Lecture 5: Actor-Critic Methods

1 Introduction

Admin:

- HW1 due Monday (we're going to add another day, might extend by a couple of days if there is more trouble submitting, raise hands if you need more time).
- Next HW will be posted at the end of next week, you'll again have two weeks.
- Start thinking about the project!

1.1 Where we are

Throughout this course we've been going back and forth between two key objects: the **policy** and the **value function**. Last week we focused on the policy side of policy gradient methods. But often, they combine together with value functions to form the *actor-critic* framework (Konda and Tsitsiklis, 1999). The actor is the policy, and the critic is the value function (critiquing, or evaluating, the policy's actions).

Today we're going to dig deeper into the *critic* side of actor-critic, and in particular the question of **advantage estimation** which is commonly thought to reduce the variance of policy gradient updates. Our roadmap:

1. **Brief recap: stabilizing policy updates** — the problem with vanilla policy gradients, importance weighting, TRPO, and PPO's clipped surrogate.
2. **The value function baseline** — why subtracting $V(s)$ from the policy gradient helps, a proof that it doesn't change the expected gradient, and the variance-minimizing baseline.
3. **Advantage estimation and GAE** — how to estimate advantages using k-step returns and generalized advantage estimation.
4. **PPO: putting it all together** — the full algorithm with all the pieces in place, what we know and don't know about baselines, and GRPO as a value-function-free alternative.
5. **From Q-learning to continuous actions: DDPG and TD3** — a different flavor of actor-critic where the critic is a Q-function and the actor approximates the argmax. Group exercise: reading the TD3 code.

2 Recap: Stabilizing Policy Gradient Updates

2.1 The problem with vanilla policy gradients

Define the RL objective in terms of the policy parameters θ as

$$\mathcal{J}(\theta) \triangleq \mathbb{E}_{\pi_\theta(\tau)}\left[\sum_t \gamma^t r_t\right]. \tag{1}$$

We would like to do gradient ascent on this objective: $\theta \leftarrow \theta + \eta \nabla_\theta \mathcal{J}(\theta)$. Using the likelihood ratio trick, the policy gradient is

$$\nabla_\theta \mathcal{J} = \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'}\right) \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right]. \tag{2}$$

The standard REINFORCE algorithm uses this gradient directly. We sample a trajectory $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$ under π_θ, compute the return $R_t = \sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'}$ for each time step, and perform the update:

$$\theta \leftarrow \theta + \eta \sum_t R_t \nabla_\theta \log \pi_\theta(a_t \mid s_t). \tag{3}$$

This is an unbiased estimate of $\nabla_\theta \mathcal{J}$, but it has very high variance. The return R_t sums over stochastic transitions, so the gradient signal is noisy, especially if our sample size is not large enough.

There are also problems if your sampling policy is different from the policy you are optimizing. Even if you are trying to behave on-policy, this can happen in a couple of instances. First, if you are using a GPU for training and a different GPU for evaluation, you might have a mismatch in log probabilities due to floating point imprecision.

Pitfall

Numerical precision matters more than you might think. Recent work by Qi et al. (2025) shows that BF16 precision introduces rounding errors that break consistency between training and inference in RL fine-tuning. This **training-inference mismatch** means the log probabilities $\log \pi_\theta(a|s)$ computed during rollout can differ from those computed during the training pass, even for the *same* model weights. These small numerical differences compound through importance ratios and can cause significant performance degradation. Thinking Machines Lab (Thinking Machines Lab, 2025) traced this further to batch-sensitive nondeterminism in GPU operations — the same model can produce different log probabilities depending on the batch composition. See also Zhang et al. (2026) for an argument that the mismatch is fundamentally an optimization problem, not just a precision problem.

Second, as soon as you take one gradient step, your old batch of rollouts is no longer on-policy. If you want to re-use that information, you need to either re-sample or modify the objective. To do this, we use data sampled from an old policy $\pi_{\theta_{\text{old}}}$ to evaluate the quality of a new policy π_θ. Define the importance weight $r(s, a; \theta) \triangleq \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$. The surrogate objective is:

$$\mathcal{L}(\theta) = \mathbb{E}_{\substack{s_t \sim \rho^{\pi_{\theta_{\text{old}}}} \\ a_t \sim \pi_{\theta_{\text{old}}}}}\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} R_t\right]. \tag{4}$$

This surrogate almost (but not quite) gives us the correct policy gradient — the missing part is that the expectation is over the state distribution of $\pi_{\theta_{\text{old}}}$, not π_θ. If the two policies visit similar states, the approximation is good.

2.2 Constraining policy updates

The question becomes: how much can the new policy differ from the old policy before the surrogate objective becomes unreliable? Schulman et al. (2015) answer this with a formal guarantee, based in constrained policy iteration (notice how a lot of methods tie back to dynamic programming foundations?):

Theorem 1 (Monotonic Improvement Bound)**.**

$$\mathbb{E}_{\pi_\theta}\left[\sum_t \gamma^t r_t\right] \geq \mathbb{E}_{\pi_{\theta_{old}}}\left[\sum_t \gamma^t r_t\right] + \mathcal{L}_{\pi_{\theta_{old}}}(\pi_\theta) - C \max_s D_{\text{KL}}\big(\pi_{\theta_{old}}(\cdot \mid s) \| \pi_\theta(\cdot \mid s)\big)\,, \tag{5}$$

where $C = \frac{4\epsilon\gamma}{(1-\gamma)^2}\alpha^2$ *and* $\epsilon = \max_{s,a} |A^\pi(s,a)|$.

In words: the new policy's return is at least the old policy's return, plus the surrogate objective, minus a penalty that grows with the KL divergence between the two policies. We want to maximize the surrogate while minimizing the divergence. An operationalization of this is to solve the following constrained optimization problem:

$$\max_\theta \quad \mathbb{E}_{\pi_{\theta_{\text{old}}}(s_t)\pi_{\theta_{\text{old}}}(a_t|s_t)}\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} R_t\right] \tag{6}$$

$$\text{s.t.} \quad \mathbb{E}_s\big[D_{\text{KL}}\big(\pi_{\theta_{\text{old}}}(\cdot \mid s) \| \pi_\theta(\cdot \mid s)\big)\big] \leq \delta. \tag{7}$$

In practice, TRPO handles the KL constraint using second-order optimization: it computes the natural gradient via the conjugate gradient method with the Fisher information matrix, avoiding an explicit Hessian inversion. This is significantly more expensive per iteration than standard first-order methods.

2.3 PPO: clipping instead of constraining

PPO (Schulman et al., 2017) replaces the KL constraint with a simpler mechanism: **clipping** the importance weight. For an action the old policy took frequently but the new policy wants to take even more, the importance weight $r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$ becomes very large — this is where instability occurs. PPO simply clips this ratio to the interval $[1-\epsilon, 1+\epsilon]$.

We can then modify the objective above as:

$$\boxed{\mathbb{E}_{\rho^{\pi_{\theta_{\text{old}}}}(s_t)\pi_{\theta_{\text{old}}}(a_t|s_t)}\Big[\min\Big(r(s_t,a_t;\theta)\,R_t,\ \text{clip}(r(s_t,a_t;\theta), 1-\epsilon, 1+\epsilon)\,R_t\Big)\Big]\,.} \tag{8}$$

The min ensures that when the return is positive, we increase the action's probability by at most $(1+\epsilon)\times$; when the return is negative, we decrease it, but the clipping prevents us from decreasing

too aggressively. The net effect is that policy updates are kept small and stable, without needing to compute a KL divergence and solve the optimization problem in the outer loop.

But notice a problem: using the raw return R_t in the objective means that *all* actions with positive return get upweighted, even mediocre ones. In expectation of all samples, we will end up upweighting good actions more than bad actions, but this is wasteful.

3 Baselines and the Advantage Function

There are really two separate motivations for subtracting something from the return in the policy gradient. The first comes from the Monte Carlo simulation literature: the idea of **control variates**. If you subtract a quantity with zero expectation from your estimator, you don't change the mean but you can reduce the variance. The second comes from the policy improvement perspective. In conservative policy iteration (Kakade and Langford, 2002), the improvement step is defined in terms of the advantage $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$. The advantage tells you how much better action a is compared to the average action under the current policy. This is a helpful mechanism by which we can prove convergence properties. Recall that if we can guarantee that our advantage is always positive, we can always improve the policy.

Since we covered the (Kakade and Langford, 2002) view last time. Let's focus on the control variate view today. In this view, we can subtract any function of the state without changing the expected gradient, so long as it is independent of the action.

For any state-dependent baseline $b(s_t)$, the policy gradient remains unbiased:

$$\nabla_\theta \mathcal{J} = \mathbb{E}\left[\sum_t (Q^\pi(s_t,a_t) - b(s_t)) \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right].$$

We need to show that the baseline term drops out. Rewriting the expectation in terms of the discounted state-occupancy measure P^π_γ:

$$\begin{aligned}
&\mathbb{E}_{s_t\sim P^\pi_\gamma(\cdot),\, a_t\sim\pi_\theta(\cdot|s_t)}[b(s_t)\,\nabla_\theta \log \pi_\theta(a_t \mid s_t)] \\
&\quad = \mathbb{E}_{s_t\sim P^\pi_\gamma(\cdot)}\big[b(s_t)\,\mathbb{E}_{a_t\sim\pi_\theta(\cdot|s_t)}[\nabla_\theta \log \pi_\theta(a_t \mid s_t)]\big] &(9)\\
&\quad = \mathbb{E}_{s_t\sim P^\pi_\gamma(\cdot)}\left[b(s_t)\,\nabla_\theta \sum_a \pi_\theta(a \mid s_t)\right] &(10)\\
&\quad = \mathbb{E}_{s_t\sim P^\pi_\gamma(\cdot)}[b(s_t)\,\nabla_\theta\, 1] = 0. &(11)
\end{aligned}$$

The key step is Eq. 10. We use the log-derivative trick in reverse:

$$\mathbb{E}_{a\sim\pi}[\nabla_\theta \log \pi(a|s)] = \sum_a \pi(a|s)\frac{\nabla_\theta \pi(a|s)}{\pi(a|s)} = \nabla_\theta \sum_a \pi(a|s) = \nabla_\theta\, 1 = 0.$$

Since $b(s_t)$ does not depend on the action, it factors out of the inner expectation and the baseline drops out regardless of the choice of b.

Pitfall

This proof relies on b depending only on the state, not the action. If $b = b(s, a)$, then it cannot be factored out of the inner expectation, and the gradient is no longer unbiased in general. In circa 2018, several folks thought about using the action-dependent baseline anyway. But Tucker et al. (2018) systematically investigated these claims and found issues. First, an action-dependent baseline is *not* guaranteed to leave the gradient unbiased. Recall our proof in Eq. 11: the key step was that $b(s)$ factors out of the expectation over actions. If b depends on a, this step fails:

$$\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[b(s,a)\,\nabla_\theta \log \pi_\theta(a \mid s)] \neq 0 \quad \text{in general.} \tag{12}$$

To correct for this, you need a bias correction term, which itself introduces additional variance. Second, the empirical findings of Tucker et al. (2018) showed that learned state-action baselines did **not** reduce variance over a simple state-dependent baseline in standard benchmarks (MuJoCo). The apparent improvements in prior work were traced to implementation details (network architecture, optimization) rather than the baseline itself. When implementation details were controlled, the state-dependent baseline was as good or better.

3.1 The variance-minimizing baseline

How do we know baselines reduce variance, beyond intuition? We can do an explicit calculation. Consider the single-sample gradient estimator at a fixed state s:

$$g = (\nabla_\theta \log \pi_\theta(a \mid s))\, Q(s, a). \tag{13}$$

With baseline $b(s)$, this becomes $g_b = (\nabla_\theta \log \pi_\theta(a \mid s))\,(Q(s, a) - b(s))$. For notational clarity, define

$$d(a) \triangleq \nabla_\theta \log \pi_\theta(a \mid s) \qquad \text{and} \qquad Q \triangleq Q(s, a), \tag{14}$$

and use $\mathbb{E}_a$ as shorthand for $\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}$. The baseline-subtracted estimator is $Z = d(a)(Q - b)$.

We want to choose b to minimize the variance of Z:

$$f_{\text{var}}(b) = \mathbb{E}_a\left[\|d(a)\|^2 (Q - b)^2\right] - \Big(\mathbb{E}_a[d(a)(Q - b)]\Big)^2. \tag{15}$$

The second term loses dependence on b by the score-function trick (Eq. 11). So we only need to minimize the first term. Taking the derivative:

$$\frac{d}{db}\mathbb{E}\left[\|d(a)\|^2 (Q - b)^2\right] = -2\,\mathbb{E}\left[\|d(a)\|^2 (Q - b)\right]. \tag{16}$$

Setting this to zero and solving:

$$\mathbb{E}\left[\|d(a)\|^2 Q\right] = b\,\mathbb{E}\left[\|d(a)\|^2\right]. \tag{17}$$

Definition 1 (Optimal Baseline)**.** The variance-minimizing baseline is

$$b^*(s) = \frac{\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[\|\nabla_\theta \log \pi_\theta(a \mid s)\|^2\, Q(s, a)\right]}{\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[\|\nabla_\theta \log \pi_\theta(a \mid s)\|^2\right]}. \tag{18}$$

This is a score-function-weighted average of Q (Greensmith et al., 2004, Theorem 8).

There's an approximate argument for why $V^\pi(s)$ is a good baseline in practice (h/t Daniel Seita), though it's a little more hand-wavy and makes some assumptions. Consider the variance of the full gradient estimator. We can approximate:

$$\begin{aligned}\mathrm{Var}\left(\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)(R_t - b(s_t))\right) &\overset{(i)}{\approx} \sum_t \mathbb{E}_\tau\left[(\nabla_\theta \log \pi_\theta(a_t|s_t)(R_t - b(s_t)))^2\right] \\ &\overset{(ii)}{\approx} \sum_t \mathbb{E}_\tau\left[(\nabla_\theta \log \pi_\theta(a_t|s_t))^2\right] \mathbb{E}_\tau\left[(R_t - b(s_t))^2\right]\end{aligned}$$

We can replace the variance of a sum with the sum of variances (valid when samples are approximately uncorrelated, e.g. from parallel actors) (i). If we also assume independence between the score function and the return (ii), we can factor the expectation. Under these approximations, optimizing over $b(s_t)$ is just a least-squares problem: the optimal $b(s_t)$ is $\mathbb{E}[R_t(\tau)] = V^\pi(s_t)$. Greensmith et al. (2004) provide a thorough analysis of these variance reduction techniques, formalizing both baselines and actor-critic methods as instances of **additive control variates** from the Monte Carlo simulation literature.

With the advantage function in hand, the PPO objective (Eq. 8) becomes:

$$\boxed{\mathbb{E}_{\rho^{\pi_{\theta_{\text{old}}}}(s_t)\pi_{\theta_{\text{old}}}(a_t|s_t)}\left[\min\Big(r(s_t,a_t;\theta)\,A(s_t,a_t),\ \text{clip}(r(s_t,a_t;\theta),1-\epsilon,1+\epsilon)\,A(s_t,a_t)\Big)\right].} \tag{19}$$

This is what PPO actually optimizes in practice.

4 Advantage Estimation

Now that we understand *why* baselines help, the question becomes: how do we estimate the advantage $A^\pi(s_t,a_t) = Q^\pi(s_t,a_t) - V^\pi(s_t)$ in practice? We don't have access to the true Q^π or V^π, so we need to estimate them from data.

4.1 Learning the value function

We learn a value function V_ϕ (parameterized by ϕ, typically a neural network) by regression to the empirical returns:

$$\min_\phi \frac{1}{2}\Big(V_\phi(s_t) - \sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'}\Big)^2. \tag{20}$$

While the value function and policy are often implemented as sharing many of the layers (since they both take states as input), this seems to degrade performance in practice (Huang et al., 2022).

Importantly, though, these updates will have error too. If there is error, our policy gradient will end up either being biased and potentially not even reducing variance (under some conditions). What tools do we have to deal with errors and bias-variance tradeoff in *value function learning*?

4.2 k-step advantage estimators

We can estimate the Q function using the empirically observed returns up to time step $t+k$, whereupon we substitute the learned value function. By varying k, we get different estimates:

$$\begin{aligned} Q(s_t, a_t) &\approx r_t + \gamma V(s_{t+1}) && \text{(1-step)} && (21)\\ &\approx r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) && \text{(2-step)} && (22)\\ &\approx r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V(s_{t+3}) && \text{(3-step)} && (23)\\ &\vdots \\ &= \textstyle\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} && \text{(Monte Carlo)} && (24) \end{aligned}$$

Each of these can be turned into an advantage estimator by subtracting the baseline $V(s_t)$:

$$\begin{aligned} \hat{A}_t^{(1)} &= r_t + \gamma V(s_{t+1}) - V(s_t) && \text{(1-step: low var, high bias)} && (25)\\ \hat{A}_t^{(2)} &= r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t) && \text{(2-step)} && (26)\\ \hat{A}_t^{(3)} &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V(s_{t+3}) - V(s_t) && \text{(3-step)} && (27)\\ &\vdots \\ \hat{A}_t^{(\infty)} &= \textstyle\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} - V(s_t) && \text{(MC: high var, low bias)} && (28) \end{aligned}$$

Which should we use? There is a tradeoff. If we immediately plug in the value function at $t+1$ (the 1-step estimator), we rely heavily on V being accurate. If V is wrong, the advantage estimate is wrong introducing **bias**. If we wait longer and use more actual rewards before substituting V (the Monte Carlo estimator), the estimate becomes unbiased, but we're summing over many stochastic transitions introducing **variance**. Recall Kearns and Singh (2000) bounds on the bias and variance of the multi-step update.

4.3 Generalized advantage estimation (GAE)

The idea behind **generalized advantage estimation** (Schulman et al., 2016) is to use *all* the k-step estimators simultaneously, taking a weighted combination with geometrically-decreasing weights:

$$\hat{A}_t^{\text{GAE}} = (1-\lambda)\,\hat{A}_t^{(1)} + (1-\lambda)\lambda\,\hat{A}_t^{(2)} + (1-\lambda)\lambda^2\,\hat{A}_t^{(3)} + \cdots \tag{29}$$

The weights $(1-\lambda), (1-\lambda)\lambda, (1-\lambda)\lambda^2, \ldots$ sum to 1, so this is a proper weighted average. The parameter $\lambda \in [0,1]$ controls how much weight we put on shorter versus longer backups.

Expanding and simplifying this expression (a nice exercise in manipulating geometric series):

$$\begin{aligned} \hat{A}_t^{\text{GAE}} &= -V_t + r_t + (1-\lambda)\gamma V_{t+1} + \lambda\gamma r_{t+1} + \lambda(1-\lambda)\gamma^2 V_{t+2} + \lambda^2\gamma^2 r_{t+2} + \cdots && (30)\\ &= \underbrace{(r_t + \gamma V_{t+1} - V_t)}_{\delta_t^V} + \lambda\gamma \underbrace{(r_{t+1} + \gamma V_{t+2} - V_{t+1})}_{\delta_{t+1}^V} + \lambda^2\gamma^2 \underbrace{(r_{t+2} + \gamma V_{t+3} - V_{t+2})}_{\delta_{t+2}^V} + \cdots && (31) \end{aligned}$$

The expression naturally factors into a sum of **TD residuals** $\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)$:

$$\boxed{\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \sum_{t'=t}^{\infty} (\lambda\gamma)^{t'-t} \delta_{t'}^V = \sum_{t'=t}^{\infty} (\lambda\gamma)^{t'-t} (r_{t'} + \gamma V_{t'+1} - V_{t'}).} \tag{32}$$

A few observations:

- GAE looks like a discounted sum of rewards, where the rewards are *shaped* by the value function. The effective discount is $\hat{\gamma} = \lambda\gamma$.
- Each term $\delta_{t'}^V$ is a TD error, so GAE is a discounted sum of TD errors.
- In practice, we truncate at $t' = t + T$ for a finite horizon T, collecting T steps from N parallel actors.
- Many implementations normalize the advantages across all transitions before computing the policy gradient.

4.4 Special cases of GAE

Let's verify that GAE reduces to known estimators at the boundary values of λ.

Case $\lambda = 0$. The estimator collapses to the 1-step TD residual:

$$\hat{A}_t^{\text{GAE}(\gamma,0)} = \delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t). \tag{33}$$

Low variance (only one stochastic step), but high bias: everything depends on V being accurate.

Case $\lambda = 1$. The sum telescopes — the V terms cancel pairwise:

$$\hat{A}_t^{\text{GAE}(\gamma,1)} = \sum_{t'=t}^{\infty} \gamma^{t'-t} \delta_{t'}^V = \left(\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} \right) - V(s_t). \tag{34}$$

This is the full Monte Carlo return minus the baseline. Unbiased (assuming V is only used as a baseline, not for bootstrapping), but high variance because we sum over many stochastic rewards.

Key idea

Note, we can think of the evaluator of actions as the critic in an actor-critic algorithm, which we then use to improve the actor (the policy)

5 PPO: Putting It All Together

We now have all the ingredients for the full PPO algorithm. PPO has two main components: (1) the clipped surrogate objective (Eq. 19) that stabilizes policy updates, and (2) the GAE advantage estimator (Eq. 32) that balances bias and variance in the critic.

Algorithm 1 PPO (Schulman et al., 2017)

Initialize policy $\pi_\theta(a \mid s)$, value function $V_\phi(s)$.
while not converged **do**

> Collect T environment steps from N actors (in parallel).
> Compute advantages for all $N \cdot T$ transitions using GAE (Eq. 32).
> Take gradient of clipped surrogate objective (Eq. 19) w.r.t. θ, doing a total of K passes over the data (corresponding to many minibatches). ▷ Uses off-policy data!
> Update value function V_ϕ by minimizing Eq. 20 with gradient descent.

Key design decisions.

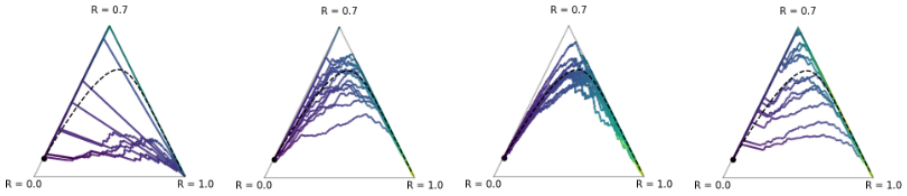
- **Clipped surrogate:** keeps policy close to data-generating policy, justified by TRPO's monotonic improvement bound. This is what makes PPO different from REINFORCE.
- **GAE ($\lambda = 0.95$):** balances bias/variance in advantage estimation, motivated by Kearns–Singh (Eq. **??**).
- **Advantage normalization:** subtracting the batch mean and dividing by std. Note: this isn't anywhere in the theory, but basically all repos do this.
- **Multiple epochs (K passes):** this is off-policy! Exactly why we need importance weighting and clipping. The clipping prevents catastrophically large updates when the policy has drifted far from the data-generating policy.
- **Separate value network:** sharing layers between policy and value networks is common but can hurt performance (Huang et al., 2022). The "37 implementation details" paper is worth reading.

5.1 Understanding what we know and what we don't know about baselines

While in theory, baselines as control varaites should reduce variance, recall that all theory relies on assumptions. Under other theoretical assumptions, baselines may play a different role. Mei et al. (2022) study the role of baselines in policy gradient optimization and show that for natural policy gradient methods, the variance of gradient estimates remains *unbounded* with or without a baseline. Instead, the primary benefit of the baseline is to **reduce the aggressiveness of updates**. Without a baseline, all actions with positive return get upweighted, potentially making very large updates that overshoot. They make the case that the baseline centers the gradient signal, which has a regularizing effect on step sizes. And that "finite variance is not necessary for almost sure convergence of stochastic NPG, while controlling update aggressiveness is both necessary and sufficient."

Chung et al. (2021) show that even when two baselines produce the *same variance*, they can lead to completely different optimization dynamics. In certain cases, a lower-variance baseline actually performs worse. Some baselines cause convergence to suboptimal policies *for any stepsize*. They suggest that the baseline interacts with the curvature of the policy parameterization. In short:

Different baselines can give rise to very different learning dynamics, even when they reduce the variance of the gradi- ents equally. They do that by either making a policy quickly tend towards a deterministic one (committal behaviour) or by maintaining high-entropy for a longer period of time (non-committal behaviour). We showed that committal be- haviour can be problematic and lead to convergence to a suboptimal policy. Specifically, we showed that stochas- tic natural policy gradient does not always converge to the optimal solution due to the unusual situation in which the iterates converge to the optimal policy in expectation but not almost surely. Moreover, we showed that baselines that lead to lower-variance can sometimes be detrimental to op- timization, highlighting the limitations of using variance to analyze the convergence properties of these methods. We also showed that standard convergence guarantees for PG methods do not apply to some settings because the assump- tion of bounded variance of the updates is violated.



(a) $b_\theta^- = b_\theta^* - 1/2$ (b) $b_\theta = b_\theta^*$ (c) $b_\theta^+ = b_\theta^* + 1/2$ (d) $b_\theta = V^{\pi_\theta}$

Figure 1: We plot 15 different trajectories of natural policy gradient with softmax parameterization, when using various baselines, on a 3-arm bandit problem with rewards $(1, 0.7, 0)$ and stepsize $\alpha = 0.025$ and $\theta_0 = (0, 3, 5)$. The black dot is the initial policy and colors represent time, from purple to yellow. The dashed black line is the trajectory when following the true gradient (which is unaffected by the baseline). Different values of ϵ denote different perturbations to the minimum-variance baseline. We see some cases of convergence to a suboptimal policy for both $\epsilon = -1/2$ and $\epsilon = 0$. This does not happen for the larger baseline $\epsilon = 1/2$ or the value function as baseline. Figure made with Ternary (Harper & Weinstein, 2015).

Implications for Frontiers

Baselines and dynamics of learning, particularly for language models, are a rich and active area of research. Note: the result from Chung et al. (2021) might be different if you consider exploration regularization, like entropy regularizers, which we will talk about next week. John Schulman (co-founder of Thinking Machines and first author on PPO/TRPO) discussed this on the Cursor podcast [video] in a segment titled "The Absence of Value Functions" and suggests that value functions don't seem to provide much variance reduction compared to simpler alternatives. This appears to reflect the state of LLM-RL research today. Why might this be the case? Let's reason through this using the theoretical tools we've developed. Discuss with the group!

5.2 Group Relative Policy Optimization (GRPO)

Instead, what people do in practice right now is something like Group Relative Policy Optimization (GRPO) from DeepSeek (DeepSeek-AI, 2024) generates a group of G completions for each prompt and estimates advantages by normalizing within the group:

$$\hat{A}_i = \frac{r_i - \text{mean}(\{r_1, \ldots, r_G\})}{\text{std}(\{r_1, \ldots, r_G\})}, \tag{35}$$

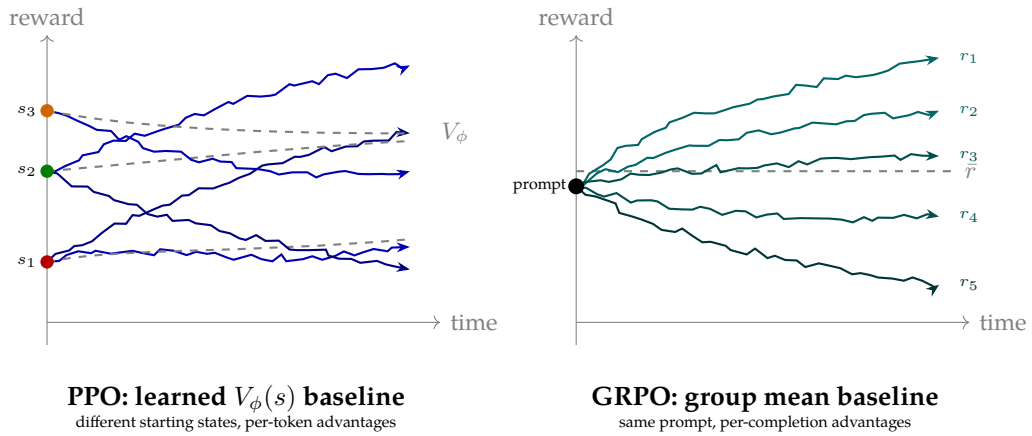


Figure 1: PPO vs. GRPO baseline strategies. **Left:** PPO collects trajectories from different states and uses a learned value function $V_\phi(s)$ as a per-state, per-token baseline. **Right:** GRPO generates G completions from the same prompt and uses the group mean reward $\bar{r}$ as a per-completion baseline — no learned value function needed.

where r_i is the reward for completion i. This requires no learned value function since the group mean serves as the baseline. The GRPO objective has the same PPO-style clipping:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \frac{1}{G}\sum_{i=1}^{G}\frac{1}{|a_i|}\sum_{t=1}^{|a_i|}\min\Big(r_t(\theta)\,\hat{A}_i,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}_i\Big) - \beta\, D_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}], \tag{36}$$

where $r_t(\theta) = \frac{\pi_\theta(a_{i,t}|s, a_{i,<t})}{\pi_{\theta_{\text{old}}}(a_{i,t}|s, a_{i,<t})}$. Note, the added KL term is typical in LLM post-training to prevent forgetting of everything you pretrained on. You can remove the term if you want to.

Implications for Frontiers

How does this make the bias variance trade-off from Kearns and Singh (2000)? What properties of the tasks is it counting on?

6 Break

Take 20 minutes. When we come back, we'll shift gears.

7 From Q-Learning to Continuous Actions: DDPG and TD3

So far, everything we've covered flows from policy gradient methods. The value function shows up as a mechanism to reduce variance (baselines, GAE) or to stabilize updates (TRPO's improvement bound). The policy is the primary object, and the critic is there to help.

But there's another way to think about actor-critic methods. What if we start from the *value function* side — from Q-learning — and use the policy as a mechanism to approximate the max over Q-values?

7.1 The problem with Q-learning in continuous actions

Recall that in Q-learning, the update involves $\max_a Q(s', a)$. In discrete action spaces, this is easy: just enumerate all actions and pick the largest. In continuous action spaces, this is an optimization problem:

$$\max_{a \in \mathcal{A}} Q(s, a). \tag{37}$$

We could try to solve this with gradient ascent at every step, but that's expensive. Instead, we can learn a *deterministic policy* $\mu_\theta(s)$ that approximates $\arg\max_a Q(s, a)$. This is the idea behind DDPG (Lillicrap et al., 2016).

7.2 DDPG: Deep Deterministic Policy Gradient

DDPG (Lillicrap et al., 2016) maintains two networks: a deterministic policy (actor) $\mu_\theta(s)$ and a Q-function (critic) $Q_\phi(s, a)$. The actor is trained to maximize the critic:

$$\max_\theta \ \mathbb{E}_s[Q_\phi(s, \mu_\theta(s))]\,. \tag{38}$$

Taking the gradient with respect to θ:

$$\nabla_\theta \mathbb{E}_s[Q_\phi(s, \mu_\theta(s))] = \mathbb{E}_s\Big[\nabla_a Q_\phi(s, a)\big|_{a=\mu_\theta(s)} \nabla_\theta \mu_\theta(s)\Big]\,. \tag{39}$$

This is the **deterministic policy gradient** (Silver et al., 2014). Notice: no log probabilities, no score function, no REINFORCE. We're just backpropagating through the Q-function into the policy via the chain rule.

The critic is trained with standard Bellman error, using transitions from a replay buffer:

$$\min_\phi \ \mathbb{E}_{(s,a,r,s')}\left[\Big(Q_\phi(s, a) - \big(r + \gamma Q_{\bar{\phi}}(s', \mu_{\bar{\theta}}(s'))\big)\Big)^2\right], \tag{40}$$

where $\bar{\phi}$ and $\bar{\theta}$ are **target networks** — slowly-updated copies of the critic and actor. These stabilize the bootstrap target and prevent the kind of divergence we saw with naive Q-learning.

Connection to amortized optimization. There's a nice way to see what DDPG is doing. Think of the policy μ_θ as *amortizing* the optimization $\arg\max_a Q(s, a)$. Instead of solving a new optimization problem for every state (expensive), we train a neural network to predict the answer directly (cheap at test time). This is the same idea as amortized inference in variational autoencoders.

7.3 TD3: Group exercise

DDPG works in principle, but has a well-known failure mode: the Q-function tends to *overestimate* values, and the policy then exploits these errors. The actor finds states and actions where the critic is wrong, leading to poor performance. TD3 (Twin Delayed DDPG) (Fujimoto et al., 2018) fixes this.

Rather than me just telling you what TD3 does, let's read the code. Below is the reference DDPG implementation from Fujimoto et al., followed by their TD3 implementation. Both are from `github.com/sfujim/TD3`.

DDPG implementation. Listing 1: DDPG (from `sfujim/TD3/DDPG.py`)

```python
import copy
import torch
import torch.nn as nn
import torch.nn.functional as F

class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, max_action):
        super(Actor, self).__init__()
        self.l1 = nn.Linear(state_dim, 256)
        self.l2 = nn.Linear(256, 256)
        self.l3 = nn.Linear(256, action_dim)
        self.max_action = max_action

    def forward(self, state):
        a = F.relu(self.l1(state))
        a = F.relu(self.l2(a))
        return self.max_action * torch.tanh(self.l3(a))

class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        self.l1 = nn.Linear(state_dim + action_dim, 256)
        self.l2 = nn.Linear(256, 256)
        self.l3 = nn.Linear(256, 1)

    def forward(self, state, action):
        q = F.relu(self.l1(torch.cat([state, action], 1)))
        q = F.relu(self.l2(q))
        return self.l3(q)

class DDPG(object):
    def __init__(self, state_dim, action_dim, max_action,
                 discount=0.99, tau=0.005):
        self.actor = Actor(state_dim, action_dim, max_action)
        self.actor_target = copy.deepcopy(self.actor)
        self.actor_optimizer = torch.optim.Adam(
            self.actor.parameters(), lr=3e-4)

        self.critic = Critic(state_dim, action_dim)
        self.critic_target = copy.deepcopy(self.critic)
        self.critic_optimizer = torch.optim.Adam(
            self.critic.parameters(), lr=3e-4)

        self.discount = discount
        self.tau = tau

    def select_action(self, state):
```

```
        state = torch.FloatTensor(state.reshape(1, -1))
        return self.actor(state).cpu().data.numpy().flatten()

    def train(self, replay_buffer, batch_size=256):
        state, action, next_state, reward, not_done = \
            replay_buffer.sample(batch_size)

        # Compute target Q value
        target_Q = self.critic_target(
            next_state, self.actor_target(next_state))
        target_Q = reward + (not_done * self.discount * target_Q).detach()

        # Critic loss
        current_Q = self.critic(state, action)
        critic_loss = F.mse_loss(current_Q, target_Q)

        # Optimize critic
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # Actor loss: maximize Q
        actor_loss = -self.critic(state, self.actor(state)).mean()

        # Optimize actor
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()

        # Soft update target networks
        for param, target_param in zip(
                self.critic.parameters(),
                self.critic_target.parameters()):
            target_param.data.copy_(
                self.tau * param.data
                + (1 - self.tau) * target_param.data)

        for param, target_param in zip(
                self.actor.parameters(),
                self.actor_target.parameters()):
            target_param.data.copy_(
                self.tau * param.data
                + (1 - self.tau) * target_param.data)
```

TD3 implementation. Now read this carefully and spot the differences.

Listing 2: TD3 (from `sfujim/TD3/TD3.py`)

```
import copy
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, max_action):
        super(Actor, self).__init__()
        self.l1 = nn.Linear(state_dim, 256)
        self.l2 = nn.Linear(256, 256)
        self.l3 = nn.Linear(256, action_dim)
        self.max_action = max_action

    def forward(self, state):
        a = F.relu(self.l1(state))
        a = F.relu(self.l2(a))
        return self.max_action * torch.tanh(self.l3(a))

class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        # Q1
        self.l1 = nn.Linear(state_dim + action_dim, 256)
        self.l2 = nn.Linear(256, 256)
        self.l3 = nn.Linear(256, 1)
        # Q2
        self.l4 = nn.Linear(state_dim + action_dim, 256)
        self.l5 = nn.Linear(256, 256)
        self.l6 = nn.Linear(256, 1)

    def forward(self, state, action):
        sa = torch.cat([state, action], 1)
        q1 = F.relu(self.l1(sa))
        q1 = F.relu(self.l2(q1))
        q1 = self.l3(q1)

        q2 = F.relu(self.l4(sa))
        q2 = F.relu(self.l5(q2))
        q2 = self.l6(q2)
        return q1, q2

    def Q1(self, state, action):
        sa = torch.cat([state, action], 1)
        q1 = F.relu(self.l1(sa))
        q1 = F.relu(self.l2(q1))
        q1 = self.l3(q1)
        return q1

class TD3(object):
    def __init__(self, state_dim, action_dim, max_action,
                 discount=0.99, tau=0.005,
                 policy_noise=0.2, noise_clip=0.5,
                 policy_freq=2):
        self.actor = Actor(state_dim, action_dim, max_action)
        self.actor_target = copy.deepcopy(self.actor)
        self.actor_optimizer = torch.optim.Adam(
            self.actor.parameters(), lr=3e-4)

        self.critic = Critic(state_dim, action_dim)
```

```
        self.critic_target = copy.deepcopy(self.critic)
        self.critic_optimizer = torch.optim.Adam(
            self.critic.parameters(), lr=3e-4)

        self.max_action = max_action
        self.discount = discount
        self.tau = tau
        self.policy_noise = policy_noise
        self.noise_clip = noise_clip
        self.policy_freq = policy_freq
        self.total_it = 0

    def select_action(self, state):
        state = torch.FloatTensor(state.reshape(1, -1))
        return self.actor(state).cpu().data.numpy().flatten()

    def train(self, replay_buffer, batch_size=256):
        self.total_it += 1
        state, action, next_state, reward, not_done = \
            replay_buffer.sample(batch_size)

        with torch.no_grad():
            # Target policy smoothing
            noise = (torch.randn_like(action) * self.policy_noise
                ).clamp(-self.noise_clip, self.noise_clip)
            next_action = (
                self.actor_target(next_state) + noise
            ).clamp(-self.max_action, self.max_action)

            # Clipped double Q: take the min
            target_Q1, target_Q2 = self.critic_target(
                next_state, next_action)
            target_Q = torch.min(target_Q1, target_Q2)
            target_Q = reward + not_done * self.discount * target_Q

        # Both critics against same target
        current_Q1, current_Q2 = self.critic(state, action)
        critic_loss = (F.mse_loss(current_Q1, target_Q)
                       + F.mse_loss(current_Q2, target_Q))

        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # Delayed policy updates
        if self.total_it % self.policy_freq == 0:
            # Actor loss: maximize Q1 only
            actor_loss = -self.critic.Q1(
                state, self.actor(state)).mean()

            self.actor_optimizer.zero_grad()
            actor_loss.backward()
            self.actor_optimizer.step()
```

```
        # Soft update target networks
        for param, target_param in zip(
                self.critic.parameters(),
                self.critic_target.parameters()):
            target_param.data.copy_(
                self.tau * param.data
                + (1 - self.tau) * target_param.data)

        for param, target_param in zip(
                self.actor.parameters(),
                self.actor_target.parameters()):
            target_param.data.copy_(
                self.tau * param.data
                + (1 - self.tau) * target_param.data)
```

Group exercise. With your group, identify the changes TD3 makes relative to DDPG. For each one, find the exact lines of code where the change happens and discuss.

7.4 Comparing the two families

We now have two families of actor-critic methods, stemming from different origins:

	PPO / TRPO / GRPO	**DDPG / TD3 / SAC**
Origin	Policy gradient	Q-learning
Policy type	Stochastic	Deterministic (or entropy-regularized)
Role of critic	Reduce variance / stabilize	Approximate the max
Data usage	On-policy (with IS)	Off-policy (replay buffer)
Action space	Discrete or continuous	Continuous

Both families are actor-critic methods, but they differ in where the "intelligence" lives. In PPO, the policy is the primary object and the value function is a helper. In DDPG, the Q-function is the primary object and the policy is a helper.

References

DeepSeek-AI (2024). DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Chung, W., Thomas, V., Machado, M. C., and Le Roux, N. (2021). Beyond variance reduction: Understanding the true impact of baselines on policy optimization. In *International Conference on Machine Learning (ICML)*, pages 1999–2009. PMLR.

Fujimoto, S., van Hoof, H., and Meger, D. (2018). Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning (ICML)*, pages 1587–1596. PMLR.

Greensmith, E., Bartlett, P. L., and Baxter, J. (2004). Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5:1471–1530.

Huang, S., Dossa, R. F. J., Raffin, A., Kanervisto, A., and Wang, W. (2022). The 37 implementation details of proximal policy optimization. *The ICLR Blog Track 2023*.

Kakade, S. and Langford, J. (2002). Approximately optimal approximate reinforcement learning. In *Proceedings of the 19th International Conference on Machine Learning* (*ICML*), pages 267–274.

Konda, V. R. and Tsitsiklis, J. N. (1999). Actor-critic algorithms. In *Advances in Neural Information Processing Systems* (*NeurIPS*), volume 12, pages 1008–1014.

Kearns, M. and Singh, S. (2000). Bias-variance error bounds for temporal difference updates. In *Proceedings of the 13th Annual Conference on Computational Learning Theory* (*COLT*), pages 24–34.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2016). Continuous control with deep reinforcement learning. In *International Conference on Learning Representations* (*ICLR*).

Mei, J., Chung, W., Thomas, V., Dai, B., Szepesvári, C., and Schuurmans, D. (2022). The role of baselines in policy gradient optimization. In *Advances in Neural Information Processing Systems* (*NeurIPS*).

Qi, P., Liu, Z., Zhou, X., Pang, T., Du, C., Lee, W. S., and Lin, M. (2025). Defeating the training-inference mismatch via FP16. *arXiv preprint arXiv:2510.26788*.

Ng, A. Y., Harada, D., and Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *International Conference on Machine Learning* (*ICML*), pages 387–395. PMLR.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897. PMLR.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations* (*ICLR*).

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Schulman, J. (2024). Dead ends, scaling RL, and building research institutions. Interview on the *Cursor Podcast*. Segment: "The Absence of Value Functions" (16:54).

Thinking Machines Lab (2025). Defeating nondeterminism in LLM inference. Blog post, https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

Tucker, G., Bhupatiraju, S., Gu, S., Turner, R., Ghahramani, Z., and Levine, S. (2018). The mirage of action-dependent baselines in reinforcement learning. In *International Conference on Machine Learning* (*ICML*), pages 5015–5024. PMLR.

Zhang, Y., Li, Y., Liu, J., Xu, J., Li, Z., Liu, Q., and Li, H. (2026). Beyond precision: Training-inference mismatch is an optimization problem and simple LR scheduling fixes it. *arXiv preprint arXiv:2602.01826*.