

Policy Gradients

Aayush Bajaj

July 10, 2026

Contents

1	the objective	1
2	the policy gradient theorem	2
2.1	the likelihood-ratio derivation	2
2.2	causality tightens it	2
3	reinforce	3
4	baselines	3
4.1	why this doesn't bias the gradient	3
5	actor-critic, and the advantage	4
6	natural gradients, and ppo	4
7	a corridor, from scratch	5
8	see also	6

value-based methods (q-learning and family) learn *how good actions are* and act by argmax. policy-gradient methods skip the middleman: parameterise the policy itself, $\pi_\theta(a \mid s)$, and do gradient ascent on expected return. the entire family — reinforce, actor-critic, trpo, ppo, and by extension rlhf — rests on one identity, the policy gradient theorem, whose derivation is three lines of calculus and one very good idea. the standard reference is sutton & Barto, free at <http://incompleteideas.net/book/the-book-2nd.html>.

the argmax is also the problem: it is undefined over continuous actions and allergic to stochastic policies. differentiation handles both

1 the objective

let $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \dots)$ be a trajectory generated by running π_θ in the environment, with total (discounted) reward $R(\tau) = \sum_t \gamma^t r_{t+1}$. the objective is plain:

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta}[R(\tau)], \quad p_\theta(\tau) = p(s_0) \prod_t \pi_\theta(a_t \mid s_t) P(s_{t+1} \mid s_t, a_t). \quad (1)$$

maximise J by gradient ascent. the obstruction: θ lives inside the *distribution* the expectation is taken over, not inside the thing being averaged — you cannot push ∇_θ through the expectation naively, and the trajectory distribution contains the unknown dynamics P .

2 the policy gradient theorem

2.1 the likelihood-ratio derivation

write the expectation as an integral and use the **log-derivative trick** — for any positive function, $\nabla p = p \nabla \log p$, because $\nabla \log p = \nabla p / p$:

$$\begin{aligned}\nabla_{\theta} J(\theta) &= \nabla_{\theta} \int p_{\theta}(\tau) R(\tau) d\tau = \int \nabla_{\theta} p_{\theta}(\tau) R(\tau) d\tau \\ &= \int p_{\theta}(\tau) \nabla_{\theta} \log p_{\theta}(\tau) R(\tau) d\tau = \mathbb{E}_{\tau \sim p_{\theta}}[R(\tau) \nabla_{\theta} \log p_{\theta}(\tau)].\end{aligned}$$

the swap of ∇_{θ} and $\int$ is legitimate for the smooth policy classes used in practice. now expand $\log p_{\theta}(\tau)$ — and watch the magic:

$$\log p_{\theta}(\tau) = \log p(s_0) + \sum_t \log \pi_{\theta}(a_t | s_t) + \sum_t \log P(s_{t+1} | s_t, a_t), \quad (2)$$

the initial-state term and every *dynamics* term are constant in θ , so their gradients vanish:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau} \left[R(\tau) \sum_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right]. \quad (3)$$

this is the punchline of the whole field: **the gradient of expected return through an unknown environment requires no model of that environment** — only the ability to differentiate your own policy and to sample rollouts. (Goodfellow, Ian, 2016)

the same score-function estimator turns up wherever gradients must pass through sampling: variational inference, discrete latent variables, black-box optimisation

2.2 causality tightens it

action a_t cannot influence rewards already banked $(r_1, \dots, r_t)$, and indeed the expectation of $\nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$ against any *past* reward is zero (same argument as for baselines below). dropping those terms replaces $R(\tau)$ with the **return-to-go** $G_t = \sum_{l \geq t} \gamma^{l-t} r_{l+1}$:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau} \left[\sum_t \gamma^t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) G_t \right], \quad (4)$$

strictly lower variance, identical expectation. (the γ^t out front is owed to the discounted objective; almost every implementation drops it and optimises the undiscounted visitation version instead — a deliberate, mildly biased simplification worth knowing you are making.) in its textbook form the **policy gradient theorem**¹

¹sutton, mcallester, singh & mansour (2000), *policy gradient methods for reinforcement learning with function approximation*, nips 12 — which also proves the compatible-function-approximation condition under which a learned critic leaves the gradient exact. reinforce itself is williams (1992), machine learning 8(3-4).

references

Goodfellow, Ian (2016). *Deep Learning*, MIT Press.

states the same thing distributionally: $\nabla_{\theta} J \propto \mathbb{E}_{s \sim d_{\pi}, a \sim \pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a | s) q_{\pi}(s, a)]$, with d_{π} the on-policy state distribution — the gradient wants log-probability mass pushed onto actions in proportion to how good they are.

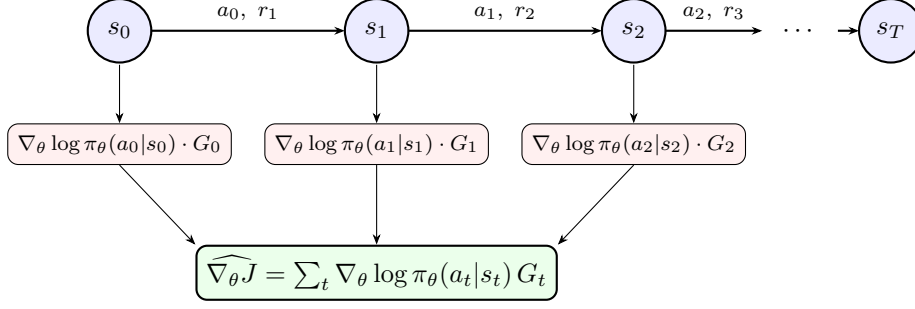


Figure 1: reinforce’s computational flow: roll a trajectory, score each action by its return-to-go G_t , weight the score-function terms, sum into one gradient estimate.

3 reinforce

williams’ 1992 algorithm is the theorem read as pseudocode — monte carlo policy gradient:

- **sample** a trajectory τ by running π_{θ} in the environment.
- **compute returns** G_t for every step, backwards from the end ($G_t = r_{t+1} + \gamma G_{t+1}$).
- **accumulate** the gradient estimate $\hat{g} = \sum_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) G_t$.
- **step** $\theta \leftarrow \theta + \alpha \hat{g}$, and repeat.

the estimator is exactly unbiased, and that is its *only* virtue: a single trajectory’s returns entangle every action’s randomness with every other’s, so the variance is enormous and grows with horizon. reinforce on raw returns learns — slowly, noisily, and with a learning rate you will tune by seance. everything after 1992 is a variance-reduction programme.

4 baselines

subtract a **state-dependent baseline** $b(s_t)$ from the return before weighting:

$$\hat{g} = \sum_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) (G_t - b(s_t)). \quad (5)$$

4.1 why this doesn’t bias the gradient

the subtracted term has expectation zero. fix a state s ; the baseline term contributes $\mathbb{E}_{a \sim \pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a | s) b(s)]$, and

$$\mathbb{E}_{a \sim \pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a | s)] = \sum_a \pi_{\theta}(a | s) \frac{\nabla_{\theta} \pi_{\theta}(a | s)}{\pi_{\theta}(a | s)} = \nabla_{\theta} \sum_a \pi_{\theta}(a | s) = \nabla_{\theta} 1 = 0, \quad (6)$$

so $b(s)$ — *any* function of the state, however terrible — multiplies a zero-mean quantity and leaves $\mathbb{E}[\hat{g}]$ untouched. the catch in the proof: b must not depend on the action a_t (or anything downstream of it), or the factorisation breaks and bias returns. variance, meanwhile, *does* change with b , and is roughly minimised near $b(s) = v_\pi(s)$ — centre the returns on what you expected to get, and only genuine surprise moves the policy.

the score function has mean zero under its own distribution — the same fact that makes the fisher information a covariance

5 actor-critic, and the advantage

learning the baseline v_π with a second function approximator turns reinforce into an **actor-critic**:

- the **actor** π_θ takes the gradient steps;
- the **critic** v_w is regressed toward observed returns (monte carlo) or bootstrapped td targets, and supplies the baseline — or more aggressively, replaces G_t entirely with the one-step estimate $r_{t+1} + \gamma v_w(s_{t+1})$, trading reinforce’s variance for td’s bias exactly as in **q-learning**’s world.

the quantity being estimated in all cases is the **advantage function**

$$A_\pi(s, a) = q_\pi(s, a) - v_\pi(s), \quad (7)$$

“how much better is this action than my average behaviour here” — mean zero under the policy, which is precisely what a well-centred gradient weight should be. generalised advantage estimation (gae) interpolates between the one-step td estimate and the full monte carlo return with an exponentially-weighted λ -average, giving the practitioner one dial for the whole bias–variance spectrum.

6 natural gradients, and ppo

natural gradients. vanilla gradient ascent measures step size in parameter space, but the same policy change costs wildly different amounts of $\|\Delta\theta\|$ in different parameterisations. the natural gradient $F^{-1}\nabla_\theta J$ — with F the fisher information matrix of π_θ — is steepest ascent measured in kl divergence between *policies*, invariant to how the policy is parameterised. trpo made this practical for deep networks by maximising the objective subject to a hard kl trust region, solved approximately with conjugate gradients; the cost is second-order machinery on every update.

ppo clipping. ppo gets most of trpo’s stability for a fraction of the machinery. with probability ratio $\rho_t(\theta) = \pi_\theta(a_t | s_t) / \pi_{\theta_{\text{old}}}(a_t | s_t)$, maximise the pessimistic clipped surrogate

$$L(\theta) = \mathbb{E}_t \left[\min(\rho_t(\theta) \hat{A}_t, \text{clip}(\rho_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t) \right], \quad (8)$$

which flattens the objective once the ratio strays outside $[1 - \epsilon, 1 + \epsilon]$ *in the direction that would improve it* — the incentive to push further dies, so several epochs of first-order sgd on recycled rollouts stay approximately on-policy. that is the entire trick, it is one min and one clip, and it is the default deep-rl (and rlhf) algorithm of the current era largely because it is hard to implement wrong.

7 a corridor, from scratch

reinforce with a learned per-state baseline on a five-cell corridor (start at cell 0, +10 at cell 4, −1 per step), softmax policy over the two actions left/right; then a direct check of the baseline claims — same 3000 rollouts fed to both estimators at a frozen uniform policy:

```
1 import numpy as np
2
3 # 1d corridor: cells 0..4, start at 0, goal at 4 (terminal, +10).
4 # actions: 0=left 1=right, -1 per step, episodes capped at 30 steps.
5 N, GOAL, GAMMA = 5, 4, 0.99
6
7 def softmax(logits):
8     p = np.exp(logits - logits.max())
9     return p / p.sum()
10
11 def episode(theta, rng):
12     """roll out one episode under the softmax policy; return trajectory."""
13     s, traj = 0, []
14     for _ in range(30):
15         a = rng.choice(2, p=softmax(theta[s]))
16         ns = max(0, s - 1) if a == 0 else s + 1
17         r = 10.0 if ns == GOAL else -1.0
18         traj.append((s, a, r))
19         if ns == GOAL:
20             break
21         s = ns
22     return traj
23
24 def returns_to_go(traj):
25     G, Gs = 0.0, []
26     for (_, _, r) in reversed(traj):
27         G = r + GAMMA * G
28         Gs.append(G)
29     return Gs[::-1]
30
31 # --- reinforce with a learned per-state baseline -----
32 rng = np.random.default_rng(7)
33 theta, baseline = np.zeros((N, 2)), np.zeros(N)
34 hist = []
35 for ep in range(2000):
36     traj = episode(theta, rng)
37     Gs = returns_to_go(traj)
38     hist.append(Gs[0])
39     for (s, a, _), G in zip(traj, Gs):
40         adv = G - baseline[s] # advantage estimate
41         grad_logp = -softmax(theta[s])
42         grad_logp[a] += 1.0 # grad of log pi(a|s) wrt
43         ↪ theta[s]
44         theta[s] += 0.1 * adv * grad_logp # ascend
45         baseline[s] += 0.1 * (G - baseline[s]) # track the value
46     if ep + 1 in (100, 500, 2000):
47         print(f"ep {ep+1:>4}: mean return (last 100) =
48             ↪ {np.mean(hist[-100:]).2f}")
49
50 p = np.array([softmax(t) for t in theta])
51 print("final policy P(right|s):",
52       " ".join(f"s{s}:{p[s,1]:.3f}" for s in range(N - 1)))
53
54 # --- does the baseline bias the gradient? measure at a fixed policy -----
55 rng = np.random.default_rng(0)
56 theta0 = np.zeros((N, 2)) # uniform random policy
57 b = np.mean([returns_to_go(episode(theta0, rng))[0] for _ in range(3000)])
58 raw, based = [], []
```

```

57 for _ in range(3000):                                # same episodes for both
58     traj = episode(theta0, rng)
59     G0 = returns_to_go(traj)[0]
60     s0, a0, _ = traj[0]
61     g = -softmax(theta0[s0]); g[a0] += 1.0            # single-sample gradient, t=0
62     ↪ term
63     raw.append(G0 * g[1])                             # no baseline
64     based.append((G0 - b) * g[1])                     # baseline subtracted
65 raw, based = np.array(raw), np.array(based)
66 print(f"grad estimate without baseline: mean {raw.mean():+.3f} std
67 ↪ {raw.std():.3f}")
68 print(f"grad estimate with baseline: mean {based.mean():+.3f} std
69 ↪ {based.std():.3f}")
70 print(f"variance ratio: {based.var() / raw.var():.2f}")

```

```

ep 100: mean return (last 100) = 5.89
ep 500: mean return (last 100) = 6.69
ep 2000: mean return (last 100) = 6.73
final policy P(right|s): s0:0.998 s1:0.998 s2:0.998 s3:0.998
grad estimate without baseline: mean +0.310 std 6.765
grad estimate with baseline: mean +0.378 std 5.627
variance ratio: 0.69

```

reading the output:

- the policy converges to “go right everywhere” with near-certainty, and the mean return settles at 6.73 — exactly the optimal $-1 - 0.99 - 0.99^2 + 10 \cdot 0.99^3$ for the four-step corridor.
- the two gradient estimators agree in expectation up to sampling noise — the gap between +0.310 and +0.378 is b times the *empirical* mean of the score, a quantity that is zero only in expectation (3000 samples of a coin flip do not average to exactly one half). unbiasedness is a statement about means, and both point the same way: increase $P(\text{right})$.
- the baseline cuts the estimator’s variance to $0.69\times$ — with a crude constant baseline on a toy problem. per-state value baselines on long-horizon tasks are the difference between learning and thrashing.

8 see also

- [q-learning](#) — the value-based road to the same destination
- [cartpole](#) — where reinforce implementations traditionally earn their keep
- [optimisers](#) — the SGD machinery all of this rides on
- [loss functions](#) — surrogate objectives, of which ppo’s clip is a particularly sneaky one