Chapter 4

Policy Gradients

4.1 Motivation

The scope of our problem has been gradually expanding.

1. In the first chapter, we considered *bandits* with a finite number of arms, where the only stochasticity involved was their rewards.
2. In the second chapter, we considered *MDPs* more generally, involving a finite number of states and actions, where the state transitions are Markovian.
3. In the third chapter, we considered *continuous* state and action spaces and developed the *Linear Quadratic Regulator.* We then showed how to use it to find *locally optimal solutions* to problems with nonlinear dynamics and non-quadratic cost functions.

Now, we'll continue to investigate the case of finding optimal policies in large MDPs using the self-explanatory approach of *policy optimization.* This is a general term encompassing many specific algorithms we've already seen:

- *Policy iteration* for finite MDPs,
- *Iterative LQR* for locally optimal policies in continuous control.

Here we'll see some general algorithms that allow us to optimize policies for general kinds of problems. These algorithms have been used in many groundbreaking applications, including AlphaGo, OpenAI Five. These methods also bring us into the domain where we can use *deep learning* to approximate complex, nonlinear functions.

4.2 (Stochastic) Policy Gradient Ascent

Let's suppose our policy can be *parameterized* by some parameters θ. For example, these might be a preferences over state-action pairs, or in a high-dimensional case, the weights and biases of a deep neural network. We'll talk more about possible parameterizations in section 4.5

Remember that in reinforcement learning, the goal is to *maximize reward.* Specifically, we seek the parameters that maximize the expected total reward, which we can express concisely using the value function we defined earlier:

$$\begin{aligned} J(\theta) := \mathop{\mathbb{E}}_{s_0 \sim \mu_0} V^{\pi_\theta}(s_0) = & \mathbb{E} \sum_{t=0}^{T-1} r_t \\ \text{where} \quad & s_0 \sim \mu_0 \\ & s_{t+1} \sim P(s_t, a_t), \\ & a_h = \pi_\theta(s_h) \\ & r_h = r(s_h, a_h). \end{aligned} \tag{4.1}$$

We call a sequence of states, actions, and rewards a **trajectory** $\tau = (s_i, a_i, r_i)_{i=0}^{T-1}$, and the total time-discounted reward is also often called the **return** $R(\tau)$ of a trajectory. Note that the above is the *undiscounted, finite-horizon case,* which we'll continue to use throughout the chapter, but analogous results hold for the *discounted, infinite-horizon case.*

Note that when the state transitions are Markov (i.e. s_t only depends on s_{t-1}, a_{t-1}) and the policy is stationary (i.e. $a_t \sim \pi_\theta(s_t)$), we can write out the *likelihood of a trajectory* under the policy π_θ:

$$\begin{aligned} \rho_\theta(\tau) = & \mu(s_0)\pi_\theta(a_0|s_0) \\ & \times P(s_1|s_0, a_0)\pi_\theta(a_1|s_1) \\ & \times \cdots \\ & \times P(s_{H-1}|s_{H-2}, a_{H-2})\pi_\theta(a_{H-1}|s_{H-1}). \end{aligned} \tag{4.2}$$

This lets us rewrite $J(\theta) = \mathbb{E}_{\tau \sim \rho_\theta} R(\tau)$.

Now how do we optimize for this function? One very general optimization technique is *gradient ascent.* Namely, the **gradient** of a function at a given point answers: At this point, which direction should we move to increase the function the most? By repeatedly moving in this direction, we can keep moving up on the graph of this function. Expressing this iteratively, we have:

$$\theta_{t+1} = \theta_t + \eta \nabla_\theta J(\pi_\theta) \Big|_{\theta = \theta_t},$$

Where η is a *hyperparameter* that says how big of a step to take each time.

In order to apply this technique, we need to be able to evaluate the gradient $\nabla_\theta J(\pi_\theta)$. How can we do this?

In practice, it's often impractical to evaluate the gradient directly. For example, in supervised learning, $J(\theta)$ might be the sum of squared prediction errors across an entire **training dataset.** However, if our dataset is very large, we might not be able to fit it into our computer's memory!

Instead, we can *estimate* a gradient step using some estimator $\widetilde{\nabla} J(\theta)$. This is called ***stochastic*** **gradient descent** (SGD). Ideally, we want this estimator to be **unbiased,** that is, on average, it matches a single true gradient step:

$$\mathbb{E}[\widetilde{\nabla} J(\theta)] = \nabla J(\theta).$$

If J is defined in terms of some training dataset, we might randomly choose a *minibatch* of samples and use them to estimate the prediction error across the *whole* dataset. (This approach is known as ***minibatch* SGD**.)

Notice that our parameters will stop changing once $\nabla J(\theta) = 0$. This implies that our current parameters are 'locally optimal' in some sense; it's impossible to increase the function by moving in any direction. If J is convex, then the only point where this happens is at the *global optimum.* Otherwise, if J is nonconvex, the best we can hope for is a *local optimum.*

We can actually show that in a finite number of steps, SGD will find a θ that is "close" to a local optimum. More formally, suppose we run SGD for T steps, using an unbiased gradient estimator. Let the step size η_t scale as $O(1/\sqrt{t})$. Then if J is bounded and β-smooth, and the norm of the gradient estimator has a finite variance, then after T steps:

$$\|\nabla_\theta J(\theta)\|^2 \leq O\left(M\beta\sigma^2/T\right).$$

In another perspective, the local "landscape" of J around θ becomes flatter and flatter the longer we run SGD.

4.3 REINFORCE and Importance Sampling

Note that the objective function above, $J(\theta) = \mathbb{E}_{\tau\sim\rho_\theta} R(\tau)$, is very difficult to compute! It requires playing out every possible trajectory, which is clearly infeasible for slightly complex state and action spaces. Can we rewrite this in a form that's more convenient to implement? Specifically, suppose there is some distribution, given by a likelihood $\rho(\tau)$, that's easy to sample from (e.g. a database of existing trajectories). We can then rewrite the objective function as follows (all gradients are being taken w.r.t. θ):

$$\begin{aligned}
\nabla J(\theta) &= \nabla \mathop{\mathbb{E}}_{\tau\sim\rho_\theta} R(\tau) \\
&= \nabla \mathop{\mathbb{E}}_{\tau\sim\rho} \frac{\rho_\theta(\tau)}{\rho(\tau)} R(\tau) && \text{likelihood ratio trick} \\
&= \mathop{\mathbb{E}}_{\tau\sim\rho} \frac{\nabla\rho_\theta(\tau)}{\rho(\tau)} R(\tau) && \text{switching gradient and expectation}
\end{aligned}$$

Note that setting $\rho = \rho_\theta$ gives us an alternative form of J that's easier to implement. (Notice the swapped order of ∇ and $\mathbb{E}$!)

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta} [\nabla \log \rho_\theta(\tau) \cdot R(\tau)].$$

Consider expanding out ρ_θ. Note that taking its log turns it into a sum of log terms, of which only the $\pi_\theta(a_t|s_t)$ terms depend on θ, so we can simplify even further to obtain

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right]$$

In fact, we can perform one more simplification. Intuitively, the action at step t does not affect the reward at previous timesteps. You can also show rigorously that this is the case, and that we only need to consider the present and future rewards to calculate the policy gradient:

$$\begin{aligned}\nabla J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1}\nabla_\theta\log\pi_\theta(a_t|s_t)\sum_{t'=t}^{T-1}r(s_{t'},a_{t'})\right]\\ &= \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1}\nabla_\theta\log\pi_\theta(a_t|s_t)Q^{\pi_\theta}(s_t,a_t)\right]\end{aligned}\tag{4.3}$$

Note that in the discounted case, the Q^{π_θ} term must becomes $\lambda^t Q^{\pi_\theta}$. (Make sure this makes sense!) **Exercise:** Prove that this is equivalent to the previous definitions. Also show that this works in the undiscounted case and for infinite horizon.

This expression allows us to estimate the gradient by sampling a few sample trajectories from π_θ, calculating the likelihoods of the chosen actions, and substituting these into the expression above.

For some intuition into how this method works, recall that we update our parameters according to

$$\begin{aligned}\theta_{t+1} &= \theta_t + \nabla J(\theta_t)\\ &= \theta_t + \mathop{\mathbb{E}}_{\tau\sim\rho_{\theta_t}}\nabla\log\rho_{\theta_t}(\tau)\cdot R(\tau).\end{aligned}$$

Consider the “good” trajectories where $R(\tau)$ is large. Then θ gets updated so that these trajectories become more likely. To see why, recall that $\rho_\theta(\tau)$ is the likelihood of the trajectory τ under the policy π_θ, so evaluating the gradient points in the direction that makes τ more likely.

This is an example of **importance sampling:** updating a distribution to put more density on “more important” samples (in this case trajectories).

4.4 Baselines and advantages

A central idea from supervised learning is the bias-variance tradeoff. So far, our method is *unbiased,* meaning that its average is the true policy gradient. Can we find ways to reduce the variance of our estimator as well?

We can instead subtract a **baseline function** $b_t : \mathcal{S} \to \mathbb{R}$ at each timestep t. This modifies the policy gradient as follows:

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{H-1}\nabla\log\pi_\theta(a_t|s_t)\left(\left(\sum_{t'=t}^{H-1}r_t\right)-b_t(s_t)\right)\right].$$

For example, we might want b_t to estimate the average reward-to-go at a given timestep: $b_t^\theta = \mathbb{E}_{\tau\sim\rho_\theta} R_t(\tau)$. This way, the random variable $R_t(\tau) - b_t^\theta$ is centered around zero, making certain algorithms more stable.

As a better baseline, we could instead choose the *value function.* Note that the random variable $Q_t^\pi(s,a) - V_t^\pi(s)$, where the randomness is taken over the actions, is also centered around zero. (Recall $V_t^\pi(s) = \mathbb{E}_{a\sim\pi} Q_t^\pi(s,a)$.) In fact, this quantity has a particular name: the **advantage function.** In a sense, it measures how much better this action does than the average for that policy. We can now alternatively and concisely express the policy gradient as follows. Note that the advantage function effectively replaces the Q-function from Equation 4.3:

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1} \nabla \log \pi_\theta(a_t|s_t) A_t^{\pi_\theta}(s_t,a_t)\right]$$

Additionally, note that for an optimal policy $\pi^\star$, the advantage of a given state-action pair is always nonpositive. (Why?)

4.5 Policy parameterizations

What are some different ways we could parameterize our policy?

If both the state and action spaces are finite, perhaps we could simply learn a preference value $\theta_{s,a}$ for each state-action pair. Then to turn this into a valid distribution, we exponentiate each of them, and divide by the total:

$$\pi_\theta^{\text{softmax}}(a|s) = \frac{\exp(\theta_{s,a})}{\sum_{s,a'} \exp(\theta_{s,a'})}.$$

However, this doesn't preserve any structure in the states or actions. While this is flexible, it is also prone to overfitting.

4.5.1 Linear in features

Instead, what if we map each state-action pair into some **feature space** $\phi(s,a) \in \mathbb{R}^p$? Then, to map a feature vector to a probability, we take a linear combination $\theta \in \mathbb{R}^p$ of the features and take a softmax:

$$\pi_\theta^{\text{linear in features}}(a|s) = \frac{\exp(\theta^\top \phi(s,a))}{\sum_{a'} \exp(\theta^\top \phi(s,a'))}.$$

Another interpretation is that θ represents the feature vector of the "ideal" state-action pair, as state-action pairs whose features align closely with θ are given higher probability.

The score for this parameterization is also quite elegant:

$$\begin{aligned}\nabla \log \pi_\theta(a|s) &= \nabla\left(\theta^\top \phi(s,a) - \log\left(\sum_{a'} \exp(\theta^\top \phi(s,a'))\right)\right)\\ &= \phi(s,a) - \mathop{\mathbb{E}}_{a'\sim\pi_\theta(s)} \phi(s,a')\end{aligned}$$

Plugging this into our policy gradient expression, we get

$$\begin{aligned}
\nabla J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1} \nabla \log \pi_\theta(a_t|s_t) A_t^{\pi_\theta}\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1} \left(\phi(s_t, a_t) - \mathop{\mathbb{E}}_{a'\sim\pi(s_t)} \phi(s_t, a')\right) A_t^{\pi_\theta}(s_t, a_t)\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1} \phi(s_t, a_t) A_t^{\pi_\theta}(s_t, a_t)\right]
\end{aligned}$$

Why can we drop the $\mathbb{E}\,\phi(s_t, a')$ term? By linearity of expectation, consider the dropped term at a single timestep: $\mathbb{E}_{\tau\sim\rho_\theta}\left[\left(\mathbb{E}_{a'\sim\pi(s_t)}\,\phi(s, a')\right) A_t^{\pi_\theta}(s_t, a_t)\right]$. By Adam's Law, we can wrap the advantage term in a conditional expectation on the state s_t. Then we already know that $\mathbb{E}_{a\sim\pi(s)}\,A_t^\pi(s, a) = 0$, and so this entire term vanishes.

4.5.2 Neural policies

More generally, we could map states and actions to unnormalized scores via some parameterized function $f_\theta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, such as a neural network, and choose actions according to a softmax:

$$\pi_\theta^{\text{general}}(a|s) = \frac{\exp(f_\theta(s, a))}{\sum_{a'} \exp(f_\theta(s, a'))}.$$

The score can then be written as

$$\nabla \log \pi_\theta(a|s) = \nabla f_\theta(s, a) - \mathop{\mathbb{E}}_{a\sim\pi_\theta(s)} \nabla f_\theta(s, a')$$