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. RLHF: PPO, GRPO, and the Training Pipeline
4. 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 — a measure of how spread out vs. peaked the distribution of singular values is —

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

  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
- Because the singular values decay so fast, truncating the SVD to its top r components $\Delta W_r = U_r \Sigma_r V_r^\top$ retains most of the energy of ΔW (i.e. $\|\Delta W_r\|_F \approx \|\Delta W\|_F$). By the Eckart–Young theorem this is the best rank-r approximation in Frobenius norm, so a low-rank matrix can faithfully represent the fine-tuning update
- Note that $\Delta W_r = U_r \Sigma_r V_r^\top = (U_r \Sigma_r)(V_r^\top) = BA$ where $B \in \mathbb{R}^{d_{\text{out}} \times r}$ and $A \in \mathbb{R}^{r \times d_{\text{in}}}$. This motivates directly *parametrising* ΔW as a product of two low-rank factors BA and learning them 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: The α/r Scaling Factor

- Problem: at initialisation, $\|BA\|_F = \Theta(\sqrt{r})$ — doubling the rank increases the perturbation size, making hyperparameter tuning rank-dependent
- Fix: the $\frac{\alpha}{r}$ factor absorbs this rank-dependence, so the *same learning rate works across different values of r* without retuning
- Two common choices of α:
  - $\alpha = r$ (original LoRA default): update norm grows as $\sqrt{r}$. Higher rank = larger total update — useful when the task benefits from more capacity
  - $\alpha = \sqrt{r}$: update norm $\sim O(1)$. The total perturbation is rank-independent — better for controlled experiments that isolate the effect of rank from that of update magnitude

Why Not Just SFT?

- SFT teaches the model to *imitate* a fixed dataset of expert responses
- But imitation has a fundamental practical limit: 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

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)

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 policy will exploit errors in R_ψ — concentrating mass on outputs where R_ψ overestimates R^*. This is reward hacking
- 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:

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

The original objective is now $\max_\theta \mathbb{E}[\sum_{t=1}^{T} \tilde{r}_t]$, which has the form of maximising a cumulative return in a standard MDP. This means we can directly apply off-the-shelf RL algorithms (e.g. PPO) using $\tilde{r}_t$ as the per-step reward

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)

- Recall: in SFT, the model outputs a raw score vector $z_t \in \mathbb{R}^{|\mathcal{V}|}$ and we compute $\pi_\theta(y_t|s_t) = \mathrm{softmax}(z_t)_{y_t}$. The loss $\mathcal{L} = -\sum_t \log \pi_\theta(y_t^*|s_t)$ is evaluated at *fixed* target tokens y_t^* — the chain $\theta \to z_t \to \mathrm{softmax} \to \mathcal{L}$ is fully differentiable, so $\nabla_\theta \mathcal{L}$ is obtained by standard backpropagation. In RL, the reward depends on *sampled* tokens $y_t \sim \mathrm{softmax}(z_t)$, and sampling is a discrete, non-differentiable operation — 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

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

The Policy Gradient Theorem (1/2)

- 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 $r_{t'}$ has already been determined before a_t is chosen, so $r_{t'}$ is a constant with respect to the expectation over a_t. We can therefore pull it out:

$$\begin{aligned}
\mathbb{E}_{a_t \sim \pi_\theta(\cdot|s_t)}\left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot r_{t'}\right] &= r_{t'} \cdot \mathbb{E}_{a_t}\left[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\right] \\
&= r_{t'} \cdot \sum_{a_t} \pi_\theta(a_t|s_t) \cdot \frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)} \\
&= r_{t'} \cdot \nabla_\theta \underbrace{\sum_{a_t} \pi_\theta(a_t|s_t)}_{=1} = 0
\end{aligned}$$

The Policy Gradient Theorem (2/2)

- 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 $\Rightarrow$ make that token *more likely* next time
  - If a token was followed by low total reward $\Rightarrow$ 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: each term in $\hat{g}$ is a product $\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot G_t$. Two independent sources of randomness compound:
  - $G_t = \sum_{t'=t}^{T} r_{t'}$ sums rewards over all future tokens — each sampled from a vocabulary of size $|\mathcal{V}| \sim 10^5$, so G_t can vary wildly between trajectories
  - $\nabla_\theta \log \pi_\theta(a_t|s_t)$ depends on which token a_t was sampled; different tokens give gradient vectors pointing in very different directions

  Since $\mathrm{Var}(XY) = \mathrm{Var}(X)\,\mathrm{Var}(Y) + \ldots$ for independent variables, the product amplifies both sources. Summing $T \sim 10^2$–10^3 such terms makes the total variance 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)
- Why δ_t estimates A^π: recall $Q^\pi(s_t, a_t) = \mathbb{E}_\pi[r_t + \gamma V^\pi(s_{t+1}) \mid s_t, a_t]$. After taking action a_t and observing r_t, s_{t+1}, we have the one-sample estimate $Q^\pi(s_t, a_t) \approx r_t + \gamma V^\pi(s_{t+1})$. Since $A^\pi = Q^\pi - V^\pi$:

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

  Replacing the unknown V^π with the learned V_ϕ gives exactly δ_t
- Bias–variance trade-off:
  - Low variance: δ_t uses only one step of actual reward r_t, then bootstraps from V_ϕ — no sum over future randomness
  - Biased: $\delta_t = A^\pi(s_t, a_t) + \gamma(V_\phi(s_{t+1}) - V^\pi(s_{t+1})) - (V_\phi(s_t) - V^\pi(s_t))$, so the error is proportional to $\|V_\phi - V^\pi\|$. If the critic is inaccurate, δ_t is a poor estimate

Generalised Advantage Estimation (GAE)

- The TD residual δ_t uses one step of real reward then bootstraps from V_ϕ — low variance (one random term) but biased (relies on $V_\phi \approx V^\pi$). Using more real steps reduces bias (less reliance on V_ϕ) but increases variance (more random terms in the sum)
- GAE (Schulman et al., 2016) blends all horizons via an exponentially-weighted sum of TD residuals, controlled by $\lambda \in [0, 1]$:

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

- λ controls the bias–variance trade-off:
  - $\lambda = 0$: $\hat{A}_t = \delta_t$ — one-step, low variance, high bias
  - $\lambda = 1$: $\hat{A}_t = G_t - V_\phi(s_t)$ — full return, high variance, low bias
- LLM-RLHF: $\lambda \in [0.95, 0.99]$, $\gamma = 1$. High λ is preferred because the critic V_ϕ may be inaccurate early in training, so low bias matters more than low variance

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

Putting It Together: The LLM RL Objective

- Combining the KL-regularised per-token rewards $\tilde{r}_t$ with GAE, the policy gradient for LLM-RLHF is:

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

  where $\hat{A}_t^{\text{GAE}} = \sum_{\ell=0}^{T-t} \lambda^\ell \tilde{\delta}_{t+\ell}$ and $\tilde{\delta}_t = \tilde{r}_t + V_\phi(s_{t+1}) - V_\phi(s_t)$
- Remaining problem: nothing constrains the *size* of the update. If one gradient step makes π_θ assign very different probabilities to tokens than before, the model can start producing degenerate text and never recover
- PPO prevents this by bounding how much $\pi_\theta(y_t|s_t)$ can change relative to the previous policy in a single update
- GRPO removes the need to train a separate critic V_ϕ by estimating advantages from groups of sampled responses

Part 3: RLHF — PPO, GRPO, and the Training Pipeline

- PPO (Schulman et al., 2017) is the principal algorithm used for RLHF in models such as ChatGPT (Ouyang et al., 2022)
- Why constrain updates? Unlike supervised learning, in RL the policy π_θ determines *which data* is collected. A bad update $\Rightarrow$ poor trajectories $\Rightarrow$ biased gradient estimates $\Rightarrow$ worse updates: a vicious cycle leading to policy collapse
- Core idea: constrain each update so that π_θ cannot change too much at once, by clipping the probability ratio between the new and old policies
- Before presenting PPO's clipping mechanism, we first examine its predecessor TRPO — the theoretically principled (but expensive) approach to the same problem

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*.

Trust Region Policy Optimisation (TRPO)

- How do we formalise "don't change the policy too much"? TRPO (Schulman et al., 2015) defines a *trust region* — a KL ball around the current policy within which the surrogate objective reliably approximates true performance:

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

  where L^{CPI} is the *surrogate objective* ("conservative policy iteration") and $\delta > 0$ is the trust region radius

- The family $\{\pi_\theta(\cdot|s) : \theta \in \Theta\}$ is a statistical manifold. The KL divergence induces a Riemannian metric on Θ: to second order,

$$D_{\mathrm{KL}}\big(\pi_\theta \| \pi_{\theta+d\theta}\big) \;=\; \tfrac{1}{2}\, d\theta^\top F(\theta)\, d\theta + O(\|d\theta\|^3)$$

  where $F(\theta) = \mathbb{E}_{\pi_\theta}\big[\nabla_\theta \log \pi_\theta \,(\nabla_\theta \log \pi_\theta)^\top\big]$ is the Fisher information matrix

- Hence F is the metric tensor of the manifold: it measures how fast the distribution changes in each parameter direction

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

TRPO: Natural Gradient

- The steepest-ascent direction of $L^{\text{CPI}}(\theta)$ under the Fisher–Rao metric (i.e. per unit of KL divergence) is the natural gradient:

$$\widetilde{\nabla}_\theta L^{\text{CPI}} = F(\theta)^{-1} \nabla_\theta L^{\text{CPI}}(\theta)$$

  This follows from the general result: the steepest-ascent direction of f w.r.t. a metric G is $G^{-1}\nabla f$ (cf. Riemannian gradient on (M, g))

- TRPO solves (2.10) approximately via:
  1. Policy gradient $g = \nabla_\theta L^{\text{CPI}}(\theta)$
  2. Natural gradient step: $\Delta\theta = F^{-1}g$
  3. The quadratic approximation $D_{\text{KL}} \approx \frac{1}{2}\Delta\theta^\top F\,\Delta\theta$ gives the largest step along $F^{-1}g$ that satisfies $D_{\text{KL}} \leq \delta$. Setting $\frac{1}{2}\Delta\theta^\top F\,\Delta\theta = \delta$ and solving: $\Delta\theta \leftarrow \sqrt{2\delta\,/\,(\Delta\theta^\top F\,\Delta\theta)}\;\Delta\theta$

Amari, S. (1998). Natural Gradient Works Efficiently in Learning. *Neural Computation*.

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: since forming F is infeasible, TRPO computes $F^{-1}g$ via the *conjugate gradient* (CG) algorithm — an iterative method that solves $Fx = g$ using only matrix-vector products Fv (computed efficiently via automatic differentiation), without ever forming F. Still expensive: requires $\sim$10 such products per update step
- 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

- Define the importance sampling ratio $\rho_t(\theta) = \pi_\theta(a_t|s_t) \,/\, \pi_{\theta_{\text{old}}}(a_t|s_t)$
- Instead of TRPO's KL constraint (expensive, second-order), PPO simply *clips* ρ_t to $[1-\varepsilon,\, 1+\varepsilon]$:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}\Big[\min\Big(\rho_t(\theta)\,\hat{A}_t,\ \text{clip}\big(\rho_t(\theta),\, 1-\varepsilon,\, 1+\varepsilon\big)\,\hat{A}_t\Big)\Big] \tag{2.11}$$

  where $\varepsilon \in [0.1, 0.2]$ is a hyperparameter
- Interpretation: once the policy moves too far from $\pi_{\theta_{\text{old}}}$ (i.e. ρ_t leaves $[1-\varepsilon,\, 1+\varepsilon]$), the gradient is zeroed out — no further incentive to move
- $L^{\text{CLIP}}(\theta) \leq L^{\text{CPI}}(\theta)$ with equality at $\theta = \theta_{\text{old}}$, so maximising L^{CLIP} amounts to maximising a *pessimistic lower bound* on the true surrogate — the same conservative update guarantee as TRPO
- Advantages over TRPO: first-order only (standard SGD), trivial to implement, compatible with minibatch training and multiple epochs per batch

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, $\varepsilon = 0.2$, $\lambda = 0.95$, $\gamma = 1.0$
- The critic V_ϕ is typically initialised from the reward model or a copy of the policy, and outputs a scalar value estimate per token position
- 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

GRPO: The Key Idea

- Problem: PPO needs a critic V_ϕ to compute advantages $\hat{A}_t = Q(s_t, a_t) - V_\phi(s_t)$. This doubles memory
- GRPO's insight (Guo et al., 2025): instead of *learning* a baseline V_ϕ, *estimate* it by sampling. For each prompt x_q, generate a *group* of G responses:

$$\{y_{q,1}, \ldots, y_{q,G}\} \sim \pi_{\theta_{\text{old}}}(\cdot \mid x_q)$$

  and score each one: $R(x_q, y_{q,g})$ for $g = 1, \ldots, G$
- The advantage $A^\pi(x, y) = Q^\pi(x, y) - V^\pi(x)$ is then estimated as:
  - $Q^\pi(x_q, y_{q,g}) \approx R(x_q, y_{q,g})$ (observed reward)
  - $V^\pi(x_q) \approx \bar{R}_q = \frac{1}{G}\sum_{g'=1}^{G} R(x_q, y_{q,g'})$ (group mean as Monte Carlo baseline)
- This is unbiased: by the law of large numbers, $\bar{R}_q \to V^{\pi_{\theta_{\text{old}}}}(x_q)$ as $G \to \infty$
- 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: Advantage and Loss

- The group-normalised advantage for response g is:

$$\hat{A}(x_q, y_{q,g}) = \frac{R(x_q, y_{q,g}) - \bar{R}_q}{\sigma_{R_q}} \tag{2.12}$$

  where $\sigma_{R_q} = \mathrm{std}\big(\{R(x_q, y_{q,g'})\}_{g'}\big)$. Division by σ_{R_q} gives unit variance across the group, stabilising gradient magnitudes

- $\hat{A} > 0$ for responses scoring above the group average; $\hat{A} < 0$ for those below. The policy is pushed to produce more of the former and less of the latter

- The GRPO loss averages over prompts q, responses g in each group, and tokens t in each response:

$$\mathcal{L}_{\mathrm{GRPO}}(\theta) = \mathbb{E}_q \, \frac{1}{G} \sum_{g=1}^{G} \frac{1}{T_g} \sum_{t=1}^{T_g} \Bigg[\underbrace{\min\Big(\rho_{t,g}\, \hat{A}_{q,g},\ \mathrm{clip}(\rho_{t,g}, 1-\varepsilon, 1+\varepsilon)\, \hat{A}_{q,g}\Big)}_{\text{PPO-Clip applied per token}} - \underbrace{\beta\, D_{\mathrm{KL}}^{(t)}(\pi_\theta \| \pi_{\mathrm{ref}})}_{\text{KL penalty from } \pi_{\mathrm{ref}}} \Bigg] \tag{2.13}$$

- Reading the formula:
  - $\rho_{t,g} = \pi_\theta(y_t|s_t)/\pi_{\theta_{\mathrm{old}}}(y_t|s_t)$: per-token importance ratio (same as PPO)
  - $\hat{A}_{q,g}$: group-normalised advantage — *constant across all tokens* in response g (unlike PPO, where each token gets a different advantage)
  - The min/clip term is exactly PPO-Clip: it caps how much $\rho_{t,g}$ can deviate from 1
  - $\beta\, D_{\mathrm{KL}}^{(t)}$: per-token KL divergence from the reference policy, preventing drift

Reinforcement Learning from Human Feedback (RLHF)

- RLHF (Ouyang et al., 2022) puts the pieces together — it is the end-to-end pipeline that turns π_{SFT} into an aligned assistant using the RL machinery from this part:
  1. Train a reward model R_ψ: collect human preference pairs $(y_w \succ y_l)$ and fit R_ψ using the Bradley–Terry model (Bradley & Terry, 1952) — a pairwise comparison model where the probability that y_w is preferred over y_l is $P(y_w \succ y_l) = \sigma\big(R_\psi(x, y_w) - R_\psi(x, y_l)\big)$, trained via binary cross-entropy
  2. Optimise the policy: use R_ψ as the reward in the KL-constrained objective (2.4), and update π_θ via PPO (clipped surrogate + critic) or GRPO (group-normalised advantages, no critic)

Ouyang, L. et al. (2022). Training Language Models to Follow Instructions with Human Feedback. *NeurIPS*.
Bradley, R. A. & Terry, M. E. (1952). Rank Analysis of Incomplete Block Designs. *Biometrika*.

Part 4: RL for Mathematical Reasoning

- In Part 3, 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

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*.

AlphaProof (DeepMind, 2025)

- AlphaProof couples an encoder-decoder Transformer with AlphaZero-style RL to prove theorems in Lean 4
- Training: pre-train on code/maths data ($\sim 3 \times 10^{12}$ decoder tokens, $\sim 12 \times 10^{12}$ encoder tokens); fine-tune on Mathlib tactics; auto-formalise 1M natural-language statements into 80M Lean statements; then RL over verified proofs
- Proof search via MCTS: each node is a proof state, each edge a tactic application. When a tactic splits a goal into subgoals, a *product node* requires all branches to be proved. A value head estimates tactics remaining; values are back-propagated via the *hardest branch*
- Test-time RL: when faced with a hard target theorem T, AlphaProof does not attempt T directly. Instead, an LLM generates millions of *variations* of T — e.g. special cases ($n = 2$), weaker hypotheses, or simpler analogues. These variants range from easy to hard, forming a natural curriculum. AlphaProof attempts to prove each variant using MCTS; every successful proof is used to *retrain the policy on the fly* (i.e. at inference time, not during a separate training phase). As the policy improves on easier variants, it gradually becomes capable of proving harder ones, eventually reaching T itself. The reward is always binary (Lean accepts or rejects) — no learned reward model is involved
- Results: proved 3 of 5 non-geometry IMO 2024 problems (P1, P2, P6), achieving silver-medal performance. P6 was the hardest — only 5 of 609 human contestants earned full marks

AlphaProof Team (2025). Olympiad-Level Formal Mathematical Reasoning with Reinforcement Learning. *Nature*.

DeepSeek-Prover-V2 (Xin et al., 2025)

- DeepSeek-Prover-V2 applies GRPO (Part 3) to Lean 4 proof generation (7B and 671B parameters)
- Cold start — the key challenge is obtaining initial training data when the model cannot yet prove anything:
  1. Prompt DeepSeek-V3 (a strong general-purpose LLM) to write a *natural-language proof sketch* of a hard theorem, and simultaneously translate it into a Lean skeleton where each intermediate step is left as a `sorry` placeholder
  2. A smaller 7B prover model attempts to fill in each `sorry` independently — these are easier subproblems that it *can* solve even before RL training
  3. When all subgoals are resolved, the pieces are assembled into a complete formal proof. This proof, paired with DeepSeek-V3's reasoning chain, becomes an SFT training example

  The resulting SFT model serves as π_{ref} for the GRPO phase
- GRPO training: the GRPO loss (2.13) with binary verifier reward ($R = 1$ if Lean accepts, 0 otherwise). Each iteration: 256 theorems $\times$ $G = 32$ candidate proofs (max 32,768 tokens)
- Results (Pass@K): miniF2F-test 88.9% (671B); PutnamBench 49/658 (671B); AIME 2024–25 6/15 (671B)

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

Aristotle (Harmonic, 2025)

- Aristotle integrates three components: a Lean proof search system, an informal reasoning module, and a geometry solver. A single >200B-parameter Transformer serves as both policy and value function
- Monte Carlo Graph Search (MCGS): generalises MCTS by treating Lean proof states as nodes in a *hypergraph* (not a tree), since different tactic sequences can reach the same proof state. Uses AND/OR structure: a state is proved if *any* action succeeds (OR); an action succeeds only if *all* resulting subgoals are proved (AND). Node selection uses a PUCT variant with an exploration bonus weighted by the policy prior
- Informal reasoning: for hard problems, an LLM generates a natural-language proof, restructures it as a sequence of lemmas with short individual proofs, then auto-formalises each lemma into Lean. Lean compiler feedback is used to iteratively correct formalisation errors. Proved lemmas are kept; failed ones are replaced with new decompositions
- Test-time training: alternate between (i) attempting the target problem via MCGS, and (ii) retraining the model on search traces from those attempts — specialising the policy to the problem at hand
- Results: solved 5 of 6 IMO 2025 problems with formally verified Lean proofs (gold-medal equivalent); only P6 remained unsolved

Harmonic Team (2025). Aristotle: IMO-Level Automated Theorem Proving. arXiv:2510.01346.

RL Training vs Iterative Refinement with Lean Feedback

- Two paradigms for LLM-based theorem proving in Lean:
  1. RL training: train a specialised model with Lean's binary reward (AlphaProof, DeepSeek-Prover-V2, Aristotle)
  2. Iterative refinement: prompt a frontier model, compile with Lean, feed diagnostics back, and repeat — no RL training at all
- The case for RL: the model internalises proof search strategies and tactic selection through millions of training episodes; test-time RL (AlphaProof) and MCTS/MCGS enable systematic exploration of the proof space
- The case against RL: frontier models improve rapidly; a compile–fix loop with Lean feedback is simple, requires no training infrastructure, and automatically benefits from each new model generation. RL-trained specialists risk being overtaken by the next frontier release

Summary

- Pre-training provides a capable base model; SFT teaches instruction-following; LoRA makes fine-tuning parameter-efficient
- The RL formulation of text generation enables reward-driven optimisation via policy gradients and advantage estimation (GAE)
- PPO: stable updates via clipping the importance ratio; requires a critic network
- GRPO: eliminates the critic via group-normalised advantages; used in DeepSeek-R1
- RLHF: reward model training (Bradley–Terry) + RL optimisation (PPO or GRPO)
- Formal theorem proving: the Lean type checker provides a non-hackable binary reward; AlphaProof (MCTS + test-time RL), DeepSeek-Prover-V2 (subgoal decomposition + GRPO), and Aristotle (MCGS + informal reasoning) achieve IMO medal-level performance