Chapter 8

Policy Gradient Methods

Reading

1. Sutton & Barto, Chapter 9–10, 13
2. Simple random search provides a competitive approach to reinforcement learning at https://arxiv.org/abs/1803.07055
3. Proximal Policy Optimization Algorithms https://arxiv.org/abs/1707.06347
4. Are Deep Policy Gradient Algorithms Truly Policy Gradient Algorithms? https://arxiv.org/abs/1811.02553
5. Asynchronous Methods for Deep Reinforcement Learning http://proceedings.mlr.press/v48/mniha16.pdf

This chapter discusses methods to learn the controller that minimizes a given cost functional over trajectories of an unknown dynamical system. We will use what is called the "policy gradient" which will be the main section of this chapter.

Recall from the last chapter that we were able to fit stochastic controllers of the form $u_{\widehat{\theta}}(\cdot \mid x)$ that is a probability distribution on the control-space U for each $x \in X$. We fitted u_θ using data from the expert in imitation learning. We did not learn the cost-to-go for the fitted controller, like we did in the lectures on dynamic programming. This is a clever choice: it is often easier to learn the controller in a typical problem than to *compute* the optimal cost-to-go as a parametric function $J^*(x)$.

Can you give another instance when we have computed a controller previously in the class without coming up with its cost-to-go?

8.1 Standard problem setup in RL

Dynamics and rewards In this and the next few chapters we will always consider discrete-time stochastic dynamical systems with a stochastic controller

with parameters (weights) θ. We denote them as follows

$$\begin{aligned} x_{k+1} &\sim p(\cdot \mid x_k, u_k) \text{ with noise denoted by } \epsilon_k \\ \mathbf{u}_k &\sim u_\theta(\cdot \mid x_k) \end{aligned}$$

We will also change perspective and instead of minimizing the infinite-horizon sum of a runtime cost, maximize the sum of a runtime reward

$$r(x, u) := -q(x, u).$$

We do so simply to confirm to tradition and standard notation in reinforcement learning; the two are mathematically completely equivalent. We are interested in maximizing the expected value of the cumulative rewards over infinite-horizon trajectories of the system

$$J(\theta) = \mathop{\mathrm{E}}_{x_1, x_2, \ldots} \left[\underbrace{\sum_{k=0}^{\infty} \gamma^k \, r(x_k, u_k)}_{\text{discounted return}} \mid x_0 \right]; \tag{8.1}$$

where each $u_k \sim u_\theta(\cdot \mid x_k)$ and each $x_{k+1} \sim p(\cdot \mid x_k, u_k)$.

Trajectory space Let us write out one trajectory of such a system a bit more explicitly. We know that the probability of the next state x_{k+1} given x_k is $p(x_{k+1} \mid x_k, u_k)$. The probability of taking a control u_k at state x_k is $u_\theta(u_k \mid x_k)$. We denote an infinite trajectory by

$$\tau = x_0, u_0, x_1, u_1, \ldots.$$

The probability this entire trajectory occurring is

$$p_\theta(\tau) = \prod_{k=0}^{\infty} p(x_{k+1} \mid x_k, u_k) \; u_\theta(u_k \mid x_k);$$

we have emphasized that the distribution of trajectories depends on the weights of controller θ. If we take the logarithm,

$$\log p_\theta(\tau) = \sum_{k=0}^{\infty} \log p(x_{k+1} \mid x_k, u_k) + \log u_\theta(u_k \mid x_k).$$

Given a trajectory $\tau = x_0, u_0, x_1, u_1, \ldots$, the sum

$$R(\tau) = \sum_{k=0}^{\infty} \gamma^k \, r(x_k, u_k) \tag{8.2}$$

is called the discounted return of the trajectory τ. Sometimes we will also talk of the undiscounted return of the trajectory which is the sum of the rewards up to some fixed finite horizon T without the discount factor pre-multiplier.

Using this notation, we can write out objective from (8.1) as

$$J(\theta) = \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} [R(\tau) \mid x_0] \tag{8.3}$$

where $p(\tau)$ is the probability distribution of an infinitely long trajectory τ.

Observe what is probably the most important point in policy-gradient based reinforcement learning: the probability of trajectory is an infinite product of terms all of which are smaller than 1 (they are probabilities), so it is essentially zero even if the state-space and the control-space are finite (even if they are small). Any given infinite (or long) trajectory is quite rare under the probability distribution of the stochastic controller. Policy-gradient methods sample lots of trajectories from the system and average the returns across these trajectories. Since the set of trajectories of even a small MDP is so large, sampling lots of trajectories, or even the most likely ones, is also very hard. This is a key challenge in getting RL algorithms to work.

Our goal in this chapter is to compute the best stochastic controller which maximizes the average discounted return. Mathematically, this amounts to finding

$$\widehat{\theta} = \operatorname*{argmax}_\theta J(\theta) := \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} [R(\tau) \mid x_0] \,. \tag{8.4}$$

The objective $J(\theta)$ is called the average return of the controller u_θ.

Computing the average return $J(\theta)$ Before we move on to optimizing $J(\theta)$, let us discuss how to compute it for given weights θ of the stochastic controller. We can sample n trajectories from the system and compute the an estimate of the expectation

$$\widehat{J}(\theta) \approx \frac{1}{n} \sum_{i=0}^{n} \sum_{k=0}^{T} \gamma^k r(x_k^i, u_k^i) \tag{8.5}$$

for some large time-horizon T and where each $u_k^i \sim u_\theta(\cdot \mid x_k^i)$.

Contrast (8.5) with the complexity of policy evaluation which was simply a system of linear equations. Evaluating the policy without having access to the dynamical system is harder.

8.2 Cross-Entropy Method (CEM)

Let us first consider a simple method to compute the best controller. The basic idea is to solve the problem

$$\widehat{\theta} = \operatorname*{argmax}_\theta J(\theta)$$

using gradient descent. We would like to update weights θ iteratively

$$\theta^{k+1} = \theta^k + \eta \, \nabla J(\theta).$$

where the step-size is $\eta > 0$ and $\nabla J(\theta)$ is the gradient of the objective $J(\theta)$ with respect to weights θ. Instead of computing the exact $\nabla J(\theta)$ which

we will do in the next section, let us simply compute the gradient using a finite-difference approximation. The i^{th} entry of the gradient is

$$(\widehat{\nabla} J(\theta))_i = \frac{J(\theta + \epsilon\, e_i) - J(\theta - \epsilon\, e_i)}{2\epsilon} \approx \frac{\widehat{J}(\theta + \epsilon\, e_i) - \widehat{J}(\theta - \epsilon\, e_i)}{2\epsilon}.$$

where $e_i = [0, 0, \ldots, 0, 1, 0, \ldots]$ is a vector with 1 on the i^{th} entry. Each quantity $\widehat{J}$ is computed as the empirical average return of n trajectories from the system. We compute all entries of the objective using this approximation and update the parameters using

$$\theta^{k+1} = \theta^k + \eta \widehat{\nabla}\, J(\theta^k).$$

A more efficient way to compute the gradient using finite-differences Instead of picking perturbations e_i along the cardinal directions, let us sample them from a Gaussian distribution

$$\xi^i \sim N(0, \sigma^2 I)$$

for some user-chosen covariance σ^2. We can however no longer use the finite-difference formula to compute the derivative because the noise e is not aligned with the axes. We can however use a Taylor series approximation as follows. Observe that

$$J(\theta + \xi) \approx J(\theta) + \langle \nabla\, J(\theta), \xi \rangle$$

where $\langle \cdot, \cdot \rangle$ is the inner product. Given m samples $\xi^1, \ldots, \xi^m$ observe that

$$\begin{aligned} \widehat{J}(\theta + \xi^1) &= \widehat{J}(\theta) + \left\langle \nabla\, J(\theta),\ \xi^1 \right\rangle \\ \widehat{J}(\theta + \xi^2) &= \widehat{J}(\theta) + \left\langle \nabla\, J(\theta),\ \xi^2 \right\rangle \\ &\vdots \\ \widehat{J}(\theta + \xi^m) &= \widehat{J}(\theta) + \langle \nabla\, J(\theta),\ \xi^m \rangle\,. \end{aligned} \tag{8.6}$$

is a linear system of equations in $\nabla\, J(\theta) \in \mathbb{R}^p$ where $\theta \in \mathbb{R}^p$. All quantities $\widehat{J}$ are estimated as before using trajectories drawn from the system. We solve this linear system, e.g., using least-squares if $m > p$, to get an estimate of the gradient $\widehat{\nabla} J(\theta)$.

The Cross-Entropy Method is a more crude but simpler to implement version of the above least-squares formulation. At each iteration it updates the parameters using the formula

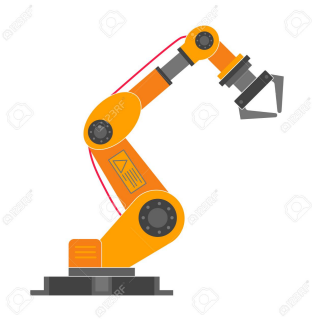
$$\theta^{k+1} = \mathop{\mathrm{E}}_{\theta \sim N(\theta^k, \sigma^2 I)} \left[\theta\, \mathbf{1}_{\{\widehat{J}(\theta) > \widehat{J}(\theta^k)\}} \right]. \tag{8.7}$$

In simple words, the CEM samples a few stochastic controllers u_θ from a Gaussian (or any other distribution) centered around the current controller u_{θ^k} and updates the weights θ^k in a direction that leads to an increase in $\widehat{J}(\theta) > \widehat{J}(\theta^k)$.

8.2.1 Some remarks on sample complexity of simulation-based methods

The CEM may seem to be a particularly bad method to maximize $J(\theta)$, after all we are perturbing the weights of the stochastic controller randomly and updating the weights if they result in a better average return $\widehat{J}(\theta)$. This is likely to work well if the dimensionality of weights $\theta \in \mathbb{R}^p$, i.e., p, is not too large. But is unlikely to work well if we are sampling θ in high-dimensions. Typical applications are actually the latter, remember that we are interested in using a deep network as a stochastic controller and θ are the weights of the neural networks.

Let us do a quick computation, if the state is $x \in \mathbb{R}^d$ and $u \in \mathbb{R}^m$ with $d = 12$ (joint angles and velocities) and $m = 6$ for a six-degree of freedom robot manipulator



and if we use a two-layer neural network with 64 neurons in the hidden layer, the total number of weights $\theta \in \mathbb{R}^p$ for the function $u_\theta(\cdot \mid x) = N(\mu_\theta(x), \sigma^2_\theta(x)I)$ where $\sigma^2(x)$ is a vector in $\mathbb{R}^m$, is

$$p = \underbrace{(12 \times 64 + 64) + (64 \times 6 + 6)}_{\text{for } \mu_\theta(x)} + \underbrace{(12 \times 64 + 64) + (64 \times 6 + 6)}_{\text{for } \sigma^2_\theta(x)}$$
$$= 2,444.$$

This is a very high-dimensional space to sample exhaustively, and it is quite large even if the input and output dimensions of the neural network are not too large. To appreciate the complexity of computing the gradient $\nabla J(\theta)$, let us think of how to compute it using finite-differences, we need two estimates $\widehat{J}(\theta - \epsilon e_i)$ and $\widehat{J}(\theta + \epsilon e_i)$ for every dimension $i \in \{1, \ldots, p\}$. Each estimate requires us to obtain n trajectories from the system. Since the number of trajectories that a robot can take is quite diverse, we should use a large n, so let's pick $n = 200$. The total number of trajectories required to update the parameters θ^k *at each iteration* is

$$2\, p\, n \approx 10^6.$$

This is an absurdly large number, and things are even more daunting when we realize that each update of the weights requires us to sample these many trajectories from the system. It is not reasonable to sample such a large number of trajectories from a physical robot, that too for each update of the weights.

ℹ For comparison, a busy espresso bar in a city makes about 500 shots per day. The espresso machine would have to work for 5 years *without breaking down* to make 10^6 shots.

Using fast simulators for RL If we expand our horizon and think of learning controllers in simulation, things feel much more reasonable. While running a large number of trajectories may degrade a robot beyond use, doing so requires just computation time in a robot simulator. There is a large number of simulators that are available with various capabilities, e.g., Gazebo (http://gazebosim.org) is a sophisticated simulator inside ROS that uses a number of Physics engines such as Bullet (https://pybullet.org/wordpress), MuJoCo (http://www.mujoco.org) is incredibly fast although not very good modeling contact, Unity is a popular platform to simulate driving and complicated scenes (https://docs.nvidia.com/isaac/isaac/doc/simulation/unity3d.html), Drake (https://drake.mit.edu) is better at contact modeling but more complex and slower; most autonomous driving companies have developed their own driving simulators in-house. The assigned reading (#2) for this chapter is a paper which develops a very fast implementation of the CEM for use in simulators.

Working well in simulation does not mean that a controller works well on the real robot It is important to realize that a simulator is not equivalent to the physical robot. Each simulator makes certain trade-offs in capturing the dynamics of the real system and it is not a given that a controller that was learned using data from a simulator will work well on a real robot. For instance, OpenAI had to develop a large number of tricks (which took about a year) to modify the simulator to enable the learned policy to work well on a robot (https://openai.com/blog/learning-dexterity) for a fairly narrow set of tasks.

8.3 The Policy Gradient

> In this section, we will study how to take the gradient of the objective $J(\theta)$, without using finite-differences.

We would like to solve the optimization problem

$$\max_{\theta} J(\theta) := \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} \left[R(\tau) \mid x_0 \right]$$

We will suppress the dependence on x_0 to keep the notation clear. The expectation is taken over all trajectories starting at state x_0 realized using the stochastic controller $u_\theta(\cdot \mid x)$. We to update weights θ using gradient descent which amounts to

$$\theta^{k+1} = \theta^k + \eta \, \nabla_\theta \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} \left[R(\tau) \right].$$

First let us note that the distribution p_θ using which we compute the expectation also depends on the weights θ. This is why we cannot simply move the derivative ∇_θ inside the expectation

$$\nabla_\theta \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} \left[R(\tau) \right] \neq \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} \left[\nabla_\theta \, R(\tau) \right].$$

We need to think of a new technique to compute the gradient above. Essentially, we would like to do the chain rule of calculus but where one of the functions in the chain is an expectation. The likelihood-ratio trick described next allows us to take such derivatives. Here is how the computation goes

$$\begin{aligned}
\nabla_\theta \mathop{\mathrm{E}}_{\tau\sim p_\theta}[R(\tau)] &= \nabla_\theta \int R(\tau) p_\theta(\tau) \mathrm{d}\tau \\
&= \int R(\tau)\, \nabla_\theta\, p_\theta(\tau)\, \mathrm{d}\tau \\
&\text{(move the gradient inside, integral is over trajectories } \tau \text{ which do not depend on} \theta \text{ themselves)} \\
&= \int R(\tau) p_\theta(\tau) \frac{\nabla\, p_\theta(\tau)}{p_\theta(\tau)} \mathrm{d}\tau \\
&= \int R(\tau) p_\theta(\tau)\, \nabla \log p_\theta(\tau) \mathrm{d}\tau \\
&= \mathop{\mathrm{E}}_{\tau\sim p_\theta(\tau)}[R(\tau)\, \nabla \log p_\theta(\tau)] \\
&\approx \frac{1}{n} \sum_{i=1}^{n} R(\tau^i)\, \nabla \log p_\theta(\tau^i)
\end{aligned} \tag{8.8}$$

This is called the likelihood-ratio trick to compute the policy gradient. It simply multiplies and divides by the term $p_\theta(\tau)$ and rewrites the term $\frac{\nabla\, p_\theta}{p_\theta} = \nabla \log p_\theta$. It gives us a neat way to compute the gradient: we sample n trajectories $\tau^1, \ldots, \tau^n$ from the system and average the return of each trajectory $R(\tau^i)$ weighted by the gradient of the likelihood of taking each trajectory $\log p_\theta(\tau^i)$. The central point to remember here is that the gradient

$$\begin{aligned}
\nabla_\theta \log p_\theta(\tau^i) &= \nabla_\theta \sum_{k=0}^{T} \log p(x_{k+1}^i \mid x_k^i, u_k^i) + \log u_\theta(u_k^i \mid x_k^i) \\
&= \sum_{k=0}^{T} \nabla_\theta \log u_\theta(u_k^i \mid x_k^i)
\end{aligned} \tag{8.9}$$

is computed using backpropagation for a neural network. This expression is called the policy gradient because it is the gradient of the objective $J(\theta)$ with respect to the parameters of the controller/policy θ.

Variance of policy gradient The expression for the policy gradient may seem like a sleight of hand. It is a clean expression to get the gradient of the objective but also comes with a number of problems. Observe that

$$\begin{aligned}
\nabla \mathop{\mathrm{E}}_{\tau\sim p_\theta(\tau)}[R(\tau)] &= \mathop{\mathrm{E}}_{\tau\sim p_\theta(\tau)} \left[R(\tau) \frac{\nabla\, p_\theta(\tau)}{p_\theta(\tau)} \right] \\
&\approx \frac{1}{n} \sum_{i=1}^{n} R(\tau^i)\, \frac{\nabla\, p_\theta(\tau^i)}{p_\theta(\tau^i)}.
\end{aligned}$$

If we sample trajectories τ^i that are not very likely under the distribution $p_\theta(\tau)$, the denominator in some of the summands can be very small. For trajectories that are likely, the denominator is large. The empirical estimate

of the expectation using n trajectories where some terms are very small and some others very large, therefore has a large variance. So one does need lots of trajectories from the system/simulator to compute a reasonable approximation of the policy gradient.

8.3.1 Reducing the variance of the policy gradient

Control variates You will perhaps appreciate that computing the accurate policy gradient is very hard in practice. Control variates is a general concept from the literature on Monte Carlo integration and is typically introduced as follows. Say we have a random variable X and we would like to guess its expected value $\mu = \mathrm{E}[X]$. Note that X is an unbiased estimator of μ but it may have a large variance. If we have another random variable Y with known expected value $\mathrm{E}[Y]$, then

$$\hat{X} = X + c(Y - \mathrm{E}[Y]) \tag{8.10}$$

is also an unbiased estimator for μ for any value of c. The variance of $\hat{X}$ is

$$\mathrm{var}(\hat{X}) = \mathrm{var}(X) + c^2\,\mathrm{var}(Y) + 2c\;\mathrm{cov}(X, Y).$$

which is minimized for

$$c^* = -\frac{\mathrm{cov}(X, Y)}{\mathrm{var}(Y)}$$

for which we have

$$\mathrm{var}(\hat{X}) = \mathrm{var}(X) - c^{*2}\;\mathrm{var}(Y) = \left(1 - \left(\frac{\mathrm{cov}(X, Y)}{\mathrm{var}(Y)}\right)^2\right)\mathrm{var}(X).$$

By subtracting $Y - \mathrm{E}[Y]$ from our observed random variable X, we have reduced the variance of X if the correlation between X and Y is non-zero. Most importantly, note that no matter what Y we plug into the above expression, we can never increase the variance of X; the worst that can happen is that we pick a Y that is completely uncorrelated with X and end up achieving nothing.

Baseline We will now use the concept of a control variate to reduce the variance of the policy gradient. This is known as "building a baseline". The simplest baseline one can build is to subtract a constant value from the return. Consider the PG given by

$$\begin{aligned}\nabla J(\theta) &= \mathop{\mathrm{E}}_{\tau\sim p_\theta}\left[R(\tau)\;\nabla \log p_\theta(\tau)\right] \\ &= \mathop{\mathrm{E}}_{\tau\sim p_\theta(\tau)}\left[(R(\tau) - b)\;\nabla \log p_\theta(\tau)\right].\end{aligned}$$

Observe that

$$\begin{aligned}\mathop{\mathrm{E}}_{\tau\sim p_\theta(\tau)}\left[b\,\nabla \log p_\theta(\tau)\right] &= \int \mathrm{d}\tau b\; p_\theta(\tau)\,\nabla \log p_\theta(\tau) \\ &= \int \mathrm{d}\tau b\;\nabla\, p_\theta(\tau) = b\;\nabla \int \mathrm{d}\tau\; p_\theta(\tau) = b\;\nabla 1 = 0.\end{aligned}$$

Example 1: Using the average returns of a mini-batch as the baseline What is the simplest baseline b we can cook up? Let us write the mini-batch version of the policy gradient

$$\widehat{\nabla} J(\theta) := \frac{1}{\cancel{b}} \sum_{i=1}^{\cancel{b}} \left[R(\tau^i) \nabla \log p_\theta(\tau^i) \right].$$

where $\tau^1, \ldots, \tau^{\cancel{b}}$ are trajectories that are a part of our mini-batch. We can set

$$b = \frac{1}{\cancel{b}} \sum_{i=1}^{\cancel{b}} R(\tau^i)$$

can use the variance-reduced gradient

$$\widehat{\nabla} J(\theta) = \frac{1}{\cancel{b}} \sum_{i=1}^{\cancel{b}} \left[\big(R(\tau^i) - b \big) \ \nabla \log p_\theta(\tau^i) \right].$$

This is a one-line change in your code for policy gradient so there is no reason not to do it.

Example 2: A weighted averaged of the returns using the log-likelihood of the trajectory The previous example showed how we can use one constant baseline, namely the average of the discounted returns of all trajectories in a mini-batch. What is the **best** constant b we can use? We can perform a similar computation as done in the control variate introduction to minimize the variance of the policy gradient to get the following.

$$\begin{aligned}
\text{var}\left(\widehat{\nabla}_{\theta_i} J(\theta) \right) &= \mathop{\mathrm{E}}_{\tau} \left[\left(\left(R(\tau) - b_i \right) \nabla_{\theta_i} \log p_\theta(\tau) \right)^2 \right] - \left(\mathop{\mathrm{E}}_{\tau} \left[\left(\left(R(\tau) - b_i \right) \right) \nabla_{\theta_i} \log p_\theta(\tau) \right] \right)^2 \\
&= \mathop{\mathrm{E}}_{\tau} \left[\left(\left(R(\tau) - b_i \right) \nabla_{\theta_i} \log p_\theta(\tau) \right)^2 \right] - \left(\widehat{\nabla}_{\theta_i} J(\theta) \right)^2.
\end{aligned}$$

Set

$$\frac{\text{var}\left(\widehat{\nabla}_{\theta_i} J(\theta) \right)}{\mathrm{d}b_i} = 0$$

in the above expression to get

$$b_i = \frac{\mathrm{E}_\tau \left[\left(\nabla_{\theta_i} \log p_\theta(\tau) \right)^2 R(\tau) \right]}{\mathrm{E}_\tau \left[\left(\nabla_{\theta_i} \log p_\theta(\tau) \right)^2 \right]}$$

which is the baseline you should subtract from the gradient of the i^{th} parameter θ_i to result in the largest variance reduction. This expression is just the expected return but it is weighted by the magnitude of the gradient, this again is 1–2 lines of code.

❓ Show that any function that only depends on the state x can be used as a baseline in the policy gradient. This technique is known as reward shaping.

8.4 An alternative expression for the policy gradient

We will first define an important quantity that helps us think of RL algorithms.

Definition 8.1 (Discounted state visitation frequency). Given a stochastic controller $u_\theta(\cdot \mid x)$ the discounted state visitation frequency for a discrete-time dynamical system is given by

$$d^\theta(x) = \sum_{k=0}^{\infty} \gamma^k \, \mathrm{P}(x_k = x \mid x_0, u_k \sim u_\theta(\cdot \mid x_k)).$$

The distribution $d^\theta(x)$ is the probability of visiting a state x computed over all trajectories of the system that start at the initial state x_0. If $\gamma = 1$, this is the steady-state distribution of the Markov chain underlying the Markov Decision Process where at each step the MDP choses the control $u_k \sim u_\theta(\cdot \mid x_k)$. The fact that we have defined the discounted distribution is a technicality; this version is seen in the policy gradient expression. You will also notice that $d^\theta(x)$ is not a normalized distribution. The normalization constant is difficult to characterize both theoretically and empirically and we will not worry about it here; RL algorithms do not require it.

Q-function Using the discounted state visitation frequency, the policy gradient that we saw before can be written in terms of the value function as follows.

$$\begin{aligned} \nabla J(\theta) &= \mathop{\mathrm{E}}_{\tau \sim p_\theta} \left[R(\tau) \nabla \log p_\theta(\tau) \right] \\ &= \mathop{\mathrm{E}}_{x \sim d^\theta} \mathop{\mathrm{E}}_{u \sim u_\theta(\cdot \mid x)} \left[q^\theta(x,u) \nabla_\theta \log u_\theta(u \mid x) \right]. \end{aligned} \tag{8.11}$$

The function $q^\theta(x,u)$ is similar to the cost-to-go that we have studied in dynamic programming and is called the Q-function

$$q^\theta(x,u) = \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} \left[R(\tau) \mid x_0 = x, u_0 = u \right]. \tag{8.12}$$

It is the infinite-horizon discounted cumulative reward (i.e., the return) if the system starts at state x and takes the control u in the first step and runs the controller $u_\theta(\cdot \mid x)$ for all steps thereafter. We make the dependence of q^θ on the parameters θ of the controller explicit.

ℹ The derivation of this expression is easy although tedious, you can find it in the Appendix of the paper "Policy gradient methods for reinforcement learning with function approximation" at https://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf.

Compare the above formula for the policy gradient with the one we had before in (8.8)

$$\begin{aligned} \widehat{\nabla} J(\theta) &= \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} \left[R(\tau) \ \nabla \log p_\theta(\tau) \right] \\ &= \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{k=0}^{T} \gamma^k r(x_k, u_k) \right) \ \left(\sum_{k=0}^{T} \nabla \log u_\theta(u_k \mid x_k) \right) \right]. \end{aligned}$$

It is important to notice that this is an expectation over trajectories; whereas

(8.11) is an expectation over states x sampled from the discounted state visitation frequency. The control u_k for both is sampled from the stochastic controller at each time-step k. The most important distinction is that (8.11) involves the expectation of the Q-function q^θ weighted by the gradient of the log-likelihood of picking each control action. There are numerous hacky ways of deriving (8.11) from (8.8) but remember that they are fundamentally different expressions of the *same policy gradient*.

This expression allows understanding of a number of properties of reinforcement learning.

1. While the algorithm collects the data, states that are unlikely under the distribution d^θ contribute little to (8.11). In other words, the policy gradient is insensitive to such states. The policy update will not consider these unlikely states that the system is prone to visit infrequently using the controller u_θ.
2. The opposite happens for states which are very likely. For two controls u_1, u_2 at the same state x, the policy increases the log-likelihood of taking the controls weighted by their values $q^\theta(x, u_1)$ and $q^\theta(x, u_2)$. This is sort of the "definition" of reinforcement learning. In the expression (8.8) the gradient was increasing the likelihood of trajectories with high returns, here it deals with states and controls individually.

8.4.1 Implementing the new expression

Suppose we have a stochastic control that is a Gaussian

$$u_\theta(u \mid x) = \frac{1}{(2\pi\sigma^2)^{p/2}} \, e^{-\frac{\|u - \theta^\top x\|^2}{2\sigma^2}}$$

where $\theta \in \mathbb{R}^{d \times p}$ and $u \in \mathbb{R}^p$; the variance σ can be chosen by the user. We can easily compute $\log u_\theta(u \mid x)$ in (8.11). How should one compute $q^\theta(x, u)$ in (8.12)? We can again estimate it using sample trajectories from the system; each of these trajectories would have to start from a state x and the control at the first step would be u, with the controller u_θ being used thereafter. Note that we have one such trajectory, namely the remainder of the trajectory where we encountered (x, u) while sampling trajectories for the policy gradient in (8.11). In practice, we do not sample trajectories a second time, we simply take this trajectory, let us call it $\tau_{x,u}$ and set

$$q^\theta(x, u) = \sum_{k=0}^{T} \gamma^k r(x_k, u_k)$$

for some large time-horizon T where $(x_0, u_0) = (x, u)$ and the summation is evaluated for (x_k, u_k) that lie on the trajectory $\tau_{x,u}$. Effectively, we are evaluating (8.12) using one sample trajectory, a highly erroneous estimate of q^θ.

8.5 Actor-Critic methods

We can of course do more sophisticated things to evaluate the Q-function q^θ in our new expression of the policy gradient.

> Actor-Critic methods fit a Q-function to the data collected from the system using the current controller (policy evaluation step) and then use this fitted Q-function in the expression of the policy gradient (8.11) to update the controller. In this sense, Actor-Critic methods are conceptually similar to policy iteration.

In order to understand how to fit the Q-function, first recall that it should satisfy the Bellman equation. This means

$$q^\theta(x,u) = r(x,u) + \gamma \underset{u\sim u_\theta(\cdot|x'),x'\sim P(\cdot|x,u)}{\mathrm{E}} \left[q^\theta(x',u')\right]. \tag{8.13}$$

We do not know a model for the system so we cannot evaluate the expectation over $x' \sim \mathrm{P}(\cdot \mid x,u)$ like we used to in dynamic programming. But we do have the ability to get trajectories τ^i from the system.

Let's say (x_k^i, u_k^i) lie on τ^i at time-step k. We can then estimate the expectation over $\mathrm{P}(\cdot \mid x_k^i, u_k^i)$ using simply x_{k+1}^i and the expectation over the controls using simply u_{k+1}^i to write

$$q^\theta(x_k^i, u_k^i) \approx r(x_k^i, u_k^i) + \gamma\, q^\theta(x_{k+1}^i, u_{k+1}^i) \quad \text{for all } i \le n, k \le T.$$

This is a nice constraint on the Q-function. If this were a discrete-state, discrete-control MDP, it is a set of linear equations for the q-values. These constraints would be akin to our linear equations for evaluating a policy in dynamic programming except that instead of using the dynamics model (the transition matrix), we are using trajectories sampled from the system.

Parameterizing the Q-function using a neural network If we are dealing with a continuous state/control-space, we can think of parameterizing the q-function using parameters φ

$$q_\varphi^\theta(x,u) : X \times U \to \mathbb{R}.$$

The parameterization is similar to the parameterization of the controller, e.g., just like we would write a deterministic controller as

$$u_\theta(x) = \theta^\top x$$

we can think of a linear Q-function of the form

$$q_\varphi^\theta(x,u) = \varphi^\top \begin{bmatrix} x \\ u \end{bmatrix}, \quad \varphi \in \mathbb{R}^{m+d}$$

which is a linear function in the states and controls. You can also think of using something like

$$q^\theta_\varphi(x,u) = \begin{bmatrix}1 & x & u\end{bmatrix} \varphi \begin{bmatrix}1\\ x\\ u\end{bmatrix} \qquad \varphi \in \mathbb{R}^{(m+d+1)\times(m+d+1)}.$$

which is quadratic in the states and controls, or in general a deep network with weights φ as the Q-function.

Fitting the Q-function We can now "fit" the parameters of the Q-function by solving the problem

$$\widehat{\varphi} = \underset{\varphi}{\operatorname{argmin}} \frac{1}{n(T+1)} \sum_{i=1}^{n}\sum_{k=0}^{T} \|q^\theta_\varphi(x^i_k,u^i_k) - r(x^i_k,u^i_k) - \gamma\, q^\theta_\varphi(x^i_{k+1},u^i_{k+1})\|^2. \tag{8.14}$$

which is nothing other than enforcing the Bellman equation in (8.13). If the Q-function is linear in $[x,u]$ this is a least squares problem, if it is quadratic the problem is a quadratic optimization problem which can also be solved efficiently, in general we would solve this problem using stochastic gradient descent if we are parameterizing the Q-function using a deep network. Such a Q-function is called the "critic" because it *evaluates* the controller u_θ, or the "actor". This version of the policy gradient where one fits the parameters of both the controller and the Q-function are called Actor-Critic methods.

ℹ We will be pedantic and always write the q-function as q^θ_φ. The superscript θ denotes that this is the q-function corresponding to the θ-parameterized controller u_θ. The subscript denotes that the q-function is parameterized by parameters φ.

Actor-Critic Methods We fit a deep network with weights θ to parameterize a stochastic controller $u_\theta(\cdot \mid x)$ and another deep network with weights φ to parameterize the Q-function of this controller, $q^\theta_\varphi(x,u)$. Let the controller weights at the k^{th} iteration be θ^k and the Q-function weights be φ^k.

1. Sample n trajectories, each of T timesteps, using the current controller $u^k_\theta(\cdot \mid x)$.
2. Fit a Q-function $q^{\theta^k}_{\varphi^{k+1}}$ to this data using (8.14). using stochastic gradient descent. While performing this fitting (although it is not mathematically sound), it is common to use initialize the Q-function weights to their values from the previous iteration φ_κ.
3. Compute the policy gradient using the alternative expression in (8.11) and update parameters of the policy to θ^{k+1}.

8.5.1 Advantage function

The new expression for the policy gradient in (8.11) also has a large variance; this should be no surprise, it is after all equal to the old expression. We can however perform variance reduction on this using the value function.

Our goal as before would be construct a relevant baseline to subtract from the Q-function. It turns out that any function that depends only upon the state

x is a valid baseline. This gives a powerful baseline for us to to use in policy gradients. We can use the value function as the baseline. The value function for controls taken by the controller $u_\theta(\cdot \mid x)$ (notice that this is not the optimal value function, it is simply the policy evaluation) is given by

$$v^\theta(x) = \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} \left[R(\tau) \mid x_0 = x \right]$$

where $u_k \sim u_\theta(\cdot \mid x_k)$ at each timestep. We also know that the value function is the expected value of the Q-function across different controls sampled by the controller

$$v^\theta(x) = \mathop{\mathrm{E}}_{u \sim u_\theta(\cdot|x)} \left[q^\theta(x,u) \right]. \tag{8.15}$$

The value function again satisfies the dynamic programming principle/Bellman equation

$$v^\theta(x) = \mathop{\mathrm{E}}_{u \sim u_\theta(\cdot|x)} \left[r(x,u) + \gamma \mathop{\mathrm{E}}_{x' \sim \mathrm{P}(\cdot|x,u)} \left[v^\theta(x') \right] \right].$$

We again parameterize the value function

$$v^\theta_\psi(x) : X \to \mathbb{R}$$

using parameters ψ and fit it to the data in the same way as (8.14) to get

$$\widehat{\psi} = \operatorname*{argmin}_\psi \frac{1}{n(T+1)} \sum_{i=1}^{n} \sum_{k=0}^{T} \| v^\theta_\psi(x^i_k) - r(x^i_k, u^i_k) - \gamma\, v^\theta_\psi(x^i_{k+1}) \|^2. \tag{8.16}$$

Using this baseline can modify the policy gradient to be

$$\nabla J(\theta) = \mathop{\mathrm{E}}_{x \sim d^\theta} \mathop{\mathrm{E}}_{u \sim u_\theta(\cdot|x)} \left[\Big(\underbrace{q^\theta_\varphi(x,u) - v^\theta_\psi(x)}_{a^\theta_{\varphi,\psi}(x,u)} \Big) \nabla_\theta \log u_\theta(u \mid x) \right]. \tag{8.17}$$

where each of the functions q^θ_φ and v^θ_ψ are themselves fitted using (8.14) and (8.16) respectively. The difference

$$\begin{aligned} a^\theta_{\varphi,\psi}(x,u) &= q^\theta_\varphi(x,u) - v^\theta_\psi(x) \\ &\approx q^\theta_\varphi(x,u) - \mathop{\mathrm{E}}_{u \sim u_\theta(\cdot|x)} \left[q^\theta_\varphi(x,u) \right] \end{aligned} \tag{8.18}$$

is called the advantage function. It measures how much better the particular control u is for a state x as compared to the average return of controls sampled from the controller at that state. The form (8.17) is the most commonly implemented form in research papers whenever they say "we use the policy gradient".

❓ The advantage function is very useful while doing theoretical work on RL algorithms. But it is also extremely useful in practice. It imposes a constraint upon our estimate q^θ_φ and the estimate v^θ_ψ. If we are not solving (8.14) and (8.16) to completion, we may benefit by imposing this constraint on the advantage function. Can you think of a way?

8.6 Discussion

This brings to an end the discussion of policy gradients. They are, in general, a complicated suite of algorithms to implement. You will see some of this complexity when you implement the controller for something as simple as the simple pendulum. The key challenges with implementing policy gradients come from the following.

1. Need lots of data, each parameter update requires fresh data from the systems. Typical problems may need a million trajectories, most robots would break before one gets this much data from them if one implements these algorithms naively.
2. The log-likelihood ratio trick has a high variance due to $u_\theta(\cdot \mid x)$ being in the denominator of the expression, so we need to implement complex variance reduction techniques such as actor-critic methods.
3. Fitting the Q-function and the value function is not easy, each parameter update of the policy ideally requires you to solve the entire problems (8.14) and (8.16). In practice, we only perform a few steps of SGD to solve the two problems and reuse the solution of k^{th} iteration update as an initialization of the $k+1^{\text{th}}$ update. This is known as "warm start" in the optimization literature and reduces the computational cost of fitting the Q/value-functions from scratch each time.
4. The Q/value-function fitted in iteration k may be poor estimates of the Q/value at iteration $k+1$ for the new controller $u_{\theta^{k+1}}(\cdot \mid x)$. If the controller parameters change quickly, θ^{k+1} is very different from θ^k, then so are $q^{\theta^{k+1}}$ and $v^{\theta^{k+1}}$. There is a very fine balance between training quickly and retaining the efficiency of warm start; and tuning this in practice is quite difficult. A large number of policy gradient algorithms like TRPO (https://arxiv.org/abs/1502.05477) and PPO (https://arxiv.org/abs/1707.06347) try to alleviate this with varying degrees of success.
5. The latter, PPO, is a good policy-gradient-based algorithm to try on a new problem. For instance, in a very impressive demonstration, it was used to build an RL agent to play StarCraft (https://openai.com/blog/openai-five).