Lecture 2: Introduction to Policy Gradient Methods

Wouter van Heeswijk

23 July 2025

Outline

From value-based to policy-based learning

Value-Based vs Policy-Based Methods

Value-Based:

- Learn action values $Q(s, a)$
- Policy: pick action with highest Q ($\arg\max$)
- Works well for discrete, small action spaces
- Difficult with continuous or combinatorial actions

Policy-Based:

- Learn policy $\pi_\theta(a|s)$ directly
- Policy outputs a distribution over actions (stochastic!)
- Suitable for continuous, structured actions
- Requires sampling and gradient estimation, which can be noisy

Tradeoff: *structured targets vs direct optimization*

Why Must Policies Be Stochastic?

To learn, we need a gradient signal from sampled actions:

- Policy gradient methods estimate how changing θ changes expected rewards by sampling actions
- This estimation requires the policy to output a **probability distribution**
- Deterministic policies pick a fixed action — no variability, no signal

In short:

Stochasticity is essential because it provides the learning signal — without it, the gradient vanishes.

Policy Gradient: Core Setup

Goal: Maximize expected return:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

Approach:

- **Sample** trajectories from π_θ
- **Estimate** gradient using rewards
- **Update** θ via gradient ascent

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

No Bellman equation, no value function, no arg max

Policy-based learning

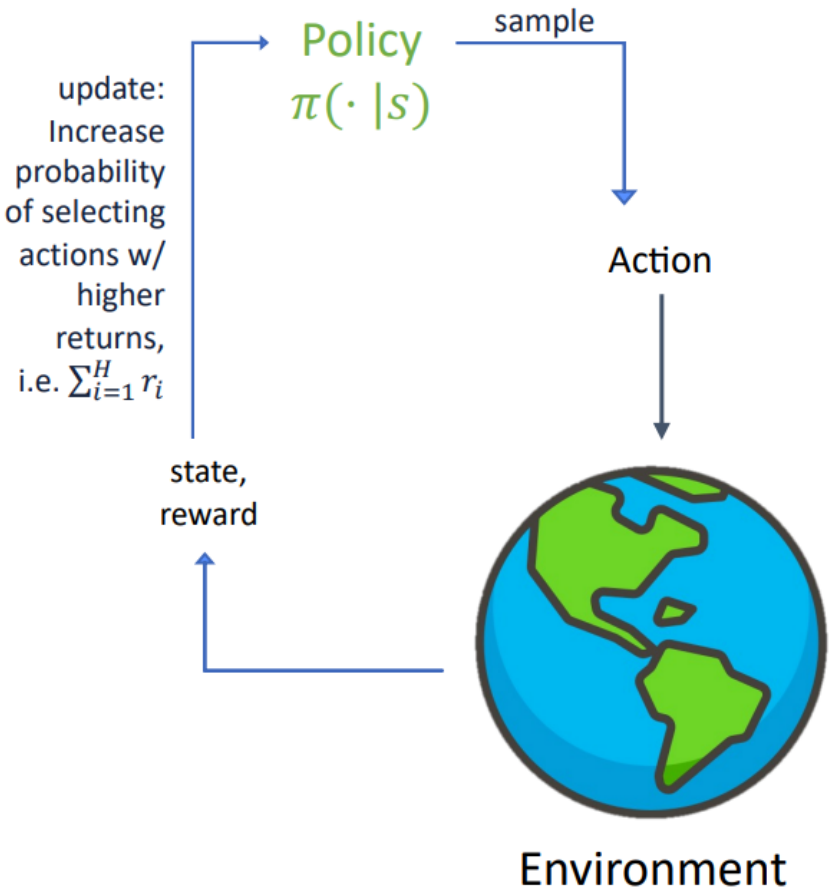

Figure: Wu, C. (2024). Policy gradient. 6.7920: Reinforcement Learning: Foundations and Methods

Gradient-Free Policy Optimization

Policies can also be updated without gradients

- **Hill Climbing / Random Search:**
  - Perturb policy parameters randomly and keep improvements
  - Simple but can be sample inefficient
- **Evolution Strategies (ES):**
  - Population-based search over policy parameters
  - Uses fitness evaluations instead of gradients
  - Scales well with parallelism
- **Cross-Entropy Method (CEM):**
  - Iteratively fit a distribution over good policies
  - Sample, evaluate, and update the distribution
- **Finite-Difference Methods:**
  - Approximate gradient via small parameter perturbations
  - Useful when gradients are unavailable or noisy

Trade-offs: Generally simpler and more robust but less sample efficient than gradient-based methods.

Policy Gradients for Combinatorial Actions

Examples: Routing, scheduling, ranking $\Rightarrow$ Action = a structured object (e.g., sequence, set)

Why not use DQN?

- Must compute $Q(s, a)$ for every possible action
- Combinatorial explosion: $|\mathcal{A}| = n!$ or 2^n
- Even $\arg\max_a Q(s, a)$ is intractable

Policy Gradient Benefit:

- Learn **structured stochastic policies** $\pi_\theta(a|s)$
- Sample complex actions step-by-step
- No need to enumerate or score all actions

Key: PG methods scale better to large structured action spaces

Derivation of policy gradient theorem

Objective of Policy Gradient Methods

Goal: Maximize the expected return of a stochastic policy $\pi_\theta(a|s)$:

$$\boxed{J(\theta) = \mathbb{E}_{\tau \sim P(\cdot;\theta)}[R(\tau)]} \quad \text{where } R(\tau) = \sum_{t=0}^{T} r(s_t, a_t)$$

Policy-based RL:

- Directly parameterizes the policy π_θ
- Optimizes $J(\theta)$ via gradient ascent:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta_k)$$

Next step: How do we compute $\nabla_\theta J(\theta)$?

Why Gradient Estimation Is Hard

The objective $J(\theta) = \mathbb{E}_{\tau \sim P(\cdot;\theta)}[R(\tau)]$ is an expectation over trajectories: $\tau = (s_0, a_0, \ldots, s_T) \sim P(\tau;\theta)$.

Trajectory distribution:

$$P(\tau;\theta) = \underbrace{\rho(s_0)}_{\text{initial state}} \cdot \prod_{t=0}^{T-1} \underbrace{\pi_\theta(a_t|s_t)}_{\text{policy (learned)}} \cdot \underbrace{P(s_{t+1}|s_t, a_t)}_{\text{environment (fixed)}}$$

Problem:

- $P(\tau;\theta)$ mixes learnable terms (π_θ) with unknown, non-differentiable ones ($P(s'|s,a)$)
- Makes direct computation of $\nabla_\theta J(\theta)$ infeasible

We need a workaround to isolate the effect of the policy.

Trajectory sampling

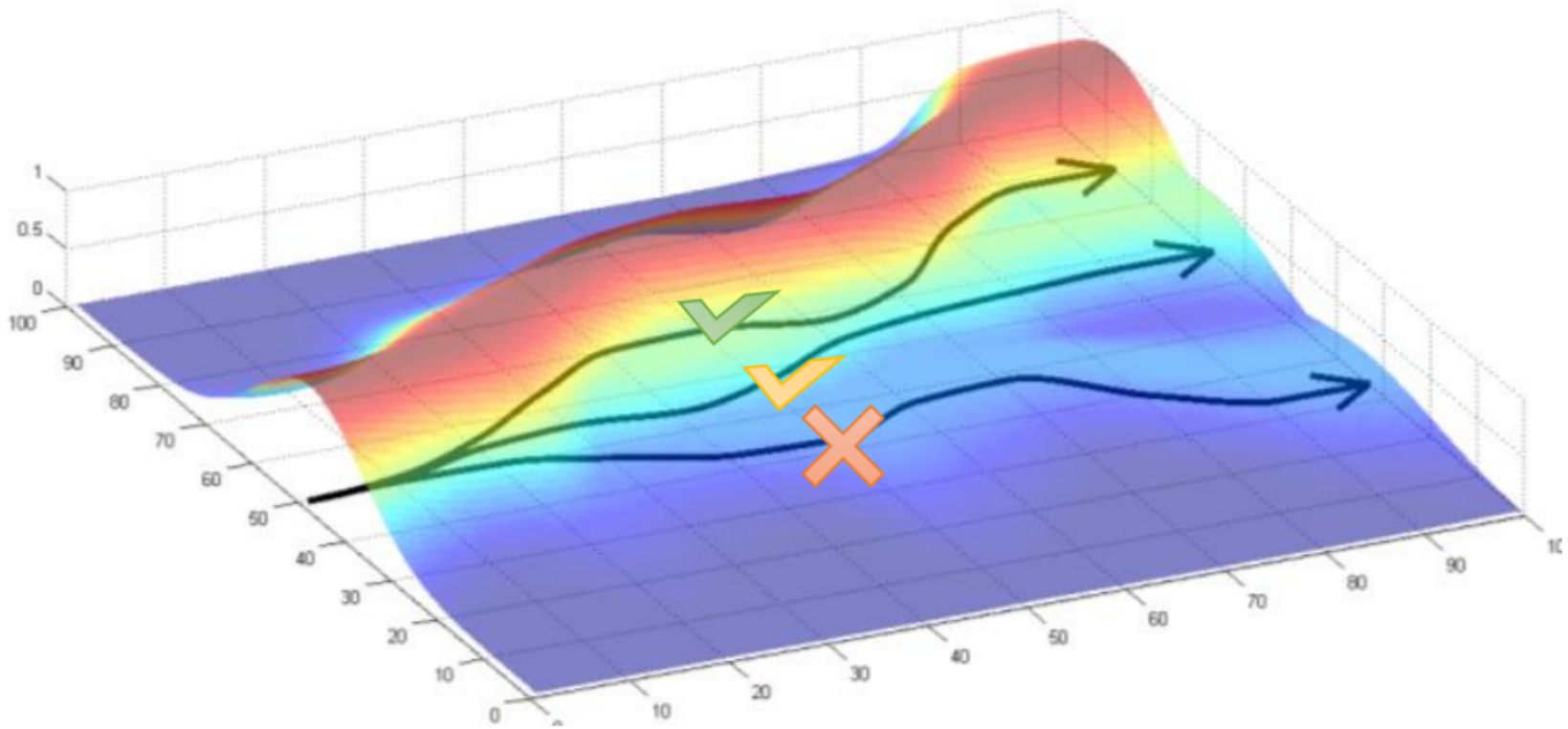

Figure: Levine, S. Policy Gradients. CS 294-112: Deep Reinforcement Learning

Why Not Just Differentiate?

We want to compute:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim P(\cdot;\theta)} [R(\tau)]$$

But: $P(\tau;\theta)$ includes both:

- $\pi_\theta(a_t|s_t)$: learnable
- $P(s_{t+1}|s_t, a_t)$: non-differentiable, potentially unknown

We can sample from $P(\tau;\theta)$, but not differentiate through it.

Key idea: Rewriting the gradient to only involve terms we control — the policy.

This is where the log-derivative trick comes in.

Starting Point: Naive Gradient

First, rewrite from gradient of sum to sum of gradients.

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_\tau P(\tau;\theta) R(\tau) = \sum_\tau \nabla_\theta P(\tau;\theta) R(\tau)$$

Problem: $P(\tau;\theta)$ is a product of many terms $\rightarrow$ hard to differentiate directly, especially with sampling!

We will further rewrite this to allow sampling and get stable gradients.

Rewriting the gradient expression

$$\nabla_\theta J(\theta) = \sum_\tau \nabla_\theta P(\tau;\theta) R(\tau)$$

$$= \sum_\tau \frac{P(\tau;\theta)}{P(\tau;\theta)} \nabla_\theta P(\tau;\theta) R(\tau) \quad // \text{ multiply by 1}$$

$$= \sum_\tau P(\tau;\theta) \underbrace{\left(\frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} \right)}_{\text{Rewriteable!}} R(\tau)$$

Now we have an expectation, as $\sum_\tau P(\tau;\theta) = \mathbb{E}_{\tau \sim P(\cdot;\theta}$.

Apply the Log-Derivative Identity

$$\frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} = \nabla_\theta \log P(\tau;\theta) \quad // \text{ identity: } \nabla \log f = \frac{\nabla f}{f}$$

$$\Rightarrow \nabla_\theta J(\theta) = \sum_\tau P(\tau;\theta) \nabla_\theta \log P(\tau;\theta) R(\tau)$$

$$= \mathbb{E}_{\tau \sim P(\cdot;\theta)} \left[\nabla_\theta \log P(\tau;\theta) R(\tau) \right]$$

Much better: Now we can sample trajectories and compute unbiased gradient estimates.

Only the Policy Depends on θ

$$\log P(\tau;\theta) = \log\left(\rho(s_0)\prod_{t=0}^{T-1}\pi_\theta(a_t|s_t)P(s_{t+1}|s_t,a_t)\right)$$

Use log of product: $\log(ab) = \log a + \log b$

$$= \underbrace{\log\rho(s_0)}_{\text{independent}} + \sum_{t=0}^{T-1}\left[\underbrace{\log\pi_\theta(a_t|s_t)}_{\text{depends on }\theta} + \underbrace{\log P(s_{t+1}|s_t,a_t)}_{\text{independent of }\theta}\right]$$

$$\Rightarrow \nabla_\theta \log P(\tau;\theta) = \sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t)$$

Only the policy terms matter for the gradient!

From Objective to Sample-Based Learning

$$J(\theta) = \mathbb{E}_{\tau \sim P(\cdot;\theta)}[R(\tau)] \quad \Rightarrow \quad \nabla_\theta J(\theta) = \mathbb{E}_\tau \left[\nabla_\theta \log \pi_\theta(\tau) R(\tau)\right]$$

Core Steps:

1. Express objective as expectation over trajectories
2. Rewrite gradient of expectation into expectation of gradient
3. Use log-derivative trick to isolate policy terms
4. Simplify trajectory log into per-step policy log terms

Now: we can use Monte Carlo samples to compute unbiased gradient estimates!

Vanilla policy gradient (REINFORCE)

Objective and Gradient (REINFORCE)

Objective: Maximize expected return under stochastic policy π_θ:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} r(s_t, a_t)\right]$$

Gradient (log-derivative trick):

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot R_t\right]$$

- $R_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$ (return from step t)
- Fully on-policy and unbiased
- High variance $\rightarrow$ motivates baseline/advantage refinements

REINFORCE Algorithm

Input: Stochastic policy $\pi_\theta(a|s)$, learning rate α
Loop:

1. Sample a trajectory $\tau = (s_0, a_0, r_0, \ldots, s_T)$ from π_θ
2. Compute return for all time epochs $t \in \{0, \ldots, T\}$:

$$G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$$

3. Gradient ascent update $\forall t \in \{0, \ldots, T\}$:

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot G_t$$

Simple, general, but sample inefficient.

Policy Parametrizations

1. **Discrete actions: Softmax policy**

$$\pi_\theta(a|s) = \frac{\exp(h_\theta(s,a))}{\sum_{a'} \exp(h_\theta(s,a'))}$$

- Works for classification-style action spaces
- Used in scheduling, routing, resource allocation

2. **Continuous actions: Gaussian policy**

$$\pi_\theta(a|s) = \mathcal{N}(a; \mu_\theta(s), \sigma^2_\theta(s))$$

- Works for real-valued action spaces
- Suitable for control, robotics, inventory pricing

Variance Reduction By Baseline

Problem: The REINFORCE gradient estimator has high variance:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot R_t\right]$$

Reduce variance by subtracting baseline:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot (R_t - b(s_t))\right]$$

where $b(s_t)$ is a baseline – e.g., the average observed reward or value function $V(s_t)$ – that does not bias the gradient

Summary: Vanilla Policy Gradient (REINFORCE)

Strengths:

- Unbiased gradient estimator
- Works with any differentiable policy (discrete/continuous)
- Conceptually and computationally simple

Limitations:

- High variance $\rightarrow$ slow convergence
- On-policy $\rightarrow$ poor sample efficiency
- Requires full trajectories

Used as the backbone for:

- Actor-critic architectures
- Contemporary policy gradient methods

Example: Dynamic Pricing with Policy Gradient

Problem: Set prices over time to maximize revenue under demand uncertainty.

- **State:** Time remaining, current inventory, past demand, competitor prices
- **Action:** Price level (discrete or continuous)
- **Reward:** Revenue from accepted price
- **Transitions:** Demand is stochastic function of price

Why Policy Gradient?

- Continuous action space $\rightarrow$ Gaussian policy
- Stochastic demand $\rightarrow$ learn robust pricing strategies
- Explore pricing strategies without needing a value function

Real-world link: Airline ticket pricing, hotel room pricing, ride-hailing surge prices

Example: Inventory Control with Policy Gradient

Problem: Decide how much to order over time to minimize cost under uncertainty.

- **State:** Current stock level, time, pending orders
- **Action:** Order quantity (continuous)
- **Reward:** Negative of ordering cost + holding cost + stockout penalty
- **Transition:** Demand is random, stock evolves deterministically

Deep policy gradient methods

Deep Policy Gradient: Beyond REINFORCE

What changes compared to basic REINFORCE?

- Replace tabular policies with a deep neural network:

$$\pi_\theta(a|s) : \text{neural network with parameters } \theta$$

- Generalize across continuous or high-dimensional state spaces

Training remains the same:

- Collect trajectories using current policy
- Estimate gradient via: $\nabla_\theta J(\theta) \approx \sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) R_t$
- Update θ via SGD/Adam

Still REINFORCE:

- High variance, no bootstrapping
- Still relies on Monte Carlo returns

Discrete Policy (Actor) Network

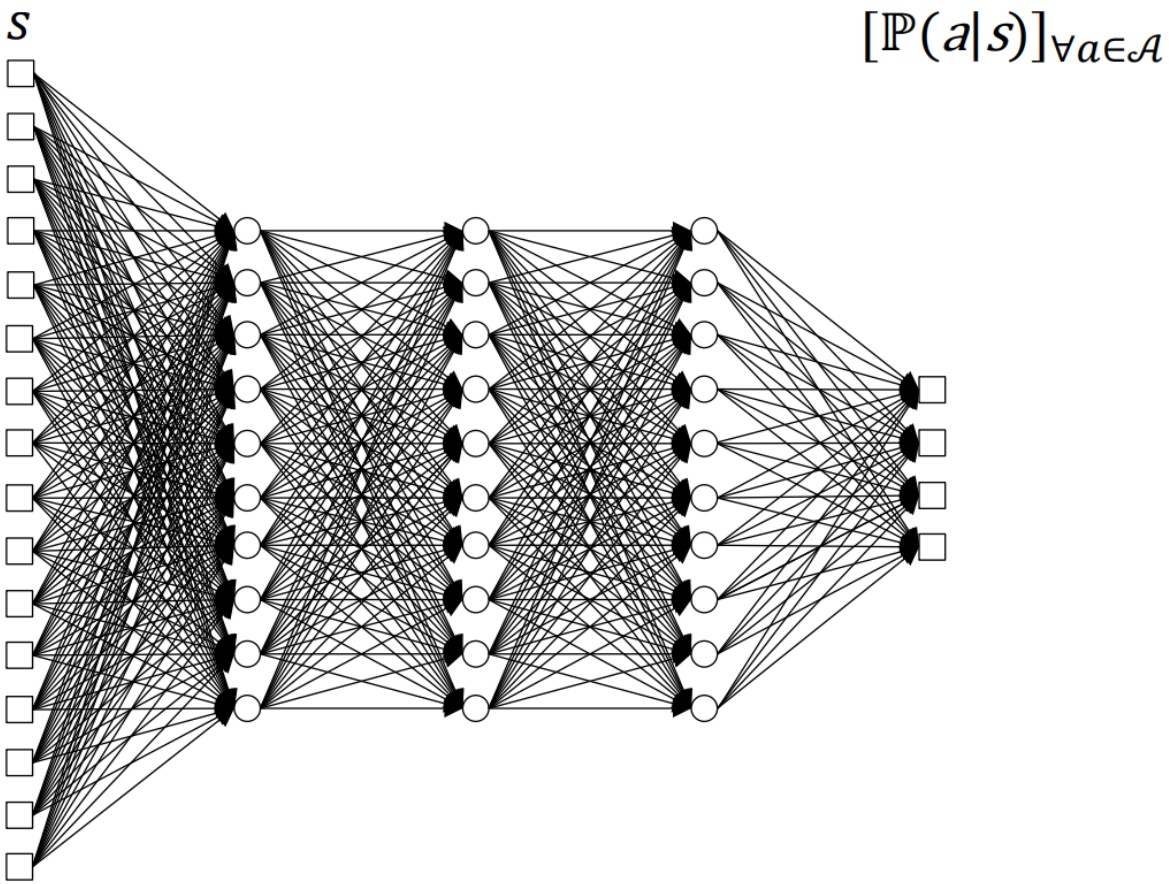

Figure: urlhttps://towardsdatascience.com/deep-policy-gradient-for-cliff-walking-37d5014fd4bc/

Continuous Policy (Actor) Network

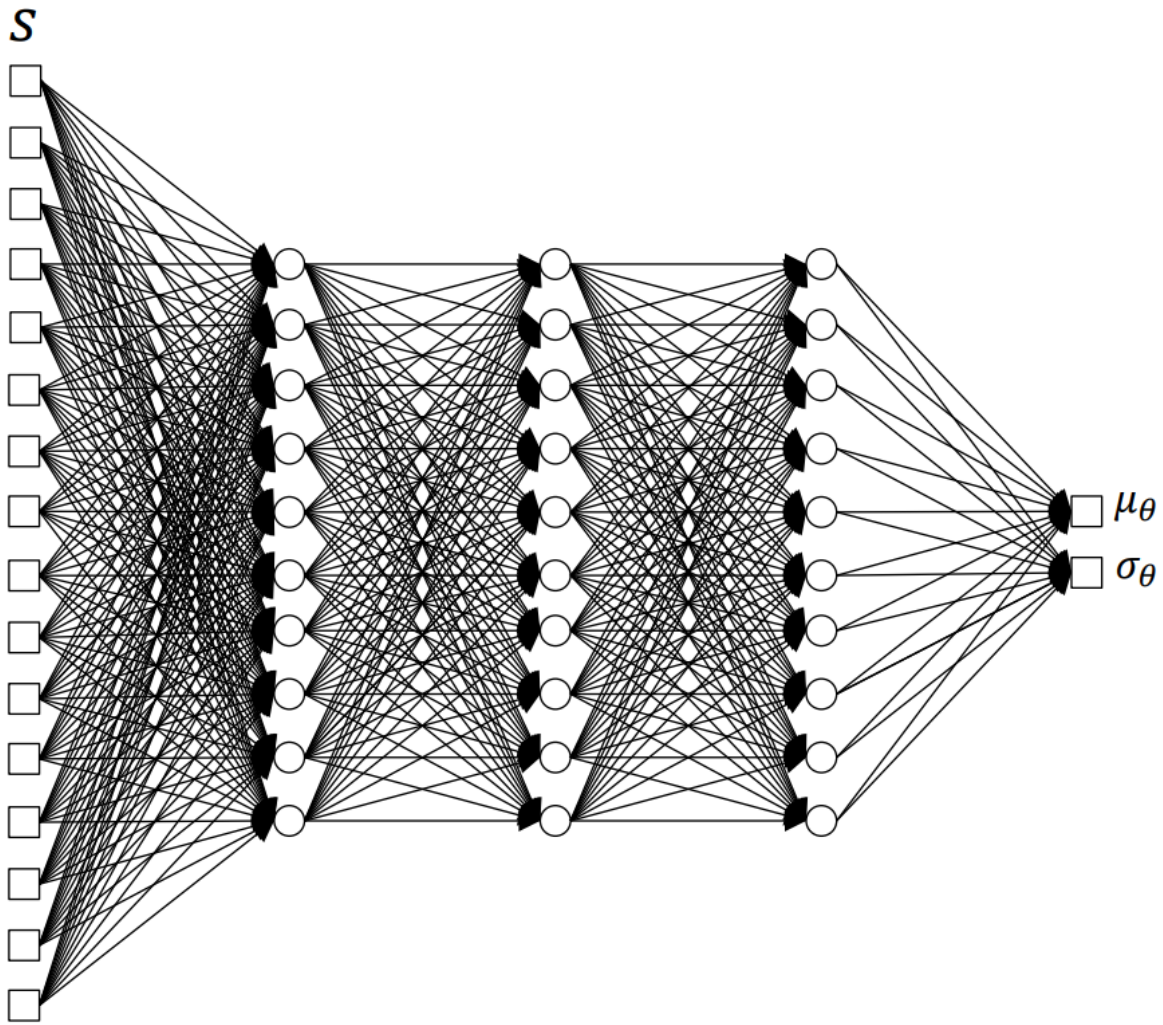

Figure: urlhttps://towardsdatascience.com/deep-policy-gradient-for-cliff-walking-37d5014fd4bc/

Cliff Walking Policy

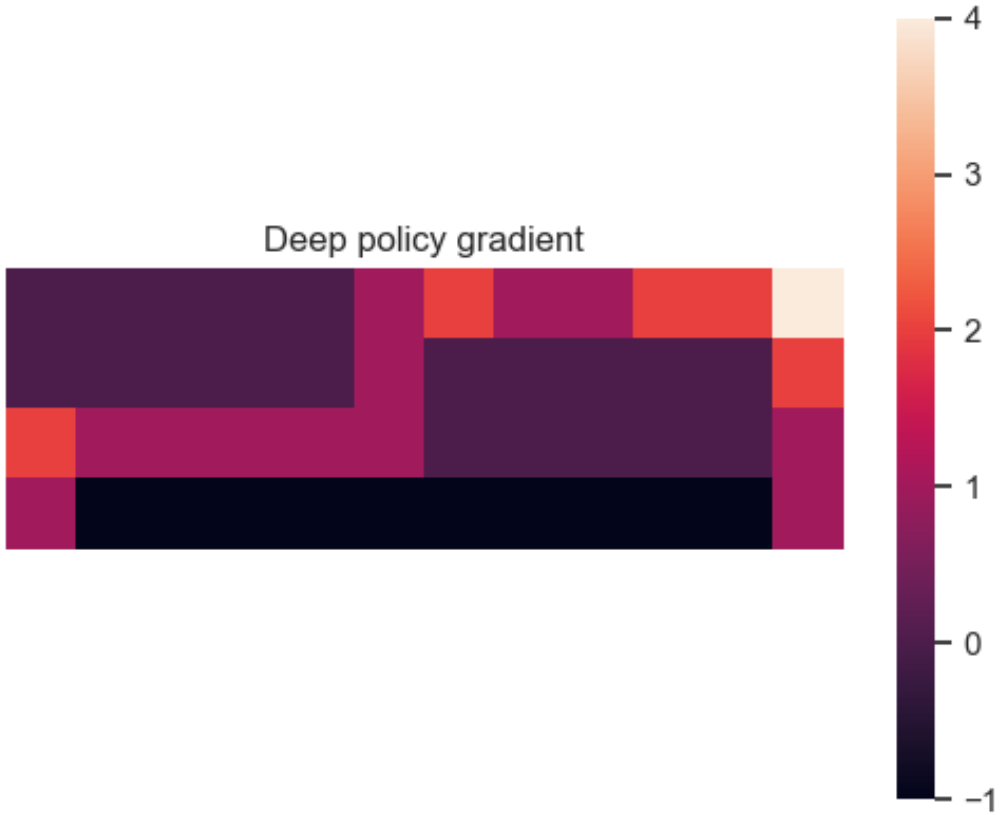

Figure: https://towardsdatascience.com/deep-policy-gradient-for-cliff-walking-37d5014fd4bc/

Actor-Critic Architectures

Actor-Critic Paradigm

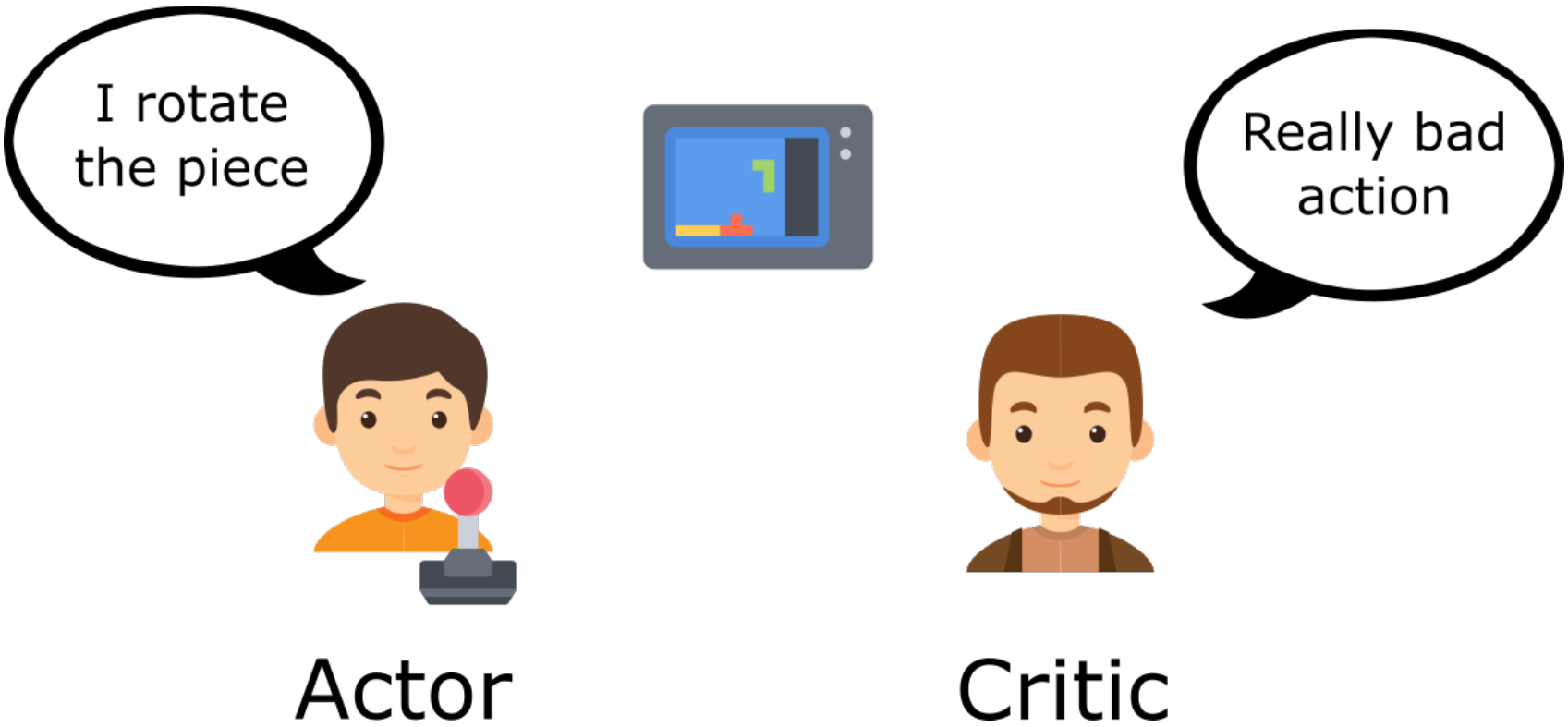

Figure: https://huggingface.co/blog/deep-rl-a2c

Actor–Critic: Learning Policies with Value Baselines

Motivation: Vanilla policy gradient (REINFORCE) suffers from high variance.

Core idea: Learn a value function to act as a baseline and reduce variance of policy gradient estimates.

What this section covers:

- Why REINFORCE is simple but noisy
- How critic networks stabilize learning
- Actor–Critic as a general architecture (not a specific algorithm)

Examples: Actor–Critic, A2C, DDPG, PPO

Actor–Critic Methods

Key idea:

- **Actor:** policy network $\pi_\theta(a|s)$ — decides what to do
- **Critic:** value network $V_w(s)$ or $Q_w(s,a)$ — evaluates the actor's choices

Policy update:

$$\nabla_\theta J(\theta) \approx \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a|s) \cdot \hat{A}(s,a)\right]$$

Critic learns:

- $V(s) \rightarrow$ used to estimate advantage: $A(s,a) = Q(s,a) - V(s)$
- Can be updated via bootstrapping (TD learning)

Advantages:

- Combines low-variance baseline (critic) with direct policy search (actor)
- More sample-efficient than REINFORCE

Actor-Critic Learning

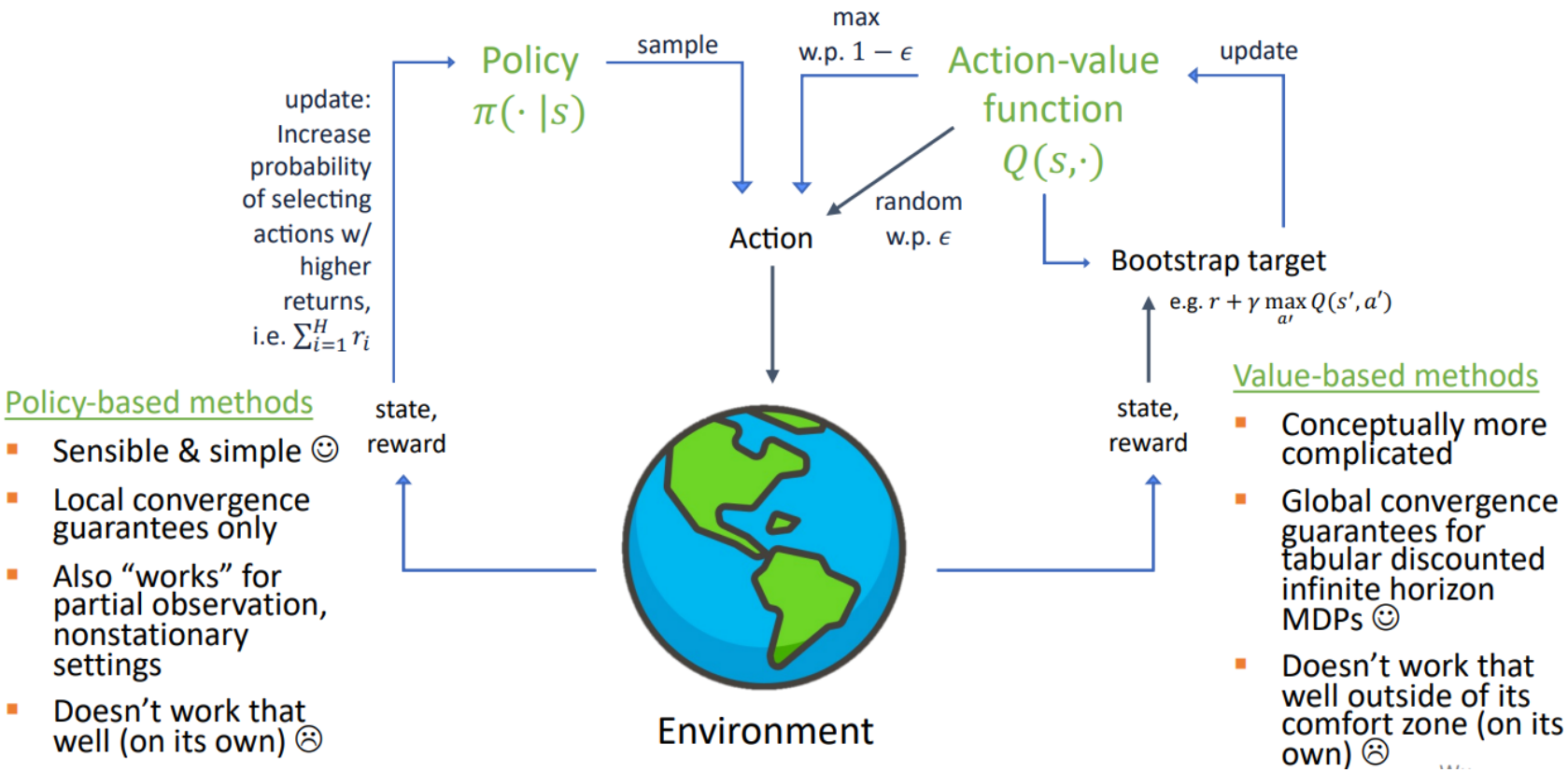

Figure: Wu, C. (2024). Policy gradient. 6.7920: Reinforcement Learning: Foundations and Methods

Advantage Actor–Critic (A2C)

A2C = Specific variant of Actor–Critic:

- On-policy
- Synchronous parallel workers with shared policy

Actor update:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot (R_t - V_\phi(s_t))\right]$$

Critic update:

$$L(\phi) = (V_\phi(s_t) - R_t)^2 \quad \text{(regression to empirical return)}$$

Summary:

- Uses full return R_t as target $\rightarrow$ high variance but unbiased
- Shared architecture across workers ensures stability

Asynchronous Advantage Actor–Critic (A3C)

A3C = Parallel, asynchronous variant of Actor–Critic:

- Multiple agents interact with independent environment copies
- Each agent computes gradients for actor and critic
- Gradients applied asynchronously to shared global parameters

Benefits:

- Diverse trajectories $\rightarrow$ better exploration
- No experience replay needed
- Efficient on CPUs

Drawbacks:

- Asynchronous updates can be unstable
- Requires careful synchronization

Policy Optimization with Trust Region Constraints

Stable Policy Optimization: From NPG to PPO

Motivation: Vanilla gradient updates can lead to large, unstable policy changes.

Core idea: Control the size and direction of policy updates — either geometrically (Fisher) or via constraints (KL or clipping).

What this section covers:

- Natural gradients and policy geometry
- KL-constrained optimization (TRPO)
- First-order approximation via clipping (PPO)

Why Vanilla Gradients May Fail

Vanilla Policy Gradient:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta)$$

Problem: Small changes in θ can cause **large, unstable shifts** in the policy $\pi_\theta(a|s)$

Intuition: In parameter space, the Euclidean step size $\|\theta_{k+1} - \theta_k\|$ doesn't reflect how much the policy changed!

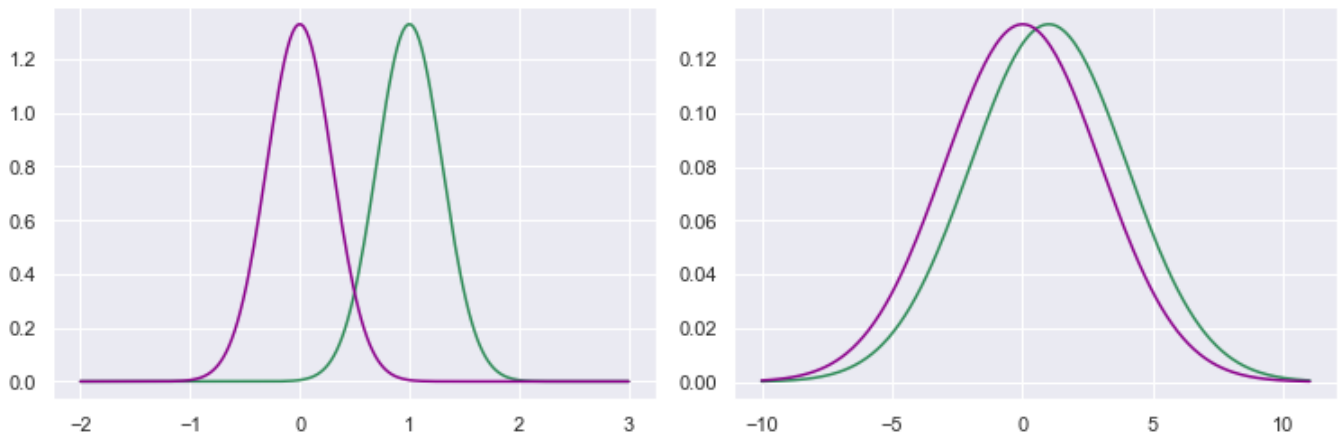

Figure: https://medium.com/data-science/natural-policy-gradients-in-reinforcement-learning-explained-2265864cf43c

From Parameter Space to Policy Space

Problem: Vanilla gradient descent ignores policy geometry → risk of **overshooting** or **inefficient updates**

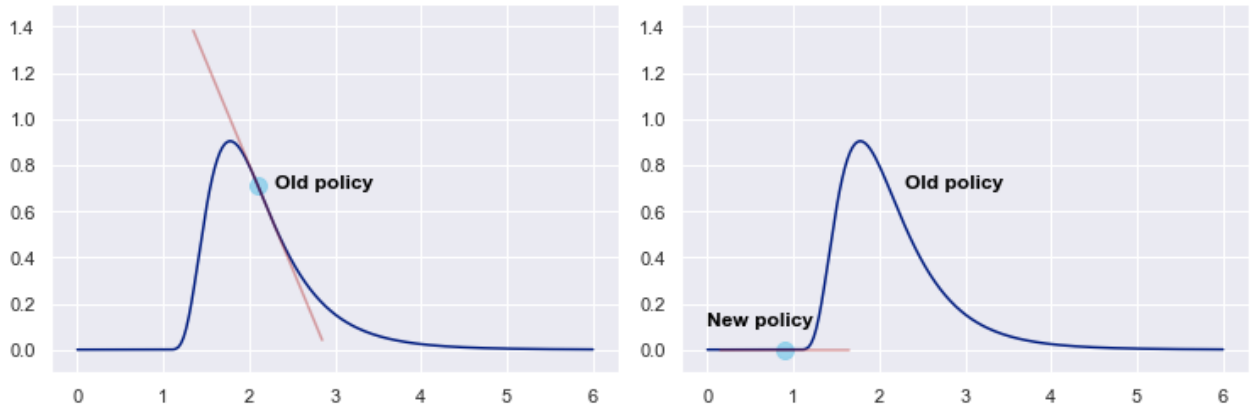

Figure: `https://medium.com/data-science/natural-policy-gradients-in-reinforcement-learning-explained-2265864cf43c`

Key idea: Use KL divergence to measure and control distance between policies:

$$D_{\mathrm{KL}}(\pi_\theta \,||\, \pi_{\theta+\delta})$$

Adjust updates to account for the **local curvature** of policy space

KL Divergence and the Fisher Matrix

Use second-order Taylor approximation:

$$D_{\mathrm{KL}}(\pi_\theta || \pi_{\theta+\delta}) \approx \frac{1}{2}\delta^\top F(\theta)\delta$$

Fisher Information Matrix can be expressed in terms of score function:

$$F(\theta) = \mathbb{E}_{s,a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)^\top\right]$$

Interpretation: Fisher Information Matrix captures local sensitivity of the policy $\rightarrow$ defines the natural metric in policy space.

Natural Gradient: KL-Aware Steepest Ascent

Optimization problem:

$$\max_{\delta} \quad \nabla_\theta J(\theta)^\top \delta \quad \text{s.t.} \quad D_{\mathrm{KL}}(\pi_\theta || \pi_{\theta+\delta}) \leq \epsilon$$

Using second-order KL approximation:

$$\Rightarrow \quad \max_{\delta} \nabla_\theta J^\top \delta \quad \text{s.t.} \quad \frac{1}{2}\delta^\top F(\theta)\delta \leq \epsilon$$

Solution:

$$\delta^\star = \alpha F(\theta)^{-1} \nabla_\theta J(\theta)$$

This is the **natural policy gradient**.

Natural Policy Gradient: Summary and Update Rule

Natural Gradient Direction:

$$\tilde{\nabla}_\theta J(\theta) = F(\theta)^{-1} \nabla_\theta J(\theta)$$

Policy Update Rule:

$$\theta_{k+1} = \theta_k + \alpha \tilde{\nabla}_\theta J(\theta)$$

Why it matters:

- Invariant to policy reparameterization
- Respects geometry of policy space
- Enables stable and efficient learning

Used in TRPO; approximated in PPO.

Limitations of Natural Policy Gradient

Shortcomings of Natural Policy Gradient:

1. **Approximate KL control:** Taylor expansion is only local $\rightarrow$ may violate actual KL trust region.
2. **No improvement guarantee:** We do not *check* if the policy is better — we just *assume* the step is good.
3. **Computational burden:** Inverting the Fisher matrix is expensive: $\mathcal{O}(n^3)$

Conclusion: We need an optimization procedure that:

- Enforces the KL constraint more reliably
- Actually checks monotonic policy improvement
- Avoids direct inversion of F

$\Rightarrow$ **Trust Region Policy Optimization (TRPO)**

TRPO: Trust Region Policy Optimization

Trust Region Idea: Constrain updates in **policy space**:

$$\max_{\theta_{k+1}} \quad \mathbb{E}_{s,a\sim\pi_{\theta_k}} \left[\frac{\pi_{\theta_{k+1}}(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a) \right] \quad \text{s.t.}$$

$$\mathbb{E}_{s\sim\mathcal{D}^{\pi_{\theta_k}}} \left[D_{KL}(\pi_{\theta_k}(\cdot|s) \| \pi_{\theta_{k+1}}(\cdot|s)) \right] \leq \delta$$

Why this works:

- Avoids explicit matrix inversion using **conjugate gradient**
- Ensures **monotonic improvement** via policy improvement bound
- KL constraint prevents overstepping $\rightarrow$ **stable learning**

TRPO: Conjugate Gradient

The **conjugate gradient method** solves $Fx = g$, where $g = \nabla_\theta J$ and F is the Fisher information matrix.

- Note that we don't benefit from knowing F^{-1} itself, it is just a step to obtain the natural gradient
- Iterative method that approximates the solution without computing or inverting F explicitly.
- Efficient: uses Hessian-vector products instead of full matrix calculations.
- Returns an approximate natural gradient direction: $x \approx F^{-1}g$.

TRPO Line Search: Surrogate Objective & Importance Sampling

Key idea: Before accepting a policy update, TRPO verifies it improves the surrogate objective.

Surrogate objective (using importance sampling):

$$\hat{J}(\theta_{k+1}) = \mathbb{E}_{s,a\sim\pi_{\theta_k}}\left[\frac{\pi_{\theta_{k+1}}(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a)\right]$$

Here, the importance sampling ratio

$$\frac{\pi_{\theta_{k+1}}(a|s)}{\pi_{\theta_k}(a|s)}$$

reweighs the samples collected under the old policy π_{θ_k} to estimate new policy's $\pi_{\theta_{k+1}}$ performance.

TRPO Line Search: Procedure

Line search procedure:

- Compute $\hat{J}(\theta_{k+1})$ for candidate update θ_{k+1} using reweighted samples
- Check if $\hat{J}(\theta_{k+1}) > \hat{J}(\theta_k)$ (improvement)
- Verify KL constraint:

$$\mathbb{E}_s \left[D_{KL} \left(\pi_{\theta_k}(\cdot|s) \| \pi_{\theta_{k+1}}(\cdot|s) \right) \right] \leq \delta$$

- If either fails, reduce step size by shrinking the step scaling factor α

TRPO Line Search: Update Rule

Explicit line search update formula:

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^\top H_k \hat{x}_k}}\, \hat{x}_k,$$

where:

- $\hat{x}_k$ is the conjugate gradient direction (approximate natural gradient)
- H_k is the Hessian of the KL (Fisher Information Matrix)
- $j = 0, 1, 2, \ldots$ indexes the line search backtracking steps
- α is typically set to 0.5 (step size halving)

Result: Guarantees monotonic improvement while respecting the trust region.

Trust Region Visualized

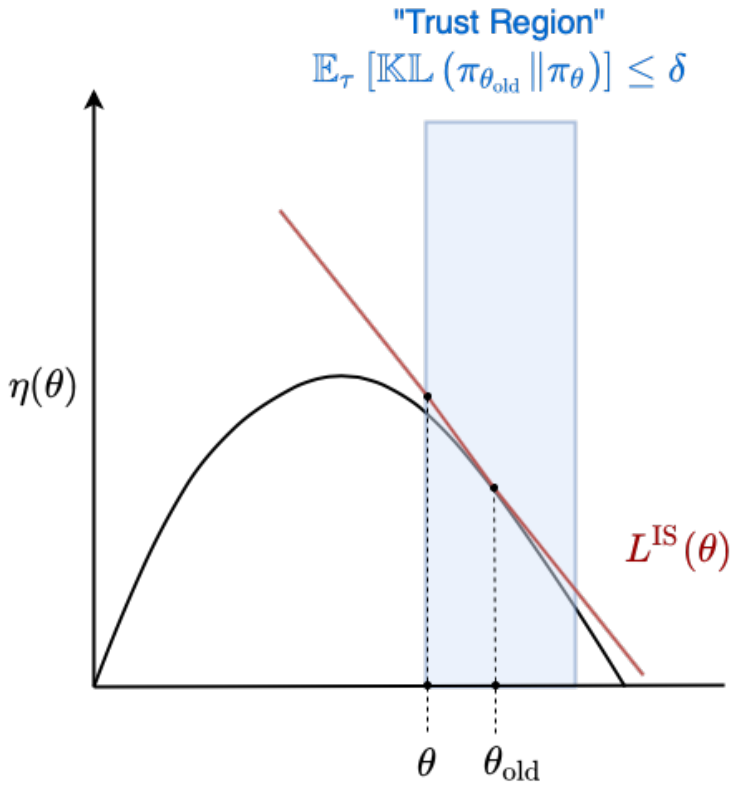

Figure: `https://avandekleut.github.io/ppo/`

Insight: As long as π^{k+1} stays in the trust region, we are guaranteed a lower bound on improvement.

From TRPO to PPO: Practical Challenges

Why not just use TRPO all the time?

1. **Too slow for deep networks:**
   - Even with conjugate gradient, second-order methods are expensive.
   - Fisher matrix is large: $|\theta| \times |\theta|$
2. **Incompatible with first-order optimizers:**
   - Can't use fast optimizers like `Adam` or `RMSprop`
3. **Hard to implement and debug:**
   - Conjugate gradient + KL constraint + line search = complex pipeline

Key Question: Can we retain the benefits of TRPO — **stability** and **KL control** — while using only first-order updates?

$\Rightarrow$ **Proximal Policy Optimization (PPO)**

Proximal Policy Optimization (PPO)

Goal: Stable, efficient policy updates without second-order methods (like TRPO)

Recall TRPO's constraint:

$$\max_{\theta_{k+1}} \quad \mathbb{E}_{s,a\sim\pi_{\theta_k}} \left[\frac{\pi_{\theta_{k+1}}(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a) \right] \quad \text{s.t.}$$

$$\mathbb{E}_{s\sim\mathcal{D}^{\pi_{\theta_k}}} \left[D_{KL}(\pi_{\theta_k}(\cdot|s) \| \pi_{\theta_{k+1}}(\cdot|s)) \right] \leq \delta$$

PPO idea: Replace hard KL constraint with a *clipped surrogate objective* to keep updates "proximal" (i.e., not too large)
Benefit: No second-order optimization, simple implementation, competitive performance

PPO Overview: What's Going On?

PPO components:

- **Semi-on-policy:** Reuse old data within an update batch, but only briefly (no full replay buffer)
- **Surrogate objective:** encourages policy improvement via advantage-weighted likelihood ratio
- **Clipping:** prevents large, destabilizing updates to the policy
- **Advantage estimation:** uses Generalized Advantage Estimation to reduce variance of policy updates
- **Mini-batching:** reuses data over multiple epochs to improve sample efficiency

Idea: Combine the best parts of:

- TRPO: trust region stability
- REINFORCE: simplicity and scalability
- GAE: low-variance advantage estimation

Importance Sampling in PPO (continued)

From TRPO to PPO:

- TRPO uses **importance sampling** to evaluate the new policy $\pi_{\theta_{k+1}}$ under trajectories collected from the old policy π_{θ_k}, ensuring valid off-policy evaluation:

$$\mathbb{E}_{\tau \sim \pi_{\theta_k}} \left[\frac{\pi_{\theta_{k+1}}(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a) \right]$$

- PPO reuses the same ratio $r_t(\theta_{k+1}) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$ to track policy drift across **multiple gradient steps**—not just one. Same concept, but even more important due to potentially larger drift from original policy.

Clipped Surrogate Objective

Clipped objective:

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

Intuition:

- $r_t(\theta_{k+1}) = \frac{\pi_{\theta_{k+1}}(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$ is the importance weight comparing new and old policies.
- Objective encourages improvement only within a small trust region.
- If $r_t(\theta)$ deviates too far from 1, surrogate is clipped $\rightarrow$ discourages overly large updates.

Why the min operator in PPO?

The min operator in $\mathbb{E}_t\left[\min\left(r_t A_t,\ \text{clip}(r_t, 1-\epsilon, 1+\epsilon)\, A_t\right)\right]$ ensures a pessimistic bound by blocking two harmful update types:

- **Large increase of bad actions (no reward):** If $A_t < 0$ but $r_t > 1+\epsilon$, the policy is boosting a bad action too much. The clipped term $(1+\epsilon)A_t$ is *lower* (more negative) than $r_t A_t$, so min picks it, $\Rightarrow$ zero gradient.
- **Large decrease of good actions (missed opportunity):** If $A_t > 0$ but $r_t < 1-\epsilon$, the policy is cutting a good action's probability too much. The clipped term $(1-\epsilon)A_t$ is *lower* than $r_t A_t$, so min picks it, $\Rightarrow$ zero gradient.

Result: All other updates (aligned with improving the policy) retain their gradient, while only truly harmful, overly large shifts are halted—creating a soft trust region.

PPO Clipping Per Case

Ratio	A_t	Value of min	Clipped?	Sign	Gradient
$1-\epsilon \le r_t(\theta) \le 1+\epsilon$	$+$	$r_t(\theta)A_t$	No	$+$	✓
$1-\epsilon \le r_t(\theta) \le 1+\epsilon$	$-$	$r_t(\theta)A_t$	No	$-$	✓
$r_t(\theta) < 1-\epsilon$	$+$	$r_t(\theta)A_t$	No	$+$	✓
$r_t(\theta) < 1-\epsilon$	$-$	$(1-\epsilon)A_t$	Yes	$-$	×
$r_t(\theta) > 1+\epsilon$	$+$	$(1+\epsilon)A_t$	Yes	$+$	×
$r_t(\theta) > 1+\epsilon$	$-$	$r_t(\theta)A_t$	No	$-$	✓

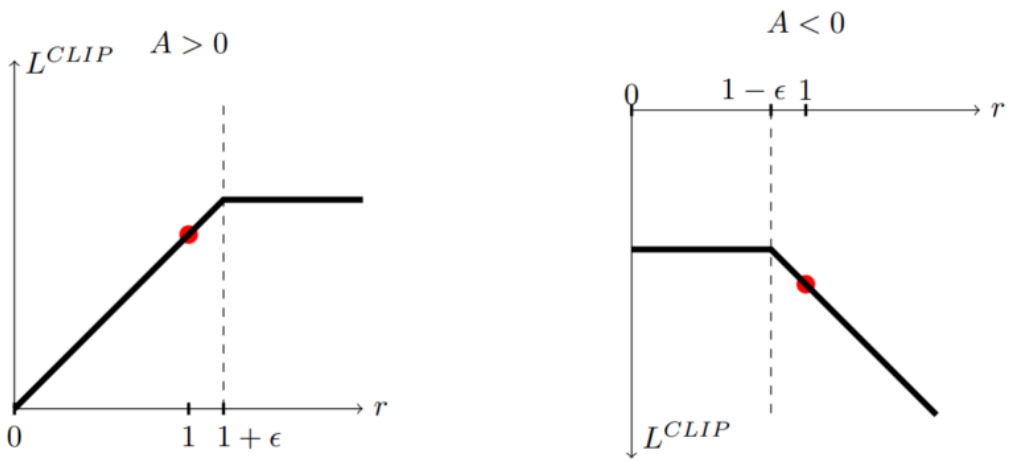

Figure: Schulman et al. (2017). Proximal Policy Optimization Algorithms

Stable Advantage Estimation for PPO (GAE)

Why we use GAE:

- Policy gradient updates depend on accurate advantage estimates
- Remind: Monte Carlo = high variance, TD = high bias

Generalized Advantage Estimation (GAE):

$$A_t^{\text{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \quad \text{with} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

Tradeoff: Smooths estimates $\rightarrow$ better learning signal

In PPO: A_t from GAE is used in the clipped objective:

$$\min\left(r_t(\theta)A_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)$$

Mini-batching and Optimization in PPO

PPO is trained via multiple epochs on collected data:

- PPO reuses the same batch of trajectories for multiple epochs $\rightarrow$ corrects for policy drift using importance sampling
- Collect T timesteps via current policy π_{θ_k}
- Compute advantages (e.g., using GAE)
- Optimize the PPO surrogate objective:

$$\theta \leftarrow \arg\max_{\theta} \; J^{\text{CLIP}}(\theta)$$

  over multiple epochs using first-order optimizer like Adam
- Split data into **mini-batches** for stable updates

Typical hyperparameters:

- Epochs per batch: 3–10
- Minibatch size: 64–512
- Timesteps per batch: 2048–8192

PPO Summary: Why It Works

- First-order, efficient, easy to implement
- Stable updates via clipped surrogate objective
- Uses GAE for variance reduction
- Strong empirical performance across environments

Limitations:

- Still on-policy $\rightarrow$ sample inefficient
- Sensitive to hyperparameters (batch size, clip range, etc.)
- Can fail in long-horizon credit assignment tasks

Differentiable Control for Continuous Actions

Stochastic PG vs Deterministic PG

Stochastic policy gradient (PG) methods

- Estimate $\nabla_\theta J$ via sampling from $\pi_\theta(a|s)$.
- High variance; require on-policy sampling.

Deterministic policy gradient (DPG) algorithms

- Differentiate *through* the critic to update the actor directly:

$$\nabla_w J(w) = \mathbb{E}_{s\sim D}\Big[\nabla_a Q_\psi(s,a)\big|_{a=\varphi_w(s)} \nabla_w \varphi_w(s)\Big].$$

- Off-policy, low variance.
- DPG family: DDPG, TD3, SAC (stochastic policy, but uses reparameterization).

DPG algorithms commonly require the critic to take actions as input, originate in continuous action spaces, and are *off-policy*.

Differentiable Control for Continuous Actions

Challenge: Standard stochastic PG (e.g., REINFORCE) struggles in continuous action spaces:

- High sampling variance
- Non-differentiable sampling step
- On-policy, so no reuse of past experiences

This block explores:

- Reparameterized policies for pathwise gradients
- Deterministic policy gradients for direct control (DDPG, TD3)
- Entropy-regularized policies for robust exploration (SAC)

Theme: Enable end-to-end differentiable optimization over continuous actions.

Deep Deterministic Policy Gradient (DDPG)

Goal: Actor–Critic method for continuous action spaces

Key ideas:

- Deterministic policy: $\mu(s;\theta^\mu) \rightarrow$ outputs action a directly
- Critic: $Q(s,a;\theta^Q)$ learns to evaluate (s,a)
- Uses target networks and replay buffer

Updates:

- Critic: minimize TD error

$$y = r + \gamma Q'(s', \mu'(s')) \quad \text{(target networks)}$$

- Actor: policy gradient via chain rule:

$$\nabla_{\theta^\mu} J \approx \mathbb{E}_s \left[\nabla_a Q(s,a)\big|_{a=\mu(s)} \nabla_{\theta^\mu} \mu(s) \right]$$

Issue: Prone to overestimation, instability

DDPG Architecture

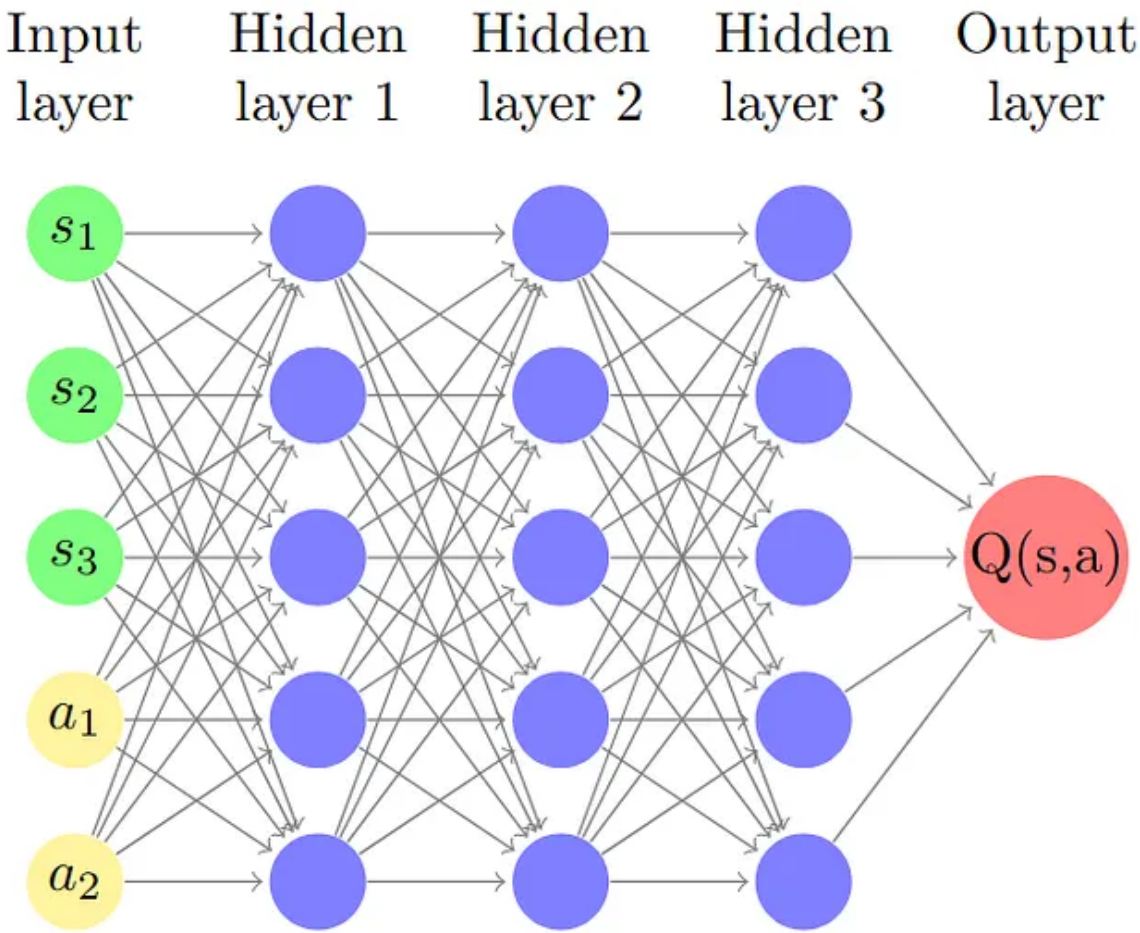

Figure:
https://medium.com/data-science/deep-deterministic-policy-gradients-explained-4643c1f71b2e

TD3: Twin Delayed Deep Deterministic Policy Gradient

Fixes to DDPG's instability:

- **Clipped Double Q-learning:** Two critics Q_1, Q_2

$$y = r + \gamma \min_{i=1,2} Q_i'(s', \mu'(s'))$$

- **Target smoothing:** Add noise to target action:

$$a' = \mu'(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \sigma), -c, c)$$

- **Delayed actor update:** Update policy less frequently than critic

Effect: Improved stability, more accurate value estimates, reduced overfitting

From Deterministic to Stochastic Policies

- **TD3 uses deterministic policies with added noise** for exploration.
- Fixed noise can be **suboptimal** for consistent, adaptive exploration.
- **Stochastic policies explicitly model action distributions**, enabling flexible and state-dependent exploration.
- However, gradient estimation for stochastic policies often relies on **likelihood ratio methods** (e.g., REINFORCE).

Next: Why likelihood ratio methods struggle with variance and gradient flow.

REINFORCE Gradient: Intuition and Limitations

$$\nabla_\theta J = \mathbb{E}_{s,a}\left[Q_\psi(s,a)\,\nabla_\theta \log \pi_\theta(a|s)\right]$$

- Uses likelihood ratios to estimate gradient without differentiating through actions.
- Cannot exploit critic gradients w.r.t. actions ($\nabla_a Q$).
- Gradient only "sees" rewards weighted by log-probability.

Next: Why this leads to high variance and sample inefficiency.

Challenges in Practice: Variance and Sample Efficiency

- **High variance** because $\log \pi_\theta(a|s)$ can explode for low-probability actions.
- Sampling breaks gradient flow (data point is not differentiable) $\rightarrow$ cannot backpropagate through critic $\rightarrow$ noisy updates.
- On-policy learning $\rightarrow$ requires fresh samples every update $\rightarrow$ sample inefficient.
- **SAC tackles these with reparameterization and off-policy learning.**

Reparameterization: Deterministic Sampling

Key idea: Replace stochastic draw $a \sim \pi_\theta(\cdot|s)$ with

$$a = g_\theta(\epsilon, s), \quad \epsilon \sim p(\epsilon)$$

—for example, Gaussian: $a = \mu_\theta(s) + \sigma_\theta(s)\,\epsilon$.

Now ϵ carries all randomness, and g_θ is differentiable.

Pathwise gradient:

$$\nabla_\theta \mathbb{E}_\epsilon\big[f(g_\theta(\epsilon, s))\big] = \mathbb{E}_\epsilon\big[\nabla_a f(a)\,\nabla_\theta g_\theta(\epsilon, s)\big].$$

- Leverages $\nabla_a f$ (e.g. critic's slope).
- **No** $\log \pi$ **term** $\rightarrow$ much lower variance.
- Turns sampling into a local deterministic sensitivity analysis.

From Pathwise Gradients to SAC

Actor–critic objective with reparameterization:

$$J(\theta) = \mathbb{E}_{s,\epsilon}\big[Q_\psi\big(s,\, g_\theta(\epsilon, s)\big)\big]$$

$$\nabla_\theta J = \mathbb{E}_{s,\epsilon}\big[\nabla_a Q_\psi(s,a)\, \nabla_\theta g_\theta(\epsilon, s)\big].$$

Payoff: A direct, low-variance update telling the actor "which way" in action space to improve Q.

Reparameterization unifies policy gradient estimation; DDPG/TD3 correspond to deterministic policies (zero noise limit), SAC uses stochastic policies with entropy regularization.

Entropy Bonus: Encouraging Exploration

Motivation: Avoid premature convergence to deterministic policies and encourage **exploration** by keeping policy "soft".

Entropy of policy:

$$\mathcal{H}(\pi_\theta(\cdot|s)) = -\mathbb{E}_{a\sim\pi_\theta}\left[\log \pi_\theta(a|s)\right]$$

Entropy-regularized objective:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t \left(r_t + \alpha\mathcal{H}(\pi_\theta(\cdot|s_t))\right)\right]$$

- α controls exploration-exploitation tradeoff
- Larger $\alpha \rightarrow$ more stochastic policies (more exploration)
- Integral part of SAC and related methods

Soft Actor-Critic (SAC): Overview

Key idea: Learn a stochastic policy that maximizes expected return **plus entropy**, encouraging exploration.

Objective (policy-level):

$$\pi^* = \arg\max_{\pi} \sum_t \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} \left[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)) \right]$$

Main components:

- **Actor:** Stochastic Gaussian policy $a = f_\theta(s, \epsilon), \epsilon \sim \mathcal{N}(0, 1)$
- **Critic(s):** Two Q-functions Q_{ϕ_1}, Q_{ϕ_2}
- **Entropy coefficient** α**:** Automatically tuned to balance exploitation vs exploration

Off-policy: Learns from replay buffer (like DQN/DDPG), improving sample efficiency over PPO/TRPO

SAC: Critic Loss (Q-function)

Minimize Bellman residual using entropy-regularized target:

$$\mathcal{L}_Q(\phi) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left(Q_\phi(s,a) - \left(r + \gamma \mathbb{E}_{a'\sim\pi_\theta}\left[Q_{\bar{\phi}}(s',a') - \alpha \log \pi_\theta(a'|s')\right]\right)^2$$

- $\mathcal{D}$: Replay buffer
- $\bar{\phi}$: Target network (slow moving average of ϕ)
- α: Entropy coefficient
- Trains the Q-network to predict soft (entropy-regularized) future returns.

SAC: Policy Loss (Actor)

Maximize expected return and entropy:

$$J_\pi(\theta) = \mathbb{E}_{s\sim\mathcal{D},\epsilon\sim\mathcal{N}}\left[Q_\phi(s,a) - \alpha \log \pi_\theta(a|s)\right], \quad \text{with } a = f_\theta(s,\epsilon)$$

Train via gradient ascent (minimize loss):

$$\mathcal{L}_\pi(\theta) = \mathbb{E}_{s\sim\mathcal{D},\epsilon\sim\mathcal{N}}\left[\alpha \log \pi_\theta(f_\theta(s,\epsilon)|s) - Q_\phi(s, f_\theta(s,\epsilon))\right]$$

- $f_\theta(s,\epsilon)$: Reparameterized action
- Enables low-variance gradients via backprop through f
- Optimizes the policy to choose actions that are both rewarding and diverse.

SAC: Entropy Temperature Loss

Loss: Minimize deviation from target entropy $\mathcal{H}_{\text{target}}$ by adjusting α.

$$\mathcal{L}_\alpha = \mathbb{E}_{a\sim\pi_\theta}\left[\alpha\left(-\log \pi_\theta(a|s) - \mathcal{H}_{\text{target}}\right)\right]$$

- α is a learnable parameter
- Trained via **gradient descent** on $\mathcal{L}_\alpha$
- Increases α if entropy is too low, decreases if too high
- Enables automatic control of policy stochasticity

SAC: Implementation Tricks and Stability Mechanisms

Key implementation tricks:

- **Twin Q-networks:** Use Q_{ϕ_1}, Q_{ϕ_2}, and take minimum to reduce overestimation bias (as in TD3)
- **Target networks:** Soft updates through Polyak averaging:

$$\bar{\phi}_j \leftarrow \tau\phi_j + (1-\tau)\bar{\phi}_j \quad \text{with small } \tau \ll 1$$

- **Reparameterization trick:** Use $a = f_\theta(s, \epsilon)$ with $\epsilon \sim \mathcal{N}(0, 1)$ to backpropagate through stochasticity
- **Entropy coefficient tuning:** Adapt α to maintain a target entropy $\bar{\mathcal{H}}$ (e.g., based on action space dimensionality)
- **Off-policy updates:** Leverage replay buffer $\mathcal{D}$ for sample reuse and training stability

Which Policy Gradient Algorithm to Use?

Algorithm	**Use case**	**Why / Trade-offs**
REINFORCE	Toy problems, teaching	Simple, but very high variance
Actor–Critic	Small/medium tasks	Lower variance, still sample-inefficient
TRPO / PPO	Robotics, high-dim control	Stable, robust, but data-hungry (on-policy)
DDPG	Continuous control, low noise	Efficient, but brittle and poor exploration
SAC	Complex continuous control (robotics, finance, games)	Efficient, stable, good exploration (entropy)

Take-away: On-policy = stable but data-hungry. Off-policy = sample efficient, often better for real-world.

Wrapping up

Lecture 2 Summary: Advanced Introduction to Policy Gradients

What we covered:

- Introduced policy-based reinforcement learning and stochastic policies, contrasted policy- and value-based learning
- Derived policy gradient theorem and outlined vanilla policy gradient (REINFORCE)
- Introduced advanced policy gradient methods: actor-critic, natural gradient, PPO, DDPG, SAC

Takeaway: Policy-based RL directly adjusts policies by tuning a parameterized stochastic policy. It forms the foundation for actor-critic models.

Contact details

Wouter van Heeswijk
Assistant Professor
Operations Research & Financial Engineering
University of Twente
w.j.a.vanheeswijk@utwente.nl
+31 53 489 8460