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 (the expected total reward)? 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, or even intractable, to compute exactly! This is because it involves taking an expectation over all possible trajectories τ. Can we rewrite this in a form that's more convenient to implement?

Specifically, suppose there is some distribution over trajectories $\rho(\tau)$ that's easy to sample from (e.g. a database of existing trajectories). We can then rewrite the gradient of objective function, a.k.a. the *policy gradient*, as follows (all gradients are being taken w.r.t. θ):

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

Note that setting $\rho = \rho_\theta$ allows us to express ∇J as an expectation. (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 the following expression for the policy gradient, known as the "REINFORCE" policy gradient:

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

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.

In fact, we can perform one more simplification. Intuitively, the action taken at step t does not affect the reward from previous timesteps, since they're already in the past! 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:

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

Exercise: Prove that this is equivalent to the previous definitions. What modification to the expression must be made for the discounted, infinite-horizon setting?

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:

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{H-1} \nabla \log \pi_\theta(a_t|s_t)\left(\left(\sum_{t'=t}^{H-1} r_t\right) - b_t(s_t)\right)\right]. \tag{4.4}$$

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.** This measures how much better this action does than the average for that policy. (Note that for an optimal policy $\pi^\star$, the advantage of a given state-action pair is always nonpositive.)

We can now express the policy gradient as follows. Note that the advantage function effectively replaces the Q-function from Equation 4.3:

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1} \nabla \log \pi_\theta(a_t|s_t) A_t^{\pi_\theta}(s_t,a_t)\right]. \tag{4.5}$$

Note that to avoid correlations between the gradient estimator and the value estimator (i.e. baseline), we must estimate them with independently sampled trajectories:

Definition 4.4.1: Policy gradient with a learned baseline

Require: Learning rate $\eta_0, \ldots, \eta_{K-1}$
Require: Initialization θ^0
for $k = 0, \ldots, K-1$ **do**
Sample N trajectories from π_{θ^k} to estimate a baseline $\widetilde{b}$ such that $\widetilde{b}_h(s) \approx V_h^{\theta^k}(s)$
Sample M trajectories $\tau_0, \ldots, \tau_{M-1} \sim \rho_{\theta^k}$
Compute the policy gradient estimate

$$\widetilde{\nabla}_\theta J(\theta^k) = \frac{1}{M}\sum_{m=0}^{M-1}\sum_{h=0}^{H-1} \nabla \log \pi_{\theta^k}(a_h \mid s_h)(R_h(\tau_m) - \widetilde{b}_h(s_h))$$

Gradient ascent update $\theta^{k+1} \leftarrow \theta^k + \widetilde{\nabla}_\theta J(\theta^k)$
end for

The baseline estimation step can be done using any appropriate supervised learning algorithm. Note that the gradient estimator will be unbiased regardless of the baseline.

4.5 Example 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 perform a "softmax" operation: 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 make use of any structure in the states or actions, so 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 function 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

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

4.5.3 Continuous action spaces

Consider a continuous n-dimensional action space $\mathcal{A} = \mathbb{R}^n$. Then for a stochastic policy, we could use a function to predict the *mean* action and then add some random noise about it. For example, we could use a neural network to predict the mean action $\mu_\theta(s)$ and then add some noise $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$ to it:

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

Exercise: Can you extend the "linear in features" policy to continuous action spaces in a similar way?

4.6 Local policy optimization

4.6.1 Motivation for policy gradient

Recall the policy iteration algorithm discussed in the MDP section: We alternate between these two steps:

- Estimating the Q-function of the current policy
- Updating the policy to be greedy w.r.t. this approximate Q-function.

(Note that we could equivalently estimate the advantage function.)

What advantages does the policy gradient algorithm have over policy iteration? Both policy gradient and policy iteration are iterative algorithms.

To analyze the difference between them, we'll make use of the **performance difference lemma**.

Theorem 4.6.1: Performance difference lemma

Let $\rho_{\pi,s}$ denote the distribution induced by the policy π over trajectories starting in state s.

Given two policies $\pi, \widetilde{pi}$, the PDL allows us to express the difference between their value functions as follows:

$$V_0^{\widetilde{\pi}}(s) - V_0^{\pi}(s) = \mathop{\mathbb{E}}_{\tau \sim \rho_{\widetilde{\pi},s}} \left[\sum_{h=0}^{H-1} A_h^{\pi}(s_h, a_h) \right] \tag{4.6}$$

Some intuition: Recall that $A_h^{\pi}(s,a)$ tells us how much better the action a is in state s than average, supposing actions are chosen according to π. How much better is $\widetilde{\pi}$ than π? To answer this, we break down the trajectory step-by-step. At each step, we compute how much better actions from $\widetilde{\pi}$ are than the actions from π. But this is exactly the average π-advantage, where the expectation is taken over actions from $\widetilde{\pi}$. This is exactly what the PDL describes.

Let's analyze why fitted approaches such as PI don't work as well in the RL setting. To start, let's ask, where *do* fitted approaches work well? They are commonly seen in *supervised learning*, where a prediction rule is fit using some labelled training set, and then assessed on a test set from the same distribution. Does this assumption still hold when doing PI?

Let's consider a single iteration of PI. Suppose the new policy $\widetilde{\pi}$ chooses some action with a negative advantage w.r.t. π. Define $\Delta_\infty = \min_{s \in \mathcal{S}} A_h^{\pi}(s, \widetilde{\pi}(s))$. If this is negative, then the PDL shows that there may exist some state s and time h such that

$$V_h^{\widetilde{\pi}}(s) \geq V_h^{\pi}(s) - H \cdot |\Delta_\infty|.$$

In general, PI cannot avoid particularly bad situations where the new policy $\widetilde{\pi}$ often visits these bad states, causing an actual degradation. It does not enforce that the trajectory distributions ρ_π and $\rho_{\widetilde{\pi}}$ be close to each other. In other words, the "training distribution" that our prediction rule is fitted on, ρ_π, may differ significantly from the "evaluation distribution" $\rho_{\widetilde{\pi}}$ — we must address this issue of *distributional shift*.

How can we enforce that the *trajectory distributions* do not change much at each step? In fact, policy gradient already does this to a small extent: Supposing that the mapping from parameters to trajectory distributions is relatively smooth, then, by adjusting the parameters a small distance from the current iterate, we end up at a new policy with a similar trajectory distribution. But this is not very rigorous, and in practice the parameter-to-distribution mapping may not be smooth. Can we constrain the distance between the resulting distributions more explicitly? This brings us to the next two methods: **trust region policy optimization** (TRPO) and the **natural policy gradient** (NPG).

4.6.2 Trust region policy optimization

TRPO is another iterative algorithm for policy optimization. It is similar to policy iteration, except we constrain the updated policy to be "close to" the current policy in terms of the trajectory distributions they induce.

To formalize "close to", we typically use the **Kullback-Leibler divergence (KLD)**:

Definition 4.6.1: Kullback-Leibler divergence

For two PDFs p, q,

$$\mathrm{KL}\left(p \parallel q\right) := \mathop{\mathbb{E}}_{x\sim p}\left[\log \frac{p(x)}{q(x)}\right] \tag{4.7}$$

This can be interpreted in many different ways, many stemming from information theory. Note that $\mathrm{KL}\left(p \parallel q\right) = 0$ if and only if $p = q$. Also note that it is generally not symmetric.

Additionally, rather than estimating the Q-function of the current policy, we can use the RHS of the Performance Difference Lemma (4.6.1) as our optimization target.

Definition 4.6.2: Trust region policy optimization (exact)

Require: Trust region radius δ
  Initialize θ^0
  for $k = 0, \ldots, K-1$ **do**
    $\theta^{k+1} \leftarrow \arg\max_\theta \mathbb{E}_{s_0,\ldots,s_{H-1}\sim\pi^k}\left[\sum_h \mathbb{E}_{a_h\sim\pi_\theta(s_h)} A^{\pi^k}(s_h, a_h)\right]$ ▷ See below
    where $\mathrm{KL}\left(\rho_{\pi^k} \parallel \rho_{\pi_\theta}\right) \le \delta$
  end for
  return π^K

Note that the objective function is not identical to the r.h.s. of the Performance Difference Lemma. Here, we still use the *states* sampled from the old policy, and only use the *actions* from the new policy. This is because it would be computationally infeasible to sample entire trajectories from π_θ as we are optimizing over θ. This approximation is also reasonable in the sense that it matches the r.h.s. of the Performance Difference Lemma to first order in θ. (We will elaborate more on this later.)

Both the objective function and the KLD constraint involve a weighted average over the space of all trajectories. This is intractable in general, so we need to estimate the expectation. As before, we can do this by taking an empirical average over samples from the trajectory distribution. However, the inner expectation over $a_h \sim \pi_\theta$ involves the optimizing variable θ, and we'd like an expression that has a closed form in terms of θ to make optimization tractable. Otherwise, we'd need to resample many times each time we made an update to θ. To address this, we'll use a common technique known as **importance sampling**.

Definition 4.6.3: Importance sampling

Suppose we want to estimate $\mathbb{E}_{x\sim\widetilde{p}}[f(x)]$. However, $\widetilde{p}$ is difficult to sample from, so we can't take an empirical average directly. Instead, there is some other distribution p that is easier to sample from, e.g. we could draw samples from an existing dataset, as in the case of **offline RL**.

Then note that

$$\mathop{\mathbb{E}}_{x\sim\widetilde{p}}[f(x)] = \mathop{\mathbb{E}}_{x\sim p}\left[\frac{\widetilde{p}(x)}{p(x)}f(x)\right]$$

so, given i.i.d. samples $x_0, \dots, x_{N-1} \sim p$, we can construct an unbiased estimate of $\mathbb{E}_{x\sim\widetilde{p}}[f(x)]$ by *reweighting* these samples according to the likelihood ratio $\widetilde{p}(x)/p(x)$:

$$\frac{1}{N}\sum_{n=0}^{N-1}\frac{\widetilde{p}(x_n)}{p(x_n)}f(x_n)$$

Doesn't this seem too good to be true? If there were no drawbacks, we could use this to estimate *any* expectation of any function on any arbitrary distribution! The drawback is that the variance may be very large due to the likelihood ratio term. If the sampling distribution p assigns low probability to any region where $\widetilde{p}$ assigns high probability, then the likelihood ratio will be very large and cause the variance to blow up.

Applying importance sampling allows us to estimate the TRPO objective as follows:

Definition 4.6.4: Trust region policy optimization (implementation)

Initialize θ^0
for $k = 0, \dots, K-1$ **do**
Sample N trajectories from ρ^k to learn a value estimator $\widetilde{b}_h(s) \approx V_h^{\pi^k}(s)$
Sample M trajectories $\tau_0, \dots, \tau_{M-1} \sim \rho^k$

$$\theta^{k+1} \leftarrow \arg\max_\theta \frac{1}{M}\sum_{m=0}^{M-1}\sum_{h=0}^{H-1}\frac{\pi_\theta(a_h \mid s_h)}{\pi^k(a_h \mid s_h)}[R_h(\tau_m) - \widetilde{b}_h(s_h)]$$
$$\text{where } \sum_{m=0}^{M-1}\sum_{h=0}^{H-1}\log\frac{\pi_k(a_h^m \mid s_h^m)}{\pi_\theta(a_h^m \mid s_h^m)} \le \delta$$

end for

4.6.3 Natural policy gradient

Instead, we can solve an approximation to the TRPO optimization problem. This will link us back to the policy gradient from before. We take a first-order approximation to the objective function and a second-order approximation to the KLD constraint. This results in the optimization problem

$$\begin{aligned}&\max_\theta \nabla_\theta J(\pi_{\theta^k})^\top(\theta - \theta^k)\\ &\text{where } \frac{1}{2}(\theta-\theta^k)^\top F_{\theta^k}(\theta-\theta^k) \le \delta\end{aligned} \tag{4.8}$$

where F_{θ^k} is the **Fisher information matrix** defined below.

Definition 4.6.5: Fisher information matrix

Let p_θ denote a parameterized distribution. Its Fisher information matrix F_θ can be defined equivalently as:

$$\begin{aligned} F_\theta &= \mathop{\mathbb{E}}_{x\sim p_\theta}\left[(\nabla_\theta \log p_\theta(x))(\nabla_\theta \log p_\theta(x))^\top\right] && \text{covariance matrix of the Fisher score} \\ &= \mathop{\mathbb{E}}_{x\sim p_\theta}\left[-\nabla_\theta^2 \log p_\theta(x)\right] && \text{average Hessian of the negative log-likelihood} \end{aligned}$$

Recall that the Hessian of a function describes its curvature: That is, for a vector $\delta \in \Theta$, the quantity $\delta^\top F_\theta \delta$ describes how rapidly the negative log-likelihood changes if we move by δ.

In particular, when $p_\theta = \rho_\theta$ denotes a trajectory distribution, we can further simplify the expression:

$$F_\theta = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{h=0}^{H-1}(\nabla \log \pi_\theta(a_h \mid s_h))(\nabla \log \pi_\theta(a_h \mid s_h))^\top\right] \tag{4.9}$$

Note that we've used the Markov property to cancel out the cross terms corresponding to two different time steps.

This is a convex optimization problem, and so we can find the global optima by setting the gradient of the Lagrangian to zero:

$$\begin{aligned} \mathcal{L}(\theta,\eta) &= \nabla_\theta J(\pi_{\theta^k})^\top(\theta-\theta^k) - \eta\left[\frac{1}{2}(\theta-\theta^k)^\top F_{\theta^k}(\theta-\theta^k) - \delta\right] \\ \nabla_\theta \mathcal{L}(\theta^{k+1},\eta) &= 0 \\ \nabla_\theta J(\pi_{\theta^k}) &= \eta F_{\theta^k}(\theta^{k+1}-\theta^k) \\ \theta^{k+1} &= \theta^k + \eta F_{\theta^k}^{-1}\nabla_\theta J(\pi_{\theta^k}) \\ \text{where } \eta &= \sqrt{\frac{\delta}{\nabla_\theta J(\pi_{\theta^k})^\top F_{\theta^k}\nabla_\theta J(\pi_{\theta^k})}} \end{aligned}$$

Definition 4.6.6: Natural policy gradient

Require: Learning rate $\eta > 0$
  Initialize θ^0
  for $k = 0, \ldots, K-1$ **do**
    Estimate the policy gradient $\hat{g} \approx \nabla_\theta J(\pi_{\theta^k})$ ▷ See (4.5)
    Estimate the Fisher information matrix $\hat{F} \approx F_{\theta^k}$ ▷ See (4.9)
    $\theta^{k+1} \leftarrow \theta^k + \eta\hat{F}^{-1}\hat{g}$ ▷ Natural gradient update
  end for

How many trajectory samples do we need to accurately estimate the Fisher information

matrix? As a rule of thumb, the sample complexity should scale with the dimension of the parameter space. This makes this approach intractable in the deep learning setting where we might have a very large number of parameters.

For some intuition: The typical gradient descent algorithm treats the parameter space as "flat", treating the objective function as some black box value. However, in the case here where the parameters map to a *distribution*, using the natural gradient update is equivalent to optimizing over distribution space rather than distribution space.

Example 4.6.1: Natural gradient on a simple problem

Let's step away from reinforcement learning specifically and consider the following optimization problem over Bernoulli distributions $\pi \in \Delta(\{0, 1\})$:

$$J(\pi) = 100 \cdot \pi(1) + 1 \cdot \pi(0)$$

Clearly the optimal distribution is the constant one $\pi(1) = 1$. Suppose we optimize over the parameterized family $\pi_\theta(1) = \frac{\exp(\theta)}{1+\exp(\theta)}$. Then our optimization algorithm should set θ to be unboundedly large. Then the vanilla gradient is

$$\nabla_\theta J(\pi_\theta) = \frac{99 \exp(\theta)}{(1 + \exp(\theta))^2}.$$

Note that as $\theta \to \infty$ that the increments get closer and closer to 0. However, if we compute the Fisher information scalar

$$\begin{aligned} F_\theta &= \mathop{\mathbb{E}}_{x \sim \pi_\theta}[(\nabla_\theta \log \pi_\theta(x))^2] \\ &= \frac{\exp(\theta)}{(1 + \exp(\theta))^2} \end{aligned}$$

resulting in the natural gradient update

$$\begin{aligned} \theta^{k+1} &= \theta^k + \eta F_{\theta^k}^{-1} \nabla_\theta J(\theta^k) \\ &= \theta^k + 99\eta \end{aligned}$$

which increases at a constant rate, i.e. improves the objective more quickly than vanilla gradient ascent.

4.6.4 Proximal policy optimization

Can we improve on the computational efficiency of the above methods?

We can relax the TRPO objective in a different way: Rather than imposing a hard constraint on the KL distance, we can instead impose a *soft* constraint by incorporating it into the objective:

Definition 4.6.7: Proximal policy optimization (exact)

Require: Regularization parameter λ
Initialize θ^0
for $k = 0, \dots, K-1$ **do**
$\theta^{k+1} \leftarrow \arg\max_\theta \mathbb{E}_{s_0,\dots,s_{H-1}\sim\pi^k}\left[\sum_h \mathbb{E}_{a_h\sim\pi_\theta(s_h)} A^{\pi^k}(s_h, a_h)\right] - \lambda \mathrm{KL}\left(\rho_{\pi^k} \parallel \rho_{\pi_\theta}\right)$
end for
return θ^K

Note that like the original TRPO algorithm 4.6.2, PPO is not gradient-based; rather, at each step, we try to maximize local advantage relative to the current policy.

Let us now turn this into an implementable algorithm, assuming we can sample trajectories from π_{θ^k}.

Let us simplify the $\mathrm{KL}\left(\rho_{\pi^k} \parallel \rho_{\pi_\theta}\right)$ term first. Expanding gives

$$\begin{aligned}
\mathrm{KL}\left(\rho_{\pi^k} \parallel \rho_{\pi_\theta}\right) &= \mathop{\mathbb{E}}_{\tau\sim\rho_{\pi^k}}\left[\log \frac{\rho_{\pi^k}(\tau)}{\rho_{\pi_\theta}(\tau)}\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\rho_{\pi^k}}\left[\sum_{h=0}^{H-1} \log \frac{\pi^k(a_h \mid s_h)}{\pi_\theta(a_h \mid s_h)}\right] && \text{state transitions cancel} \\
&= \mathop{\mathbb{E}}_{\tau\sim\rho_{\pi^k}}\left[\sum_{h=0}^{H-1} \log \frac{1}{\pi_\theta(a_h \mid s_h)}\right] + c
\end{aligned}$$

where c is some constant relative to θ.

As we did for TRPO (4.6.4), we can use importance sampling (4.6.3) to rewrite the inner expectation. Combining the expectations together, this gives the (exact) objective

$$\max_\theta \mathop{\mathbb{E}}_{\tau\sim\rho_{\pi^k}}\left[\sum_{h=0}^{H-1}\left(\frac{\pi_\theta(a_h \mid s_h)}{\pi^k(a_h \mid s_h)} A^{\pi^k}(s_h, a_h) - \lambda \log \frac{1}{\pi_\theta(a_h \mid s_h)}\right)\right]$$

Now we can use gradient ascent on the parameters θ until convergence to maximize this function, completing a single iteration of PPO (i.e. $\theta^{k+1} \leftarrow \theta$).