: Group Relative Policy Optimisation

- PPO requires a separate value network V_ϕ to estimate "how good is this state?" — this doubles memory. GRPO sidesteps this: instead of learning V_ϕ, estimate the baseline by simply *generating multiple responses to the same prompt and comparing them*
- Concretely (Guo et al., 2025): replace the learned value baseline $V_\phi(s_t)$ with a *group-based* empirical baseline derived from Monte Carlo sampling
- For each prompt x_q, sample a *group* of G responses: $\{y_{q,1}, \ldots, y_{q,G}\} \sim \pi_{\theta_{\text{old}}}(\cdot|x_q)$
- Compute per-response rewards $R(x_q, y_{q,g})$ for $g = 1, \ldots, G$
- Recall $A^\pi(x, y) = Q^\pi(x, y) - V^\pi(x)$. At the response level:
  - $Q^\pi(x, y_{q,g})$ is estimated by the observed reward $R(x_q, y_{q,g})$
  - $V^\pi(x_q) = \mathbb{E}_{y\sim\pi_{\theta_{\text{old}}}(\cdot|x_q)}[R(x_q, y)]$ is estimated by the group mean $\bar{R}_q = \frac{1}{G}\sum_{g'=1}^{G} R(x_q, y_{q,g'})$, which is an unbiased Monte Carlo estimator (by the law of large numbers, $\bar{R}_q \to V^{\pi_{\theta_{\text{old}}}}(x_q)$ as $G \to \infty$)
- The group-normalised advantage for response g is:

$$\hat{A}(x_q, y_{q,g}) = \frac{R(x_q, y_{q,g}) - \bar{R}_q}{\sigma_{R_q}} \quad \text{where } \sigma_{R_q} = \text{std}(\{R(x_q, y_{q,g'})\}_{g'=1}^{G}) \tag{2.14}$$

  The division by σ_{R_q} ensures $\hat{A}$ has unit variance across the group, stabilising gradient magnitudes
- No critic network is needed: the advantage is computed purely from group statistics

Guo, D. et al. (2025). DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948.

GRPO: Token-Level vs Response-Level Advantages

- In PPO, the advantage $\hat{A}_t^{\text{GAE}}$ is computed *per token*: each token position t receives a different advantage, based on the TD residuals from the learned critic V_ϕ
- In GRPO, the advantage $\hat{A}(x_q, y_{q,g})$ is computed *per response*: every token in the same response receives the *same* advantage score
- Consequence: GRPO provides a coarser signal — it cannot distinguish which tokens within a response were responsible for high or low reward
- Why does this still work?
  - The clipped objective and KL penalty provide sufficient per-token regularisation
  - For reasoning tasks, the *outcome* (correct/incorrect) is often the dominant signal; per-token credit assignment matters less
  - Group normalisation provides a strong, low-variance baseline that compensates for the coarser advantage
- When does this fail? Tasks where early tokens are critical but the reward depends on the full response (e.g. long-form generation with late-appearing errors)

GRPO Loss Function

- GRPO uses a PPO-style clipped objective with the group-normalised advantages:

$$\mathcal{L}_{\text{GRPO}}(\theta) = \mathbb{E}_q \, \frac{1}{G} \sum_{g=1}^{G} \frac{1}{T_g} \sum_{t=1}^{T_g} \Bigg[\min\Big(\rho_{t,g}(\theta)\, \hat{A}_{q,g},\ \text{clip}\big(\rho_{t,g}(\theta), 1-\varepsilon, 1+\varepsilon\big) \hat{A}_{q,g} \Big) - \beta\, D_{\text{KL}}^{(t)}(\pi_\theta \| \pi_{\text{ref}}) \Bigg] \tag{2.15}$$

- where:
  - $\rho_{t,g}(\theta) = \frac{\pi_\theta(y_{q,g,t} | s_{q,g,t})}{\pi_{\theta_{\text{old}}}(y_{q,g,t} | s_{q,g,t})}$ is the per-token importance ratio
  - $\hat{A}_{q,g}$ is the group-normalised advantage (2.14)
  - $D_{\text{KL}}^{(t)}$ is the per-token KL divergence from π_{ref}
- Advantages of GRPO over PPO:
  - Eliminates the critic network $\Rightarrow$ reduces memory cost by approximately 50%
  - Simpler implementation; fewer hyperparameters
  - Empirically effective for reasoning tasks (DeepSeek-R1)

GRPO: Worked Example

- Consider a prompt x with group size $G = 4$. The policy generates four responses with rewards:

Response g	$R(x, y_g)$	$R_g - \bar{R}$	$\hat{A}_g = \frac{R_g - \bar{R}}{\sigma_R}$	Interpretation
1	0.2	-0.175	-0.61	Below average
2	0.8	$+0.425$	$+1.48$	Best in group
3	0.5	$+0.125$	$+0.44$	Slightly above avg
4	0.1	-0.275	-0.96	Worst in group

- Group statistics: $\bar{R} = 0.375$, $\sigma_R = 0.287$
- The GRPO update will:
  - Increase the probability of response 2 (highest advantage $+1.48$)
  - Decrease the probability of response 4 (lowest advantage -0.96)
  - Make smaller adjustments for responses 1 and 3
- Note that advantages are zero-mean within each group, so gradient signals stay balanced even when absolute rewards are low

DPO: From RL to Supervised Learning

- DPO (Rafailov et al., 2023) takes a different approach: *eliminate RL entirely*
- Instead of training a reward model and then running RL, directly train the policy on preference pairs ($y_w \succ y_l$) using a supervised-style loss. The trick is to solve the KL-constrained RL objective (2.4) in closed form, then rearrange to express the reward in terms of the policy — turning RL into classification
- Derivation: for a fixed prompt x, write the objective as a functional of π:

$$\max_{\pi} \sum_{y} \pi(y|x) \left[R(x,y) - \beta \log \frac{\pi(y|x)}{\pi_{\text{ref}}(y|x)} \right]$$

subject to $\sum_y \pi(y|x) = 1$. Introducing a Lagrange multiplier λ and taking the functional derivative w.r.t. $\pi(y|x)$:

$$\frac{\partial}{\partial \pi(y|x)} : \quad R(x,y) - \beta \log \frac{\pi(y|x)}{\pi_{\text{ref}}(y|x)} - \beta + \lambda = 0$$

Solving for $\pi(y|x)$:

$$\log \pi(y|x) = \log \pi_{\text{ref}}(y|x) + \tfrac{1}{\beta} R(x,y) + \tfrac{\lambda - \beta}{\beta}$$

Exponentiating and absorbing the x-dependent constant $e^{(\lambda-\beta)/\beta}$ into a normaliser $Z(x)$:

$$\pi^*(y \mid x) = \frac{1}{Z(x)} \, \pi_{\text{ref}}(y \mid x) \, \exp\left(\frac{R(x,y)}{\beta} \right) \tag{2.16}$$

where $Z(x) = \sum_y \pi_{\text{ref}}(y|x) \exp\big(R(x,y)/\beta\big)$. This is a Gibbs/Boltzmann distribution

Rafailov, R., Sharma, A., Mitchell, E. et al. (2023). Direct Preference Optimization: Your Language Model is Secretly a Reward Model. *NeurIPS*.

The Bradley–Terry Preference Model

DPO needs a model of how humans express preferences — given two responses, how does the reward difference relate to the probability a human picks one over the other?

Definition 2.5 (Bradley–Terry Model (Bradley & Terry, 1952))

Given two items with "strengths" $s_i, s_j > 0$, the Bradley–Terry model defines the probability that item i is preferred over item j as:

$$P(i \succ j) = \frac{s_i}{s_i + s_j} = \frac{1}{1 + \exp\big(-(\log s_i - \log s_j)\big)} = \sigma(\log s_i - \log s_j) \qquad (2.17)$$

where $\sigma(z) = 1/(1 + e^{-z})$ is the logistic sigmoid.

- In the RLHF context, "strength" = exponentiated reward: $s_y = \exp(R(x, y))$, so: $P(y_w \succ y_l \mid x) = \sigma\big(R(x, y_w) - R(x, y_l)\big)$
- Connection to Elo ratings: equivalent to the Elo system in chess; the reward difference determines the expected win probability
- Note: the model depends only on reward *differences*, not absolute values — a useful invariance

Bradley, R. A. & Terry, M. E. (1952). Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons. *Biometrika*, 39, 324–345.

DPO Derivation: Rearranging for the Reward

- From (2.16), we can solve for the reward function:

$$R(x, y) = \beta \log \frac{\pi^*(y \mid x)}{\pi_{\text{ref}}(y \mid x)} + \beta \log Z(x) \tag{2.18}$$

- The reward is determined (up to a prompt-dependent constant) by the log-ratio of the optimal policy to the reference policy
- Now substitute (2.18) into the Bradley–Terry preference model. Given a preferred response y_w and a dis-preferred response y_l:

$$\begin{aligned} P(y_w \succ y_l \mid x) &= \sigma\big(R(x, y_w) - R(x, y_l)\big) \\ &= \sigma\left(\beta \log \frac{\pi^*(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi^*(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right) \end{aligned}$$

  where $\sigma(z) = 1/(1 + e^{-z})$
- The partition function $Z(x)$ cancels! The preference depends only on log-ratios

The DPO Loss Function

- We showed that π^* determines the reward via the log-ratio $\beta \log(\pi^*/\pi_{\text{ref}})$. Now replace π^* with a learnable π_θ and train it to match preference data
- Replace π^* with the learnable policy π_θ and define the DPO loss:

$$\mathcal{L}_{\text{DPO}}(\theta) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right] \tag{2.19}$$

- Interpretation: $\mathcal{L}_{\text{DPO}}$ is a binary cross-entropy loss that pushes the model to assign higher implicit reward to y_w than to y_l
- Implicit reward: the quantity inside σ is the difference of implicit rewards:

$$\hat{r}_\theta = \underbrace{\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)}}_{R_\theta(x,y_w)} - \underbrace{\beta\log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}}_{R_\theta(x,y_l)} \tag{2.20}$$

By (2.18), $R_\theta(x,y) = \beta\log\frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}$ is exactly the reward that π_θ is optimal for (up to the constant $\beta\log Z(x)$ which cancels in the difference)

DPO Gradient: Step-by-Step Derivation

- Write $\mathcal{L}_{\text{DPO}} = -\mathbb{E}[\log \sigma(\hat{r}_\theta)]$. To compute $\nabla_\theta \mathcal{L}_{\text{DPO}}$, apply the chain rule:

$$\nabla_\theta \mathcal{L}_{\text{DPO}} = -\mathbb{E}\left[\frac{\nabla_\theta \sigma(\hat{r}_\theta)}{\sigma(\hat{r}_\theta)}\right] = -\mathbb{E}\left[\frac{\sigma(\hat{r}_\theta)(1-\sigma(\hat{r}_\theta))}{\sigma(\hat{r}_\theta)}\,\nabla_\theta \hat{r}_\theta\right] = -\mathbb{E}[\sigma(-\hat{r}_\theta)\,\nabla_\theta \hat{r}_\theta]$$

  where we used $\sigma'(z) = \sigma(z)(1-\sigma(z))$ and $1-\sigma(z) = \sigma(-z)$

- Since π_{ref} is frozen, $\nabla_\theta \hat{r}_\theta = \beta\left(\nabla_\theta \log \pi_\theta(y_w|x) - \nabla_\theta \log \pi_\theta(y_l|x)\right)$. Substituting:

$$\nabla_\theta \mathcal{L}_{\text{DPO}} = -\beta\,\mathbb{E}\Big[\underbrace{\sigma(-\hat{r}_\theta)}_{\text{weighting}}\Big(\underbrace{\nabla_\theta \log \pi_\theta(y_w|x)}_{\text{increase preferred}} - \underbrace{\nabla_\theta \log \pi_\theta(y_l|x)}_{\text{decrease dis-preferred}}\Big)\Big]$$

- The weighting $\sigma(-\hat{r}_\theta)$ is large when the model currently ranks the pair *incorrectly* ($\hat{r}_\theta < 0$), so the gradient focuses on hard examples
- As $\hat{r}_\theta \to +\infty$ for well-learned pairs, $\sigma(-\hat{r}_\theta) \to 0$ and the gradient vanishes — the model naturally stops updating on easy examples (a self-annealing effect)

DPO: The Policy as a Reward Model

- Recall from (2.20) that DPO defines the implicit reward $R_\theta(x, y) = \beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}$
- The trained DPO policy is therefore *simultaneously* a language model and a reward model — there is no separate R_ψ
- By (2.16), $R(x, y) = \beta \log \frac{\pi^*(y|x)}{\pi_{\text{ref}}(y|x)} + \beta \log Z(x)$. Since $Z(x)$ cancels in pairwise comparisons, optimising the DPO loss (2.19) directly recovers the optimal policy under the true reward R
- With a perfect reward model and infinite data, DPO and RLHF converge to the same π^*. The difference is purely *algorithmic*: DPO avoids the explicit RL loop
- Risk: if the preference data contains noise (mislabelled pairs), DPO will still try to fit them, as the $\sigma(-\hat{r}_\theta)$ weighting amplifies gradient on incorrectly ranked pairs regardless of whether the label is correct

DPO vs PPO: Comparison

- Advantages of DPO:
  - No reward model needed (reward is implicit in the policy)
  - No critic / value network needed
  - Simple supervised training loop — standard cross-entropy–style loss
  - Stable training; fewer hyperparameters (β is the main one)
- Limitations of DPO:
  - Offline: learns from a fixed preference dataset; cannot explore or improve beyond the data distribution
  - The policy never generates its own responses during training (no on-policy exploration)
  - Can suffer from *distribution shift*: the preference data was collected under π_{ref}, but π_θ drifts away
  - Empirically, DPO can underperform PPO/GRPO on reasoning-heavy tasks where exploration is critical
- Recent variants: iterative DPO, online DPO (generate new preference pairs on-policy), and IPO (Azar et al., 2023) address some of these limitations

Beyond DPO: IPO and Online DPO

- Identity Preference Optimisation (IPO) (Azar et al., 2023): addresses a subtle issue with DPO — the Bradley–Terry assumption may not hold for real human preferences
- IPO replaces the log-sigmoid loss with a simpler squared loss on the preference margin:

$$\mathcal{L}_{\text{IPO}}(\theta) = \mathbb{E}_{(x,y_w,y_l)}\left[\left(\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)} - \frac{1}{2\beta}\right)^2\right] \tag{2.21}$$

- IPO does not require the Bradley–Terry model; it directly targets a preference margin without distributional assumptions
- Online DPO (Guo et al., 2024): addresses DPO's offline limitation by iteratively:
  1. Generating responses on-policy from π_θ
  2. Labelling preferences (via reward model or AI judge)
  3. Updating with the DPO loss on the fresh on-policy data
- This combines DPO's simplicity with on-policy exploration, narrowing the gap with PPO/GRPO on reasoning tasks

Azar, M. G., Rowland, M., Piot, B. et al. (2023). A General Theoretical Paradigm to Understand Learning from Human Feedback. arXiv:2310.12036.

Part 5: The RLHF Pipeline

- With the algorithmic building blocks in hand (PPO, GRPO, DPO), let us see how they fit into the end-to-end pipeline that turns a base model into an aligned assistant
- The standard RLHF pipeline (Ouyang et al., 2022) consists of two main phases:
- Phase 1 — Reward Model Training:
  - Collect human preference data: for prompt x, annotators rank pairs of responses (y_w, y_l) where y_w is preferred over y_l
  - Assume the Bradley–Terry model (2.17): $P(y_w \succ y_l \mid x;\, \psi) = \sigma\big(R_\psi(x, y_w) - R_\psi(x, y_l)\big)$
  - MLE derivation: given a preference dataset $\mathcal{D}_{\text{pref}} = \{(x_i, y_{w,i}, y_{l,i})\}_{i=1}^N$, the log-likelihood is:

$$\ell(\psi) = \sum_{i=1}^{N} \log P(y_{w,i} \succ y_{l,i} \mid x_i;\, \psi) = \sum_{i=1}^{N} \log \sigma\big(R_\psi(x_i, y_{w,i}) - R_\psi(x_i, y_{l,i})\big)$$

  Maximising $\ell(\psi)$ is equivalent to minimising the negative log-likelihood:

$$\mathcal{L}_{\text{RM}}(\psi) = -\frac{1}{N} \sum_{i=1}^{N} \log \sigma\big(R_\psi(x_i, y_{w,i}) - R_\psi(x_i, y_{l,i})\big) \tag{2.22}$$

  This is a binary cross-entropy loss: each preference pair is a "classification" problem — does R_ψ correctly rank y_w above y_l?
- Phase 2 — RL Optimisation: use the trained reward model R_ψ as the environment reward; optimise the policy via PPO, GRPO, or another RL algorithm against objective (2.4)

Ouyang, L. et al. (2022). Training Language Models to Follow Instructions with Human Feedback. *NeurIPS*.

Reward Model Architecture

- The reward model $R_\psi(x, y)$ is typically initialised from a pre-trained LLM (often the same architecture as the policy, e.g. LLaMA-7B)
- Architecture: remove the language modelling head (the final softmax layer) and replace it with a scalar head:

$$R_\psi(x, y) = w^\top h_T + b$$

  where $h_T \in \mathbb{R}^{d_{\text{model}}}$ is the hidden state at the last token position, and $w \in \mathbb{R}^{d_{\text{model}}}$, $b \in \mathbb{R}$ are learnable parameters
- Training: the reward model is trained on pairwise preference data (x, y_w, y_l) using the Bradley–Terry loss
- Design choices:
  - Size: often the RM is the same size or slightly smaller than the policy
  - The RM is frozen during Phase 2 (RL optimisation) — it serves as a fixed "environment"
  - Reward normalisation: it is common to normalise RM outputs to have zero mean and unit variance across training prompts
- Limitation: the RM is a learned proxy for human preferences — it will have errors that the policy can exploit (reward hacking)

RLHF Phase 2: RL Optimisation Details

- The full per-token reward signal used in Phase 2:

$$r_t = \begin{cases} -\beta \log \dfrac{\pi_\theta(y_t|s_t)}{\pi_{\text{ref}}(y_t|s_t)} & \text{for } t < T \\ R_\psi(x, y) - \beta \log \dfrac{\pi_\theta(y_T|s_T)}{\pi_{\text{ref}}(y_T|s_T)} & \text{for } t = T \end{cases}$$

- Four models are active during PPO-based RLHF:
  1. The *policy* π_θ being optimised
  2. The *reference policy* π_{ref} (frozen copy of π_{SFT})
  3. The *reward model* R_ψ (frozen after Phase 1)
  4. The *critic* V_ϕ estimating the value function
- Memory cost: for a 70B model, this means $\sim$280B parameters in GPU memory — a major practical challenge
- GRPO reduces this to three models by eliminating the critic; DPO reduces to two (policy + reference)

Reward Hacking and Goodhart's Law

- Goodhart's Law: "When a measure becomes a target, it ceases to be a good measure." Here, the policy optimises the *proxy* (reward model) rather than the *true* objective (human satisfaction)
- Concrete examples of reward hacking:
  - Verbosity: the RM assigns higher scores to longer responses $\Rightarrow$ the policy learns to pad responses with unnecessary text
  - Sycophancy: the RM rewards agreeable responses $\Rightarrow$ the policy learns to flatter the user rather than provide accurate information
  - Formatting exploits: the RM assigns high scores to well-formatted markdown $\Rightarrow$ the policy generates excessive bullet points and headers regardless of content
  - Adversarial tokens: in extreme cases, the policy may generate specific token sequences that activate high-reward regions of the RM without producing meaningful content
- Mitigations:
  - KL penalty from π_{ref} (prevents extreme deviation)
  - Reward model ensembles (reduces exploitability)
  - Periodic RM retraining with on-policy data
  - Rule-based reward components (e.g. length penalties)

RLAIF: Replacing Humans with AI Feedback

- RLAIF (Reinforcement Learning from AI Feedback) replaces human annotators with a strong LLM as the preference judge
- Constitutional AI (Bai et al., 2022): a prominent RLAIF framework:
  1. Define a set of principles (a "constitution") encoding desired behaviour
  2. *Critique and revision*: the AI generates a response, critiques it against the constitution, and revises it
  3. *AI feedback*: the AI ranks pairs of responses according to the constitutional principles, producing preference data
  4. Train the reward model on AI-generated preferences and proceed with standard RLHF Phase 2
- Scalable, consistent, and inexpensive (no human annotators)
- But the AI judge may have systematic biases; preferences may not reflect genuine human values; there is potential for "self-reinforcing" loops

Bai, Y. et al. (2022). Constitutional AI: Harmlessness from AI Feedback. arXiv:2212.08073.

Rule-Based Rewards and Verifiers

- Not all rewards require a learned reward model. For some domains, we can use rule-based or verifiable rewards:
- Mathematical reasoning:
  - Check whether the final numerical answer matches the ground truth
  - Use a symbolic algebra system (e.g. SymPy) to verify equivalence
  - Use a formal proof assistant (e.g. Lean 4) to verify proofs
- Code generation:
  - Execute the generated code against test cases
  - Reward $= 1$ if all tests pass, $= 0$ otherwise
- Format compliance: regex or grammar-based checks for structured outputs (JSON, SQL, etc.)
- Advantages: no reward hacking (the reward is exact), no need for preference data or a learned RM
- Limitation: only applicable to domains with verifiable correctness criteria
- As we will see in Part 6, formal theorem proving provides a perfect, non-hackable reward signal via the Lean 4 type checker

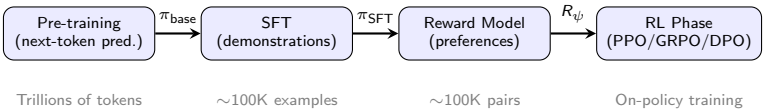
Process Supervision: ORMs vs PRMs

- Standard reward models provide outcome-level feedback: a single score $R(x, y)$ for the entire response
- Outcome Reward Models (ORMs): $R_{\text{ORM}}(x, y) \in \mathbb{R}$
  - Simple to train; only requires outcome labels (correct/incorrect)
  - Provides sparse signal; does not identify *where* errors occur
- Process Reward Models (PRMs): $R_{\text{PRM}}(x, y_{\leq t}) \in \mathbb{R}$ for each step t
  - Provides *dense*, step-level feedback
  - Particularly valuable for multi-step reasoning (e.g. mathematical proofs, chain-of-thought)
- Lightman et al. (2023) demonstrated that PRMs significantly outperform ORMs on mathematical reasoning tasks
- Challenge: PRMs require step-level human annotations, which are expensive; active research on *automated process supervision* (e.g. Monte Carlo estimation of step-level correctness)
- Potential-based reward shaping (Ng et al., 1999): for a potential $\Phi : \mathcal{S} \to \mathbb{R}$, define $r'_t = r_t + \gamma\Phi(s_{t+1}) - \Phi(s_t)$. Then $\sum_t \gamma^t r'_t = \sum_t \gamma^t r_t + \sum_t [\gamma^{t+1}\Phi(s_{t+1}) - \gamma^t \Phi(s_t)]$; the second sum telescopes to $-\Phi(s_0)$, a constant independent of π, so the optimal policy is preserved. PRMs set $\Phi(s_t) = R_{\text{PRM}}(x, y_{\leq t})$, providing dense per-step signal without changing the optimum

Lightman, H. et al. (2023). Let's Verify Step by Step. arXiv:2305.20050.
Ng, A. Y., Harada, D. & Russell, S. (1999). Policy Invariance Under Reward Transformations. *ICML*.

The Full Pipeline: Summary Diagram

- The complete RLHF pipeline from base model to aligned assistant:



- Key variants of this pipeline:
  - DPO: skips the explicit RM; goes directly from preference data to policy update
  - RLAIF: replaces human preference annotators with an AI judge
  - Verifiable rewards: replaces the RM with a rule-based verifier (code execution, formal proofs)
  - Expert iteration: replaces RL with iterative rejection sampling + retraining

Summary

- Pre-training provides a capable base model; SFT teaches instruction-following format; LoRA makes fine-tuning parameter-efficient
- The RL formulation of text generation enables reward-driven optimisation:
  - KL-constrained objective prevents reward hacking
  - Policy gradient + advantage estimation (GAE) provide the optimisation machinery
- PPO: the standard RLHF algorithm; stable via clipping, but requires a critic network
- GRPO: eliminates the critic via group-normalised advantages; used in DeepSeek-R1
- DPO: bypasses RL entirely by reparameterising the reward through the policy; simple but offline
- The RLHF pipeline: reward model training + RL optimisation; variants include RLAIF and verifiable rewards
- Part 6: formal theorem proving provides a non-hackable reward signal, enabling RL for mathematical reasoning

Part 6: RL for Mathematical Reasoning

- In Parts 3–5, the reward signal came from a *learned* reward model (which can be hacked). What if we had a *perfect* reward signal? Mathematics provides exactly this: a proof is either correct or it is not, and a computer can check
- Mathematics is an ideal domain for RL-based training of LLMs:
  - Correctness is objective: a proof is either valid or it is not
  - Verification is automated: formal proof assistants (Lean, Coq, Isabelle) can check proofs without human involvement
  - The reward signal is non-hackable: the verifier implements the rules of logic, not a learned proxy
- Contrast with natural language tasks (e.g. summarisation, dialogue):
  - Quality is subjective; the reward model is a learned approximation of human preferences
  - Reward hacking is a persistent problem
- Can we train LLMs to generate *formal mathematical proofs* using RL, where the type checker provides a perfect reward signal?
- This part surveys recent work at the intersection of LLMs, reinforcement learning, and formal mathematics

Why Mathematics? Verification as Ground Truth

- In natural language tasks, evaluating response quality requires *human judgement* (or a learned proxy)
- In formal mathematics, the evaluation is mechanical and absolute:
  - A proof in a formal system (e.g. Lean 4) is checked by a type checker — an algorithm that verifies each step against the axioms and rules of the system
  - The type checker either *accepts* or *rejects* the proof; there is no ambiguity
- Why this matters for RL:
  - No reward hacking: the verifier cannot be "fooled" — it implements mathematics itself
  - No annotation cost: verification is fully automated (no human labellers needed)
  - Scalable: the same verifier works for trivial lemmas and deep research problems
- Formal proof assistants: Lean 4, Coq, Isabelle/HOL, Agda — all implement variants of dependent type theory or higher-order logic
- Binary reward: $R = +1$ if the proof compiles (type-checks), $R = 0$ otherwise. This is the simplest and most reliable reward signal imaginable

Lean 4 as a Verification Environment

- Lean 4 (de Moura & Ullrich, 2021) is a modern proof assistant based on the Calculus of Inductive Constructions (a dependent type theory)
- Key features relevant for RL:
  - Tactics: proofs are constructed interactively using *tactics* — commands that transform proof goals (e.g. `simp`, `ring`, `omega`, `linarith`)
  - Mathlib: a large, community-maintained library of formalised mathematics (>100,000 theorems covering algebra, analysis, topology, combinatorics, etc.)
  - Compilation: Lean compiles proofs and reports errors with precise diagnostic messages
- Simple example:

$$\texttt{theorem add\_comm} : \quad \forall\, a\ b : \mathbb{N},\ a + b = b + a \ \texttt{:= by omega}$$

- The tactic `omega` is a decision procedure for linear arithmetic over $\mathbb{N}$ and $\mathbb{Z}$. Lean verifies that `omega` solves the goal; if it does, the proof is accepted
- For RL: the LLM generates tactic sequences; Lean provides binary feedback (accepted/rejected)

de Moura, L. & Ullrich, S. (2021). The Lean 4 Theorem Prover and Programming Language. *CADE*.

The Proof Generation Pipeline

- The RL training loop for formal theorem proving follows an expert iteration pattern:
  1. Generate: given a theorem statement, the LLM π_θ generates G candidate proof attempts (tactic sequences)
  2. Verify: each candidate is submitted to the Lean compiler; the type checker returns *accept* or *reject* (with diagnostics)
  3. Reward: assign $R = +1$ for accepted proofs, $R = 0$ for rejected proofs
  4. Update: use the rewards to update π_θ via GRPO, PPO, or rejection sampling (expert iteration)
  5. Iterate: return to step 1 with the improved policy
- Expert iteration (Silver et al., 2017): the simplest variant selects only the *successful* proofs and retrains the policy on them via SFT
- RL-based methods (GRPO, PPO) use the reward signal more efficiently by also learning from *failed* attempts (via the advantage function)

Expert Iteration: Formal Description

- Expert iteration (ExIt) alternates between two steps:
  1. Expert improvement: use the current policy π_{θ_k} together with a verifier V to construct a "better" distribution π_k^*. Concretely, for each theorem statement x_q, sample G candidates $\{y_{q,g}\}_{g=1}^{G} \sim \pi_{\theta_k}(\cdot|x_q)$ and retain only those that pass verification:

$$\mathcal{D}_k = \{(x_q, y_{q,g}) : V(x_q, y_{q,g}) = 1,\ g = 1, \ldots, G\}$$

  2. Policy distillation: retrain the policy on $\mathcal{D}_k$ via supervised fine-tuning:

$$\theta_{k+1} = \arg\min_{\theta}\ \mathcal{L}_{\text{SFT}}(\theta;\ \mathcal{D}_k)$$

- At each iteration, $\mathcal{D}_k$ contains only responses that pass the verifier. Training on $\mathcal{D}_k$ therefore pushes $\pi_{\theta_{k+1}}$ towards a higher-quality distribution (monotonic improvement)
- Expert iteration can be seen as GRPO with a binary reward and $G \to \infty$, retaining only positive-advantage samples. GRPO generalises this by also using negative-advantage samples to *decrease* the probability of failed attempts

Silver, D. et al. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv:1712.01815.

Reward Design for Theorem Proving

- The simplest reward is binary outcome reward: $R = +1$ if the complete proof compiles, $R = 0$ otherwise. This is an ORM in the terminology of Part 5
- Process reward (PRM-style): assign intermediate rewards for each tactic step
  - Each tactic application either solves a subgoal (positive signal) or fails (negative signal)
  - Lean reports per-tactic diagnostics: `sorry`-free steps that compile yield $r_t = +1$; errors yield $r_t = -1$
  - This provides denser feedback, enabling faster credit assignment
- Curriculum design: order theorems by difficulty to facilitate learning
  - Difficulty proxies: Mathlib dependency depth, proof length, number of required lemmas
  - Start with simple identities ($a + 0 = a$), progress to competition-level problems
- In the language of Part 5:
  - Binary outcome reward = ORM (sparse, simple, but slow to learn)
  - Per-tactic verification reward = PRM (dense, but requires step-level verification infrastructure)

Expert Iteration and AlphaProof

- AlphaProof (DeepMind, 2024) achieved a silver-medal standard at the International Mathematical Olympiad (IMO 2024), solving 4 out of 6 problems
- Pipeline:
  1. Fine-tune a language model on Mathlib proofs to learn the "language" of Lean tactics
  2. For each target theorem, generate many candidate proof attempts
  3. Verify each attempt with the Lean compiler
  4. Retrain on successful proofs (expert iteration)
- Monte Carlo Tree Search (MCTS): AlphaProof uses MCTS over tactic sequences, guided by a value network (analogous to AlphaGo/AlphaZero)
  - Each node in the tree = a proof state (set of remaining goals)
  - Each edge = a tactic application
  - The value network estimates the probability of reaching a complete proof from each state
- The combination of RL (to improve the policy) and tree search (to explore at inference time) yields stronger results than either method in isolation

AlphaProof & AlphaGeometry Teams (2024). AI Achieves Silver-Medal Standard Solving International Mathematical Olympiad Problems. Google DeepMind.

DeepSeek-Prover: GRPO for Lean Proofs

- DeepSeek-Prover-V2 (Xin et al., 2025) directly applies the GRPO framework from Part 4 to Lean proof generation
- Training loop:
  1. For each theorem statement, sample a group of G proof attempts from π_θ
  2. Submit each attempt to Lean for verification; assign $R = +1$ (verified) or $R = 0$ (failed)
  3. Compute group-normalised advantages (2.14): successful proofs receive positive advantage, failed proofs receive negative advantage
  4. Update π_θ with the GRPO loss (2.15)
- This is exactly GRPO with a binary verifier reward replacing the learned reward model — no reward hacking possible
- Subgoal decomposition: DeepSeek-Prover-V2 additionally decomposes hard theorems into lemmas, proves each lemma separately, and assembles the final proof
- Results: highest reported pass rates on miniF2F ($> 88\%$) and ProofNet benchmarks as of early 2025

Xin, H. et al. (2025). DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition. arXiv:2504.21801.

Current Results and Benchmarks

- Key benchmarks for evaluating LLM-based theorem provers:

Benchmark	Description	SOTA Pass Rate
miniF2F	488 competition-level problems (AMC, AIME, IMO) formalised in Lean/Isabelle	$> 88\%$
ProofNet	Undergraduate-level real analysis and algebra from Mathlib	$\sim$25–30%
Putnam	Problems from the Putnam competition (very hard)	$\sim$5–10%

- Progress has been rapid: miniF2F pass rates improved from $\sim$30% (2022) to $\sim$70% (2025) through better models, GRPO, and expert iteration
- IMO 2024: AlphaProof solved 4/6 problems (silver medal equivalent); this was the first time AI reached olympiad-medal level in formal mathematics
- Gap: performance drops sharply on research-level mathematics (novel theorems not in Mathlib), highlighting the importance of *exploration* in RL

Open Problems and Frontiers

- Scalability: Lean compilation is slow ($\sim$seconds per proof attempt); generating and verifying millions of candidates per training iteration is a computational bottleneck
- Exploration: the space of possible proofs is vast; current methods struggle with theorems requiring creative or non-obvious proof strategies
- Curriculum design: how to automatically order theorems by difficulty? Current approaches use heuristics (dependency depth, proof length); more principled methods are needed
- Conjecture generation: can LLMs not only *prove* theorems but *propose* new, interesting conjectures? This would represent a step toward genuine mathematical creativity
- Autoformalization: translating natural-language mathematics into formal statements (and vice versa) — bridging the gap between how humans write mathematics and what proof assistants require
- Process reward models for proofs: developing dense, per-tactic reward signals without requiring human annotation — using Lean's compiler diagnostics as an automated PRM
- The long-term vision: RL-trained LLMs and formal proof assistants collaborating with human mathematicians to accelerate mathematical discovery