Policy Optimization and RL Algorithms

Arushi Somani

November 22, 2025

1 Introduction

With the benefit of hindsight, the story of algorithms that shape modern LLM RL is surprisingly linear. Each new algorithm can be seen as a solution to the most painful problems the previous algorithm(s) had. Now, the real history is messier— discoveries in parallel, dead ends, happy accidents. But we're going to discuss the clean version. The clean version, even though it's somewhat fictional, teaches something true about why these algorithms work.

Picture a robot stuck in a maze. The robot has no map, no guidance— just the ability to move around and try things. We train it by letting it loose in the maze until something happens— maybe it runs out of battery, maybe it finds the exit— and then give it a reward based on how it did.

Now let's be precise. We have:

- An environment: states s, actions a, rewards r
- A robot operating according to a policy $\pi_\theta(a \mid s)$—a probability distribution over actions given its current state, controlled by parameters θ.
- Our goal is to maximize expected total reward $J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)]$, where τ is a complete trajectory from start to finish.

The fundamental question: how do we adjust θ to make our robot better?

2 REINFORCE

In a perfect world, we'd compute $\nabla_\theta J(\theta)$—the direction in parameter space that increases our expected reward. We have $J(\theta) = \mathbb{E}_{x\sim p_\theta}[f(x)]$, where x is a trajectory, p_θ is the distribution we control (through θ) and $f(x)$ is the total reward the environment gives us.

If we could differentiate through the entire process— from θ to actions to the reward $f(x)$— we'd just back-propagate and call it a day. This would be optimizing $J(\theta)$ directly. But the environment is a black box. We observe samples $(x, f(x))$ but can't differentiate f with respect to θ.

So we need to find an estimate that we can differentiate. We want $\nabla_\theta J(\theta)$. Start with the definition:

$$J(\theta) = \mathbb{E}_{x\sim p_\theta}[f(x)] = \int p_\theta(x) \cdot f(x)\, dx$$

Take the gradient:

$$\begin{aligned}\nabla_\theta \mathbb{E}_{x\sim p_\theta}[f(x)] &= \nabla_\theta \int p_\theta(x)\cdot f(x)\,dx \\ &= \int \nabla_\theta p_\theta(x)\cdot f(x)\,dx\end{aligned}$$

Here's the key move: We want to turn this into an expectation so we can estiate it by sampling. Expectations look like $\int p(x)(\text{something})\ dx$. But right now we have $\int \nabla p(x)(\text{something})\ dx$.

So what if we multiplied the numerator and denominator by $p_\theta(x)$?

$$\begin{aligned}\nabla_\theta \mathbb{E}_{x\sim p_\theta}[f(x)] &= \int \frac{p_\theta(x)}{p_\theta(x)}\nabla_\theta p_\theta(x)\cdot f(x)\,dx \\ &= \int p_\theta(x)\frac{\nabla_\theta p_\theta(x)}{p_\theta(x)}\cdot f(x)\,dx\end{aligned}$$

Notice that:

$$\nabla_\theta \log p_\theta(x) = \frac{1}{p_\theta(x)}\nabla_\theta p_\theta(x)$$

By the chain rule.

Substitute:

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{x\sim p_\theta}[f(x)] \\ &= \int p_\theta(x)\cdot \nabla_\theta \log p_\theta(x)\cdot f(x)\,dx \\ &= \mathbb{E}_{x\sim p_\theta}[\nabla_\theta \log p_\theta(x)\, f(x)]\end{aligned}$$

This is the **log-derivative trick**, also called the score function trick. It's beautiful: we can now estimate this gradient just by sampling trajectories. Collect some $x^{(1)},\ldots,x^{(N)}$ from p_θ, then:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} f(x^{(i)})\,\nabla_\theta \log p_\theta(x^{(i)}).$$

What does this mean? Instead of directly "pushing up the reward f" (which we can't differentiate), we push up the log-probability of trajectories in proportion to how good they are.

Now we apply this to our robot. A trajectory is $\tau = ((s_1,a_1),...(s_\tau,a_\tau))$, the reward is $R(\tau)$. The probability of a trajectory factors as $\pi_\theta(\tau) = \prod_{t=1}^{\tau}\pi_\theta(a_t|s_t)$.

The gradient becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)\,\nabla_\theta \log \pi_\theta(\tau)]$$

For better credit assignment— so early actions aren't blamed for late rewards they had less effect on, we use *reward-to-go*:

$$G_t = \sum_{t'=t}^{T}\gamma^{\,t'-t} r_{t'},$$

giving us the standard per-time update:

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

In practice, we sample N episodes and estimate:

$$\hat{g} = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} G_t^{(i)} \,\nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)}).$$

REINFORCE works, it is not without flaw. Suppose our robot finds the exit and gets $R(\tau) = 100$. REINFORCE says "great! make every action in that trajectory more likely—they all contributed equally to this success!"

Even with discounting, where $G_t = \sum_{t'=t}^{T} \gamma^{\,t'-t} r_{t'}$, so earlier mistakes are weighted less— we still can't attribute good actions (earlier or later in the trajectory) to the cause of the success.

This blunt assignment creates very high variance. In practice, you might need millions of samples to train even simple tasks, because a lot of the gradient signal is noise.

3 Advantage

How can we fix this problem with REINFORCE? The key insight is this: we don't actually care about the absolute reward. We care about *surprise*. If an action does better than we expected in that state—reward it. If it does worse than expected—punish it. If it does exactly what we expected—don't update at all. This notion of "better or worse than expected" is called the **advantage** of the action.

To calculate advantage, we need a *baseline* for each state—some measure of what we typically expect to happen from there. Then we compare: did this particular trajectory do better or worse than that baseline? A simple version looks like

$$A_t \approx G_t - b(s_t).$$

However, when we modify our gradient estimator, we have to be careful not to make it *biased*. If the expected value of our estimator no longer equals the true gradient, we're optimizing the wrong thing entirely—we're no longer climbing toward $J(\theta)$ at all.

What baselines can we subtract without introducing bias? Here's the wonderful thing. Any baseline that depends only on the *state*—not on the action—leaves the estimator completely unbiased. Why? Look at what happens:

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

The baseline pulls out of the expectation over actions, and the remaining term is zero. This makes intuitive sense too: a state-only baseline doesn't favor any particular action, so it can't systematically push the policy in any direction.

Therefore we can subtract $b(s_t)$ inside our gradient estimator *without changing its expectation*:

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

This brings us to two functions we'll use constantly:

- The **value function**: the expected return if we start in state s and follow policy π.

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s],$$

- The **action-value function**: the expected return if we are in state s, take action a once, and then follow policy π thereafter.

$$Q^\pi(s,a) = \mathbb{E}_\pi[G_t \mid s_t = s, a_t = a],$$

The **advantage** is:

$$A(s,a) = Q^\pi(s,a) - V^\pi(s),$$

and we'll use this often below.

4 Actor-Critic

So we want to use the advantage $A(s,a) = Q(s,a) - V(s)$ to improve REINFORCE. But immediately we run into a practical problem: we don't actually *know* V^π or Q^π. We only see samples from our rollouts.

One simple idea: for each state, keep track of the average return we've seen from it. That works—but only if we see the same state many times. In chess, or in any complex environment, that exact board position might never appear twice. We need some way to *generalize* our value estimates across similar states.

The problem: we do not know V^π or Q^π. We only see samples, so we must find a way to practically estimate them. The key observation is to note the relationship between V and Q. Q looks one step in the future using V. After we take an action a_t, we get a reward r_t, reach state s_{t+1} and then the expected value from that state is $V(s_{t+1})$.

This is the Bellman one-step lookahead for Q:

$$Q(s_t, a_t) = \mathbb{E}\big[r_t + \gamma V(s_{t+1}) \mid s_t, a_t\big].$$

So Q, the value of taking action a_t in state s_t is the immediate reward and a discounted expected value of the next state.

Now let's plug this into the advantage equation:

$$\begin{aligned} A(s_t, a_t) &= Q(s_t, a_t) - V(s_t) \\ &= \mathbb{E}\big[r_t + \gamma V(s_{t+1}) - V^\pi(s_t) \mid s_t, a_t\big]. \end{aligned}$$

This value inside of the expectation:

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

is called the **temporal difference (TD) error**. It's an unbiased estimator of the advantage, but it only needs one step of experience.

Of course, we don't actually know the true $V^\pi(s)$. So here's what we do: we learn *two* neural networks that work together. The first network is the **actor** $\pi_\theta(a \mid s)$—the policy that chooses actions. The second is the **critic** $V_\phi(s)$—a function that learns to predict the value of each state. The critic's job is to tell the actor: "you're in state s, and based on everything I've seen, I expect you to get this much total return from here."

We train them in tandem:

- **Training the critic:** We want the critic's predictions to be accurate. So we collect experience (s_t, a_t, r_t, s_{t+1}) and minimize the squared TD error:

  $$\mathcal{L}(\phi) = \mathbb{E}\left[\big(V_\phi(s_t) - (r_t + \gamma V_\phi(s_{t+1}))\big)^2\right].$$

  We're saying "based on the reward we just got and our current estimate of the next state's value, this is what $V_\phi(s_t)$ should have predicted." We're bootstrapping: using our own estimates to improve our own estimates.

- **Training the actor:** We update the policy using the TD error as our advantage estimate:

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

  where $\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$.

Here's the full picture. At each time step:

1. The actor takes action a_t in state s_t.
2. We observe reward r_t and next state s_{t+1}.
3. The critic evaluates: "I thought state s_t was worth $V_\phi(s_t)$. Now I see we got reward r_t and landed in a state worth $V_\phi(s_{t+1})$. The TD error δ_t tells me whether this was better or worse than expected."
   (a) If $\delta_t > 0$, the outcome was better than expected—increase the probability of action a_t.
   (b) If $\delta_t < 0$, the outcome was worse than expected—decrease the probability of action a_t.
4. Update the critic so its prediction $V_\phi(s_t)$ moves closer to the observed target $r_t + \gamma V_\phi(s_{t+1})$.

This makes credit assignment *local in time*. Each action gets immediate feedback: did it lead somewhere better or worse than expected? We don't wait for the full episode to finish. The critic V_ϕ summarizes "everything that happens after this point," and the TD error δ_t tells us whether this particular action improved or hurt our prospects.

Compare this to REINFORCE, which waits until the episode ends to compute the full return $G_t = \sum_{t=1}^{\tau} \gamma^{t-1} r_t$. If the episode is long, early actions wait hundreds of time steps for a learning signal. And that signal has high variance because it sums many random rewards. Actor-critic bootstraps instead: we learn from each step immediately, using V_ϕ to estimate the future rather than waiting to observe it.

5 TRPO

We've made improvements to variance and credit assignment, but there is still a way that learning might become unstable.

Picture this: our maze robot discovers that going `LEFT` at some intersection sometimes leads to the exit. The critic estimates a positive advantage, so the actor update increases the probability of `LEFT`. Next batch: mostly `LEFT` trajectories, advantage still looks good, so we crank `LEFT` even higher. A few more iterations and we've collapsed to

$$\pi_\theta(a \mid s) \approx \mathbf{1}[a = \texttt{LEFT}],$$

a nearly deterministic policy that always goes left.

If those initial trajectories were lucky, buggy, or unrepresentative, we've locked ourselves onto the wrong answer. And now we're stuck: the policy only explores left, so we never see evidence that other directions might work better.

Here's the deeper problem. We compute our gradient estimate using data from the current policy π_{old}. That gradient is only guaranteed to point in the right direction *locally*—for small changes to the policy. If we take a huge step in that direction, we might end up with a new policy π_θ that behaves completely differently: it visits different states, puts all its probability mass on different actions. The data we collected under π_{old} no longer reflects what π_θ would actually experience.

The gradient at π_{old} wasn't wrong. The problem is we trusted a local signal to make a non-local change. We need a way to *constrain* how much the policy's behavior can shift in a single update.

Here's the setup. We've collected data under a fixed old policy π_{old}: states s_t, actions a_t, and advantage estimates A_t. Now we want to find a better policy π_θ. Ideally, we'd like to evaluate how well π_θ would do on the states we've seen:

$$\mathbb{E}_{a \sim \pi_\theta(\cdot \mid s_t)} \big[A(s_t, a) \big],$$

where the states s_t come from π_{old} but we're asking what π_θ would do in those states.

Expanding this expectation:

$$\mathbb{E}_{a \sim \pi_\theta(\cdot \mid s_t)}[A(s_t, a)] = \sum_a \pi_\theta(a \mid s_t)\, A(s_t, a).$$

Now here's a trick. We don't have samples from π_θ—we only have samples from π_{old}. But we can

rewrite this sum by multiplying and dividing by $\pi_{\text{old}}(a \mid s_t)$:

$$\sum_a \pi_\theta(a \mid s_t)\, A(s_t, a) = \sum_a \pi_{\text{old}}(a \mid s_t)\, \underbrace{\frac{\pi_\theta(a \mid s_t)}{\pi_{\text{old}}(a \mid s_t)}}_{r_t(\theta)}\, A(s_t, a)$$
$$= \mathbb{E}_{a \sim \pi_{\text{old}}(\cdot \mid s_t)}[r_t(\theta)\, A(s_t, a)]\,.$$

This ratio $r_t(\theta) = \pi_\theta(a_t \mid s_t)/\pi_{\text{old}}(a_t \mid s_t)$ is called the **importance ratio**. It reweights old samples to estimate what would happen under the new policy. If π_θ puts more probability on action a_t than π_{old} did, the ratio is greater than 1 and we weight that sample more heavily. If π_θ puts less probability on it, the ratio is less than 1 and we weight it less.

This gives us the surrogate objective:

$$L(\theta) = \mathbb{E}_{t \sim \pi_{\text{old}}}[r_t(\theta)\, A_t]\,, \quad \text{where} \quad r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\text{old}}(a_t \mid s_t)}.$$

If $L(\theta)$ increases, it means π_θ is putting more weight on high-advantage actions and less on low-advantage actions—an improvement.

So why not just maximize $L(\theta)$? Because importance sampling can go badly wrong when π_θ and π_{old} differ too much. Imagine some state-action pair (s, a) where $\pi_{\text{old}}(a \mid s)$ is tiny—say, 0.01—but $\pi_\theta(a \mid s)$ is large—say, 0.5. The importance ratio is 50. If we happened to see a positive advantage for that action (maybe just by luck), it dominates the entire objective. We're amplifying noise.

Worse: we can only evaluate $L(\theta)$ on states that π_{old} visited. If π_θ would visit completely different states—states where the advantage is actually terrible—we have no way to see that in our data. The surrogate $L(\theta)$ might look great while the true performance $J(\theta)$ is awful. The core issue: $L(\theta)$ is a first-order approximation of $J(\theta)$ around π_{old}. It's valid locally, but if π_θ strays too far, the approximation breaks down.

TRPO's solution is to explicitly constrain how far we're allowed to move. We solve for:

$$\max_\theta\ L(\theta) \quad \text{subject to} \quad \mathbb{E}_{s \sim \pi_{\text{old}}}\left[D_{\text{KL}}\big(\pi_{\text{old}}(\cdot \mid s) \,\|\, \pi_\theta(\cdot \mid s)\big)\right] \le \delta,$$

for some small δ (typically around 0.01).

This constraint measures how different the new policy is from the old one. For each state s that π_{old} visits, we compute the KL divergence between the old action distribution and the new one. KL divergence is always non-negative and equals zero only when the distributions are identical—the bigger the KL, the more different the policies.

By keeping the average KL small, we create a "trust region" around π_{old}. Inside this region, we trust that $L(\theta)$ is a good proxy for true performance. We're allowing the policy to improve, but forcing it to do so gradually—small, safe steps rather than wild leaps.

Under reasonable assumptions, you can prove that each TRPO update either improves performance or stays roughly the same—it won't catastrophically collapse. The trust region keeps us honest: we only make changes we can justify with our current data.

6 PPO

TRPO works, but it's complicated. That constraint $D_{\mathrm{KL}} \leq \delta$ requires solving a constrained optimization problem, which means computing second derivatives, inverting matrices—expensive operations we'd rather avoid. **Proximal Policy Optimization (PPO)** keeps the core idea—"don't change too much"—but implements it much more simply.

Instead of adding a constraint, PPO modifies the objective itself. The key insight: if the importance ratio $r_t(\theta) = \pi_\theta(a_t \mid s_t)/\pi_{\text{old}}(a_t \mid s_t)$ gets too far from 1, something's wrong. So let's just not allow that to contribute.

Here's PPO's clipped objective:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \Big(r_t(\theta) A_t,\ \text{clip}\big(r_t(\theta), 1-\epsilon, 1+\epsilon\big) A_t \Big) \right],$$

where $\text{clip}(r, 1-\epsilon, 1+\epsilon)$ clamps r to the range $[1-\epsilon, 1+\epsilon]$ (typically $\epsilon = 0.2$).

The min enforces a pessimistic bound: we only get credit when both the clipped and unclipped objectives agree that we're improving. If either one thinks we're going too far, we stop benefiting. The objective naturally saturates when importance ratios drift too far from 1.

7 GRPO

GRPO keeps PPO's "small step" spirit but recognizes that the value networks are a lot of fuss—we have to train $V_\phi(s)$ and worry about its learning dynamics. Is there a simpler way?

Here's an observation: in LLM training, we already sample *multiple* completions for each prompt. We don't just generate one response and update on it—we generate a whole batch, maybe 8 or 16 completions per prompt, and score them all. That's a lot of information about what's good and bad for this particular prompt. What if we use the group itself as the baseline?

Group-Relative Policy Optimization (GRPO) does exactly this. For each prompt q, sample a group of K completions $\{o_i\}_{i=1}^K$ from the current policy. Score each one with a reward model or programmatic verifier to get rewards $\{r_i\}_{i=1}^K$. The insight: we don't care about the absolute reward values—we care about which completions did better or worse *relative to each other*.

Normalize within the group:

$$\tilde{r}_i = \frac{r_i - \bar{r}}{\text{std}(r)}, \qquad \bar{r} = \frac{1}{K} \sum_{j=1}^{K} r_j.$$

This gives us a z-score for each completion. If $\tilde{r}_i > 0$, this completion did better than the group average—reinforce it. If $\tilde{r}_i < 0$, it did worse—suppress it. The normalization by standard deviation accounts for how spread out the rewards are: if all completions got similar rewards, the group doesn't tell us much, so advantages stay small.

Now we need per-token advantages. The completion-level score $\tilde{r}_i$ applies to the entire output o_i, but we update token by token. The simplest approach: spread the score evenly across all L_i tokens:

$$\hat{A}_{i,t} \ = \ \frac{1}{L_i} \tilde{r}_i \quad \text{for } t = 1, \ldots, L_i.$$

This length-normalization ensures that longer completions don't accumulate more gradient just because they have more tokens.

With advantages in hand, we optimize a PPO-style objective:

$$\mathcal{L}_{\text{GRPO}}(\theta) = \mathbb{E}_{q,\,i,\,t}\Big[\min\Big(i_{i,t}(\theta)\,\hat{A}_{i,t},\ \text{clip}\big(i_{i,t}(\theta), 1-\epsilon, 1+\epsilon\big)\,\hat{A}_{i,t}\Big)\Big]\ -\ \beta\,\text{KL}\big(\pi_\theta \,\|\, \pi_{\text{ref}}\big)\,,$$

where

$$i_{i,t}(\theta) = \frac{\pi_\theta(o_{i,t} \mid q, o_{i,<t})}{\pi_{\text{old}}(o_{i,t} \mid q, o_{i,<t})}.$$

The first term is standard PPO clipping—same as before. The second term is a KL penalty that keeps the policy π_θ close to a fixed reference policy π_{ref} (usually the initial supervised fine-tuned model). This prevents the policy from drifting too far from sensible behavior as it optimizes for reward.

By using group-relative baselines, GRPO achieves variance reduction without training a critic, cutting memory/compute, and works especially well when rewards are *verifiable* in domains like math and code correctness. GRPO was invented in February 2024, a reminder that RL for LLMs is still young and there are optimizations yet to be discovered.

8 Entropy Regularization

We use this opportunity to also talk about entropy regularization, a long-standing trick to keep the policy from collapsing too quickly and to encourage exploration. We simply add the policy's entropy to the objective:

$$\mathcal{L}_{\text{ent}}(\theta) = \mathbb{E}_t[L(\theta)]\ +\ \beta\,\mathbb{E}_t\big[\mathcal{H}\big(\pi_\theta(\cdot \mid s_t)\big)\big]\,,$$

with weight $\beta > 0$ (often decayed over training).

For a categorical policy,

$$\mathcal{H}(\pi) = -\sum_a \pi(a \mid s)\log\pi(a \mid s).$$

By adding entropy to the objective, we reward the policy for staying a bit "uncertain". Why does this help? Without entropy regularization, the policy can collapse prematurely onto a single action that looked good early in training, cutting off exploration.

In practice, β often starts moderate and decays over time. Early in training, we want broad exploration. Later, once we've found promising regions, we can afford to become more deterministic and exploit what we've learned. The decay schedule balances exploration early with exploitation late.

This provides a chance for me to talk more about algorithm design, and add a bit of editorializing: I believe that we'd ideally want entropy regularization to emerge naturally from the algorithm itself (like soft actor-critics) rather than being bolted on as an auxiliary term.

9 Further Readings

Curious readers can further explore by learning about: Soft Actor-Critics, A3C, DPO, Constitutional AI and Multi-Agent RL.