UVA | SCHOOL of DATA SCIENCE

Extensions to Policy Gradient

Reinforcement Learning
School of Data Science
University of Virginia

Last updated: March 18, 2025

Agenda

> Recap of REINFORCE

> Actor-Critic Methods

> Trust Region Methods

> Deep Deterministic Policy Gradient (DDPG)

Recap of REINFORCE

REINFORCE

This was our first policy gradient (PG) method

All PG methods are on-policy algorithms

REINFORCE is a Monte Carlo Gradient algorithm (uses full trajectories)

It works by:
1) simulating paths
2) calculating returns G from each path (evaluate)
3) taking update steps based on return and gradient of policy (improve)

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta}), \forall a \in \mathcal{A}, s \in \mathcal{S}, \boldsymbol{\theta} \in \mathbb{R}^n$
Initialize policy weights $\boldsymbol{\theta}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G_t \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t, \boldsymbol{\theta})$

REINFORCE Intuition

As we generate sample paths, we calculate return G over the paths.

The update step will increase parameter vector in direction of:
- greater return
- greater increase of probability repeating action A_t on future visits to state S_t

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta}), \forall a \in \mathcal{A}, s \in \mathcal{S}, \boldsymbol{\theta} \in \mathbb{R}^n$
Initialize policy weights $\boldsymbol{\theta}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G_t \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t,\boldsymbol{\theta})$

REINFORCE Limitations

REINFORCE can have high variance and converge slowly
From any sampled *(s,a)* there may be many very different return estimates

We are estimating the gradient of the performance measure $\nabla \hat{J}(\theta_t)$

Can be noisy due to factors including:

> Noise in environment

> Sparse rewards

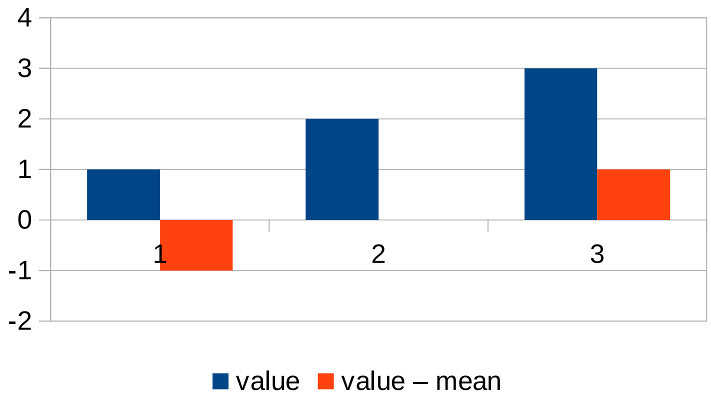
> Length of sample trajectories can vary

REINFORCE with Baseline

REINFORCE can have high variance and converge slowly

We want to take positive gradient steps in direction of parameters that lead to high-reward trajectories
We want to take negative steps for low-reward trajectories

Rather than working w rewards, we can work with relative rewards



REINFORCE with Baseline, contd.

Positive gradient steps in direction of parameters that lead to high ***relative*** reward trajectories

Negative steps for low ***relative*** reward trajectories

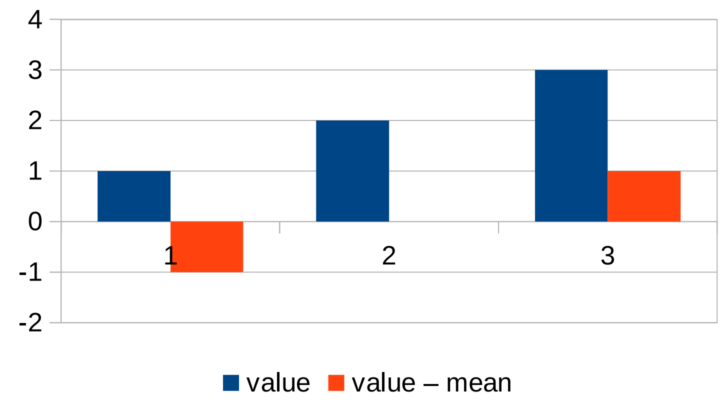
Q: What makes sense for relative reward?

One solution: subtract the average trajectory reward

This is actually the state value function V(*s*)

By subtracting off a baseline, it reduces variance

We will still have an unbiased estimate for gradient

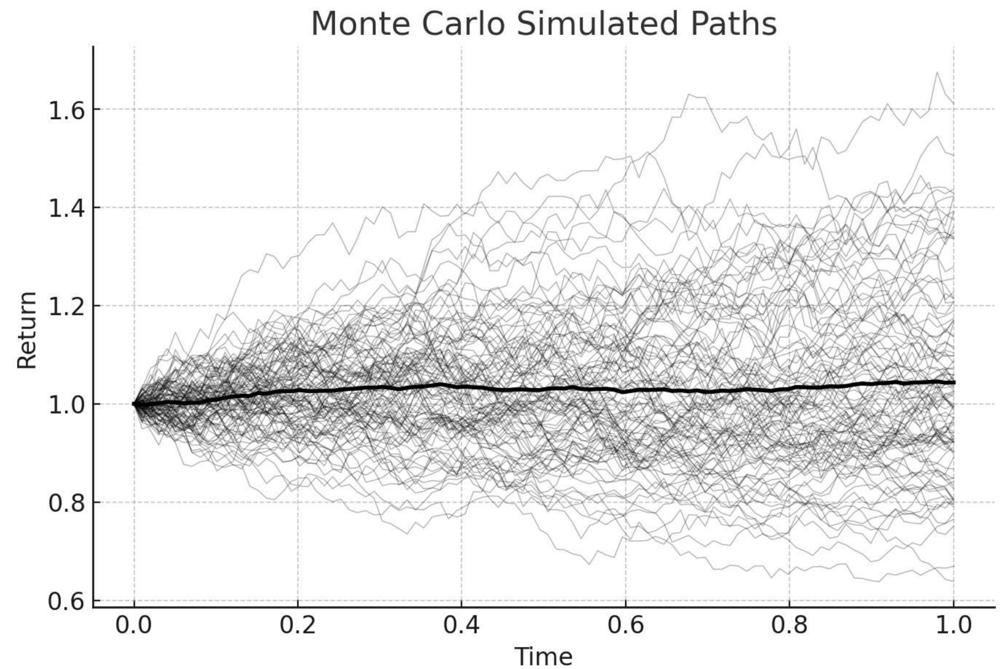


REINFORCE with Baseline, contd.

Plot showing many simulated paths. A gain can be computed for each path.

The dark curve shows average trajectory values. The gain can be computed and it represents V(s).

This can be treated as baseline.



REINFORCE with Baseline Algorithm

REINFORCE can have high variance and converge slowly

One solution: subtract baseline, which can be state value function

REINFORCE with Baseline (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} R_k$ $\quad (G_t)$
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$ **Value function as baseline**
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \gamma^t \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Actor-Critic Methods

Introducing the Critic

The state value function estimate is based on first state of each transition (time *t*)

$$\delta \leftarrow G - \hat{v}(S_t, \mathbf{w})$$

Another option is to use state value estimates from multiple time steps $\gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$

Given two time steps, we can estimate a one-step return

We can collect many trajectories, calculate the returns, and train a neural network

By using this model, it reduces the variance from the samples

It can be used to evaluate the policy, and is called the *critic*

The Actor

The policy network is called the *actor*

It guides the behavior of the agent

The actor uses a separate set of parameters

Actor-Critic

We now have two different parametrized functions:

> Policy function (*actor*)

> State-value function (*critic*). Evaluates the policy.

For each function the params are updated on each pass

Can use neural networks for the functions, given sufficient training data

Actor-Critic Algorithm

There are two parametrized functions: one for policy (*actor*), one for state value (*critic*)

For each function the params are updated on each pass

One-step Actor–Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} I \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Estimate of one-step return

$$\gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$$

Reappearance of the Advantage Function

Recall the advantage function which measures the benefit of a particular action:

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

This appears in the critic term (line 4 circled at right)

While S is not terminal:
$A \sim \pi(\cdot|S,\boldsymbol{\theta})$
Take action A, observe S', R
$\delta \leftarrow \boxed{R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})}$
$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} I \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
$I \leftarrow \gamma I$
$S \leftarrow S'$

The Q-value estimate can be calculated from single step transitions.
We have done this with TD learning.

When estimating the advantage this way, the approach is sometimes called *Advantage Actor Critic* or A2C

Trust Region Methods

Trust Region Policy Optimization (TRPO)

Source: OpenAI Spinning Up

TRPO methods have shown a large improvement over Actor Critic methods

TRPO updates policies by taking largest possible step to improve performance
However, there is a constraint: new policy must not differ “too much” from old policy

Constraint is formalized using KL-divergence: distance between prob distributions

K-L Divergence

Given probability distributions *P* and *Q* defined on sample space $\mathcal{X}$,

KL-divergence measures log-difference between *P* and *Q*, where expectation is measured relative to *P*

Discrete case:

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right),$$

Continuous case:

$$D_{\mathrm{KL}}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx,$$

See here for details and a nice example

TRPO in Practice

TRPO is relatively hard to implement in practice.
Proximal Policy Optimization (PPO) is easier to implement

Thus, we see a quick overview of TRPO

Relative Policy Performance

Define *surrogate advantage* which measures relative performance of policies

π_θ	New policy
π_{θ_k}	Old policy
$\mathcal{L}(\theta_k, \theta)$	Surrogate advantage of new policy over old policy

$$\mathcal{L}(\theta_k, \theta) = \underset{s,a \sim \pi_{\theta_k}}{\mathrm{E}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a) \right]$$

Expectation is taken under new policy

As usual, we estimate expectation by sampling

Taylor Approximation

$$\mathcal{L}(\theta_k, \theta) = \mathop{\mathrm{E}}_{s,a\sim\pi_{\theta_k}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a) \right]$$

TRPO uses first-order Taylor expansion for easier solution

Recall:

For function *f(x)* differentiable at point *x* = *a*, can consider approximations:

$$P_1(x) = f(a) + f'(a)(x-a)$$ (First order)

$$P_2(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2.$$ (Second order)

Taylor Approximation, contd.

$$\mathcal{L}(\theta_k, \theta) = \underset{s,a \sim \pi_{\theta_k}}{\mathrm{E}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a) \right]$$

TRPO uses first-order Taylor expansion for easier solution

Approximation has this form:

$$\mathcal{L}(\theta_k, \theta) \approx g^T(\theta - \theta_k)$$
$$\bar{D}_{KL}(\theta || \theta_k) \approx \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k)$$

where *g* is gradient and *H* is Hessian (2nd order derivative) of surrogate

This can be useful where parameters are sufficiently close

Optimization Problem

Framing as an optimization problem we have:

$$\theta_{k+1} = \arg\max_{\theta} g^T(\theta - \theta_k)$$
$$\text{s.t. } \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \leq \delta.$$

The objective of TRPO is to maximize the gradient of the surrogate advantage function

This is equivalent to policy gradient method, which moves in direction of gradient

Solving the Optimization Problem

Framing as an optimization problem we have:

$$\theta_{k+1} = \arg\max_{\theta} \, g^T(\theta - \theta_k)$$
$$\text{s.t. } \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \leq \delta.$$

This can be solved with this update rule:

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g,$$

Notice each parameter has different learning rate

Hessian *H* needs to be inverted

TRPO Challenges

Several challenges make this hard to implement (see Bilgin p249 for list)

Includes:

> Taylor approximation may be violated

> For policy network with massive number of parameters,

- Inverting Hessian H may be hard
- May be hard to store H^{-1}

Proximal Policy Optimization (PPO)

Same approach as TRPO:
> Take policy improvement steps
> Limit distance between old and new policy to retain good performance

TRPO used second-order Taylor expansion

PPO uses first-order methods and tricks
Easier to implement, competitive performance

PPO Variants

PPO-Penality : penalize KL-divergence in objective function

PPO-Clip : No KL-divergence term. Specialized clipping of objective fcn.

OpenAI uses PPO-Clip

Objective function from earlier:

$$\mathcal{L}(\theta_k, \theta) = \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a) \right]$$

We will limit policy changes based on ratio

PPO-Clip

Clipped form of objective function *L* has form below:

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \quad g(\epsilon, A^{\pi_{\theta_k}}(s,a))\right),$$

This can be computing based on sample trajectories

The *min*() operator will serve to limit changes based on ratio of new and old policies

PPO-Clip : Example of Positive Case

If advantage for a state-action pair is positive, the contribution to objective reduces to

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1+\epsilon)\right) A^{\pi_{\theta_k}}(s, a).$$

Any benefit of changing the policy beyond clipped limit is removed

This acts as a regularizer

PPO-Clip Algorithm

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0

2: **for** $k = 0, 1, 2, ...$ **do**

3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

4: Compute rewards-to-go $\hat{R}_t$.

5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.

7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

8: **end for**

Deep Deterministic Policy Gradient (DDPG)

DDPG - Motivation

We studied a lot of approaches using Q-function

For discrete / low cardinality action space, this makes sense: $a^*(s) = \arg\max_a Q^*(s, a)$

For continuous / high cardinality action space, the *max* is too costly

Instead, a function approximator is used in DDPG

This is the same idea that we used for state spaces with DQN

DDPG - Parametrization

The idea is to use: $\max_a Q(s, a) \approx Q(s, \mu(s))$

for policy $\mu(s)$

This assumes the policy function is differentiable wrt action

The policy is now *deterministic*

Uses the same features as DQN:

- Replay buffer for stored experiences

- Target network with soft updates (Polyak averaging): $\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi$

DDPG - Training

The policy is deterministic

If the agent explores on-policy, it may not try enough actions

The trick for better exploration is to add noise to the actions during training

DDPG - Algorithm

Add noise for more exploration

Soft updates

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$

Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$

repeat

  Observe state s and select action $\boxed{a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High}), \text{ where } \epsilon \sim \mathcal{N}}$

  Execute a in the environment

  Observe next state s', reward r, and done signal d to indicate whether s' is terminal

  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$

  If s' is terminal, reset environment state.

  if it's time to update **then**

    for however many updates **do**

      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\begin{aligned} \phi_{\text{targ}} &\leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi \\ \theta_{\text{targ}} &\leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta \end{aligned}$$

    end for

  end if

until convergence

Source: https://spinningup.openai.com/en/latest/algorithms/ddpg.html