Lecture 5: Post-Training

Edouard Oyallon
edouard.oyallon@cnrs.fr
CNRS, ISIR

What you will learn today

- Today, we will cover:
  - Supervised Fine Tuning
  - Preference Policy Optimization to match user preferences
  - Policy Gradients methods to learn from a reward
  - Distillation and useful techniques for reusing your model.

Post-Training: How a Raw LLM Becomes a Helpful Assistant

- **Pretraining (base model):** learns language + world knowledge via next-token prediction on huge corpora, but is unaligned, unsafe, format-inconsistent, and domain-blurry.

- **Post-training (behavior shaping):** adds extra training to make the model **follow instructions**, match preferences, be safer, and perform on targeted tasks/domains.

- Two main families:

  - **Supervised demonstrations** (input → target answer)

  - **Preference-based alignment** ("A is better than B")

  - **Reinforcement Learning from Verifiable Rewards** (solving maths problems)

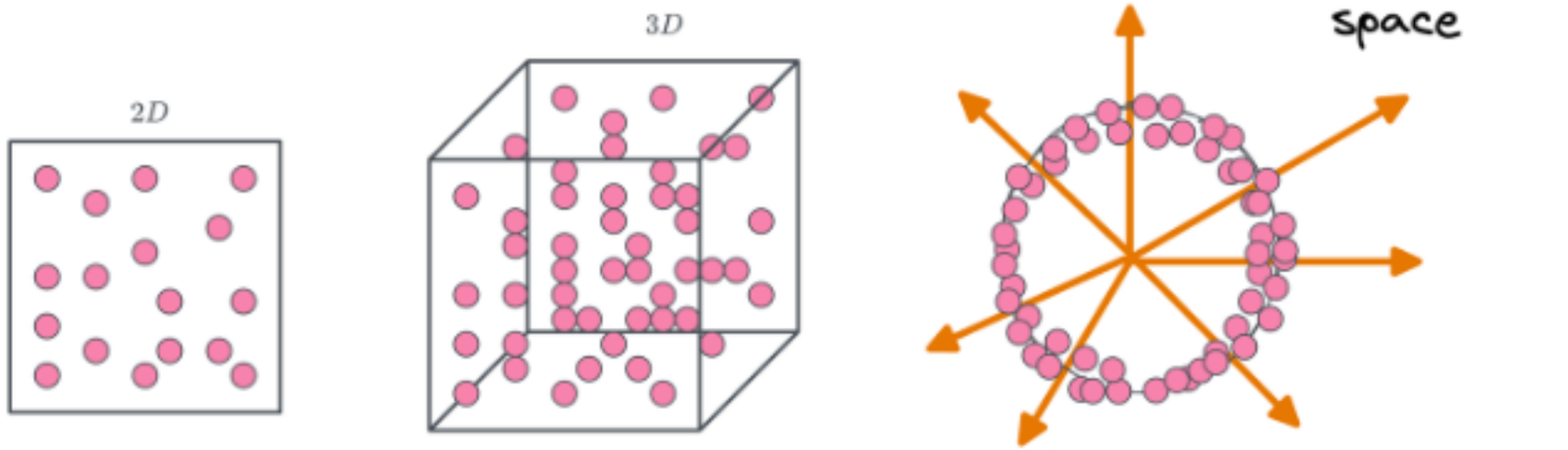
- Many competing goals: post-training jointly pushes for helpfulness, safety, style, tool use, and domain adaptation, and these objectives can conflict.

- Yet it can cause **regressions** (some abilities get worse) and **forgetting** of pretraining skills; behavior remains that of a huge **black box**.

- Note: Post-training can be **very cheap** (<5% of pretraining compute) or **quite large** (up to 50% in some systems, e.g. GROK-style models).

Disclaimer: Understanding LLMs

- **High-dimensional statistical problems are hard:** optimizing in very large parameter spaces has complex, poorly characterized geometry.



- **Neural nets are extreme cases:** millions/billions of parameters, highly non-linear; the theory is limited so there is a classic *curse of dimensionality*. Post-training adds another layer of complexity.

- **Result: conflicting empirical findings:** different setups lead to opposite-looking conclusions (some works say post-training learns new behaviors, others that it mainly amplifies pretrained one, e.g., "Echo Chamber: RL Post-training Amplifies Behaviors Learned in Pretraining")

What Kinds of Tasks Does Post-Training Improve?

- **Instruction following** – do what you *ask*, not just continue text (follow steps, respect constraints, use tools).

- **Math & reasoning** – structured problem solving, multi-step derivations, checking intermediate steps.

- **Code generation & debugging** – write, refactor, and explain code; follow specs; generate tests.

- **Format-sensitive tasks** – reliable **JSON**, API payloads, tables, style guides, report templates.

- **Dialogue & assistance** – stay on topic, ask clarifying questions, remember context, be helpful.

- **Safety & preference alignment** – avoid harmful outputs, follow policies, match desired tone/persona.

How Do We Evaluate LLM Outputs at Scale?

- **LLM outputs are hard to evaluate**: quality depends on meaning, nuance, and context, not just exact string match.

- **Generation is often stochastic**: the same prompt can yield different valid (or invalid) answers across samples.

- Human eval is the gold standard, but it's slow, costly, and hard to scale.

- So we rely on **automatic or semi-automatic evaluation**: heuristics, model-based judges, and task-specific checks to measure quality at scale.

Approaches to Automatic LLM Evaluation

- **Multiple-choice QA**: compare model's choice against the correct option

- **Exact answer matching**: check whether the model's answer matches a gold reference or unit tests

- **Constraint verification** (e.g., regex and schema checks): validate format: JSON, email, dates, code style, etc.

- **Ad hoc ML tools or LLM-as-a-judge**: rain a classifier to score correctness, toxicity, etc. or another LLM.

Generating some text from LLMs

- Given a context $x = [x_1, ..., x_L]$, a model $\varphi(x)$ outputs a distribution over next tokens where β is some temperature and a sigmoid σ:

$$p_{\beta,\varphi}(.|x) = \sigma(\frac{\varphi(x)}{\beta})$$

- Decoding strategies:

  - **Greedy:** pick the most probable token at each step

  - **Sampling:** sample from the softmax distribution

  - **Beam search:** keep K best sequences, return highest log-prob one.

- Generation loop: append the chosen token, update the context, and repeat until EOS, max length, or a stop sequence is reached.

- We will study this in more detail during the next lecture.

Type of datasets

- **How they're built:** mix of public sources (chat logs, code, math, web, textbooks), heavily filtered and normalized (deduplicate, safety filters, format unification) and often converted into *instruction* $\rightarrow$ *response* pairs or *preference* pairs.

- https://github.com/mlabonne/llm-datasets

- https://meta-pytorch.org/torchtune/main/basics/datasets_overview.html

Dataset	Category	Size	Short description
Nemotron-Post-Training-Dataset-v2	General-purpose SFT	6.34M	Multilingual instructions
open-perfectblend	General-purpose SFT	1.42M	Mixed instructions
OpenMathInstruct-2	Math	14M	Math reasoning
Ling-Coder-SFT	Code	4.48M	Multilingual code
luth-sft	Multilingual	570K	FR/EN instructions
WildChat-4.8M	Real conversations	3.2M	Clean chat logs
Skywork-Reward-Preference-80K-v0.2	Preference	77K	Preference pairs

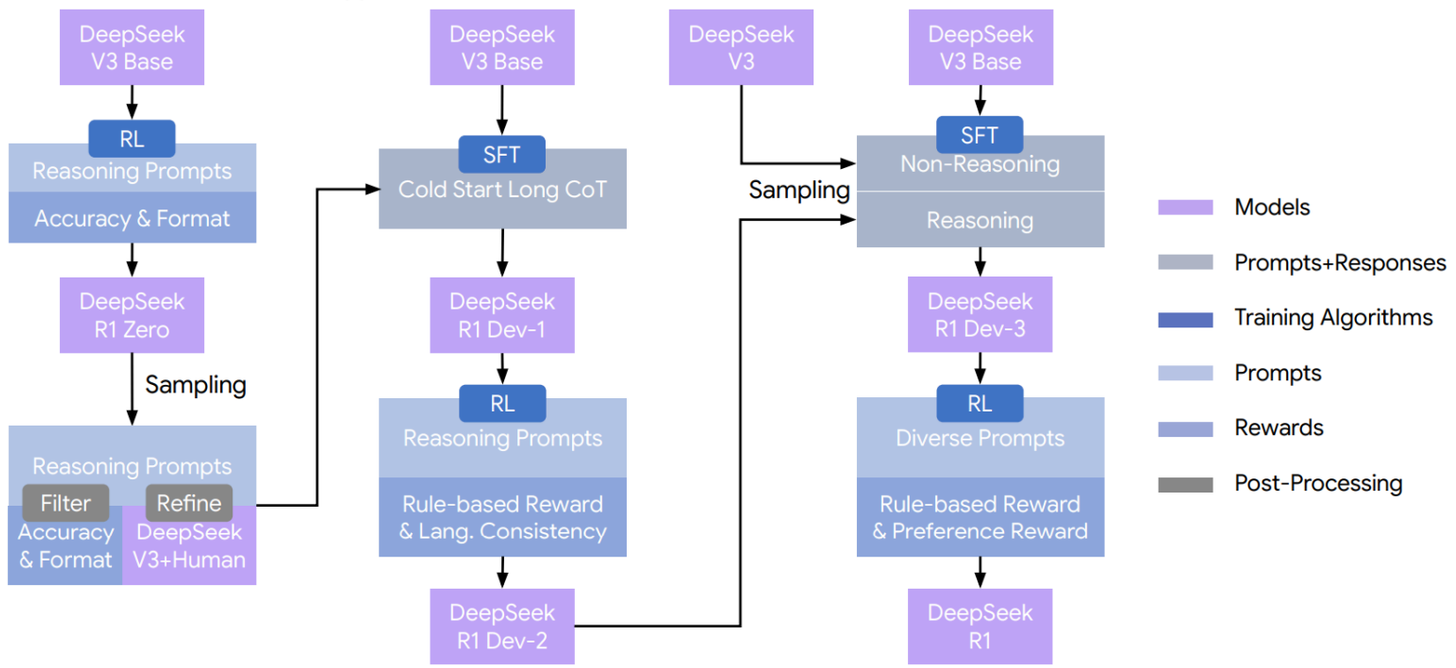
SFT, RL, both?

- It's still unclear what the *right recipe* is for our setting: SFT only, RL only, or a combination of both.

- **DeepSeek-R1-zero** reports strong results using *pure RL* post training, suggesting SFT may not be strictly necessary for high performance.

- However, their approach also highlights **downsides in interpretability (human readability)**...

- We should treat the **SFT vs RL (or hybrid)** choice as an *open design decision*, not a settled best practice.



Outline of the lecture

- We will cover:
  - **First**: post-training via Supervised Fine-Tuning (SFT)
  - **Then:** how to leverage **Reinforcement Learning (RL)** for alignment and performance
  - **Finally:** Other post-training strategies useful for inference (e.g., continual learning, safety alignment)
- Post-training is a big space: we'll focus on the **core ideas and standard building blocks**.

Supervised Demonstration Learning

Supervised Fine-Tuning

- Provide **task-specific examples** tailored to your use case: Pairs of **(input, desired output)**: prompts and target responses

- Train the model to **imitate these targets** (minimize loss between prediction and target)

instruction string · *lengths* 37↔202 74.3%	**output** string · *lengths* 82↔814 95.2%	**code** string · *lengths* 27↔759 95.2%
Write a python function `max_subarray_sum` that finds the maximum sum of a contiguous subarray within a one-dimensional array of numbers which has at least one positive number.	Here is the code to solve this problem: ```python def max_subarray_sum(nums): if not nums: return 0 current_sum = max_sum = nums[0] for num in nums[1:]: current_sum = max(num, current_sum + num) max_sum = max(max_sum, current_sum) return max_sum ```	def max_subarray_sum(nums): if not nums: return 0 current_sum = max_sum = nums[0] for num in nums[1:]: current_sum = max(num, current_sum + nur max_sum = max(max_sum, current_sum) return max_sum

Parameter-Efficient Fine-Tuning

- **Fine-tuning is popular in research** because it's **cheap and fast** compared to full training from scratch.
- You can adapt a base model to a **domain, task, or style** with relatively little data and compute.
- **Parameter-Efficient Fine-Tuning (PEFT)** updates only a small set of parameters instead of the full model.
- Common PEFT methods:
  - **Adapters / LoRA**: add small trainable matrices/modules; base weights stay frozen.
  - **Prompt tuning / Prefix tuning**: learn a small set of "virtual tokens" or prefixes prepended to the input.
- Benefits of PEFT:
  - Lower GPU memory + faster training
  - **Store and swap many task variants** cheaply (one base model + small adapter files)
  - Often comparable quality to full fine-tuning for many tasks.

Example 1: QLoRA. Efficient Finetuning of Quantized LLMs

- **Objective:** fine-tune large LLMs on a **single GPU** while keeping quality close to full fine-tuning.

- **Key idea: freeze the base model** and train only a small number of parameters.

- **Step 1 — 4-bit quantize the pretrained weights:** store the model in **4-bit** to cut memory drastically (base weights stay frozen).

- **Step 2 — add LoRA adapters L_1, L_2 to linear layers W:** for a weight matrix the adapters are typically small rank with few trainable parameters.

$$y = Wx + L_1 L_2 x$$

- **Training:** forward pass uses **dequantized weights** (typically into bf16/fp16) for compute, but gradients update **only the LoRA adapters** (not the frozen 4-bit base weights).

- One gets the memory savings of 4-bit storage + the flexibility of LoRA updates, enabling practical fine-tuning on limited hardware.

Example 2: Assistant with Instruction Fine tuning

- **Assistant SFT** is still **supervised fine-tuning** — the main difference is **how you design the dataset**, not the optimization algorithm.

- Data is formatted as **multi-turn conversations** with explicit roles (system / user / assistant).

- Training target is the **assistant's messages only** (user/system tokens are context).

- Typical prompt format:
  `<s>[INST] <<SYS>> {{ system_prompt }} <</SYS>>`
  `{{ user_message }} [/INST]`

- Often includes:

  - multiple turns (follow-ups, clarifications)

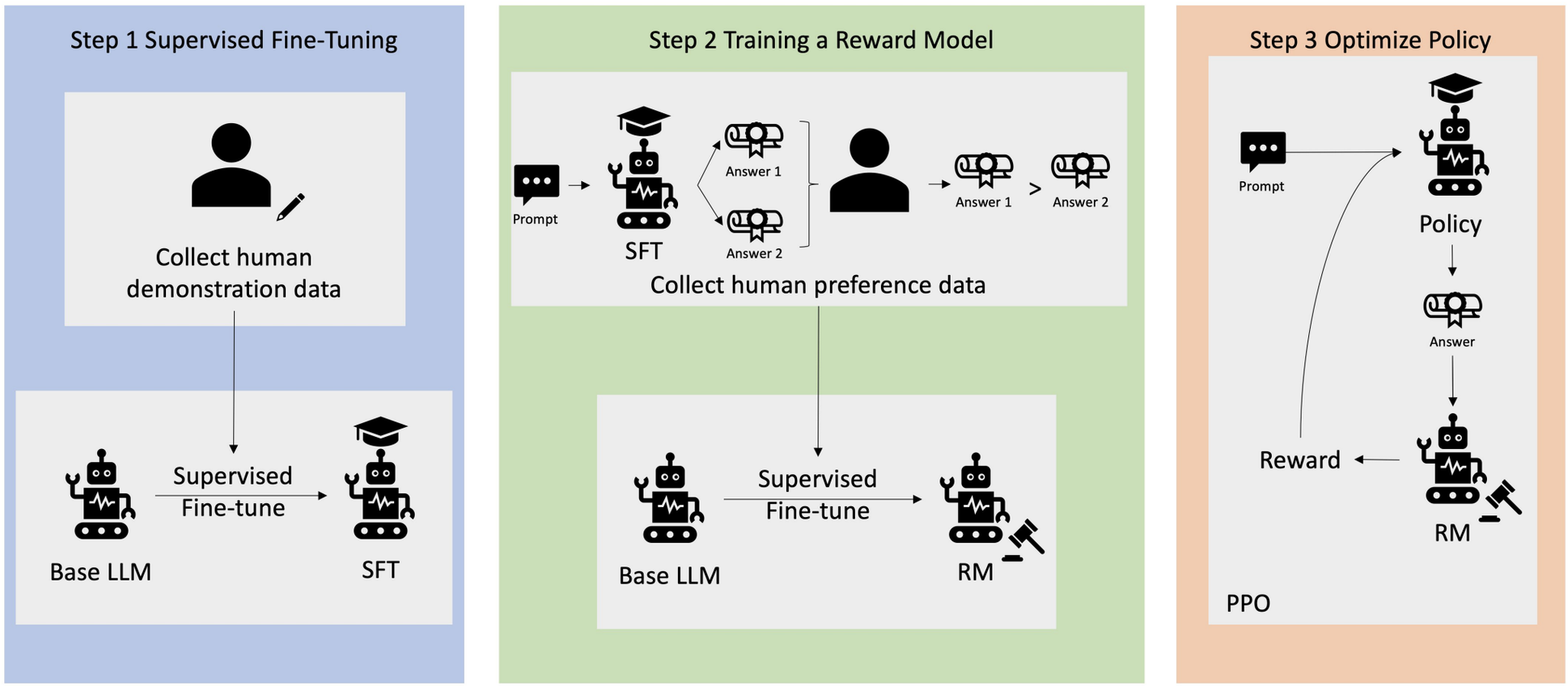
  - safety and style constraints in the system message

  - curated "good" assistant behavior (helpful, concise, refuses when needed)

Reinforcement Learning Basics

General idea of RL-HF (here PPO)

- RLHF (Reinforcement Learning from Human Feedback) aligns a model to human preferences. Typical pipeline:
  - **(Optional) SFT warm-start:** supervised fine-tune the base model on high-quality instruction/assistant data.
  - **Reward model training:** collect human preference comparisons (A vs B) and train a model to score responses.
  - **RL fine-tuning:** optimize the main model to **maximize reward** (often with a KL constraint to stay close to the SFT/base model).



Crash Course in RL

- Let x be a prompt and y a completion.
- View the LLM as a **policy** $\pi_\phi(y|x)$**:** it generates tokens sequentially, parameterized by ϕ.
- Assume a **reward signal** $r_\theta(y|x)$ that scores the completion (learned from human preferences or defined by rules).
- RL objective: find a policy that maximizes expected reward

$$\phi^\star \;=\; \arg\max_{\phi} \; \mathbb{E}_{x\sim\mathcal{D}_{\text{train}},y\sim\pi_\phi(.|x)}[r_\theta(x,y)]$$

  often with a **KL regularization** term to keep the policy close to a reference model π_{ref} (stability / preserving language quality).
- Key questions (RLHF):
  - **How do we learn** the reward model r_θ from comparisons?
  - **How do we optimize** the policy π_ϕ to improve the expected reward (e.g., PPO/DPO-style methods)?
- Note: RL is one framing for "optimize toward preferences," but other solutions might exist.

Preference Optimization

From Rewards to Preferences

- We model evaluation as preferences between two completions for the same prompt. Let x be a prompt and y_a, y_b two candidate answers.

- Humans tend to prefer the answer with **higher latent "utility"** $r(x, y)$**.**

- We turn utilities into probabilistic comparisons: the chance that y_a is preferred increases with the **utility gap**

$$\Pr(y_a \succ y_b \mid x) \text{ depends on } r(x, y_a) - r(x, y_b).$$

- Consequence: only **differences within a prompt** are observable $\rightarrow$ the reward is not identifiable in absolute value.

- Adding any term that depends only on the prompt x cancels out in pairwise comparisons, which motivates the following equivalence relation.

$$r \sim r' \iff \exists f : \ \forall x, y, \quad r(x, y) = r'(x, y) + f(x).$$

- In particular (but the induced equivalence relation would be too weak):

$$r(x, y) \leq r(x, y) \Rightarrow r'(x, y) \leq r'(x, y)$$

Training Reward Models

- Data setup (pairwise preferences): For each prompt x, we collect a preferred response y_+ and a rejected response y_- to form a dataset $\mathcal{D}$.

- Then we model the probability to prefer y_+ over y_- via

$$\Pr(y_+ \succ y_- \mid x) = \sigma\Big(r_\theta(x, y_+) - r_\theta(x, y_-)\Big)$$

with $\sigma(t) = \dfrac{1}{1+e^{-t}}$.

- It gives a smooth, well-behaved **binary classification** objective (preferred vs rejected) via logistic regression:

$$\mathcal{L}_R(\theta) = -\,\mathbb{E}_{(x,y_+,y_-)\sim\mathcal{D}}\Big[\log\sigma\big(r_\theta(x, y_+) - r_\theta(x, y_-)\big)\Big].$$

- Note that $r_\theta \sim r_{\theta'}$ give the same preference model.

- This is the **Bradley–Terry** model: r_θ is a latent "strength", and preferences depend on score differences; with multiple answers it extends to **Plackett–Luce** (ranking) models.

RL Fine-Tuning Phase

- Recall that the KL divergence between π and π_{ref} is given by

$$\mathrm{KL}(\pi(\cdot \mid x) \,\|\, \pi_{\text{ref}}(\cdot \mid x)) = \mathbb{E}_{y \sim \pi(\cdot|x)} \left[\log \frac{\pi(y \mid x)}{\pi_{\text{ref}}(y \mid x)} \right] = \sum_y \pi(y \mid x) \log \frac{\pi(y \mid x)}{\pi_{\text{ref}}(y \mid x)}.$$

- **Policy optimization (RLHF):** Once we have a reward model, we fine-tune the LLM policy π_ϕ by maximizing **reward** while staying close to a **reference policy** π_{ref} (the supervised/SFT model), via a **KL-regularized objective**.

$$\max_\phi \; \mathbb{E}_{x \sim \mathcal{D},\, y \sim \pi_\phi(\cdot|x)} \Big[r_\theta(x, y) \; - \; \beta \, \mathrm{KL}\big(\pi_\phi(\cdot \mid x) \,\|\, \pi_{\text{ref}}(\cdot \mid x)\big) \Big].$$

- The KL penalty prevents π_ϕ from drifting too far from π_{ref}, providing essential regularization and stabilizing training.

- **Nice property:** this KL-regularized objective admits a **closed-form optimal policy** given by

$$\pi^*(y \mid x) \propto \pi_{\text{ref}}(y \mid x)\, e^{r_\theta(x,y)/\beta}.$$

Proof

- We can minimise point wise in x the following:

$$\mathbb{E}_{y\sim\pi(\cdot|x)}\Big[r(x,y)-\beta\,\mathrm{KL}\big(\pi(\cdot\mid x)\,\|\,\pi_{\mathrm{ref}}(\cdot\mid x)\big)\Big]=\sum_y \pi(y\mid x)\,r(x,y)-\beta\sum_y \pi(y\mid x)\log\frac{\pi(y\mid x)}{\pi_{\mathrm{ref}}(y\mid x)}$$

- It's easy to see that the right term is concave in $\pi\big(y\big|x\big)$, so we introduce the Lagrangian:

$$\mathcal{L}_x(\pi,\lambda)=\sum_y \pi(y\mid x)\,r(x,y)-\beta\sum_y \pi(y\mid x)\log\frac{\pi(y\mid x)}{\pi_{\mathrm{ref}}(y\mid x)}+\lambda\left(\sum_y \pi(y\mid x)-1\right)$$

- Use KKT conditions and the Lagrangian multiplier:

$$\frac{\partial\mathcal{L}_x}{\partial\pi(y\mid x)}=r(x,y)-\beta(\log\pi(y\mid x)-\log\pi_{\mathrm{ref}}(y\mid x)+1)+\lambda_x\;=\;0.$$

$$\log\pi(y\mid x)=\log\pi_{\mathrm{ref}}(y\mid x)+\frac{1}{\beta}\big(r(x,y)+\lambda_x-\beta\big)\quad\Longrightarrow\quad\pi(y\mid x)\propto\pi_{\mathrm{ref}}(y\mid x)\,e^{\,r(x,y)/\beta}.$$

Direct Policy Optimization

- Parameterize the optimal policy: given a reward function the KL-regularized optimum has an explicit form:

$$\pi_\theta(y \mid x) = Z(x,\theta)\, \pi_{\text{ref}}(y \mid x)\, e^{r_\theta(x,y)/\beta}.$$

The **partition function** normalizes this distribution.

$$Z(x,\theta) \;=\; \int_y \pi_{\text{ref}}(y \mid x)\, \exp\!\left(\frac{1}{\beta}\, r_\theta(x,y)\right)\, dy.$$

- **Rewrite reward via the optimal policy:** This relationship can be inverted to express the reward (up to an additive constant) in terms of the **log-ratio**:

$$r_\theta(x,y) = \beta \log \frac{\pi_\theta(y \mid x)}{\pi_{\text{ref}}(y \mid x)} \;+\; \beta \log Z(x,\theta).$$

- Plugging this parameterization into the reward-model preference loss shows it is **equivalent to training the policy directly from preferences**:

$$\mathcal{L}_R(\theta) = -\,\mathbb{E}_{(x,y_+,y_-)\sim\mathcal{D}}\Big[\log \sigma\big(r_\theta(x,y_+) - r_\theta(x,y_-)\big)\Big].$$

or, equivalently

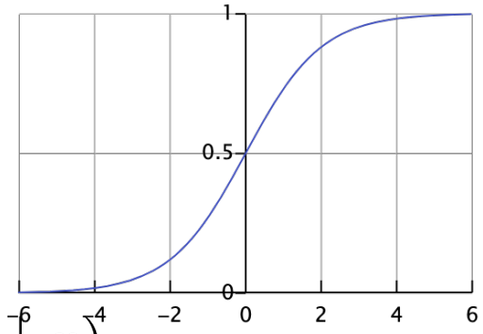
$$\mathcal{L}_{DPO}(\theta) = -\,\mathbb{E}_{(x,y_+,y_-)\sim\mathcal{D}}\left[\log \sigma\Big(\beta \log \frac{\pi_\theta(y_+ \mid x)\, \pi_{\text{ref}}(y_- \mid x)}{\pi_\theta(y_- \mid x)\, \pi_{\text{ref}}(y_+ \mid x)}\Big)\right].$$

- **Outcome:** We eliminate the need for an **ad-hoc standalone reward model:** we learn π_θ from preference pairs while π_{ref} acts as the anchor (regularizer).

DPO's gradients

- Recall that the likelihood of a preference is given by

$$\Pr(y_+ \succ y_- \mid x) = \sigma\Big(r_\theta(x, y_+) - r_\theta(x, y_-)\Big)$$



and the implicit reward is given here by $r_\theta(x, y) = \beta \log \dfrac{\pi_\theta(y \mid x)}{\pi_{\text{ref}}(y \mid x)} + \beta \log Z(x, \theta)$

- So that DPOs gradient write

Increase likelihood of y_+ — Decrease likelihood of y_-

$$\nabla_\theta \mathcal{L}_{\text{DPO}} = -\beta\, \mathbb{E}_{(x, y_+, y_-) \sim \mathcal{D}} \Big[\sigma(r_\theta(x, y_-) - r_\theta(x, y_+)) \left(\nabla_\theta \log \pi_\theta(y_+ \mid x) - \nabla_\phi \log \pi_\theta(y_- \mid x)\right)\Big].$$

- Updates are largest when the reference ratio doesn't clearly separate winner vs loser, and shrink when the preference is already strongly satisfied

- If the model is sure i.e. $r_\theta(x, y_+) \gg r_\theta(x, y_-)$, then the gradient vanishes.

- If the model is unsure $r_\theta(x, y_+) \approx r_\theta(x, y_-)$, the the update is moderate.

- If the model strongly disagrees $r_\theta(x, y_+) \ll r_\theta(x, y_-)$, then the gradient is huge.

Policy Gradients

Policy Gradient

- **Policy-based RL:** parameterize a stochastic policy $\theta \in \mathbb{R}^d$, that satisfies

$$\sum_a \pi_\theta(a|s) = 1$$

For LLMs: actions = tokens, state = prompt + prefix with a much more general scope than DPO.

- **Objective:** maximize the expected return, obtained from generated samples $\{(s_t, a_t, r_t)\}_t$ by sampling actions from $a_t \sim \pi_\theta(.|s_t)$ and maximise

$$J(\theta) = \mathbb{E}_{\pi_\theta}[\sum_t r_t]$$

- **Policy gradient** (REINFORCE):

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \hat{A}_t\right].$$

where the advantage function is obtained from any baseline b:

$$\hat{A}_t = r_t - b(s_t).$$

Deriving REINFORCE: One Trick + Baselines

- One key identity (score-function trick), for any function f:

$$\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[f(a,s)] = \int \pi_\theta(a \mid s)\, f(a,s)\, da$$

and this implies:

$$\begin{aligned}\nabla_\theta \mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[f(a,s)] &= \int \nabla_\theta \pi_\theta(a \mid s)\, f(a,s)\, da \\ &= \int \frac{\nabla_\theta \pi_\theta(a \mid s)}{\pi_\theta(a|s)} \pi_\theta(a|s)\, f(a,s)\, da \\ &= \mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[\nabla \log \pi_\theta(a|s) f(a,s)]\end{aligned}$$

- Baseline property (unbiased), for any b that does not depend on a:

$$\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[\nabla_\theta \log \pi_\theta(a \mid s)\, b(s)] = b(s)\, \mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[\nabla_\theta \log \pi_\theta(a \mid s)] = 0.$$

- So we can subtract a baseline without changing the expected gradient, and to reduce the variance of $\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[f(a)]$ via

$$b(s) \approx \mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[f(a)]$$

- **Autodiff trick:** implement the estimator by maximizing the surrogate objective

$$L_{\mathrm{PG}}(\theta) = \sum_t \log \pi_\theta(a_t \mid s_t)\ \hat{A}_t$$

Policy Gradients Training Loop

- A typical training loop looks like this:
  - **Collect data (rollouts):** run the current behavior policy μ to generate N trajectories:

$$M := |\mathcal{D}| = \sum_{i=1}^{N} T_i, \qquad \mathcal{D} = \{(s_t, a_t, r_t, s_{t+1}, \log \pi_\mu(a_t \mid s_t))\}_{t=1}^{M}.$$

  - Compute learning signals: estimate returns and advantages
  - **Update the policy:** for several epochs, sample **mini-batches** $B_k \subset \mathcal{D}$ from the rollout buffer and optimize a surrogate objective (you might sample multiple epochs)

$$\begin{aligned} \theta^0 &\leftarrow \theta^{\text{old}} \\ \theta^{k+1} &\leftarrow \theta^k + \alpha \nabla_\theta L(\theta^k; B_k), \qquad k = 0, \ldots, K-1, \\ \theta^{\text{new}} &\leftarrow \theta^K \end{aligned}$$

  - Repeat.

- **On-policy:** updates use data generated by (nearly) the same policy being optimized, i.e.:

$$\mu \approx \pi_{\text{old}}$$

- **Off policy:** updates use data generated by a **different behavior policy** μ (often older or mixed), so you need corrections

- After one iteration, most of the gradient policies are approximately on-policy in the updates they allow. However, *seen as an outer/inner loop*, the procedure is on-policy.

Why not REINFORCE?

- REINFORCE is the simplest **Monte-Carlo policy-gradient** method to train a neural network policy via

$$\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t$$

- **Unbiased, but high variance:** the update uses sampled advantages, so gradients are **very noisy:** slow learning and sensitivity to hyperparameters.

- Without an explicit constraint, a single update can **change the policy too much** and hurt performance (a baseline helps variance, but doesn't fully prevent destructive steps).

- To be unbiased, it has to be **on-policy** and uses limited data reuse; needs lots of rollouts.

- However, we'd like to reuse rollouts via **multiple epochs** (GPU-friendly) and keep updates conservative while improving sample efficiency.

Stabilizing Policy Gradients: on-Policy and Advantage Baselines

- Policy-gradient estimators typically look like

$$\sum_t \rho_t \nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t$$

where ρ_t is a normalisation factor. (potentially linked to Importance Sampling)

- Off-policy policy gradients are possible, but they require importance sampling to correct the mismatch. This can cause **high (sometimes exploding) variance**, especially over long horizons or when policies differ a lot.

- **Variance reduction:** use an **advantage** instead of raw returns:

$$\hat{A}_t = r_t - v_\psi(s_t)$$

It can greatly reduce variance and a similar model to the reward model is trained in adhoc manner via

$$\mathcal{L}_V(\psi) = \mathbb{E}_t\left[\left(v_\psi(s_t) - \sum_{k=0}^{T-t-1} r_{t+k}\right)^2\right].$$

Proximal Policy Optimization

Trust Region Policy Optimization

- TRPO is an on-policy policy-gradient method with monotonic improvement guarantees (under assumptions): one needs to solve a sequence of optimization problems.

- **Core idea:** improve the policy, but **don't move too far** from the old policy each update (a "trust region").

- The optimization problem becomes:

$$\max_{\theta} \quad \mathbb{E}_{\pi_{\theta_{\text{old}}}}\left[\sum_t \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t\right]$$

$$\text{s.t.} \quad \left[\mathrm{KL}\left(\pi_{\theta_{\text{old}}}(\cdot \mid s_t) \,\|\, \pi_\theta(\cdot \mid s_t)\right)\right] \leq \delta.$$

- In practice, it uses a **second-order approximation** to the KL constraint (Fisher and natural-gradient style), solved with **conjugate gradient** + **backtracking line search** to satisfy the KL limit. It's thus hard to optimize.

Proximal Policy Optimization Algorithm

- While TRPO is conceptually appealing, we want a simpler objective that's easier to optimize in practice.

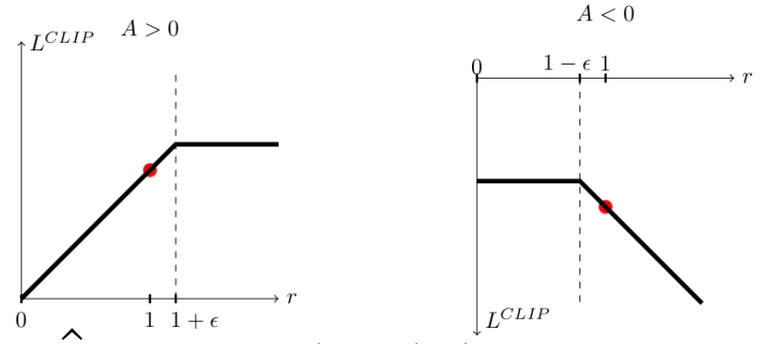
- PPO approximates TRPO's trust-region constraint using the **policy ratio:**

$$\rho_t(\theta) \triangleq \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \geq 0.$$

- **The advantage $\hat{A}_t$ is typically** estimated with an **actor–critic** v_ψ **.**

- The clipped objective (stability) becomes

I'll drop this variable often



$$L^{\text{PPO}}(\theta, \theta_{\text{old}}) = \mathbb{E}_{\pi_{\theta_{\text{old}}}}\left[\sum_t \min\left(\rho_t(\theta)\,\hat{A}_t,\ \text{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}_t\right)\right].$$

This is differentiable, keeps the update **close to old policy, while maintaing.**

- **In this case the gradient becomes** where $\mathbf{1}_{\text{active}} = \begin{cases} \mathbf{1}[\rho_t(\theta) < 1+\epsilon], & \hat{A}_t \geq 0, \\ \mathbf{1}[\rho_t(\theta) > 1-\epsilon], & \hat{A}_t < 0\,. \end{cases}$

$$\nabla_\theta L^{\text{PPO}}(\theta) = \mathbb{E}_{\pi_{\theta_{\text{old}}}}\left[\sum_t \mathbf{1}_{\text{active}} \hat{A}_t\, \rho_t(\theta)\, \nabla_\theta \log \pi_\theta(a_t|s_t)\right].$$

PPO Intuition: Improve, but Don't Move Too Far

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_{\pi_{\theta_{\text{old}}}}\Big[\sum_t \min\left(\rho_t(\theta)\,\hat{A}_t,\ \text{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}_t\right)\Big] \text{ with } \rho_t(\theta) \triangleq \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \geq 0.$$

- PPO objective can also rewrite

$$L^{\text{PPO}}(\theta) = \mathbb{E}_{\pi_{\theta_{\text{old}}}}\left[\sum_{t=0} \min\Big(\rho_t\,\hat{A}_t,\ g\Big(\epsilon, \hat{A}_t\Big)\Big)\right] \quad \text{with} \quad g(\epsilon, A) = \begin{cases} (1+\epsilon)\,A & \text{if } A \geq 0, \\ (1-\epsilon)\,A & \text{if } A < 0. \end{cases}$$

- This allows the update to remain close to the old policy, aligned with TRPO incentive. Let's quickly analyse the term:

- Meaning of ρt: it measures how the new policy changes the probability of the sampled action: $\rho_t > 1$ (more likely), $\rho_t < 1$ (less likely).

- If $\hat{A}_t > 0$ **(good action)**: increase probability, but cap the gain: if $\rho_t > 1+\epsilon$ then the objective **clips** to $(1+\epsilon)\hat{A}_t$.

- Case $\hat{A}_t < 0$ **(bad action):** decrease its probability, but cap the penalty if $\rho_t < 1-\epsilon$ the objective **clips** to $(1-\epsilon)\hat{A}_t$.

- **Takeaway:** clipping creates a **trust-region-like update**: PPO improves the policy using the advantage signal, while preventing overly large policy shifts that can wreck performance.

PPO Training Loop

- The typical training loop of PPO looks like
  - **Collect data (rollouts):** run the current behavior policy π_{θ_n} to generate N trajectories:

$$M := |\mathcal{D}| = \sum_{i=1}^{N} T_i, \qquad \mathcal{D} = \{(s_t, a_t, r_t, s_{t+1}, \log \pi_{\theta_n}(a_t \mid s_t))\}_{t=1}^{M}.$$

  - Compute learning signals: estimate returns and advantages
  - **Update the policy:** for several epochs, sample **mini-batches** $B_k \subset \mathcal{D}$ from the rollout buffer and optimize a surrogate objective (you might sample multiple epochs)

$$\begin{aligned} \theta^0 &\leftarrow \theta_n \\ \theta^{k+1} &\leftarrow \theta^k + \alpha \nabla_\theta L^{\mathrm{PPO}}(\theta^k; \theta_n, B_k), \qquad k = 0, \ldots, K-1, \\ \theta_{n+1} &\leftarrow \theta^K \end{aligned}$$

  - Repeat.

Toward Reasoning via GRPO (and DeepSeek)

GRPO: Grouped New Policy Optimization

- **PPO works well**, but in practice it is often trained as an **actor–critic** method, requiring a **value/critic** v_ψ to estimate advantages (extra memory, compute and tuning).

- **GRPO:** remove the explicit critic by estimating a **prompt-level baseline from a group** of n sampled completions.

- For a given prompt/state, sample n outputs from the policy and compute **relative (normalized) advantages** within the group:

$$\hat{A}_{i,t} = \frac{r_i - \text{mean}(\{r_i\}_{i \leq n})}{\text{std}(\{r_i\}_{i \leq n})}$$

- **Result:** similar "trust-region-style" policy updates as PPO, but with a **simpler training loop** (no value network), the group mean can be thought of as a baseline.

- DeepSeek also adds **KL regularization** to a reference policy to control drift.

DeepSeek-R1 Training Template & Rule-Based Reward

- Output template (training format): model must produce *reasoning* then *final answer*, wrapped as

  `<think> … </think>` and `<answer> … </answer>`

  `<think>` isolates the reasoning trace for parsing/analysis; `<answer>` isolates the final output for verification/reward.

- It enables **programmatic parsing** (separate reasoning vs final) and supports a **format reward** that enforces the tags.

- Rule-based reward:

$$\text{reward}_{\text{rule}} = \text{reward}_{\text{acc}} + \text{reward}_{\text{format}}$$

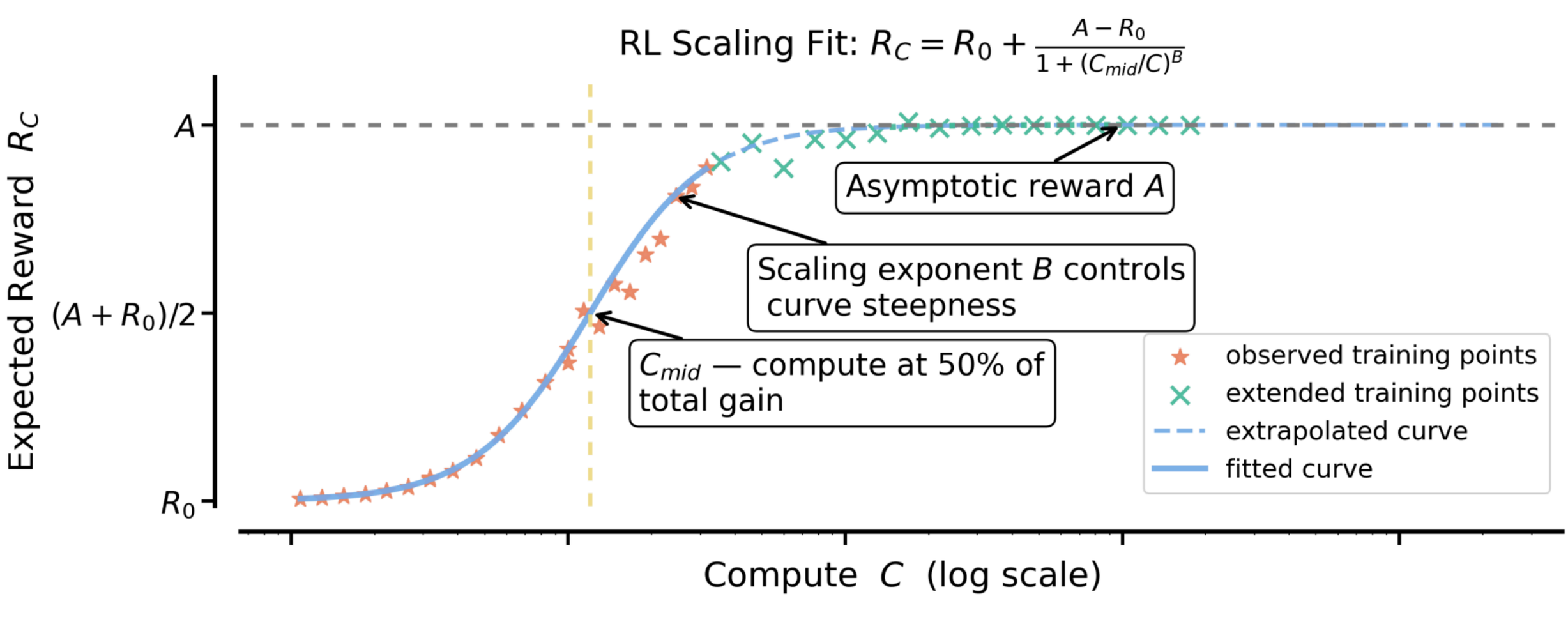
- **Verifiable accuracy signals:** e.g., math answers required in a specified format (like "boxed") for string/logic checking; code tasks can be evaluated via compiler/tests.

Scaling Laws for RL

- Recent study (*The Art of Scaling Reinforcement Learning Compute for LLMs*) claims that RL training for LLMs **follows a predictable sigmoid scaling law**: performance vs. compute can be fit and extrapolated reliably.

$$\underbrace{R_C - R_0}_{\text{Reward Gain}} = \underbrace{A - R_0}_{\text{Asymptotic Reward Gain}} \times \underbrace{\frac{1}{1 + (C_{\text{mid}}/C)^B}}_{\text{Compute Efficiency}}.$$

- They also claim that most design tweaks (curriculum, loss aggregation, normalization, etc.) mainly change **compute efficiency**, not the **asymptotic performance ceiling**.



A specific discussion: RL for maths

- Many tasks have **verifiable outcomes** (exact final answer, equivalence checks), enabling **RL from verifiable rewards (RLVR)** instead of human preference models.

- **Data/compute trick that matters: multiple samples per problem** (diverse attempts) + selection pressure from rewards improves exploration and robustness (especially on hard problems)

- **DeepSeek-Math-V2 (proof setting):** Trains an **LLM verifier** for theorem proving and uses its feedback as the **reward signal** to train a proof generator toward **self-verifiable** proofs

Dataset	**Total**	**Train**	**Test/Val**	**Typical difficulty**
GSM8K (main)	8,792	7,473	1,319	Easy–Medium (grade-school arithmetic/word problems
MATH	12,500	7,500	5,000	Hard (competition-style; multiple topics)
NuminaMath	$\approx$ 860,000	pairs (split varies)		Medium–Hard (broad mix; many competition-style)
Omni-MATH	4,428	benchmark (protocol varies)		Hard–Very Hard (olympiad / advanced problems)

Table 1: Common math datasets/benchmarks, standard sizes, and a rough difficulty level.

Other Post-Training Strategies

The Next Steps of Post-Training

- SFT and RL can push behavior fast, but they also introduce two big practical problems:
  - **how do we package the improvements efficiently?** (distillation)
  - **how do we avoid breaking the model?** (catastrophic forgetting/ collapse).
  - **how do we make sure our models are safe?** (safety alignement)
- There are much much much more questions which are raised by Post-Training and that would deserve attention.

LLM distillation

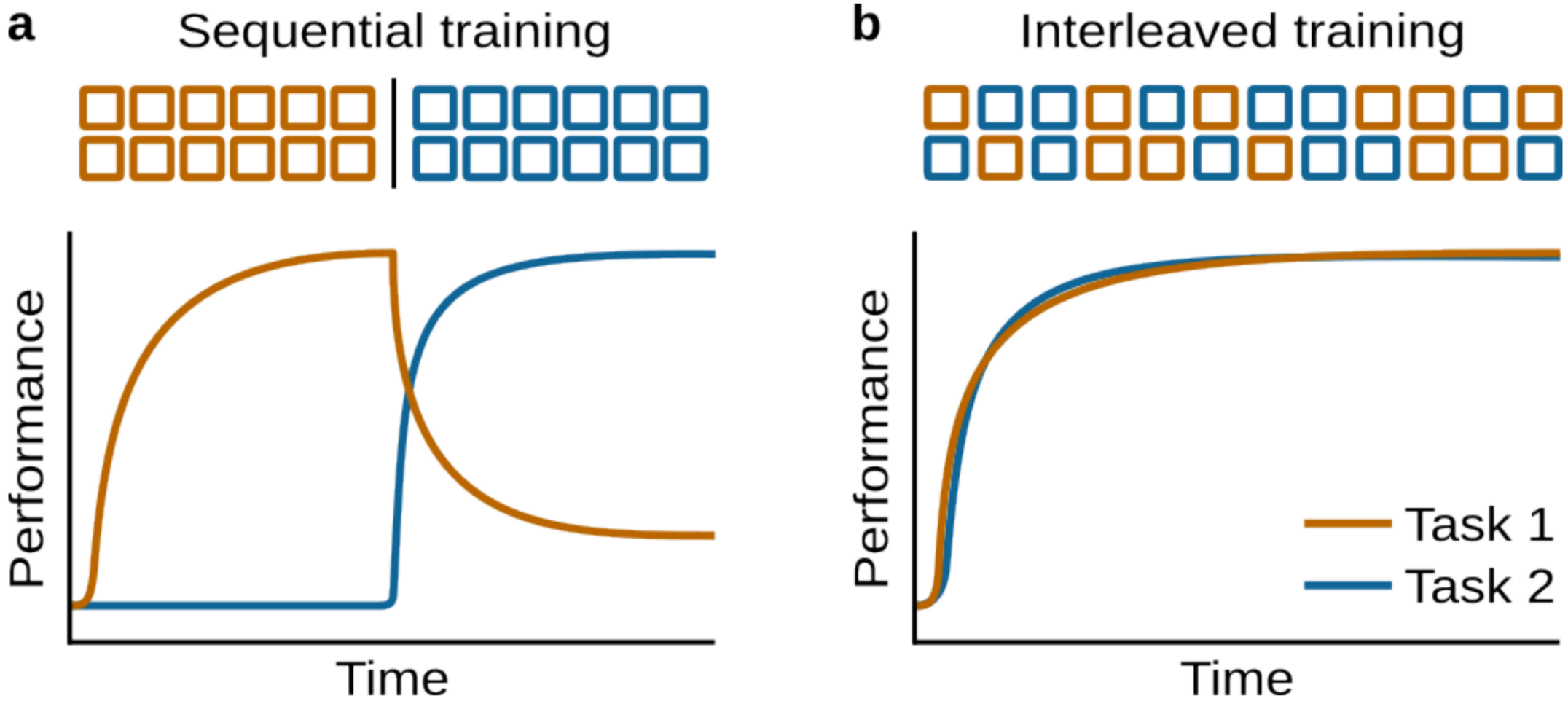
- **Goal:** transfer Post-training gains into a **cheaper, deployable** model.
- **Teacher → Student:** generate high-quality outputs from a strong teacher; train the student by imitation.
- **Common recipes:** sequence distillation (train on teacher outputs), best-of-N distillation, preference distillation...
- **Why it works:** converts expensive signals into **supervised data**, improving speed and reproducibility.

Catastrophic forgetting

- **Forgetting:** specialization on post-training data causes loss of general skills/knowledge.

- Catastrophic forgetting *does* happen in LLMs during continual instruction tuning.

- A mitigation strategy is to interleave training data.



Safety Alignment

- **Goal:** reduce harmful behavior while preserving usefulness - typically you want to minimize "over-refusal".

- **Where it fits:** safety is a **post-training objective** alongside helpfulness (SFT, preference optimization, RL).

- **Methods:** curated safety SFT, preference optimization (safe vs unsafe), RL with safety reward, policy constraints (rules), ...

- **Core trade-off:** safety vs capability. This must be managed with calibrated rewards + targeted data + **many many** evaluations.

- **Evaluation:** adversarial tests, jailbreak robustness, refusal quality, and "helpful safe completion" metrics on realistic user tasks.

TorchTune

Extensible Framework for Post-Training

- TorchTune is a **PyTorch-native** library for authoring and **fine-tuning LLMs** with composable components + reusable training "recipes."

- **Why people use it:** emphasizes **simplicity and extensibility**, with configs and simple command lines to run common post-training setups without a heavy framework.

- **Common recipes included: full fine-tune, LoRA, QLoRA, PPO, GRPO**...

- **Scale + hardware:** supports **single-GPU to distributed** and the idea is to quickly reuse and create multiple checkpoints.

- Its structure is similar to TorchTitan.

Conclusion

- Now, let's try our TorchTune!