Intelligence via generation and selection: A tutorial on reinforcement learning with LLMs and tools

Nando de Freitas and Everyone

May 15, 2026

1 Intelligence via generation and selection

The central theme of this note is that intelligent systems become powerful when they can both *generate* candidate behaviours and *select* among them. This is a useful way to connect several traditions. In philosophy, agency is the capacity of an actor to act in an environment; in AI, an intelligent agent is often described as something that perceives an environment, acts autonomously to achieve goals, and may improve through learning.

Supervised learning corresponds to the most trivial form of imitation: *mimicking*. It uses maximum likelihood–the way we pretrain and SFT LLMs–to map states of the world, such as text questions, to actions, such as text answers. We call such mappings policies. In supervised learning, the teacher demonstrates what to do, but does not grade. Hence the student is only as good as the demonstrations, and good expert data is expensive.

As an aside, there are very powerful ways of doing supervised learning with very general experts: next-step prediction, denoising, and reconstruction. This is basically how pretraining of LLMs works, and also how diffusion, flow matching, and autoencoder-style methods work in multimodal perception and generation. These are often called self-supervised or unsupervised because the target is extracted from the data itself rather than supplied as an external human label.

Reinforcement learning (RL), on the other hand, is about *selective imitation*. The agent does not have to imitate every piece of behaviour in the data. Instead, it can use rewards, value functions, reward models, critics, tests, or other evaluators to decide which behaviours are worth reinforcing and which behaviours should be ignored. In this sense RL can exploit huge amounts of cheap, noisy, or suboptimal data generated by many agents. We do not imitate everything our parents do; we select the useful bits and try to forget the rest.

Supervised learning says: imitate the demonstrated action. RL says: generate or collect candidate trajectories, evaluate them, and put more probability mass on the candidates that lead to better outcomes. The learning signal is

not just "what action did the teacher take?" but "which actions led to high return?".

RL is also about self-improvement. Agents *generate* data by acting in the environment. They can learn from their own successes and mistakes, as well as from a replay buffer containing data from other agents. When we use reward signals to construct selection mechanisms–for example, rank many sampled answers and train only on the best half–the agent can start learning from its own data and self-improve.

Moreover, because an action $\boldsymbol{a}$ changes the environment, interaction produces interventional knowledge. In causal notation, the agent is trying to learn what happens under an intervention, that is $P(\boldsymbol{o'} \mid \mathrm{do}(\boldsymbol{a}), \boldsymbol{o})$, not merely what tends to co-occur in passive data. For background see this discussion on causal models. RL researchers omit writing the do operator explicitly, but it would be reckless to assume the world model is an observational model $P(\boldsymbol{o'} \mid \boldsymbol{a}, \boldsymbol{o})$. It is an interventional model. The actions of an agent behaving in an environment are interventions, not observations.

The price RL pays is that interventions can be expensive, slow, or dangerous. For this reason, *agents can use mental models or external tools–simulators, test suites, calculators, web browsers, theorem provers, code interpreters–to run cheaper experiments before acting in the real world.* Incidentally, I haven't heard anyone proposing to use tools as world models, but this is indeed viable and I believe a great argument for RL. This is not merely an implementation trick. The extended mind thesis of Clark and Chalmers argues that cognition can extend into tools and the surrounding environment, and Dennett's intentional stance explains why treating a system as a rational, goal-directed agent can be a useful predictive strategy. Modern tool-using LLM agents make these old philosophical ideas feel very concrete.

In *The Beginning of Infinity*, David Deutsch makes a strong case for knowledge growth through conjectures and criticism: generate explanations, criticise them, and keep the ones that survive. Darwin's theory of natural selection is the biological ancestor of this view: variation generates candidates, and selection preserves the ones that survive and reproduce. Darwin's *On the Origin of Species* and Deutsch's book are therefore useful background reading for this generation-and-selection perspective.

RL has roots in psychology. In operant conditioning, voluntary behaviours are modified by association with rewards or aversive stimuli: reinforcement increases a behaviour, while punishment or extinction decreases it. B. F. Skinner developed behaviour analysis and used operant conditioning to study how consequences shape future behaviour. I highly recommend reading Skinner's books.

There is also a deep economic lineage. von Neumann and Morgenstern helped formalise decision theory and expected utility. The von Neumann–Morgenstern utility theorem says that, under suitable axioms, rational choice under uncertainty can be represented as maximising expected utility. Savage's subjective expected utility framework then made uncertainty personal: the agent combines its own probabilities with its own utility function. Modern RL inherits this optimisation language, replacing utility with reward and

expected utility with value or expected return.

Silver, Singh, Precup, and Sutton's *Reward is Enough* hypothesis says that intelligence and its associated abilities can be understood as serving reward maximisation. That view is coherent and inspiring, but it assumes that the reward signal is the right selection mechanism. In real systems the desired objective may be hard to specify, and agents may exploit proxy rewards in unintended ways–the classic reward hacking or specification-gaming problem. RL is therefore not just about maximising the reward $\boldsymbol{r}$. It also involves designing, learning, auditing, and stress-testing the selection rule. I expand on this in the last section of this tutorial.

Finally, reward-maximising agency is not the only route to purposeful behaviour. Pedro Ortega's *Universal Artificial Intelligence as Imitation* gives a complementary account based on interaction, imitation, and first-person updating. In that view, an agent's actions are choices that set up evidence, while the world's observations are the evidence used for learning. Reward can still appear as one kind of observation, but purpose does not have to be defined first as a scalar reward. This is a useful reminder for modern AI: agents may learn causal knowledge not only by maximising rewards, but also by imitating structured interactions, using tools, following protocols, absorbing norms, and learning what tends to happen after their own interventions.

2 RL systems

An *agent* is an entity that perceives its environment, takes actions autonomously to achieve goals, and may improve its performance through reinforcement learning or teaching. Agents can have internal goals (inferred subgoals, and the desire to observe more, learn more, or control more – yes, we should be thinking safety here!) or external goals either specified as reward functions or as feedback reward signals. Figure 1 shows the main ingredients of RL.

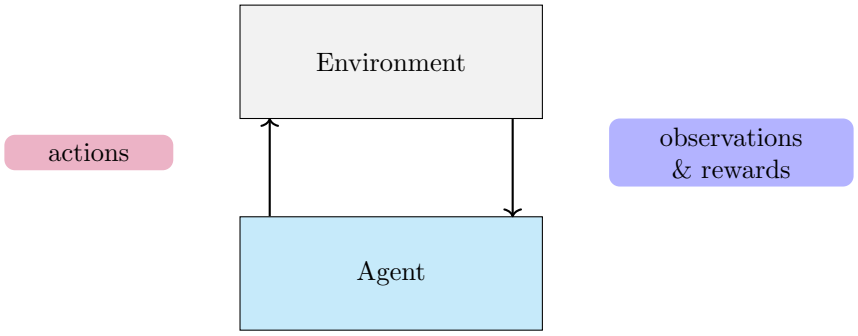

Figure 1: Reinforcement Learning entities.

An agent could be a multimodal neural net that interacts with a user (its environmnent) to empower the user with personalised education (the goal). The

more the agent observes, the easier it is for the agent to create a personalised curriculum to assist the user.

As shown in Figure 2, modern distributed RL systems are broken down into two components: Actors and Learners. Each actor interacts with the environment by generating actions with a network known as the *policy*. The actors also gather rewards and observations from the environment. The data collected are added to a common replay memory. The learner samples data from the replay memory and uses it to update the policy network. After updating the network, the weight checkpoints need to be sent to each actor. It is important when designing such systems to measure the duration of each operation, measure the bandwidth of each communication link, and so on. This demands precise engineering and comprehensive measurements and ablations.

In language, the actors are chatbot agents and the environments are people. The data from each chat is then sent to a replay memory for learning. Typically the learner may require more storage and compute than the actors. It is important to know the costs of inference at the actors and the costs of learning. In some settings, these costs allow for the agent to learn on-policy, in which case the replay memory simply acts as a buffer. Note that this is typically asynchronous because different actors work at different speeds. On the other hand, if the data is not gathered fast enough, it may be necessary to replay old examples to update the policy. This is the off-policy setting. Here, it is important to correct for the fact that the model is being learned with stale data. We will see later that people often introduce importance weights and other mechanisms to correct for this.

Finally, sometimes it is possible to learn the policy from a big replay data alone. This is known as off-line RL or batch RL. Off-line RL is better than supervised learning because it includes selection mechanisms, but it is of course not as good as on-line RL because it lacks direct generation of actions on environments. However, off-line RL can be really useful and allow for learning in situations where interaction is either costly or dangerous.

3 The many types of RL problems

RL comes in many shapes and forms. It is this variety that makes people work on the problem for years and still be suprised now and then.

To start, many examples of "finetuning LLMs with RL" are *one-step RL problems* (top left of Figure 1). Here, given a prompt the model generates a single action and gets an evaluation. A single action can be a text answer, sketch, speech, any other behaviour signal, or any sequence of tokens. The evaluation is often a *single outcome* reward, e.g. whether the answer is correct.

In a multi-step conversation with a chatbot, the user is the environment and the chatbot the agent. Deciding what to say next, while the user does not provide any inputs, is a one-step RL problem. This is clear in Figure 1 (top left) because the three actions can be easily combined into a single one without destroying the structure of the decision graph. However, planning an entire

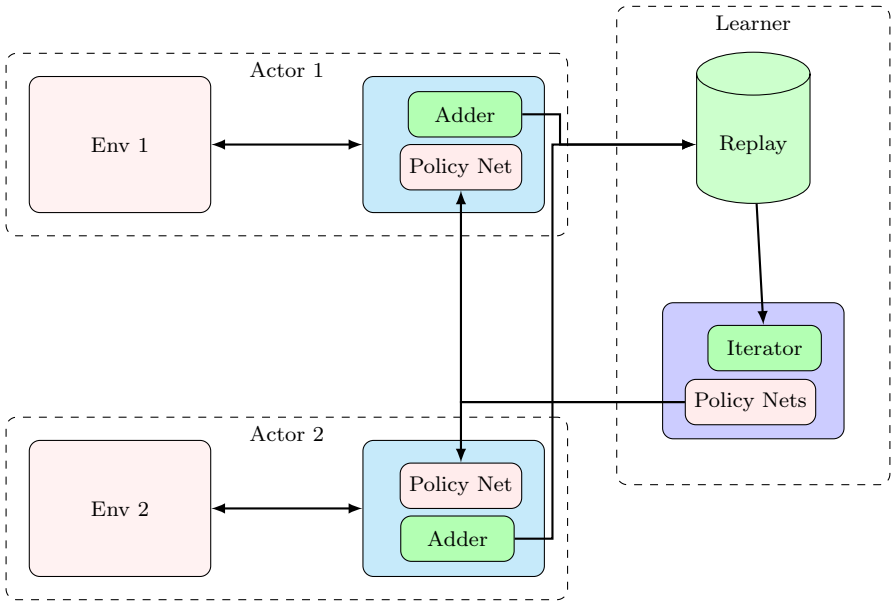

Figure 2: Distributed RL setup with two actors and a central replay buffer and learner.

conversation to achieve a goal at the end, during which both user and chat agent adapt, is a multi-step RL problem (Figure 1, bottom left). This setting could also model a chatbot using tools, e.g. a web browser or slack, to gather information.

RL can involve many steps, where at each step the world changes. As a result, when one gets a reward, one does not know which of the many decisions led to the reward. People call this the *credit assignment problem*. RL, because of the many-steps, is combinatorial and very high-dimensional. Moreover, not knowing the horizon, i.e. the number of decision steps, can create additional trans-dimensional inference complexity. In short, RL is really hard, and the variance of the solutions can be very high. Researchers have invented a series of concepts to keep the variance under check at the expense of introducing bias, including value functions (functions that represent the expected rewards, with some guess as to how to discount future rewards). These concepts are useful in multi-step decision problems, but not always needed for one-step RL.

It is dangerous to import blindly the theory and software of RL methods developed for computer games to the world of language models. Those methods had to deal with asynchronous expensive data generation with game engines and stale replay buffers. Instead, it is important to measure the efficiency of all components in the system and optimize the algorithms for the available hardware and types of interaction.

For tool-use and multi-step assistance, we need multi-step RL for LLMs.

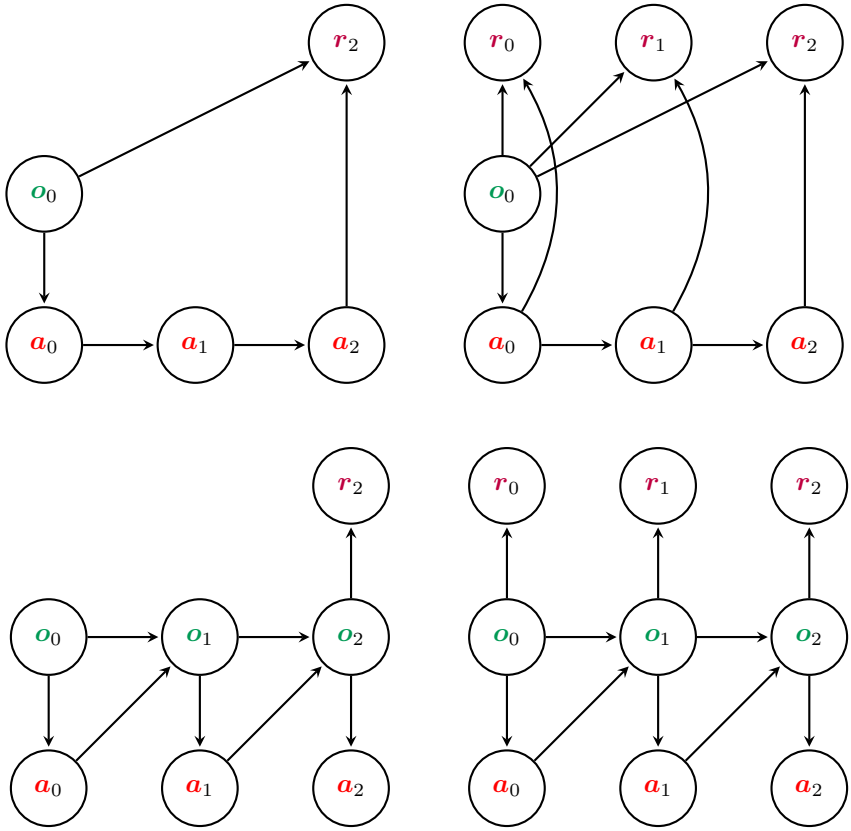

Figure 3: Four Markov decision processes (RL models) with finite horizon $T = 2$. [**Top left**] This is the setting of *outcome rewards* with LLM, such as deepseek-R1 and chatgpt-o1. The agent generates a string of actions (tokens) and gets a reward at the end. Note that we can group the three actions into a single composite action $\boldsymbol{a} = (\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3)$. [**Top right**] This is the setting of *process rewards* with LLMs. Other than observing a reward at each action step, the agent does not observe any other inputs. [**Bottom left**] This is the tool-use or multi-turn conversation setting. Here, the agent may use a tool $\boldsymbol{a}_0$ and sense $\boldsymbol{o}_1$. Note that the environment can evolve over time, something that we could model with a world-model: $P(\boldsymbol{o}_{t+1}|\boldsymbol{o}_t, \boldsymbol{a}_t)$. A chat agent planning an entire multi-turn conversation with the intent of helping a user, where each turn involves an action by the chat agent (text) and observing the action of the user (text reply) is a good example of this. As the conversation progresses the chat agent adapts and so does the user. The game of Go is another example. In both of these, the user is the environment. In truth, if the user is adversarial or manipulative, we are outside the realm of RL and in the world of game theory, but such cases are beyond our scope. [**Bottom right**] Same as previous but with process rewards.

However, to do what deepseek-R1 and chatgpt-o1 do, it may suffice to first solve the one-step RL problem, which is slightly simpler. Some of the software, e.g. for policy gradients introduced later in this document, might be very similar for both one-step and multi-step RL, so we may as well try to use a common flexible software framework.

All RL agents self-learn and self-improve. If designed well, they can construct datasets of increasing quality, which in turn lead to policies of increasing quality. This property of RL agents is essential for both performance and for safety. Note that in one-step RL, we repeat the process of solving one-step problems, and the improvements should grow with each iteration.

There can be harder RL cases not addressed here. Sometimes the decision horizon is unknown or infinite, thus increasing the complexity of inference. Moreover, the time steps can be continuous or interrupt-driven. The actions and observations can be a mix of discrete and continuous, e.g. text documents with illustrations, and the reward can combine multiple desirable components. There are also extensions to multi-agents in both collaborative and adversarial settings.

For didactic reasons, we will cover the simplest case first: one-step RL. We will cover the main algorithms, and this will be enough to explain methods such as deepseek-R1, ReST and ReST^{EM}. Subsequently, we will address multi-step settings, which allow for tool-use. We will avoid introducing more complexity purposely because the simpler methods might be enough for LLM finetuning.

4 One-step RL problem formulation

Many teams, including deepseek-r1 and chatgpt-o1, are trying to maximize the following one-step objective function:

$$J(\boldsymbol{\theta}) = \int r(\boldsymbol{a}, \boldsymbol{o}) \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \, d\boldsymbol{a} d\boldsymbol{o}$$

The symbols could for example represent the following:

- $\boldsymbol{o} \equiv$ question or prompt
- $\boldsymbol{a} \equiv$ answer
- $\pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) \equiv$ LLM model with parameters $\boldsymbol{\theta}$
- $r(\boldsymbol{a}, \boldsymbol{o}) \equiv$ evaluation metric or reward

That is, we are fine-tuning the LLM over all data strings $(\boldsymbol{o}, \boldsymbol{a})$. I'm using the integral sign to denote very large discrete sums.

4.1 Method 1: Policy gradient

This is what people think of as *on-policy* RL or Reinforce. However, in the one-step case, none of the machinery of Reinforce is needed. This is basic gradient

following, and calling it RL is unnecessary. It is said to be on-policy because the policy model that generates the samples is the same as the one being learned.

This approach makes sense when generating the samples is cheaper than learning. That is, when the learner can easily get fresh samples on demand. This seems to be the case with text, but it wasn't the case with expensive game simulation engines, where buffers and replay memories had to be introduced to cache data. The data would then go stale, necessitating off-policy methods, which we will discuss in the following section. For now, we will see how to calculate the gradient of our one-step loss. We will then simply follow the gradient to update the parameters.

Reinforced Self Training ReST is a good example of this approach. It can indeed be used for self-improvement, while also staying close to human demonstrations. In ReST the reward is used to rank samples. Subsequently, ReST re-learns the policy unsing only the highest ranked samples.

Let us start the derivation of the policy gradient with the one-step objective:

$$J(\boldsymbol{\theta}) = \int r(\boldsymbol{a}, \boldsymbol{o}) \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \, d\boldsymbol{a} d\boldsymbol{o}$$

The gradient is trivial to obtain:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \int r(\boldsymbol{a}, \boldsymbol{o}) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \, d\boldsymbol{a} d\boldsymbol{o} \\ &= \int r(\boldsymbol{a}, \boldsymbol{o}) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \, d\boldsymbol{a} d\boldsymbol{o} \end{aligned}$$

If we generate (sample) a prompt and solutions from the policy, that is $\boldsymbol{a}^i \sim \pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o})$ and $\boldsymbol{o}^i \sim P(\boldsymbol{o})$, we can approximate the gradient with Monte Carlo:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{i=1}^{N} r(\boldsymbol{a}^i, \boldsymbol{o}^i) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)$$

This is a necessary baseline when fine-tuning LLMs.

4.1.1 Improvement 1: Baseline subtraction

If we subtract the mean of r and replace it with the following reward:

$$\boldsymbol{r}^i = r(\boldsymbol{a}^i, \boldsymbol{o}^i) - \frac{1}{N} \sum_{j=1}^{N} r(\boldsymbol{a}^j, \boldsymbol{o}^j)$$

we don't change the location of the maximum (it is only a constant vertical shift of the loss function), but we reduce the variance. GPT4 can easily prove why the variance is lower! We could also scale the rewards, but the benefit of this isn't clear.

This trick should always be used. Moreover, it is extra useful when the rewards are binary and we require a more continuous gradual feedback signal. We can think of the new reward as the *advantage* of r with respect to the mean baseline.

4.1.2 Improvement 2: Stay close to a good policy

We can add a KL term to the loss so it's close to the SFT policy $\pi_{\text{SFT}}(\boldsymbol{a} \mid \boldsymbol{o})$:

$$D_{\text{KL}}(\pi_{\boldsymbol{\theta}} \| \pi_{\text{SFT}}) = \int \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) \log \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})}{\pi_{\text{SFT}}(\boldsymbol{a} \mid \boldsymbol{o})} \, d\boldsymbol{a}$$

Sample $\boldsymbol{a}^i \sim \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})$, and use the MC approximation again:

$$D_{\text{KL}}(\pi_{\boldsymbol{\theta}} \| \pi_{\text{SFT}}) \approx \frac{1}{N} \sum_{i=1}^{N} \log \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}{\pi_{\text{SFT}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}$$

The KL can also be estimated by:

$$D_{\text{KL}}(\pi_{\boldsymbol{\theta}} \| \pi_{\text{SFT}}) = \int \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) \left[\log \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})}{\pi_{\text{SFT}}(\boldsymbol{a} \mid \boldsymbol{o})} + \frac{\pi_{\text{SFT}}(\boldsymbol{a} \mid \boldsymbol{o})}{\pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})} - 1 \right] d\boldsymbol{a}$$

Note that the last two terms inside the integral cancel each other, so in effect this is still the same quantity. However it enables us to get better estimates. John Schulmann has a nice blog about it.

The MC approximation is:

$$D_{\text{KL}}(\pi_{\boldsymbol{\theta}} \| \pi_{\text{SFT}}) \approx \frac{1}{N} \sum_{i=1}^{N} \left[-\log \frac{\pi_{\text{SFT}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)} + \frac{\pi_{\text{SFT}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)} - 1 \right]$$

Note that the samples come from $\boldsymbol{a}^i \sim \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})$, and not the previous policy, so there is a typo in equation (1) of the deepseek paper. They seem to have added the math after the fact.

ReST uses the following gradient to trade-off staying close to the expert data and following the gradient:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int \Bigg(& \lambda \int r(\boldsymbol{a}, \boldsymbol{o}) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) d\boldsymbol{a} \\ & + (1 - \lambda) \int r(\boldsymbol{a}, \boldsymbol{o}) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) \pi_{data}(\boldsymbol{a} \mid \boldsymbol{o}) \, d\boldsymbol{a} \Bigg) P(\boldsymbol{o}) d\boldsymbol{o} \end{aligned}$$

The data policy $\pi_{data}(\boldsymbol{a} \mid \boldsymbol{o})$ can correspond to actually using human expert data, or samples from an SFT policy, or any other policy being distilled.

4.1.3 A minimal Python policy-gradient experiment

The accompanying notebook `minimal_policy_gradients_with_llm.ipynb` implements the one-step policy-gradient estimator in the smallest setting where all the moving parts are visible. The "LLM" is a tiny word-level causal language model. It reads a prompt $\boldsymbol{o}$ and samples exactly one answer token $\boldsymbol{a}$. The reward is an exact verifier:

$$\boldsymbol{r}(\boldsymbol{a}, \boldsymbol{o}) = \begin{cases} 1, & \text{if the sampled answer token is correct,} \\ 0, & \text{otherwise.} \end{cases}$$

The policy is written as $\pi(a \mid o; \theta)$, where π denotes the model family and θ denotes its neural-network weights. The notebook first performs a short SFT warm-start, freezes that model as a reference policy π_{ref}, and then applies policy-gradient RL.

In the code we use a mini-batch estimate with an advantage A. The proximity-to-reference penalty enters as a separate term in the loss rather than being folded into the reward. The advantage is computed from the raw reward,

$$A_i = \frac{r_i - \text{mean}(r)}{\text{std}(r) + \epsilon},$$

and a sampled KL penalty against π_{ref} is added to the loss using Schulman's k_3 estimator,

$$\begin{aligned}\widehat{\text{KL}}_{k_3}(a_i, o_i) = &\big(\log \pi(a_i \mid o_i; \theta) - \log \pi_{\text{ref}}(a_i \mid o_i)\big) \\ &+ \exp\big(\log \pi_{\text{ref}}(a_i \mid o_i) - \log \pi(a_i \mid o_i; \theta)\big) - 1,\end{aligned}$$

with $a_i \sim \pi(\cdot \mid o_i; \theta)$. Once the batch

$$\{(o_i, a_i, r_i, A_i, \log \pi_{\text{ref}}(a_i \mid o_i))\}_{i=1}^N$$

has been sampled, the observations o_i, sampled actions a_i, advantages A_i, and reference log-probabilities $\log \pi_{\text{ref}}(a_i \mid o_i)$ are treated as fixed data for the gradient step. The only quantity in the surrogate loss that depends on the current parameters θ is the current-policy log-probability $\log \pi(a_i \mid o_i; \theta)$, which appears both in the policy-gradient term and (through the log-ratio) in the k_3 KL term.

Codebases that use automatic differentiation, like PyTorch, usually use a policy gradients (PG) surrogate loss. We will show that he gradient of this surrogate, with a stop-gradient, is the correct one. The total loss is:

$$\mathcal{L}(\theta) = \underbrace{-\frac{1}{N}\sum_{i=1}^{N} \text{sg}[A_i] \log \pi(a_i \mid o_i; \theta)}_{\mathcal{L}_{\text{PG}}(\theta)} + \underbrace{\beta \frac{1}{N}\sum_{i=1}^{N} \widehat{\text{KL}}_{k_3}(a_i, o_i)}_{\mathcal{L}_{\text{KL}}(\theta)},$$

where sg$[\cdot]$ means "treat this quantity as a constant when differentiating." The stop-gradient applies only to the advantages: the gradient does flow through $\log \pi(a_i \mid o_i; \theta)$ in both the policy-gradient term and the k_3 KL term.

Let us now verify that this is correct. Differentiate the policy-gradient term $\mathcal{L}_{\text{PG}}$ term by term. By linearity of the gradient,

$$\nabla_\theta \mathcal{L}_{\text{PG}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \left(\text{sg}[A_i] \log \pi(a_i \mid o_i; \theta)\right).$$

Using the product rule,

$$\nabla_\theta \left(\text{sg}[A_i] \log \pi(a_i \mid o_i; \theta)\right) = \nabla_\theta \,\text{sg}[A_i] \log \pi(a_i \mid o_i; \theta) + \text{sg}[A_i] \nabla_\theta \log \pi(a_i \mid o_i; \theta).$$

But the stop-gradient makes

$$\nabla_{\boldsymbol{\theta}} \operatorname{sg}[\boldsymbol{A}_i] = 0.$$

Therefore

$$\nabla_{\boldsymbol{\theta}} \left(\operatorname{sg}[\boldsymbol{A}_i] \log \boldsymbol{\pi}(\boldsymbol{a}_i \mid \boldsymbol{o}_i; \boldsymbol{\theta}) \right) = \boldsymbol{A}_i \nabla_{\boldsymbol{\theta}} \log \boldsymbol{\pi}(\boldsymbol{a}_i \mid \boldsymbol{o}_i; \boldsymbol{\theta}),$$

and the policy-gradient piece of the loss reduces to

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathrm{PG}}(\boldsymbol{\theta}) = -\frac{1}{N} \sum_{i=1}^{N} \boldsymbol{A}_i \nabla_{\boldsymbol{\theta}} \log \boldsymbol{\pi}(\boldsymbol{a}_i \mid \boldsymbol{o}_i; \boldsymbol{\theta}).$$

In the code that follows, this is why the policy-gradient loss is implemented as

`pg_loss = -(advantage.detach() * logp).mean()`.

The call to `detach()` enforces the mathematical convention above: the advantage is used as a scalar weight on the log-probability, not as something through which the optimiser should backpropagate during the policy update. In contrast, the k_3 KL term is written as

```
kl_loss = beta * (log_ratio + torch.exp(-log_ratio) - 1).mean(),
```

without any `detach()` on the current-policy log-probability: we want the gradient of the KL penalty to flow back through $\log \boldsymbol{\pi}(\boldsymbol{a}_i \mid \boldsymbol{o}_i; \boldsymbol{\theta})$ and update $\boldsymbol{\theta}$ so the policy is pulled towards $\boldsymbol{\pi}_{\mathrm{ref}}$. The reference log-probability $\log \boldsymbol{\pi}_{\mathrm{ref}}(\boldsymbol{a}_i \mid \boldsymbol{o}_i)$ is computed under `torch.no_grad()` since $\boldsymbol{\pi}_{\mathrm{ref}}$ is frozen.

Experiment setup: prompts, model, and reward

```
pairs = [
    ("Question: what animal says meow ? Answer:", "cat"),
    ("Question: what animal says woof ? Answer:", "dog"),
    ("Question: what animal says moo ? Answer:", "cow"),
    ("Question: what colour is sky ? Answer:", "blue"),
    # ... four more prompt/answer pairs ...
]

class TinyCausalLM(nn.Module):
    def __init__(self, vocab_size: int, d_model: int = 48):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, d_model)
        self.rnn = nn.GRU(d_model, d_model, batch_first=True)
        self.lm_head = nn.Linear(d_model, vocab_size)

    def forward(self, input_ids, lengths):
        hidden_states, _ = self.rnn(self.embed(input_ids))
        last_hidden = hidden_states[
            torch.arange(input_ids.size(0), device=input_ids.device),
            lengths - 1,
        ]
```

```
        return self.lm_head(last_hidden)  # next-token logits

reward = (actions == targets).float()
```

The model is deliberately small: an embedding layer, a GRU, and a linear next-token head. This keeps the experiment runnable on a laptop while preserving the same mathematical object used by an autoregressive LLM: the log-probability of a sampled completion under $\pi(\cdot \mid o; \theta)$. In a real LLM, the action a is a sequence, and the log-probability is the sum of token log-probabilities.

The most important part of the notebook is the policy-gradient update:

The policy-gradient step

```
def policy_gradient_step(model, ref_model, opt, batch_size=64, beta=0.02):
    # 1. Sample prompts o_i.
    indices = torch.randint(0, len(pairs), (batch_size,)).tolist()
    x, lengths, targets = make_prompt_batch(indices)

    # 2. Sample actions a_i from pi(. | o_i; theta).
    logits = model(x, lengths)
    dist = torch.distributions.Categorical(logits=logits)
    actions = dist.sample()
    logp = dist.log_prob(actions)
    entropy = dist.entropy().mean()

    # 3. Evaluate the same actions under the frozen reference policy,
    #    and compute the verifier reward.
    with torch.no_grad():
        ref_logits = ref_model(x, lengths)
        ref_logp = torch.distributions.Categorical(
            logits=ref_logits
        ).log_prob(actions)
        reward = (actions == targets).float()

    # 4. Advantage from the *raw* reward (KL is a separate loss term).
    with torch.no_grad():
        advantage = (reward - reward.mean()) / (reward.std() + 1e-8)

    # 5. Schulman's k3 KL estimator:
    #    k3 = (log pi - log pi_ref) + exp(log pi_ref - log pi) - 1.
    #    log_ratio keeps its grad-graph; ref_logp is no_grad, so the KL
    #    term is differentiable in theta.
    log_ratio = logp - ref_logp
    kl_k3 = log_ratio + torch.exp(-log_ratio) - 1.0   # >= 0 elementwise

    # 6. Total loss: REINFORCE PG loss + beta * KL penalty.
    pg_loss = -(advantage * logp).mean()
    kl_loss = beta * kl_k3.mean()
    loss = pg_loss + kl_loss

    opt.zero_grad(set_to_none=True)
    loss.backward()
```

```
    nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
    opt.step()
```

4.2 Method 2: Importance Sampling

Sometimes we use RL systems where there are many actors collecting data and adding it to a memory system. A learner then extracts samples from this memory to learn. Some of the samples go stale in this asynchronous setup. The mechanism producing the samples (actors) is not the same as the one updating the parameters (learner), so the method is said to be *off-policy*.

Importance sampling (IS) provides a solution for correcting for be samples being off-policy. It proceeds as follows. Let us multiply and divide the integrand of the one-step objective function with an existing *behaviour* policy $\pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o})$. We will act using this policy, but learn another policy, and hence this is often called off-policy learning. In mathematics:

$$J(\boldsymbol{\theta}) = \int r(\boldsymbol{a}, \boldsymbol{o}) \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) \pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \frac{1}{\pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o})} d\boldsymbol{a} d\boldsymbol{o}$$

If we sample $\boldsymbol{a}^i \sim \pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o})$ and $\boldsymbol{o}^i \sim P(\boldsymbol{o})$, we can replace the integral with the following Monte Carlo approximation, known as the importance sampling (IS) estimate:

$$J(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{i=1}^{N} \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}{\pi_{\text{old}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)} r(\boldsymbol{a}^i, \boldsymbol{o}^i)$$

4.2.1 IS improvement: Clip the importance weights

The importance weights

$$w^i = \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}{\pi_{\text{old}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}$$

can grow and cause instability, especially because the space of all strings over which we are computing this ratio is high-dimensional. To safeguard against high variance and instability, we have to truncate/clip the importance weights in smart ways.

Let us consider our off-policy objective again:

$$\begin{aligned} J(\boldsymbol{\theta}) &= \int \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})}{\pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o})} r(\boldsymbol{a}, \boldsymbol{o}) \pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \, d\boldsymbol{a} d\boldsymbol{o} \\ &= \int w(\boldsymbol{\theta}) r(\boldsymbol{a}, \boldsymbol{o}) \pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \, d\boldsymbol{a} d\boldsymbol{o} \end{aligned}$$

where $w(\boldsymbol{\theta})$ is the importance ratio. PPO modifies this objective to penalize changes to the policy that move $w(\boldsymbol{\theta})$ away from 1, as follows:

$$J(\boldsymbol{\theta}) = \int \min\left(w(\boldsymbol{\theta}) r(\boldsymbol{a}, \boldsymbol{o}), \text{clip}[w(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon] r(\boldsymbol{a}, \boldsymbol{o})\right) \pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \, d\boldsymbol{a} d\boldsymbol{o}$$

where, as stated in the PPO paper, ϵ is a hyperparameter, say, $\epsilon = 0.2$. The first term inside the min is $J(\boldsymbol{\theta})$. The second term, $\text{clip}[w(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon]r(\boldsymbol{a}, \boldsymbol{o}))$, clips the importance ratio, thus removing the incentive for w to move outside of the interval $[1-\epsilon, 1+\epsilon]$. We take the minimum so the objective is a lower bound on the unclipped objective.

4.2.2 Deepseek

Deepseek-R1 combines clipped importance sampling, with baseline subtraction and KL proximity to a reference policy to train their reasoning models. We have now covered all the ingredients, so it is just a matter of putting them together. Of course, the real challenges arise in the implementation of the infrastructure and data.

$$J(\boldsymbol{\theta}) \approx \frac{1}{N}\sum_{i=1}^{N} \min\left(\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}{\pi_{\text{old}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}\boldsymbol{r}^i, \text{clip}\left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}{\pi_{\text{old}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}, 1-\epsilon, 1+\epsilon\right]\boldsymbol{r}^i\right)$$
$$-\frac{\beta}{N}\sum_{i=1}^{N}\left[-\log\frac{\pi_{\text{SFT}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)} + \frac{\pi_{\text{SFT}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)} - 1\right]$$

To obtain the first Monte Carlo estimate we use the samples $\boldsymbol{a}^i \sim \pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o})$, but if we want to keep things unbiased, the Monte Carlo estimate of the second (i.e. KL) term should use $\boldsymbol{a}^i \sim \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})$.

We have subtracted the mean baseline from the reward:

$$\boldsymbol{r}^i = r(\boldsymbol{a}^i, \boldsymbol{o}^i) - \frac{1}{N}\sum_{j=1}^{N} r(\boldsymbol{a}^j, \boldsymbol{o}^j)$$

but unlike deepseek-R1, we haven't divided by the standard deviation. This is worth testing empirically.

Note: in this version, we are sampling one action per observation. It is also possible to sample several actions per observation to reduce variance. Deepseek-R1 basically does this, with the gradient update consisting of several samples of the actions with a single question. This technique is known as common random numbers in stochastic approximation.

4.3 Method 3: Maximum likelihood via expectation maximization (EM)

This is an old idea recently brought back under the name ReST^{EM}. It provides us with a statistical way of interpreting the problem, as opposed to the RL perspective. It results in the same gradient as we will see. That is, this note is about different perspectives ending in similar algorithms.

The derivation proceeds as follows. Introduce the dummy variable H such that:

$$P(H = 1 \mid \boldsymbol{a}, \boldsymbol{o}) = r(\boldsymbol{a}, \boldsymbol{o})$$

The likelihood is then equal to the one-step RL objective:

$$\begin{aligned} P_{\boldsymbol{\theta}}(H=1) &= \int P(H=1 \mid \boldsymbol{a}, \boldsymbol{o}) \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \, d\boldsymbol{a} d\boldsymbol{o} \\ &= \int r(\boldsymbol{a}, \boldsymbol{o}) \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \, d\boldsymbol{a} d\boldsymbol{o} \\ &= J(\boldsymbol{\theta}) \end{aligned}$$

We can maximize the likelihood using the EM algorithm. EM uses the following Jensen inequality (this is a variational inference idea that GPT can easily prove):

$$\begin{aligned} \log P_{\boldsymbol{\theta}}(H=1) \geq & \int P_{\text{old}}(\boldsymbol{a}, \boldsymbol{o} \mid H=1) \log P_{\boldsymbol{\theta}}(\boldsymbol{a}, \boldsymbol{o}, H=1) \, d\boldsymbol{a} d\boldsymbol{o} \\ & - \int P_{\text{old}}(\boldsymbol{a}, \boldsymbol{o} \mid H=1) \log P_{\text{old}}(\boldsymbol{a}, \boldsymbol{o} \mid H=1) \, d\boldsymbol{a} d\boldsymbol{o} \end{aligned}$$

The first term is a lower bound, known as the expected complete data likelihood $L(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$. The second term, known as the entropy, is independent of $\boldsymbol{\theta}$ so we can ignore it. We only need to maximize $L(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$. We do it as follows:

$$\begin{aligned} L(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) &= \int P_{\text{old}}(\boldsymbol{a}, \boldsymbol{o} \mid H=1) \log P_{\boldsymbol{\theta}}(\boldsymbol{a}, \boldsymbol{o}, H=1) \, d\boldsymbol{a} d\boldsymbol{o} \\ &= \int \frac{P(H=1 \mid \boldsymbol{a}, \boldsymbol{o}) \pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o})}{P_{\text{old}}(H=1)} \log P_{\boldsymbol{\theta}}(\boldsymbol{a}, \boldsymbol{o}, H=1) \, d\boldsymbol{a} d\boldsymbol{o} \\ &= \frac{1}{P_{\text{old}}(H=1)} \int r(\boldsymbol{a}, \boldsymbol{o}) \pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \; \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) \, d\boldsymbol{a} d\boldsymbol{o} + \text{const}. \end{aligned}$$

We applied Bayes rule to get the second line. The normalizing constant is:

$$P_{\text{old}}(H=1) = \int P(H=1 \mid \boldsymbol{a}, \boldsymbol{o}) \pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \, d\boldsymbol{a} d\boldsymbol{o}$$

Finally, sample $\boldsymbol{a}^i, \boldsymbol{o}^i \sim \pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o})$, then the MC approximation of the loss is:

$$L(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) \approx \frac{\sum_{i=1}^N r(\boldsymbol{a}^i, \boldsymbol{o}^i) \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i)}{\sum_{j=1}^N r(\boldsymbol{a}^j, \boldsymbol{o}^j)}$$

The ML gradient approximation is as follows:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) &= \sum_{i=1}^N \frac{r(\boldsymbol{a}^i, \boldsymbol{o}^i)}{\sum_{j=1}^N r(\boldsymbol{a}^j, \boldsymbol{o}^j)} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i) \\ &= \text{const} \sum_{i=1}^N r(\boldsymbol{a}^i, \boldsymbol{o}^i) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}^i \mid \boldsymbol{o}^i) \\ &= \text{const} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \end{aligned}$$

That is, it is an irrelevant constant times the online policy gradient.

This is however assuming that $\pi_{\text{old}}(\boldsymbol{a} \mid \boldsymbol{o})$ is the the latest policy, i.e. that we are on-policy. It may be that we can also use this method off-policy (with an older policy or even the SFT distribution) as argued in the ReST EM paper, but that needs to be evaluated, and the expectation is that being on-policy for language modelling will likely be better.

4.4 Method 4: Maximum likelihood via online EM

It is possible to come up with an online version of the EM algorithm, which could be run even at pre-training time provided we have good reward signals for pre-training.

Recall the auxiliary variable $H \in \{0, 1\}$ with $P(H{=}1 \mid \boldsymbol{a}, \boldsymbol{o}) = r(\boldsymbol{a}, \boldsymbol{o})$ and the EM lower bound

$$L(\boldsymbol{\theta} \mid \boldsymbol{\theta}^{\text{old}}) \;=\; \frac{\mathbb{E}_{\boldsymbol{o}\sim P,\, \boldsymbol{a}\sim\pi_{\boldsymbol{\theta}^{\text{old}}}(\cdot|\boldsymbol{o})}[r(\boldsymbol{a}, \boldsymbol{o})\, \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})]}{\mathbb{E}_{\boldsymbol{o}\sim P,\, \boldsymbol{a}\sim\pi_{\boldsymbol{\theta}^{\text{old}}}(\cdot|\boldsymbol{o})}[r(\boldsymbol{a}, \boldsymbol{o})]} \;+\; \text{const.}$$

The corresponding gradient is

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta} \mid \boldsymbol{\theta}^{\text{old}}) \;=\; \frac{\mathbb{E}[r(\boldsymbol{a}, \boldsymbol{o})\, \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})]}{\mathbb{E}[r(\boldsymbol{a}, \boldsymbol{o})]}.$$

The online EM algorithm consists again of two steps: computing the expectations with running averages and maximising over the parameters.

4.4.1 E-step

In online learning, we typically process a stream (or mini-batches) of $(\boldsymbol{o}_t, \boldsymbol{a}_t)$ drawn on-policy, i.e. $\boldsymbol{o}_t \sim P(\boldsymbol{o})$ and $\boldsymbol{a}_t \sim \pi_{\boldsymbol{\theta}_{t-1}}(\cdot \mid \boldsymbol{o}_t)$, and observe $\boldsymbol{r}_t = r(\boldsymbol{a}_t, \boldsymbol{o}_t)$. However, with LLMs we often draw a large number of actions a for each observation o (context, instruction, question, or state). If we do this data augmentation, then we can build more interesting algorithms, for example by using the reward to filter for the top candidates. Incidentally, various versions have been explored before in statistics, see eg this paper. However, we may be able to do a few more aggressive things with $\boldsymbol{r}_t$ in LLMs.

Next, consider the log-likelihood gradient evaluated at the previous parameters:

$$s_t \;=\; \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}_{t-1}}(\boldsymbol{a}_t \mid \boldsymbol{o}_t),$$

and maintain Robbins–Monro averages with stepsizes γ_t (e.g. $\gamma_t = (t + t_0)^{-\alpha}$ with $\alpha \in (0.5, 1]$):

$$G_t \;=\; (1 - \gamma_t)\, G_{t-1} \;+\; \gamma_t\, \boldsymbol{r}_t\, s_t, \tag{1}$$

$$M_t \;=\; (1 - \gamma_t)\, M_{t-1} \;+\; \gamma_t\, \boldsymbol{r}_t. \tag{2}$$

Here $G_t \approx \mathbb{E}[\,\boldsymbol{r}\, \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})\,]$ and $M_t \approx \mathbb{E}[\,\boldsymbol{r}\,]$ are the online estimates of the sufficient statistics of the complete-data objective.

4.4.2 M-step

With learning rate η_t and a small $\varepsilon > 0$,

$$\boldsymbol{\theta}_t \;=\; \boldsymbol{\theta}_{t-1} \;+\; \eta_t \,\frac{G_t}{\max(M_t, \varepsilon)}.$$

This performs a generalized EM step that maximizes the current lower bound using the estimated sufficient statistics (1)–(2). When M_t is very small, the update can be skipped for stability.

Mini-batch of actions per single observation: For a mini-batch B_t, we could replace $\boldsymbol{r}_t s_t$ in (1) by

$$\frac{1}{|B_t|} \sum_{(\boldsymbol{a})\in B_t} r(\boldsymbol{a}, \boldsymbol{o})\, \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}_{t-1}}(\boldsymbol{a} \mid \boldsymbol{o})$$

and replace $\boldsymbol{r}_t$ in (2) by the batch mean of $\boldsymbol{r}$.

Off-policy variant with behaviour policy μ: If samples come from a behavior policy $\mu(\boldsymbol{a} \mid \boldsymbol{o})$ (e.g. an older policy or SFT), use self-normalized importance weights

$$w_t \;=\; \frac{\pi_{\boldsymbol{\theta}_{t-1}}(\boldsymbol{a}_t \mid \boldsymbol{o}_t)}{\mu(\boldsymbol{a}_t \mid \boldsymbol{o}_t)} \quad \text{(optionally clipped)},$$

and update

$$\begin{aligned} G_t &= (1-\gamma_t) G_{t-1} + \gamma_t \,(\boldsymbol{r}_t w_t)\, s_t, \\ M_t &= (1-\gamma_t) M_{t-1} + \gamma_t \,(\boldsymbol{r}_t w_t), \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} + \eta_t \,\frac{G_t}{\max(M_t, \varepsilon)}. \end{aligned}$$

Some useful remarks:

1. Using a constant $\gamma_t \equiv \gamma \in (0, 1)$ yields a forgetting factor that can track non-stationarity.
2. A moving baseline b_t may reduce variance by replacing $\boldsymbol{r}_t$ with $(\boldsymbol{r}_t - b_t)$ in G_t only (do not subtract b_t from M_t). This is a form of continual learning.
3. KL trust-region control can be added by penalizing $\mathrm{KL}\big(\pi_{\boldsymbol{\theta}} \,\|\, \pi_{\boldsymbol{\theta}_{t-1}}\big)$ or by step-size adaptation.

5 Many-step RL problem formulation

To attack tool-use or multi-turn conversations, or any other setting with multiple external feedback, we need to introduce the following concepts:

$$\begin{aligned}
\tau &= \{\boldsymbol{o}_0, \boldsymbol{a}_0, \boldsymbol{o}_1, \boldsymbol{a}_1, \ldots, \boldsymbol{o}_T\} && \text{Experience (aka, episode or trajectory)} \\
\boldsymbol{a}_t &\sim \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t \mid \boldsymbol{o}_t) && \text{Policy with parameters } \boldsymbol{\theta} \\
\boldsymbol{o}_{t+1} &\sim P(\boldsymbol{o}_{t+1} \mid \boldsymbol{a}_t, \boldsymbol{o}_t) && \text{Markov world model} \\
\boldsymbol{r}_t &= \boldsymbol{r}(\boldsymbol{o}_t, \boldsymbol{a}_t) && \text{Reward function} \\
\boldsymbol{R}_t &= \sum_{n=t}^{T} \boldsymbol{r}(\boldsymbol{o}_n, \boldsymbol{a}_n) && \text{Returns}
\end{aligned}$$

With these concepts, the (multi-step) RL objective is:

$$\begin{aligned}
J^{\pi}(\boldsymbol{\theta}) &= \mathbb{E}_{\tau}\left[\sum_{t=0}^{T} \boldsymbol{r}(\boldsymbol{o}_t, \boldsymbol{a}_t)\right] \\
&= \int \left(\sum_{t=0}^{T} \boldsymbol{r}(\boldsymbol{o}_t, \boldsymbol{a}_t)\right) P_{\boldsymbol{\theta}}(\boldsymbol{a}_{0:T}, \boldsymbol{o}_{0:T})\, d\boldsymbol{a}_{0:T} d\boldsymbol{o}_{0:T} \\
&= \int \boldsymbol{R}_0 \prod_{t=0}^{T} \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t \mid \boldsymbol{o}_t) P(\boldsymbol{o}_{t+1} \mid \boldsymbol{a}_t, \boldsymbol{o}_t)\, d\boldsymbol{a}_{0:T} d\boldsymbol{o}_{0:T}
\end{aligned}$$

We can use sampled episodes to approximate the objective:

$$J^{\pi}(\boldsymbol{\theta}) = \mathbb{E}_{\tau}\left[\boldsymbol{R}_0\right] \approx \frac{1}{N} \sum_{i=1}^{N} \boldsymbol{R}_0^i$$

When the horizon is infinite, we introduce a discount factor γ defined on the unit interval $[0, 1]$ as follows:

$$\boldsymbol{R}_0 = \sum_{n=0}^{\infty} \gamma^n \boldsymbol{r}(\boldsymbol{o}_n, \boldsymbol{a}_n)$$

5.1 Method 5: Multi-step policy gradient

The gradient of the objective is known as the policy gradient:

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} J^{\pi}(\boldsymbol{\theta}) &= \mathbb{E}_P \left[\int \sum_{t=0}^{T} \boldsymbol{r}(\boldsymbol{o}_t, \boldsymbol{a}_t) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_{0:T} \mid \boldsymbol{o}_{0:T}) \, d\boldsymbol{a}_{0:T} \right] \\
&= \mathbb{E}_P \left[\int \sum_{t=0}^{T} \boldsymbol{r}(\boldsymbol{o}_t, \boldsymbol{a}_t) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_{0:T} \mid \boldsymbol{o}_{0:T}) \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_{0:T} \mid \boldsymbol{o}_{0:T}) \, d\boldsymbol{a}_{0:T} \right] \\
&= \mathbb{E}_P \left[\int \sum_{t=0}^{T} \boldsymbol{r}(\boldsymbol{o}_t, \boldsymbol{a}_t) \left(\sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t \mid \boldsymbol{o}_t) \right) \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_{0:T} \mid \boldsymbol{o}_{0:T}) \, d\boldsymbol{a}_{0:T} \right] \\
&= \mathbb{E}_\tau \left[\sum_{t=0}^{T} \left(\sum_{n=0}^{T} \boldsymbol{r}(\boldsymbol{o}_n, \boldsymbol{a}_n) \right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t \mid \boldsymbol{o}_t) \right] \\
&= \mathbb{E}_\tau \left[\sum_{t=0}^{T} \left(\sum_{n=t}^{T} \boldsymbol{r}(\boldsymbol{o}_n, \boldsymbol{a}_n) \right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t \mid \boldsymbol{o}_t) \right] \\
&= \mathbb{E}_\tau \left[\sum_{t=0}^{T} \boldsymbol{R}_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t \mid \boldsymbol{o}_t) \right]
\end{aligned}
$$

Where in the penultimate step we used the reasoning that an action at a particular time only influences future rewards, not past rewards. As before, we can use sampled trajectories to approximate the gradient:

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} J^{\pi}(\boldsymbol{\theta}) &= \mathbb{E}_\tau \left[\sum_{t=0}^{T} \boldsymbol{R}_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t \mid \boldsymbol{o}_t) \right] \\
&\approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \boldsymbol{R}_t^i \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t^i \mid \boldsymbol{o}_t^i)
\end{aligned}
$$

The algorithm, known as *reinforce* or policy gradients, iterates between three steps:

1. Generate a batch of episodes using the current policy.
2. Compute the returns.
3. Update the parameters using the Monte Carlo gradient estimate.

Note that if the parameter updates are much faster, we can try to do multiple parameter updates per iteration. The ReST paper mentions several of these considerations.

As before, we we should also subtract the mean of the returns to reduce variance. DeepSeekMath subtracts the mean of the returns from each reward. It also divides by the standard deviation.

As before, we can also apply a KL term to stay close to another distribution, e.g. the SFT one. If we use a memory full of experiences, some off-policy, we

can also do clipped importance sampling. All these things should be ablated of course.

Putting this together, we get the DeepSeekMath objective:

$$J(\boldsymbol{\theta}) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\min\left(\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t^i \mid \boldsymbol{o}_t^i)}{\pi_{\text{old}}(\boldsymbol{a}_t^i \mid \boldsymbol{o}_t^i)}\boldsymbol{R}_t^i, \text{clip}\left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t^i \mid \boldsymbol{o}_t^i)}{\pi_{\text{old}}(\boldsymbol{a}_t^i \mid \boldsymbol{o}_t^i)}, 1-\epsilon, 1+\epsilon\right]\boldsymbol{R}_t^i\right)$$
$$-\frac{\beta}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\left[-\log\frac{\pi_{\text{SFT}}(\boldsymbol{a}_t^i \mid \boldsymbol{o}_t^i)}{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t^i \mid \boldsymbol{o}_t^i)} + \frac{\pi_{\text{SFT}}(\boldsymbol{a}_t^i \mid \boldsymbol{o}_t^i)}{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t^i \mid \boldsymbol{o}_t^i)} - 1\right]$$

The DeepSeekMath algorithm has an outer loop to learn the reward function.

There are some subleties with respect to the importance weights (marginal vs joint space) that have not been explained here. The ACER paper provides a lot more detail.

Here too one could sample several actions per question to reduce variance.

6 Imitation learning and the DAgger algorithm

Imitation learning assumes access to an expert policy $\pi^*(\boldsymbol{a} \mid \boldsymbol{o})$ and aims to learn a student policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})$ that performs well by mimicking the expert. The simplest approach is *behavioural cloning*, which uses supervised learning to minimise the cross-entropy loss:

$$\mathcal{L}_{\text{BC}}(\boldsymbol{\theta}) = -\mathbb{E}_{(\boldsymbol{o},\boldsymbol{a})\sim\mathcal{D}^*}\left[\log \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})\right],$$

where $\mathcal{D}^*$ is a dataset of expert trajectories: $(\boldsymbol{o}, \boldsymbol{a}) \sim (\boldsymbol{o}, \pi^*(\boldsymbol{a} \mid \boldsymbol{o}))$. Behavioural cloning assumes that $\pi_{\boldsymbol{\theta}}$ will only ever see inputs $\boldsymbol{o}$ drawn from the expert's distribution. In reality, the learned policy drifts from the expert and induces its own distribution of states and observations—causing a mismatch between train and test distributions known as *covariate shift*. Errors compound over time, leading to significant performance degradation.

DAgger (Dataset Aggregation) addresses this issue by closing the loop between learning and data collection. At each iteration i, we roll out the current policy π_i to generate states (observations) $\boldsymbol{o}$, query the expert for the corresponding action $\boldsymbol{a}^* \sim \pi^*(\cdot \mid \boldsymbol{o})$, and add these $(\boldsymbol{o}, \boldsymbol{a}^*)$ pairs to the dataset $\mathcal{D}$. The policy is then updated by minimizing the behavioural cloning loss on the aggregated dataset:

$$\mathcal{D} \leftarrow \mathcal{D} \cup \{(\boldsymbol{o}, \pi^*(\boldsymbol{o}))\}, \quad \pi_{i+1} = \arg\min_{\boldsymbol{\theta}} -\mathbb{E}_{(\boldsymbol{o},\boldsymbol{a})\sim\mathcal{D}}\left[\log \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})\right].$$

DAgger therefore iteratively aligns the learned policy with the expert on the distribution of observations *induced by the learner itself. This is crucial: rather than assuming expert-like observations, DAgger uses the observations that the learning policy produces.*

6.1 DAgger vs RL

Reinforcement learning is different from imitation learning in that it does not assume access to expert actions. Instead, it uses scalar reward signals $\boldsymbol{r}(\boldsymbol{a}, \boldsymbol{o})$ to improve the policy. In our one-step objective:

$$J(\boldsymbol{\theta}) = \int r(\boldsymbol{a}, \boldsymbol{o}) \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \, d\boldsymbol{a} d\boldsymbol{o},$$

we require a reward function r to assign values to actions, whereas imitation learning replaces $r(\boldsymbol{a}, \boldsymbol{o})$ with an indicator of whether the action matches the expert.

DAgger interpolates between these worlds. It allows the agent to generate its own data (like in RL), but uses full-action supervision from the expert (like in supervised learning). Unlike standard RL, DAgger does not require scalar rewards or exploration. Unlike pure supervised learning, it does not assume that expert-collected data is sufficient to learn a good policy.

6.2 When SFT and DAgger are the same

If the policy never induces new states—e.g., if every $\boldsymbol{o}$ is provided by people and the agent takes only one step—then DAgger reduces to behavioural cloning (SFT). In that case, *the learner's policy has no influence on the observation distribution:*

$$\boldsymbol{o} \sim P(\boldsymbol{o}), \qquad \boldsymbol{a}^* \sim \pi^*(\cdot \mid \boldsymbol{o}),$$

and no feedback loop is required. This is exactly the setting of supervised fine-tuning (SFT) of LLMs on human demonstrations, where the agent is always fed static prompts o and produces one output a.

6.3 Connection to one-step RL

In our one-step RL framework, we can reinterpret imitation learning by defining a synthetic reward:

$$\boldsymbol{r}_{\text{imitation}}(\boldsymbol{a}, \boldsymbol{o}) = \mathbb{I}[\boldsymbol{a} = \pi^*(\boldsymbol{o})],$$

and optimizing:

$$J_{\text{imit}}(\boldsymbol{\theta}) = \int \mathbb{I}[\boldsymbol{a} = \pi^*(\boldsymbol{o})] \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}) P(\boldsymbol{o}) \, d\boldsymbol{a} d\boldsymbol{o}.$$

This gives maximum reward to actions that exactly match the expert. Viewed this way, imitation learning is equivalent to reward maximization under a degenerate 0/1 reward.

7 Example: Agentic tool-use

For agents,the supervised datum can be an entire interaction trajectory. In a tool-use setting, the model may need to ask a clarifying question, call a tool, read the tool's observation, call a second tool, and only then answer the user.

The GLM-4.5 paper uses this idea as a cold start for agentic abilities. Their SFT data includes basic tool use, authentic project-development coding tasks, and long-context understanding tasks. The automatic construction pipeline in the GLM-4.5 paper has four steps.

1. **Collect tools and frameworks.** Gather agentic frameworks, real-world APIs, MCP servers, and simulated tools constructed by LLMs.
2. **Synthesize tasks.** Generate user tasks over those tools. Mature frameworks can be read by an LLM to produce relevant queries. The tasks include both single-step and multi-step tool use.
3. **Generate trajectories.** Use existing LLMs to produce tool-call trajectories. For some tasks, an LLM acts as a user simulator, converting a one-shot task into a multi-turn dialogue.
4. **Filter for quality.** Judge agents keep only successful trajectories. The broader rejection-sampling pipeline also removes malformed, truncated, repetitive, or too-short samples; checks objective correctness; uses reward models for subjective tasks; and verifies tool-call protocol and terminal state for tool-use tasks.

In our notation, an ordinary SFT example is roughly

$$(\boldsymbol{o}, \boldsymbol{a}) \sim \mathcal{D}_{\mathrm{SFT}}, \qquad \max_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o}).$$

An agentic SFT example is instead a trajectory

$$\tau = (\boldsymbol{o}_0, \boldsymbol{a}_1, \boldsymbol{o}_1, \boldsymbol{a}_2, \boldsymbol{o}_2, \ldots, \boldsymbol{a}_T, \boldsymbol{o}_T),$$

where $\boldsymbol{a}_t$ may be a natural-language reply, a clarifying question, or a tool call, and $\boldsymbol{o}_t$ may be a user message, a tool response, or another environment observation. The SFT loss is still imitation learning, but now it imitates selected assistant actions inside a trajectory:

$$\mathcal{L}_{\text{agentic SFT}}(\boldsymbol{\theta}) = -\sum_{t=1}^{T} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t^{\star} \mid \boldsymbol{o}_{0:t-1}, \boldsymbol{a}_{<t}^{\star}).$$

The star means that the action came from a retained expert or expert-like trajectory. In words: learn the next assistant action given the dialogue, previous tool calls, and previous observations.

Small practical examples

The following boxes are compact versions of the kinds of demonstrations that can be used as SFT targets after quality filtering.

1. Single-step tool call: calendar lookup

```
<|user|>
Do I have any meetings tomorrow afternoon?

<|assistant|>
<tool_call>list_calendar_events
<arg_key>date</arg_key><arg_value>2026-04-22</arg_value>
<arg_key>time_range</arg_key><arg_value>afternoon</arg_value>
</tool_call>

<|observation|>
<tool_response>[{"title":"Research review","time":"15:00"},
{"title":"1:1","time":"16:30"}]</tool_response>

<|assistant|>
Yes. You have Research review at 15:00 and a 1:1 at 16:30.
```

Here the trajectory is short:

$$(\boldsymbol{o}_0, \boldsymbol{a}_1, \boldsymbol{o}_1, \boldsymbol{a}_{\text{final}}).$$

The action $\boldsymbol{a}_1$ is the calendar call, and $\boldsymbol{o}_1$ is the tool response. The final answer is grounded in the observation rather than guessed from the prompt.

2. Multi-step tool use: travel assistant

```
<|user|>
Find the cheapest flight to Paris next Friday and a central hotel under
     GBP 250.

<|assistant|>
<tool_call>search_flights
<arg_key>destination</arg_key><arg_value>Paris</arg_value>
<arg_key>date</arg_key><arg_value>2026-04-24</arg_value>
</tool_call>

<|observation|>
<tool_response>{"airline":"Air France","price":"GBP 118","arrival
     ":"10:20"}</tool_response>

<|assistant|>
<tool_call>search_hotels
<arg_key>city</arg_key><arg_value>Paris</arg_value>
<arg_key>area</arg_key><arg_value>city center</arg_value>
<arg_key>max_price</arg_key><arg_value>250</arg_value>
</tool_call>

<|observation|>
<tool_response>[{"hotel":"Hotel A","price":"GBP 210"},
{"hotel":"Hotel B","price":"GBP 245"}]</tool_response>

<|assistant|>
Air France is GBP 118, arriving at 10:20. Hotel A is GBP 210 and Hotel B
```

```
    is GBP 245.
```

This is already a many-step decision problem. The second action depends on the first observation, and the final answer summarizes information gathered from multiple tools. In RL notation, a later policy call is conditioned on the partial history:

$$a_2 \sim \pi_\theta(\cdot \mid o_0, a_1, o_1).$$

3. Coding agent: edit a file with tools

```
<|user|>
Open app.py, fix division by zero, and save the patch.

<|assistant|>
<tool_call>read_file
<arg_key>path</arg_key><arg_value>app.py</arg_value>
</tool_call>

<|observation|>
<tool_response>
def score(x, y):
    return x / y
</tool_response>

<|assistant|>
<tool_call>write_file
<arg_key>path</arg_key><arg_value>app.py</arg_value>
<arg_key>content</arg_key>
<arg_value>
def score(x, y):
    if y == 0:
        return 0
    return x / y
</arg_value>
</tool_call>

<|observation|>
<tool_response>{"status":"success"}</tool_response>

<|assistant|>
I added a guard for y == 0 and saved the patch to app.py.
```

This shows why the paper cares about reducing character escaping. The code inside `<arg_value>` is readable and does not require the model to learn a large number of backslashes merely to represent a Python function.

4. Multi-turn agent with a simulated user

```
<|user|>
Please schedule lunch with Sarah next week.

<|assistant|>
What day next week works best, and should I optimize for Sarah's calendar
     or yours first?

<|user|>
Try Wednesday, and optimize for both of us.

<|assistant|>
<tool_call>check_availability
<arg_key>person</arg_key><arg_value>user</arg_value>
<arg_key>date</arg_key><arg_value>2026-04-29</arg_value>
</tool_call>
<tool_call>check_availability
<arg_key>person</arg_key><arg_value>Sarah</arg_value>
<arg_key>date</arg_key><arg_value>2026-04-29</arg_value>
</tool_call>

<|observation|>
<tool_response>{"user":["12:00","13:00"]}</tool_response>
<tool_response>{"Sarah":["13:00","13:30"]}</tool_response>

<|assistant|>
You are both free at 13:00 on Wednesday. Shall I book that?
```

This final example includes a user-simulator turn. The assistant first chooses an ordinary text action $\boldsymbol{a}_1$ because the initial observation $\boldsymbol{o}_0$ is underspecified. Only after receiving the user reply does it switch to tool actions.

Agentic SFT teaches several coupled behaviours: when to use a tool, which tool to use, how to format the call, how to read tool outputs, how to continue a multi-step plan, when to ask a clarifying question, and when to stop with a user-facing answer. The filtering stage matters because it converts many sampled trajectories into a smaller dataset of successful demonstrations. Recall the start of this paper. Selection is important for intelligent behaviour.

7.1 Function Calling RL in GLM-4.5

After SFT, the GLM-4.5 paper further trains tool use with Function Calling RL. This is the same generate-and-select principle, but now the current policy generates the actions and a reward signal selects which behaviours to reinforce. The paper divides Function Calling RL into two regimes.

Step-wise rule-based RL. For tasks with a clear tool-invocation procedure, the training data provides a ground-truth next function call $\boldsymbol{a}_t^\star$ at each step. Given the task and previous steps, the model generates the next assistant action $\boldsymbol{a}_t$, which may be either a function call or a user-facing response. The reward

is strict:

$$r_t = \begin{cases} 1, & \text{FormatCorrect}(a_t) \ \wedge \ \text{Match}(a_t, a_t^\star), \\ 0, & \text{otherwise.} \end{cases}$$

The match requires the function name, parameters, and fields to be exactly correct. This is useful when the desired decision flow is known: it gives dense, per-step supervision and strongly enforces the tool protocol. For readers familiar with causality, this is an alternative to training with causal effects weights.

End-to-end multi-turn RL. For more open-ended tasks, the model first rolls out a full trajectory

$$\tau = (o_0, a_1, o_1, \ldots, a_T, o_T),$$

after interacting with tools and sometimes an LLM-simulated user. The whole trajectory then receives a terminal return:

$$r(\tau) = \begin{cases} 1, & \text{FormatCorrect}(a_{1:T}) \ \wedge \ \text{TaskCompleted}(I, o_0, a_1, o_1, \ldots, a_T, o_T), \\ 0, & \text{otherwise.} \end{cases}$$

Here I denotes the task. The task-completion check can be a predefined environment rule or an LLM judge agent. The paper applies this setting to single-turn multi-step tasks, where the user gives enough information at the start but multiple tools are needed, and to multi-turn multi-step tasks, where the model must interact with both tools and a simulated user to obtain missing information.

A practical implementation detail from the paper is to penalize malformed tool use during agent rollouts: if the model fails to produce the correct tool format, the process can be halted and the trace receives zero reward. For policy optimization, only the model-generated tokens are optimized; environment observations such as tool outputs or user-simulator replies are conditioning context, not targets to be predicted. This is exactly the separation in the RL diagram: the policy controls a, the environment returns o and r, and learning adjusts π_θ.

8 RL as Self-Improvement

When people say a foundation model can *self-improve*, they often mean an iterative loop of the form:

$$\text{Generate} \ \rightarrow \ \text{Score/Select} \ \rightarrow \ \text{Learn} \ \rightarrow \ \text{Repeat}.$$

This loop is what RL does. Let the model be a stochastic policy $\pi_\theta(a \mid o)$, where o is the prompt/context and a is the completion. Let $r(a, o)$ be a scalar outcome reward/score.

A (vanilla) policy-gradient objective is

$$J(\theta) \ = \ \mathbb{E}_{o \sim P(o),\, a \sim \pi_\theta(\cdot \mid o)} \big[r(a, o) \big],$$

with gradient (omitting baselines/advantages for simplicity)

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \;=\; \mathbb{E}_{\boldsymbol{o}\sim P(\boldsymbol{o}),\, \boldsymbol{a}\sim\pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{o})}\Big[\boldsymbol{r}(\boldsymbol{a},\boldsymbol{o})\,\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}\mid \boldsymbol{o})\Big].$$

SFT is maximum likelihood on an (expert) dataset $\mathcal{D}$ of pairs $(\boldsymbol{o}, \boldsymbol{a})$:

$$J_{\mathrm{SFT}}(\boldsymbol{\theta}) \;=\; \mathbb{E}_{(\boldsymbol{o},\boldsymbol{a})\sim\mathcal{D}}\big[\log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}\mid \boldsymbol{o})\big], \qquad \nabla_{\boldsymbol{\theta}} J_{\mathrm{SFT}}(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{o},\boldsymbol{a})\sim\mathcal{D}}\big[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}\mid \boldsymbol{o})\big].$$

Note the differences. There are two. First, in RL the reward weights the samples. Second, in SFT the actions are produced by another external policy, but in RL they are generated by the policy being trained.

A very common "self-improvement" recipe is:

Self-improvement via RL

1. Sample K candidates $\boldsymbol{a}^{(1)}, \ldots, \boldsymbol{a}^{(K)} \sim \pi_{\mathrm{old}}(\cdot \mid \boldsymbol{o})$.
2. Score them with $\boldsymbol{r}(\boldsymbol{a}^{(k)}, \boldsymbol{o})$ (or a ranker/verifier).
3. Keep a subset (e.g. top-m) and finetune the model to increase their likelihood.

This is typically presented as self-training, ReST-style filtering, best-of-K distillation, etc. But it is RL: we are using reward to *change the policy*.

Concretely, many implementations reduce the learning step to a weighted maximum-likelihood update:

$$\max_{\boldsymbol{\theta}} \;\; \mathbb{E}_{\boldsymbol{o}\sim P(\boldsymbol{o})}\, \mathbb{E}_{\boldsymbol{a}\sim\pi_{\mathrm{old}}(\cdot|\boldsymbol{o})}\Big[w(\boldsymbol{a},\boldsymbol{o})\, \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}\mid \boldsymbol{o})\Big],$$

where $w(\boldsymbol{a}, \boldsymbol{o})$ is a weight derived from reward (e.g. $w = \boldsymbol{r}$, or $w = \mathbf{1}\{\boldsymbol{a}$ in top-$m\}$, or a softmax over the ranks). This is a policy-gradient-style update in which reward (or a monotone transform of it) acts as the advantage/weight on $\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{o})$. With hard top-$m$ filtering, the method is simply optimizing expected reward under a *binary* reward: "selected" receives reward 1, "rejected" receives reward 0.

So "generate → rank → finetune" is precisely the self-improvement strategy of RL. There are of course other self-improvement strategies. A better perception model can lead to a better generative model which in turn leads to a better perception model and so on.

9 Is reward enough?

The short answer is: not by itself. A more precise statement is: reward is a powerful selection signal, but not a complete account of intelligence.

The *Reward is Enough* thesis is plausible as a high-level organising hypothesis: many abilities associated with intelligence can arise because they help

an agent maximise long-term expectations. But reward alone does not specify enough structure. To obtain intelligent behaviour, one also needs an environment, an action space, a policy class, representations, exploration, credit assignment, computational resources, inductive biases, and learning dynamics.

Reward can select intelligent behaviour, but reward alone does not specify the agent, the representation, the environment, the action space, the learning rule, or the computational resources needed to discover that behaviour.

Potential games and Lyapunov functions clarify this distinction. They do not refute reward maximisation. A reward-maximising agent can patrol, cycle, explore, maintain homeostasis, or adapt continually. So the problem is not that reward maximisation requires gradient descent or convergence to a fixed point. It does not.

The problem is that a scalar reward does not, by itself, explain the full dynamical system. For example, in a multi-agent system, each agent may maximise its own reward, but the joint system need not maximise any single global reward. Potential games are precisely the special case where local incentives align with one global scalar potential. Following Monderer and Shapley, a differentiable game is a potential game when there exists a scalar function Φ such that each player's local improvement agrees with the corresponding coordinate gradient of Φ:

$$\nabla_{\boldsymbol{\theta}_i} J_i(\boldsymbol{\theta}_i, \boldsymbol{\theta}_{-i}) = \nabla_{\boldsymbol{\theta}_i} \Phi(\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_n).$$

In this special case, multi-agent learning can be interpreted as movement on one shared landscape.

But most games are not potential games. In adversarial, cyclic, or general-sum games, the coupled learning dynamics may contain rotational or Hamiltonian components. For example, in the bilinear zero-sum game

$$\min_{\boldsymbol{\theta}_1} \max_{\boldsymbol{\theta}_2} \boldsymbol{\theta}_1 \boldsymbol{\theta}_2,$$

the continuous-time gradient dynamics are

$$\dot{\boldsymbol{\theta}}_1 = -\boldsymbol{\theta}_2, \qquad \dot{\boldsymbol{\theta}}_2 = \boldsymbol{\theta}_1.$$

These dynamics rotate around the equilibrium instead of descending a shared scalar loss. Each agent has a clear objective, but the objectives do not combine into one global potential. This is why individual rewards can guide individual learners without creating a single objective for the whole multi-agent system.

Balduzzi et al.'s mechanics of differentiable games make this distinction explicit by decomposing differentiable games into potential-like and Hamiltonian-like components. The potential component resembles optimisation of a scalar function; the Hamiltonian component corresponds to rotational structure that cannot be reduced to descent on one global objective.

Lyapunov theory gives a related lesson. A Lyapunov function V can certify that some learning dynamics are stable:

$$\frac{\mathrm{d}}{\mathrm{d}t} V(\boldsymbol{\theta}(t)) = \nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta})^\top \boldsymbol{G}(\boldsymbol{\theta}) \leq 0.$$

But this does not imply that the dynamics are generated by

$$G(\theta) = -\nabla_\theta V(\theta).$$

A Lyapunov function can certify stability without being the objective that the system is trying to minimise. The dynamics may spiral, cycle, track a moving target, or remain on an invariant set. Thus scalar functions can sometimes summarise or certify behaviour, but they need not generate the behaviour.

For continual learning, the issue is even sharper. In stationary RL, we can define the negative expected return

$$\mathcal{L}_r(\theta) = -\mathbb{E}_{\tau\sim\pi(\cdot;\theta)}\left[\sum_{t\geq 0}\gamma^t r_t\right],$$

and policy-gradient methods can be understood as minimising this loss. In that simplified setting, reward maximisation gives a clean optimisation problem:

$$\theta^\star \in \arg\min_\theta \mathcal{L}_r(\theta).$$

But continual learning is not merely the search for one fixed parameter vector. The data distribution changes, the environment changes, other agents change, and the policy itself changes the future data. The real object is often a continuing sequence

$$\theta_0, \theta_1, \theta_2, \ldots,$$

where each parameter vector defines a new policy

$$\pi(a \mid o; \theta_0), \quad \pi(a \mid o; \theta_1), \quad \pi(a \mid o; \theta_2), \ldots$$

Reward can guide this process, but reward does not by itself determine the learning dynamics. The policy changes the data, the data changes the policy, and in games each learner's changing parameters change the learning problem faced by every other learner.

This is also why *Reward is Enough* is weaker than it first sounds. If the reward function, environment, representation, action interface, optimiser, and policy class are all well designed, then reward can be enough to select good behaviour. But much of the intelligence has already been placed into the setup: the architecture, the state representation, the simulator, the tools, the exploration scheme, the curriculum, and the prior knowledge.

This is analogous to Bayesian modelling. Bayesian inference is powerful once the likelihood, prior, and model class are specified. But Bayesian inference alone does not tell us what the right model class is. Similarly, RL is powerful once the reward, environment, policy class, and learning dynamics are specified. But reward alone does not solve the modelling problem.