# 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 Policy Gradient Ascent

Let's suppose our policy can be *parameterized* by some parameters $\theta$. For example, these might be the coefficients of a simple linear function $\pi_\theta(s) = \theta^\top s$, or in a high-dimensional case, the weights and biases of a deep neural network.

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

$$\begin{aligned} J(\theta) := \mathop{\mathbb{E}}_{s_0 \sim \mu_0} V^{\pi_\theta}(s_0) = \mathop{\mathbb{E}}_{s_0} \sum_{h=0}^{\infty} \gamma^h r_h \\ \text{where} \quad & r_h = r(s_h, a_h), \\ & s_{h+1} \sim P(s_h, a_h), \\ & a_h = \pi_\theta(s_h). \end{aligned}$$

We call a sequence of states, actions, and rewards a **trajectory,** and the total time-discounted reward is also often called the **return** of a trajectory.

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 $\eta$ 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 $\widetilde{\nabla} J(\theta)$. Ideally, we want this estimator to be unbiased; that is, $\mathbb{E}[\widetilde{\nabla} J(\theta)] = \nabla J(\theta)$. Updating using these estimators is called *stochastic* gradient descent (SGD). Continuing the example above, we might randomly choose a *minibatch* of samples from our dataset and use them to estimate the prediction error across the whole dataset. (This approach is known as *minibatch SGD.*)

You might 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.* (We'll prove this below.) Otherwise, if $J$ is nonconvex, the best we can hope for is a *local optimum.*

## 4.3 asdf