11
Policy Gradients

In this lecture, we will continue to consider the problem of directly learning a policy including from sampled trajectories. We will focus on *policy gradient methods* that use samples from the environment to get noisy gradient estimates and then update policy. Policy gradient methods take advantage of one important structure black box methods do not: the fact that we can design our policy space such that we know the relationship between the parameters of that policy space and the output actions. That is, the policy search problem need not be *entirely* a black-box operation since even without a model of the environment or cost functions as we still can have a model of how our policy space works. [1]

To take advantage of this approach, we need a method to relate the parameters of the policy class with the resulting actions. One of the simplest methods to do so is computing derivatives: for sophisticated policy classes this is often best done through automatic differentiation techniques. We begin here by reviewing the most common automatic differentiation technique used in the learning literature commonly known as backpropagation or *reverse-mode automatic differentiation*.

We demonstrate how to use this in a larger loop of policy optimization later in the lecture. Before specifying the details of this approach, we will review back-propagation and its use in neural networks and controls.

11.1 Back-propagation

A powerful way to describe many complex systems is as a composition of interconnected modules described as a *directed graph*[2]. This makes it easier to organize a complex system and debug the system by unit testing individual components. For example, robotics often leverages a "sense, act, plan" paradigm, where each component is often studied and optimized separately. However, modifying a single module can influence the overall system performance in a complicated way due to the relationship between modules. Back-propagation attempts to address this problem by offering a principled method to calculate the cascaded effects of module parameters on overall system performance. Back-propagation is also known as *the adjoint method* and *reverse-mode automatic differentiation* in the control and optimization literature.

Back-propagation makes it possible to solve a large class of problems that would be intractable using naive differentiation techniques or the use of

[1] At least one of the authors, based on disappointing experimental evidence, had largely despaired of this advantage translating into a reduction in the amount of interaction required over naive blackbox methods "A major open issue within the field is the relative merits of the these two approaches: in principle, white box methods leverage more information, but with the exception of models, the performance gains are traded-off with additional assumptions that may be violated and less mature optimization algorithms. Some recent work ... suggest that much of the benefit of policy search is achieved by black-box methods." (Kober, Peters, Bagnell, 14). In recent years, as policy classes have become more sophisticated (i.e. deep CNN based policies) with very large parameter sets, the benefit of this additional structure has become important and the methods described in this chapter have at times become preferred to black box search.

[2] Often called the *computation graph* in the learning literature

forward mode auto-differentiation. One of the best known applications is training neural networks, which has led to dramatic results in computer vision and natural language processing. [3] A common misunderstanding is that back-propagation is specific to machine learning/neural network (*aka deep network*) training. In fact, back-propagation can be used to compute gradients for any differentiable function expressed as a graph of operations and this general dynamic programming strategy for computing derivatives is quite old. In particular, the same idea has been used widely in optimal control, known as *the adjoint method*[4]. Just as back-propagation's led to tremendous success in training neural nets, the adjoint method has enabled researchers to tackle complex control problems with millions of control inputs. An elegant example is the work "Fluid Control with the Adjoint Method" [1], where the simulation of a human-shaped smoke cloud required over one million control inputs. Naive derivative computation with this number of parameters is nearly intractable, while backpropagation allows it to scale to real time animation. As the backpropagation technique has become better understood and more ubiquitous, we've entered a period of *differentiable programming* [5] where we can assemble sophisticated programs and their derivatives to enable optimization of these programs.

[3]

[4] A fine summary of the adjoint method can be found here: http://www.argmin.net/2016/05/18/mates-of-costate/

[5] https://en.wikipedia.org/wiki/Differentiable_programming

We will first look at back-propagation as a general algorithm to compute gradients, then we will see several examples including multi-layer neural networks and the LQR problem. An excellent reference on the origins and general backpropagation technique is [6]. The book *Deep Learning* [7] provides a fine introduction in section 6.5.

[6]

[7]

Total vs. Partial Derivatives

In dealing with compositions of functions, a crucial distinction must be made between two types of derivatives, **total derivatives** and **partial derivatives**. The partial derivative of a function describes the change in output resulting from a change in direct dependencies– i.e. a module has a set of inputs and we evaluate how the output of the module changes in terms of these inputs. The total derivative of a function describes the change of the output resulting from *all* dependencies, direct and indirect. For instance, in a control problem, a module describing the result of dynamics at time time t, x_{t+1} may have no direct dependence on a control u_{t-5} at time step $t-5$ and hence has *partial derivative* of 0. However, there is a potentially non-0 *total derivative* of that output in terms of u_{t-5}, as this control effects the output, albeit indirectly.

In a sense, *partial derivatives* are "syntactic" and *total derivates* are semantic, representing the complete effect of varying a single parameter or input on a resulting computation.

The Chain Rule

In other terms, the partial derivative does not account for the composition but rather direct inputs, while the total derivative does. Backpropagation is effectively a dynamic programming means to turn manually specified partial derivatives into automatic computation of total derivatives.

Before diving in and solving more sophisticated problems using back-propagation, let's review some basic calculus starting with the chain rule of calculus. First, let us consider the simplest case where $x \in \mathbb{R}$ is a real num-

ber. Let f and g be two differentiable functions that map $\mathbb{R}$ to $\mathbb{R}$. Suppose that $y = g(x)$ and $z = f(y) = f(g(x))$. Then, the chain rule tells us,

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{11.1.1}$$

The chain rule can further generalized to the case when $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ are vectors[8]. Let $f : \mathbb{R}^m \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$ be two differentiable functions. As before, suppose that $z = f(g(x))$. Then, we have,

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{11.1.2}$$

In vector notation, we rewrite the above equation as,

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y z, \tag{11.1.3}$$

where $\nabla_x z = [\frac{\partial z}{\partial x_1}, \dots \frac{\partial z}{\partial x_n}]^\top$ and $\nabla_y z = [\frac{\partial z}{\partial y_1}, \dots \frac{\partial z}{\partial y_m}]^\top$ are the *gradient* of z with respect to x and y, respectively, and

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \dots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \dots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \dots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

is the *Jacobian matrix* of the function g.

[8] In fact, the chain rule can be generalized to the case of "tensors". The use of this phrase in *deep learning* doesn't imply the geometric meaning of mathematics, but rather simply refers to a multi-dimensional array of numbers. See

Block Diagrams

Now that we are equipped with the necessary mathematical tools to compute gradients, let us take one step further and look at how we can represent how the modules are interconnected in a system using a *block diagram*.

In the language of block diagram, each *module* or *operation* is represented by a block, whereas the arrows between blocks indicate variables that are inputs to/outputs of the operations. For example, the system considered in the previous section can be represented as Figure 11.1.1.

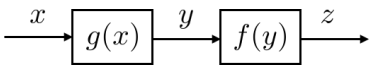

Figure 11.1.1: The block diagram representation of the simple example.

Given a block diagram and a variable x in the diagram, we say a variable y is a *parent*[9] of x if there exists a block f such that x is the output of f and y one of the inputs. Note that a variable may have multiple parents since there can be multiple inputs to block f. We denote the set of variables that are parents of x as $Parents(x)$. Conversely, we call a variable y as a *child* of x if x is a parent of y, i.e., there exists a block g such that y is the output of g and x is one of the inputs. We denote the set of variables that are children of x as $Children(x)$.

[9] Here we abuse the definition of parents by denoting an "edge" as a parent of another "edge" in the diagram. Same for children.

We say a block diagram is *acyclic* if it has no cyclic paths. For back-propagation, we assume that the associated block diagram is acyclic[10], and there exists a topological ordering (over variables) such that the output of the system is the last one in the list. In our case, we assume that the output of

[10] Recurrent neural networks and closed loop control systems (with a finite horizon) can be represented by an acyclic diagram through an operation called *unfold*. We will discuss it later.

the system is a scalar $J \in \mathbb{R}$. It could be the value of the loss function if we are training a neural network, it can also be the total cost of the trajectory(ies) if we are optimizing a policy.

Recall that we are interested how the output J is changed when we change a variable x in the diagram, which is precisely the gradient $\nabla_x J$. By the chain rule, we have,

$$\nabla_x J = \sum_{y \in Children(x)} \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y J \tag{11.1.4}$$

Examples

To make things more concrete, let us look at some examples.

- *Linear*

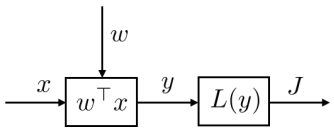

Figure 11.1.2: The block diagram of the linear module.

A *linear module* takes two inputs x and w to produce output $y = f(x, w) = w^\top x$. Assume that the system is associated with an overall output $J = L(y)$. Then, we have,

$$\left(\frac{\partial y}{\partial x}\right)^\top = w, \qquad \left(\frac{\partial y}{\partial w}\right)^\top = x \tag{11.1.5}$$

$$\nabla_x J = \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y J = \frac{dL}{dy} w \tag{11.1.6}$$

$$\nabla_w J = \left(\frac{\partial y}{\partial w}\right)^\top \nabla_y J = \frac{dL}{dy} x \tag{11.1.7}$$

- *Squared Loss*

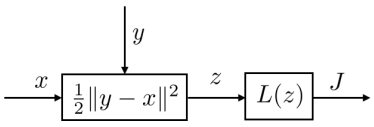

Figure 11.1.3: The block diagram of the squared loss module.

A *squared loss module* takes two inputs x and y and produces output $z = f(x, y) = \frac{1}{2}(y - x)^\top (y - x) = \frac{1}{2}\|y - x\|^2$. Assume that the system is associated with an overall output $J = L(z)$. Then, we have,

$$\left(\frac{\partial z}{\partial x}\right)^\top = x - y, \qquad \left(\frac{\partial z}{\partial y}\right)^\top = y - x \tag{11.1.8}$$

$$\nabla_x J = \left(\frac{\partial z}{\partial x}\right)^\top \nabla_z J = \frac{dL}{dz}(x - y) \tag{11.1.9}$$

$$\nabla_y J = \left(\frac{\partial z}{\partial y}\right)^\top \nabla_z J = \frac{dL}{dz}(y - x) \tag{11.1.10}$$

- *Branch*

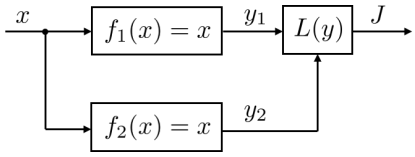

Figure 11.1.4: The block diagram of the branch module.

A *branch module* takes in one input x and produces two outputs $y_1 = f_1(x) = x$ and $y_2 = f_2(x) = x$. Assume that the system is associated with an overall output $J = L(y_1, y_2)$. Then, we have,

$$\left(\frac{\partial y_1}{\partial x}\right)^\top = \left(\frac{\partial y_2}{\partial x}\right)^\top = I \tag{11.1.11}$$

$$\nabla_x J = \left(\frac{\partial y_1}{\partial x}\right)^\top \nabla_{y_1} J + \left(\frac{\partial y_2}{\partial y}\right)^\top \nabla_{y_2} J = \nabla_{y_1} J + \nabla_{y_2} J \tag{11.1.12}$$

- *Addition*

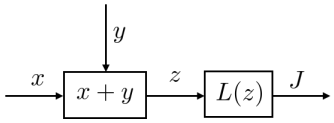

Figure 11.1.5: The block diagram of the plus module.

An *addition module* takes in two inputs x and y and produces output $z = f(x, y) = x + y$. Again, assume that the system is associated with an overall output $J = L(z)$. Then, we have,

$$\left(\frac{\partial z}{\partial x}\right)^\top = \left(\frac{\partial z}{\partial y}\right)^\top = I \tag{11.1.13}$$

$$\nabla_x J = \left(\frac{\partial z}{\partial x}\right)^\top \nabla_z J = \nabla_z J \tag{11.1.14}$$

$$\nabla_y J = \left(\frac{\partial z}{\partial y}\right)^\top \nabla_z J = \nabla_z J \tag{11.1.15}$$

Back-propagation: A Dynamic Programming Algorithm

Although given any variable x in the diagram, we can calculate the gradient of the output J with respect to the variable x by recursively applying the chain rule. However, when we are training a neural network or solving an optimal control problem, we oftentimes want to compute the gradient with respect to *a large set of variables*, such as weights in every layer of the neural network, or the control input at every time step. The question then becomes, can we do something better than calculating the gradients one by one? The answer is yes!

To see this, let us look back at the linear module example. When we calculate the $\nabla_x J$ and $\nabla_w J$ in (11.1.6) and (11.1.7), we actually use the value of $\nabla_y J$ for multiple times. Therefore, if we can somehow *store* the previously calculated gradients, and order the variables in such a way that we can make use of the gradients computed previously, then we can save a lot of computation by *reusing* these gradients. This idea of dynamic programming is the main idea behind *back-propagation*.

Recall from the previous part that the gradient with respect to a variable x can be computed based on the gradient with respect to all its children $y \in Children(x)$,

$$\nabla_x J = \sum_{y \in Children(x)} \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y J.$$

Based on this observation, we see that in order to reuse the previously computed gradient, we need to order the variables *backwards* – from the output to the inputs, from the parents to the children. Then, we need to backward *propagate* the gradients from the children to the parents, this is where the name *back-propagation* comes from. [11]

[11] This dynamic programming ordering might remind you a bit of *value iteration* from the earliest lectures. It should! If you think of the output of the final module as the value function, backpropagation is simply doing value iteration with a local, linear approximation of the value function. As such, it's essentially value-iteration in disguise.

The "Learning" Algorithm

Now, let us try to do something useful with the back-propagation algorithm. Assume that there are a set of input variables in the diagram called *parameters* that we are free to choose. We denote these parameters as $\{w_i\}_i$. Examples of these parameters include weights in the neural networks, control inputs and initial conditions, etc. Conversely, there are other input variables whose values are given and we have no control over, such as the inputs to the neural network, the system dynamics, etc. Our goal is to find a set of *parameters* such that the value of some scalar output J is minimized (loss for training neural networks, cost for optimal control problems, etc), i.e.,

$$\{w_i^*\}_i = \arg\min_{\{w_i\}_i} J \tag{11.1.16}$$

We are interested in designing a learning algorithm that updates parameters of a system to reduce the value of J. One way to perform the gradient descent algorithm,

$$w_i^{k+1} = w_i^k - \alpha \nabla_{w_i} J. \tag{11.1.17}$$

where $\alpha > 0$ is the learning rate. Note that here the gradients can be calculated by the back-propagation algorithm.

In summary, there are three main steps in the learning algorithm: *forward-propagation*, *back-propagation*, and *gradient descent*. Forward propagation consists of generating all module outputs by running the system "forward" (from the inputs to the output). This is necessary for recursively evaluating all the partial derivatives in the back propagation step, as detailed in the previous section. Finally, once all gradients have been calculated, we take a gradient descent step. Then we repeat the whole process until convergence. [12]

[12] More sophisticated algorithms than gradient descent are at times used that automatically scale individual directions or apply approximations to a second order method. See for more details.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org

11.2 System Examples

Below are examples of modular systems where back-propagation can be used.

Linear Regression Example

We can describe a linear regression by a linear module cascaded with a squared loss module, as shown below. Linear module takes two inputs x

and w, and output $z = w^\top x$. Squared loss module takes inputs z and y, and output $J = \frac{1}{2}(z-y)^\top(z-y)$. The system takes inputs x, y, and w, where x corresponds to the data, y are the respective regression targets, and w is the regression parameter we can control. Our goal is to minimize the loss z. Back-propagation is usually not used here because it is not difficult to calculate the total derivatives directly.

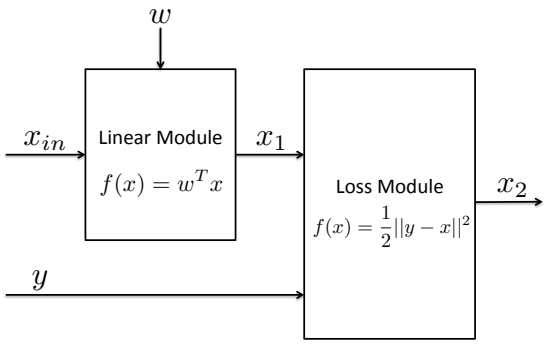

Figure 11.2.1: Linear regression represented as a cascade of modules.

Neural Networks

A neural network consists of layered linear modules and nonlinear firing units. Traditionally, the firing units are sigmoid functions such as hyperbolic tangent or the logistic function. Recently, the nonlinear rectifier function shown below has come into common practice. The sigmoid functions have small linear support regions between saturation, which require the inputs to be scaled properly. The rectifier does not suffer from these issues, and is computationally simpler, allowing for large neural networks to be applied to a variety of data.

In deep, multi-layer networks, cascading makes it difficult to directly determine total derivatives for all the parameters. Utilizing the back-propagation algorithm, however, we can efficiently tune the linear module weight parameters.

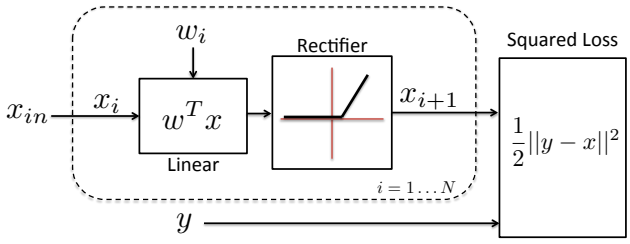

Figure 11.2.2: A neural net.

Let J be the output of the squared loss function. Then, we have,

$$\nabla_{x_{N+1}} J = x_{N+1} - y. \tag{11.2.1}$$

By the chain rule, for any $i = 1, \ldots, N$, we have,

$$\nabla_{x_i} J = \left(\frac{\partial x_{i+1}}{\partial x_i} \right)^\top \nabla_{x_{i+1}} J \tag{11.2.2}$$

$$\nabla_{w_i} J = \left(\frac{\partial x_{i+1}}{\partial w_i} \right)^\top \nabla_{x_{i+1}} J \tag{11.2.3}$$

We can use these relations to recursively calculate all the gradients of our system using only partial derivatives and *back propogating* gradients from later modules in the system. This process begins at the output.

Note that there are numerous variations on neural network architectures and update algorithms for domain-specific applications. Variations include pooling, probabilistic drop-out, autoregressive loss, and convolution layers, etc.

11.3 Relating LQR and Backprogation

By now, we have seen a few "backwards" algorithms in this class, the back-propagation algorithm that we just saw and the value-iteration/Riccatti recursion used for the LQR problem. One natural question one may ask is whether there are some connections between these. In fact, one can find multiple connections.

DDP, Model-based optimization and second-order backpropagation

Perhaps the most natural *policy gradient* approach is to consider optimizing the parameters of a policy where we can completely specify the dynamics and cost function as modules in a computation graph as well. In this complete, model-based case (similar to that of the LQR setting), we can use gradient descent to optimize parameters of a policy.

If we apply backpropagation to such a chain of modules (rather than a general Directed Acyclic Graph), backprogation can be understood as making a *linear* approximation of a value function. In this viewpoint, DDP can be understood as making a second order approximation of the value function. One can develop more sophisticated variants of DDP/iLQR that work on general directed graphs that can be seen as second order generalizations of backpropagation. [13].

[13]

Rederiving LQR with Back-propagation

Another connection is that we can think about the Ricatti back-up equation as coming about from following the same dynamic programming computational pattern as backpropagation, but propagating *analytic* derivatives.

Recall from the earlier lecture that the LQR problem is stated as the following,

$$\min_{u_0, \ldots, u_{T-1}} \quad \sum_{t=0}^{T-1} \left({x_t}^\top Q x_t + {u_t}^\top R u_t \right) \tag{11.3.1}$$

$$\text{s.t.} \quad x_{t+1} = A x_t + B u_t, \quad \forall t = 0, \ldots, T-2 \tag{11.3.2}$$

where $x_{t+1} = A x_t + B u_t$ is the system dynamics, and ${x_t}^\top Q x_t + {u_t}^\top R u_t$ is the instantaneous cost at each time step.

First, let us rewrite the LQR problem into a block diagram. The block diagram of the LQR problem is shown in Figure 11.3.1. Here we introduce a quadratic cost module at each time step and aggregate them into a total cost J.

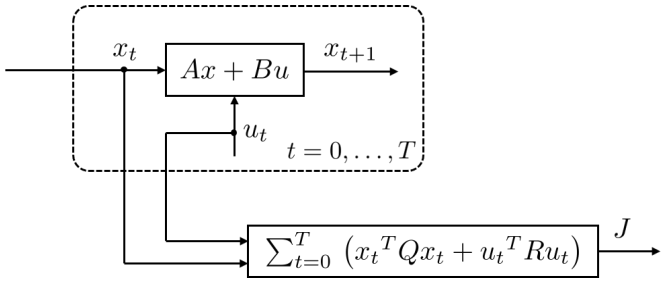

Figure 11.3.1: Finite horizon LQR realized by a block diagram.

First, we have,

$$\nabla_{u_{T-1}} J = 2R\, u_{T-1}, \tag{11.3.3}$$
$$\nabla_{x_{T-1}} J = 2Q\, x_{T-1}. \tag{11.3.4}$$

By the chain rule, for any $t = 0, \ldots, T-2$, we have,

$$\begin{aligned} \nabla_{x_t} J &= \left(\frac{\partial J}{\partial x_t}\right)^\top + \left(\frac{\partial x_{t+1}}{\partial x_t}\right)^\top \nabla_{x_{t+1}} J \\ &= 2Q\, x_t + A^\top \nabla_{x_{t+1}} J \end{aligned} \tag{11.3.5}$$

$$\begin{aligned} \nabla_{u_t} J &= \left(\frac{\partial J}{\partial u_t}\right)^\top + \left(\frac{\partial x_{t+1}}{\partial u_t}\right)^\top \nabla_{x_{t+1}} J \\ &= 2R\, u_t + B^\top \nabla_{x_{t+1}} J \end{aligned} \tag{11.3.6}$$

With the gradient we get from back-propagation, one can certainly run gradient descent for a set of controls $\{u_t\}_{t=0}^{T-1}$. The gradient descent process does not require a matrix inversion as we saw earlier, but as a cost, it requires possibly many gradient descent steps and does not provide a control policy but rather optimizes an open-loop trajectory. This can also be viewed as a policy search approach to the LQR problem.

Note, however, that we can also *solve* for optimal input using these gradients since we know that the problem is convex and we have a closed-form expression of those gradients – we can just set the gradients to zero!

- At time step $T-1$, by setting $\nabla_{u_{T-1}} J = 0$, we have,

$$2R\, u_{T-1} = 0 \qquad \Rightarrow \qquad u_{T-1} = 0. \tag{11.3.7}$$

Let $V_{T-1} = Q$, we have,

$$\nabla_{x_{T-1}} J = 2Q\, x_{T-1} \doteq 2V_{T-1}\, x_{T-1}. \tag{11.3.8}$$

- At time step $T-2$, we have,

$$\begin{aligned} \nabla_{u_{T-2}} J &= 2R\, u_{T-2} + B^\top \nabla_{x_{T-1}} J \\ &= 2R\, u_{T-2} + 2B^\top V_{T-1}\, x_{T-1} \\ &= 2R\, u_{T-2} + 2B^\top V_{T-1} (A\, x_{T-2} + B\, u_{T-2}) \\ &= 2(R + B^\top V_{T-1}\, B)\, u_{T-2} + 2B^\top V_{T-1}\, A\, x_{T-2} \end{aligned} \tag{11.3.9}$$

By setting $\nabla_{u_{T-2}} J = 0$, we have,

$$u_{T-2} = -(R + B^\top V_{T-1} B)^{-1} B^\top V_{T-1} A x_{T-2} \doteq K_{T-2} x_{T-2}. \tag{11.3.10}$$

Meanwhile,

$$\begin{aligned}
\nabla_{x_{T-2}} J &= 2Q x_{T-2} + 2A^\top \nabla_{x_{T-1}} J \\
&= 2Q x_{T-2} + 2A^\top V_{T-1} (A + B K_{T-2}) x_{T-2} \\
&= 2(Q + (A + B K_{T-2})^\top V_{T-1} (A + B K_{T-2}) \\
&\qquad - K_{T-2}^\top B^\top V_{T-1} (A + B K_{T-2})) x_{T-2} \\
&= 2(Q + (A + B K_{T-2})^\top V_{T-1} (A + B K_{T-2}) + K_{T-2}^\top R K_{T-2}) x_{T-2} \\
&\doteq 2V_{T-2} x_{T-2}.
\end{aligned} \tag{11.3.11}$$

- By repeating the process, we get,

$$\begin{aligned}
K_{t-1} &= -(R + B^\top V_t B)^{-1} B^\top V_t A \\
V_{t-1} &= Q + (A + B K_{t-1})^\top V_t (A + B K_{t-1}) + K_{t-1}^\top R_t K_{t-1}
\end{aligned} \tag{11.3.12}$$

This is precisely the Riccati equation!

11.4 Policy Gradient Methods

In the standard RL setting, we do not have access to differentiable modules that describe the dynamics (and often don't have access to the cost function in this form either). Instead, we can sample trajectories, or perhaps have access to a somewhat richer sample based model as described in earlier lectures. Value function methods like Q-learning and SARSA use information from *every* transition (s, a, r, s') in every trajectory, while black-box policy optimization methods only look at the total reward of the trajectories ignoring all structure to the reinforcement learning problem. It's natural to ask if we can use more structure without the difficulties of value function approximation.

As we have seen earlier in the lecture, if the environment model and the reward function are *known*, we can compute the policy gradient conveniently using the back-propagation algorithm. However, in reinforcement learning, we often care about the case when we don't have access to the environment model and/or the reward function. *Policy gradient methods* seek to estimate the policy gradients from trajectories without access to the environment model and the reward function.

Before we dive in to the details, we should consider whether a gradient exists for a certain policy class. This can be interpreted as a continuity condition of the mapping from the parameters in the policy class to the trajectories. This is clearly false for discrete action spaces and deterministic policies, since an infinitesimally small change in parameters can drastically change the policy and hence the trajectories. Therefore, in this lecture, we consider a class of stochastic policies parameterized by θ, $\pi_\theta : s \mapsto \pi_\theta(a|s)$. Under mild assumptions about the environment, we can safely assume that the policy gradient always exists for this policy class since stochastic policies "smooth out" the problem. [14]

Let ξ denote a *trajectory* of states and actions, $\xi = (s_0, a_0, \ldots, s_{T-1}, a_{T-1})$. We define the *total reward of the trajectory* ξ as,

$$R(\xi) = \sum_{t=0}^{T-1} r(s_t, a_t).$$

Our goal is to find the parameters that produce a policy that maximizes the expected total reward of the trajectories,

$$J(\theta) = E_{p(\xi|\theta)}[R(\xi)] = E_{p(\xi|\theta)}\left[\sum_{t=0}^{T-1} r(s_t, a_t)\right],$$

where $p(\xi|\theta)$ is the probability of the trajectory ξ given the policy parameterized by θ, which, we will see later, is also dependent on the transition model of the environment.

To find the optimal policy, we compute the policy gradient by taking the derivative with respect to θ.

$$\begin{aligned}\nabla_\theta J &= \nabla_\theta E_{p(\xi|\theta)}\left[R(\xi)\right] \\ &= \nabla_\theta \sum_{\xi \in \Xi} p(\xi|\theta)\, R(\xi),\end{aligned}$$

where Ξ denotes the set of all possible trajectories. In the case when the state and/or action space is continuous, the sum should be replaced by an

[14] The general strategy of lifting from a discrete space to a distribution to ensure continuity is used throughout machine learning and optimization. Consider, [Arora et al., 2012] as an excellent introduction to the exponentiated gradient approach to solving problems.

integral. The derivation will remain the same for integrals, although some steps would require additional justification[15].

[15] For example, the dominated convergence theorem needs to be invoked in order to swap the integral with the gradient operator in the next step.

Since $R(\xi)$ is the total reward of a *given* trajectory ξ, it has no dependence on θ. Therefore,

$$\nabla_\theta J = \sum_{\xi \in \Xi} \left(\nabla_\theta p(\xi|\theta)\right) R(\xi). \tag{11.4.1}$$

However, we cannot compute the gradient with eq. (11.4.1) because it requires us to evaluate the gradient for *all* possible trajectories. Instead, we want to obtain at least an estimate of the policy gradient using *samples* of trajectories. Therefore, we want to express the gradient as an *expectation* over probability $p(\xi|\theta)$ – the moment we do that, we can use the *law of large numbers* to draw samples from the distribution and estimate the expectation. Therefore, we use a simple trick,

$$\begin{aligned}\nabla_\theta J &= \sum_{\xi \in \Xi} \frac{p(\xi|\theta)}{p(\xi|\theta)} \left(\nabla_\theta p(\xi|\theta)\right) R(\xi) \\ &= E_{p(\xi|\theta)} \left[\frac{\nabla_\theta p(\xi|\theta)}{p(\xi|\theta)} R(\xi)\right].\end{aligned}$$

By the chain rule, we have, $\nabla_\theta \log\left(p(\xi|\theta)\right) = \frac{\nabla_\theta p(\xi|\theta)}{p(\xi|\theta)}$. So, we have an elegant expression of the policy gradient as an expectation,

$$\nabla_\theta J = E_{p(\xi|\theta)} \left[\nabla_\theta \log\left(p(\xi|\theta)\right) R(\xi)\right]. \tag{11.4.2}$$

This is sometimes called the *likelihood ratio policy gradient*. The likelihood ratio policy gradient can be interpreted as increasing the (log) probability of the trajectories with high reward and decreasing the (log) probability of the trajectories with low reward. To see this, consider a single trajectory ξ. Imagine that $R(\xi)$ is a large positive number, then if we do gradient ascent with respect to the total reward J, we are in some sense doing *gradient ascent* with respect to $\log\left(p(\xi|\theta)\right)$ according to eq. (11.4.2). Conversely, if $R(\xi)$ is a large negative number, we are performing *gradient descent* with respect to its log probability in some sense.

Note, however, that we still can not compute the policy gradient using the above equation because it requires us to evaluate $\nabla_\theta \log p(\xi|\theta)$ in the expectation, yet we do *not* know the transition model $p(s_{t+1}|a_t, s_t)$.

However, we will see that it is not a problem for policy gradient methods. If we assume the Markov property, we have,

$$p(\xi|\theta) = p(s_0) \left(\prod_{t=0}^{T-2} p(s_{t+1}|a_t, s_t)\right) \left(\prod_{t=0}^{T-1} \pi_\theta(a_t|s_t)\right).$$

Then, we have,

$$\begin{aligned}\nabla_\theta \log p(\xi|\theta) = \nabla_\theta \log p(s_0) &+ \left(\sum_{t=0}^{T-2} \nabla_\theta \log p(s_{t+1}|a_t, s_t)\right) \\ &+ \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\right).\end{aligned}$$

However, $\log p(s_0)$ and $\log p\left(s_{t+1}|s_t, a_t\right)$ do not depend on θ, so the gradients with respect to these terms are zero. Hence,

$$\nabla_\theta J = E_{p(\xi|\theta)} \left[\left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta\left(a_t|s_t\right)\right) R(\xi)\right].$$

Notice that we don't know and can't control the system dynamics, but by formulating the problem this way, we don't need to – we have control over the policy class we choose, and thus can easily compute an unbiased gradient estimate. [16] For example, we can use the *back-propagation* algorithm that we saw last week to compute the gradient $\nabla_\theta \log \pi_\theta(a_t|s_t)$.

[16] Often with outrageously high sample variance however.

As mentioned earlier, we can now use the law of large numbers to estimate this expectation,

$$\widetilde{\nabla}_\theta J = \frac{1}{N}\sum_{i=1}^{N}\left[\left(\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta\left(a_t^{(i)}|s_t^{(i)}\right)\right) R(\xi^{(i)})\right]. \tag{11.4.3}$$

By the law of large number, we know that the estimated gradient in eq. (11.4.3) is an *unbiased* estimate of the true policy gradient. Therefore, we can run *stochastic gradient ascent* with this estimated gradient. This forms the basis of the REINFORCE (Algorithm 20) algorithm (version 1, we will show some improvements soon).

Algorithm 20: The REINFORCE algorithm.

Start with an arbitrary initial policy π_θ
while *not converged* **do**
  Run simulator with π_θ to collect $\{\xi^{(i)}\}_{i=1}^N$
  Compute estimated gradient

$$\widetilde{\nabla}_\theta J = \frac{1}{N}\sum_{i=1}^{N}\left[\left(\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta\left(a_t^{(i)}|s_t^{(i)}\right)\right) R(\xi^{(i)})\right]$$

  Update parameters $\theta \leftarrow \theta + \alpha\, \widetilde{\nabla}_\theta J$
return π_θ

In step 1, we run the simulator using the current policy to collect training sequences. In step 2, we approximate the expectation by the sample mean. Step 3 is the update rule of the algorithm with α being the step size. The algorithm is then repeated until convergence or until you are bored.

An example: Tetris

We will use Tetris as an example to show how you might choose your policy function $\pi_\theta(a|s)$ and how you would compute $\nabla_\theta \log \pi_\theta(a|s)$. Suppose we have some features representing the state-action pair of the Tetris game. For instance f_1 =the number of "holes" after the placement, f_2 =the height of the highest column after the placement, etc. Due to the log in eq. (11.4.3), a convenient stochastic policy is,

$$\pi_\theta(a|s) = \frac{\exp\left(\theta^\top f(s,a)\right)}{\sum_{a'}\exp\left(\theta^\top f(s,a')\right)}.$$

This is sometimes called the Boltzmann distribution or Gibbs distribution.

The gradient of the probability distribution can be computed by any method, e.g. using back-propagation. However, it is fairly simple to solve

analytically:

$$\begin{aligned}\nabla_\theta \log \pi_\theta(a|s) &= \nabla_\theta \left[\theta^\top f(s,a) - \log \sum_{a'} \exp\left(\theta^\top f(s,a')\right)\right] \\ &= f(s,a) - \frac{\sum_{a'} f(s,a') \exp\left(\theta^\top f(s,a')\right)}{\sum_{a'} \exp\left(\theta^\top f(s,a')\right)} \\ &= f(s,a) - \sum_{a'} f(s,a')\, \pi_\theta\left(a'|s\right) \\ &= f(s,a) - E_{\pi_\theta(a'|s)}\left[f(s,a')\right]\end{aligned} \tag{11.4.4}$$

This is essentially computing the difference between the feature at state s and action a versus the expectation over all actions for that state that we could have chosen, in a way the "average" feature. Assume that we observe that feature i for action a is *larger* than the average over all actions. According to eq. (11.4.4), if performing action a at state s produces a trajectory that has high reward, we will *increase* the value of θ_i to *upweight* this particular feature. Because it seems that this feature is "helpful" for getting high rewards. On the other hand, if this state-action pair produces low reward trajectories, we may conclude that feature i is "harmful". So we make the corresponding parameter θ_i to be small or negative to reflect this observation.

11.5 *Reducing Variance*

Although the estimated gradient in eq. (11.4.3) can in theory provide an unbiased estimate, it suffers from *high variance*. In order to see this, recall that the likelihood ratio policy gradient increases the probability of the trajectories with high reward and decreases the probability of the trajectories with low reward. However, imagine when *every* trajectory has a very high reward – although some are higher than others. Then, since we only has finite number of samples at each iteration, the estimated gradient will push the probability of all these trajectories higher (if possible) since the total reward is high (and hence make the probability of other trajectories lower). However, the algorithm has no idea about the reward of trajectories *compared to other trajectories*. Therefore, we can imagine that the estimated gradients are pointing in different directions at each iteration. In fact, without making the modifications introduced in this part, the REINFORCE algorithm performs poorly compared to "black-box" approaches.

One simple modification to reduce the variance is to take advantage of causality – the actions selected now cannot affect past rewards.

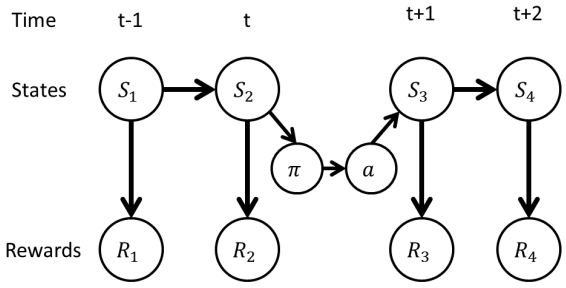

Figure 11.5.1: A trajectory of states, actions and rewards. We consider changing the action at time t in order to get a better expected future reward.

If we consider a trajectory of states and rewards, we want to change the action at time t to maximize expected reward. Intuitively, we know that changing the action at time t cannot affect the rewards obtained in the past, since we have already received them. Thus, we can represent our expected reward as only the future reward.

$$\begin{aligned}\nabla_\theta J &= E_{p(\xi|\theta)}\left[\sum_{t=0}^{T-1}\left(\nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=0}^{t-1} r(s_{t'}, a_{t'}) + \sum_{t'=t}^{T-1} r(s_{t'}, a_{t'})\right)\right)\right] \\ &= E_{p(\xi|\theta)}\left[\sum_{t=0}^{T-1}\left(\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=t}^{T-1} r(s_{t'}, a_{t'})\right)\right],\end{aligned} \tag{11.5.1}$$

where $\sum_{t'=t}^{T-1} r(s_{t'}, a_{t'})$ is sometimes called *future reward* or *reward-to-go*. We can use this idea to remove the dependence of past rewards from the calculation of our gradient.

One can reduce the variance even further by introducing *baselines* for the expected total rewards. Recall that one of the reasons for the high variance is that the algorithm does not know how well the trajectories perform *compared to other trajectories*. Therefore, by introducing a *baseline* for the total reward (or reward to go), we can update the policy based on how well the policy performs compared to a baseline. The variance can hopefully be reduced if the baseline approximates the average performance of the trajectories. But how do we know that whether the estimated gradient still makes sense?

Let's first take a look at the expectation $E_{p(\xi|\theta)}[\nabla_\theta \log p(\xi|\theta) b]$. We have,

$$\begin{aligned}E_{p(\xi|\theta)}[\nabla_\theta \log p(\xi|\theta)\, b] &= \sum_{\xi \in \Xi} \nabla_\theta\, p(\xi|\theta)\, b \\ &= \nabla_\theta \left(\sum_{\xi \in \Xi} p(\xi|\theta)\right) b \\ &= (\nabla_\theta 1)\; b = 0.\end{aligned} \tag{11.5.2}$$

Therefore, the estimated policy is still unbiased if we introduce a baseline for the total reward (or reward to go). Note here that the above equation holds as long as b does not depend on θ, hence b can potentially be a function of the state, i.e. $b = b(s_t)$.[17] In fact, a common choice of baseline is the value function or some estimate of the value function.

[17] However, some additional effort is needed to show that a time-dependent baseline actually works, including expanding $p(\xi|\theta)$ in the expectation as a product of the transition probability and the policy.

Putting everything together, we can generate another policy gradient expression,

$$\nabla_\theta J = E_{p(\xi|\theta)}\left[\sum_{t=0}^{T-1}\left(\nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T-1} r(s_{t'}, a_{t'}) - b(s_t)\right)\right)\right], \tag{11.5.3}$$

We estimate the above policy gradient as

$$\widetilde{\nabla}_\theta J = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1}\left(\nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})\left(\sum_{t'=t}^{T-1} r(s_{t'}^{(i)}, a_{t'}^{(i)}) - b(s_t^{(i)})\right)\right). \tag{11.5.4}$$

This can give us an unbiased estimate of the policy gradients with lower variance.

11.6 Eligibility Traces

Conveniently, the approach described above can be effectively implemented with a simple infinite impulse response filter, rather than by remembering entire trajectories. To lighten notation, consider the case when no baselines are introduced, i.e. $b \equiv 0$.

Given a trajectory, we can introduce an iteratively computed *eligibility vector*,

$$e_t = e_{t-1} + \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Note then that,

$$e_t \cdot r(s_t, a_t) = \sum_{t'=0}^{t} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'})\, r(s_t, a_t).$$

We will see that the gradient then is just a running sum of the expected future rewards over all visited states at each time-step,

$$\Delta_t = \Delta_{t-1} + e_t \cdot r(s_t, a_t).$$

If we expand this out, we can see that it is the same as gradient calculated by the likelihood ratio method.

$$\begin{aligned}\Delta_t = \nabla_\theta \log \pi_\theta(a_0|s_0)\, r(s_0, a_0) + \sum_{t=0}^{1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, r(s_1, a_1) + \\ \cdots + \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, r(s_T, a_T) \\ = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T-1} r(s_{t'}, a_{t'})\end{aligned}$$

11.7 REINFORCE

The REINFORCE algorithm uses the eligibility trace to calculate the gradient update. We start off with a set or parameters and several trajectories gathered by forward simulating the policy generated by those parameters. We can then use the eligibility trace to calculate the gradient and get a new set of parameters. We add a discount factor, γ, to manage the general class of infinite horizon discounted problems.

Algorithm 21: REINFORCE Algorithm

1: $e = 0$
2: $\Delta = 0$
3: **for all** t **do**
4: $\quad e \leftarrow \gamma\, e + \nabla_\theta \log \pi_\theta(a_t|s_t)$
5: $\quad \Delta \leftarrow \Delta + \frac{1}{t+1}\left[r(s_t, a_t) \cdot e - \Delta\right]$
6: **end for**

The resulting Δ_{t+1} is a noisy estimate of the gradient. We can either compute Δ_{t+1} several times to get a less noisy estimate, or we can move a small amount using the noisy estimate.

11.8 The Policy Gradient Theorem

The REINFORCE algorithm calculates the gradient using expected future reward as determined by a trajectory.

$$\nabla_\theta J = E_{p(\xi|\theta)} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T-1} r(s_{t'}, a_{t'}) \right]$$

We can instead replace the the estimate of future reward $\sum_{t'=t}^{T-1} r(s_{t'}, a_{t'})$ with the action value Q^{π_θ}, which by definition gives us the expected future reward.

$$\nabla_\theta J = E_{p(\xi|\theta)} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, Q^{\pi_\theta}(s_t, a_t) \right]$$

We can update the gradient rule to take the expectation over the distribution of *states* rather than the expectation over the *trajectories*, this leads to the *Policy Gradient Theorem*.

$$\nabla_\theta J = E_{s\sim d^{\pi_\theta}(s), a\sim\pi_\theta(a|s)} \left[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \right] \tag{11.8.1}$$

Here, $d^{\pi_\theta}(s)$ is the distribution of states under policy π_θ, i.e., the fraction of time spent in state s,

$$d^{\pi_\theta}(s) = \frac{1}{T} \sum_{t=0}^{T-1} p^{\pi_\theta}(s, t),$$

where $p^{\pi_\theta}(s,t)$ is the probability that state s is visited at step t under policy π_θ.

The policy gradient theorem states that the gradient of average reward under a policy π_θ parametrized by θ is given by

$$\nabla_\theta J = E_{d^{\pi_\theta}(s)} E_{\pi_\theta(a|s)} \left[\nabla_\theta \log(\pi_\theta(a|s)\, Q^{\pi_\theta}(s,a) \right] \tag{11.8.2}$$

The expectations are with respect to the distribution $d^{\pi_\theta}(s)$ of states given a policy π_θ and the actions taken under the policy π_θ given the state s. We can prove that, for the value function $V^{\pi_\theta}(s)$ is only a function of the state s, it can viewed as a *baseline* as we saw above. Thus, Eq. 11.8.2 is equal to:

$$\nabla_\theta J = E_{d^{\pi_\theta}(s)} E_{\pi_\theta(a|s)} \left[\nabla_\theta \log(\pi_\theta(a|s)\, (Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)) \right], \tag{11.8.3}$$

where $A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$ is referred to as the *advantage* of action a at state s under policy π_θ. So why is this true?First, consider the inner expectation. Because V^{π_θ} does not depend on a, this is equivalent to,

$$E_{\pi_\theta(a|s)} \left[\nabla_\theta \log(\pi_\theta(a|s) V^{\pi_\theta}(s) \right] = V^{\pi_\theta}(s)\, E_{\pi_\theta(a|s)} \left[\nabla_\theta \log(\pi_\theta(a|s)) \right]. \tag{11.8.4}$$

That leaves $\nabla_\theta \log(\pi_\theta(a|s)$ in the expectation. Intuitively that must be equal to zero because the probability distribution π_θ must sum to one, so the sum over all changes must be equal to zero. We show more explicitly below that this is indeed the case. We expand (Eq. 11.8.4) into sums over the states

and actions. We can show that,

$$\begin{aligned} E_{\pi_\theta(a|s)}\left[\nabla_\theta \log(\pi_\theta(a|s)\right] &= \sum_{a\in\mathbb{A}} \pi_\theta(a|s)\, \nabla_\theta \log(\pi_\theta(a|s)) \\ &= \sum_{a\in\mathbb{A}} \pi_\theta(a|s)\, \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\ &= \sum_{a\in\mathbb{A}} \nabla_\theta \pi_\theta(a|s) \\ &= \nabla_\theta \left(\sum_{a\in\mathbb{A}} \pi_\theta(a|s) \right) = \nabla_\theta 1 = 0. \end{aligned} \tag{11.8.5}$$

Through linearity of expectation, we have,

$$\begin{aligned} & E_{d^{\pi_\theta}(s)} E_{\pi_\theta(a|s)}\left[\nabla_\theta \log(\pi_\theta(a|s) V^{\pi_\theta}(s)\right] \\ =& E_{d^{\pi_\theta}(s)}\left[V^{\pi_\theta}(s)\, E_{\pi_\theta(a|s)}\left[\nabla_\theta \log(\pi_\theta(a|s)\right]\right] \\ =& E_{d^{\pi_\theta}(s)}\left[V^{\pi_\theta}(s)\cdot 0\right] = 0. \end{aligned} \tag{11.8.6}$$

Finally,

$$\begin{aligned} \nabla_\theta J &= E_{d^{\pi_\theta}(s)} E_{\pi_\theta(a|s)}\left[\nabla_\theta \log(\pi_\theta(a|s) Q^{\pi_\theta}(s,a)\right] \\ &= E_{d^{\pi_\theta}(s)} E_{\pi_\theta(a|s)}\left[\nabla_\theta \log(\pi_\theta(a|s)\, (Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s))\right] \\ &= E_{d^{\pi_\theta}(s)} E_{\pi_\theta(a|s)}\left[\nabla_\theta \log(\pi_\theta(a|s) A^{\pi_\theta}(s,a)\right] \end{aligned} \tag{11.8.7}$$

Intuitively, this shows that the algorithm wants the advantage of the action to be high, and wants to choose actions that are correlated with the advantage being high. It adjusts the policy by making small changes towards Q values that are higher than the average.

The policy gradient theorem connects estimating the gradient $\nabla_\theta J$ with estimating Q^{π_θ} or A^{π_θ}. For example,we can estimate Q^{π_θ} with some parameterized function $Q_\phi^{\pi_\theta}$ using *Approximate Dynamic Programming* methods like Fitted Q-Iteration or an advantage estimator $A_\phi^{\pi_\theta}$, to approximate the advantage function $A^{\pi_\theta}(s,a)$. Also, we can use samples trajectories under policy π_θ to estimate the expectation in Eq. (11.8.7), which results in an estimated policy gradient,

$$\widetilde{\nabla}_\theta J = \frac{1}{N} \sum_{i=1}^{N} \left(\nabla_\theta \log \pi_\theta(a_i|s_i)\, A_\phi^{\pi_\theta}(s_i,a_i) \right). \tag{11.8.8}$$

This leads to a class of methods called *Actor–Critic Methods*. Actor–Critic methods learn a *actor* (the policy) and a *critic* simultaneously. The critic produces the estimate of some value function (e.g., state-value function, action-value function, advantage function, etc.) for bootstrapping (updating the value function estimate for a state from the estimated values of other states). By introducing the critic, the variance of the gradient estimate can be further reduced. Many popular policy gradient algorithms, including TRPO, PPO and DDPG, adopt the actor–critic architecture.

Examples

Let us consider a simple example of the actor-critic algorithm. Say we have two actions that we can take from a given state and one feature for the state s. One of our actions a_0 is bad, while the other one a_1 is good.

We use the Boltzmann distribution that we have seen in the previous example,

$$\pi_\theta(a|s) = \frac{\exp[\theta^\top f(s,a)]}{\sum_{a'} \exp[\theta^\top f(s,a)]}.$$

Suppose that the features of our state and the two actions are $f(s,a_0) = 3$ and $f(s,a_1) = 1$.

Let's say our current value of the parameter θ is $\theta = 1$. Then, the probabilities for taking each action are,

$$\pi_\theta(a_0|s) = \frac{\exp[\theta^\top f(s,a_0)]}{\exp[\theta^\top f(s,a_0)] + \exp[\theta^\top f(s,a_1)]} = \frac{e^3}{e^3+e} = \frac{e^2}{e^2+1} \approx 0.88,$$

$$\pi_\theta(a_1|s) = \frac{\exp[\theta^\top f(s,a_1)]}{\exp[\theta^\top f(s,a_0)] + \exp[\theta^\top f(s,a_1)]} = \frac{e}{e^3+e} = \frac{1}{e^2+1} \approx 0.12,$$

where $e \approx 2.71828$ is the base of the natural logarithm

We then get an estimate of the future reward, possibly through our critic: $Q^\pi(s,a_0) = 1$ and $Q^\pi(s,a_1) = 100$.

We have already seen previously that we can compute the derivative of the log probability as follows:

$$\nabla_\theta \log \pi_\theta(a|s) = f(s,a) - E_{\pi_\theta(a'|s)}[f(s,a')],$$

where,

$$E_{\pi(a'|s)}[f(s,a')] \approx 0.88 \times 3 + 0.12 \times 1 = 2.76.$$

We can just compute the gradient estimate[18],

$$\begin{aligned}
\widetilde{\nabla}_\theta J &= E_{a\sim\pi_\theta(a|s)}\left[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)\right] \\
&= \pi_\theta(a_0|s)\nabla_\theta \log \pi_\theta(a_0|s) Q^{\pi_\theta}(s,a_0) \\
&\qquad + \pi_\theta(a_1|s)\nabla_\theta \log \pi_\theta(a_1|s) Q^{\pi_\theta}(s,a_1) \\
&\approx 0.88 \times (3 - 2.76) \times 1 + 0.12 \times (1 - 2.76) \times 100 \\
&\approx -20.79
\end{aligned}$$

[18] Note that this is an estimate because we are not taking expectation over state

Thus the policy gradient algorithm tells us to decrease the value of θ since the higher feature value seems to result in lower future reward. This makes the probability of choosing a_1 at s higher than the previous iteration.

11.9 Highly Correlated Features

Gradient ascent/descent methods depend greatly on the *parameterization* of the policy. To see this, consider the two parameterizations of Tetris.

Parameterization 1: $f_1 =$ # of Holes after the placement, $f_2 =$ Height after the placement. We use θ to denote the parameter for this parameterization.

Parameterization 2: $g_1 = \ldots = g_{100} =$ # of Holes after the placement, $g_{101} =$ Height after the placement. We use ϕ to denote the parameter for this parameterization

Then, for Parameterization 1, we have,

$$\theta^\top f(x,a) = \theta_1 \times \#\text{ of Holes}(x,a) + \theta_2 \times \text{Height}(x,a).$$

While for Parameterization 2, we have,

$$\phi^\top g = \left(\sum_{i=1}^{100} \phi_i\right) \times \#\text{ of Holes}(x,a) + \phi_{101} \times \text{Height}(x,a).$$

When we take the policy gradient, we have,

$$\nabla_{\theta_i} J = E_{p(\xi|\theta)}\left[\sum_{t=0}^{T-1}\left(f_i(s,a) - E_{\pi_\theta(a'|s)}[f_i(s,a')]\right) Q^{\pi_\theta}(s_t,a_t)\right]$$
$$\nabla_{\phi_i} J = E_{p(\xi|\phi)}\left[\sum_{t=0}^{T-1}\left(g_i(s,a) - E_{\pi_\phi(a'|s)}[g_i(s,a')]\right) Q^{\pi_\phi}(s_t,a_t)\right]$$

Hence, we have $\nabla_{\phi_1} J = \ldots = \nabla_{\phi_{100}} J = \nabla_{\theta_1} J$. The policy gradient algorithm takes a 100 times larger step for the *actual weight* corresponding to the number of holes using Parametrization 2 than in Parametrization 1!

Gradient ascent (or steepest ascent) poses the problem of finding $\max_{\Delta\theta} J(\theta + \Delta\theta)$ such that $\delta\theta$ is small. Gradient ascent measures "small" as the l_2 norm $\|\Delta\theta\|_2 = \sqrt{\sum_i (\Delta\theta_i)^2} \leq \epsilon$. However this version of measuring "small" depends on the parameterization of our policy. Ideally, we want the descent to measure "small" based on changes in our policy and not depend on the parameterization of the policy. We will address this problem in the next lecture.

11.10 Natural Policy Gradient

In the general formulation of steepest descent, as given by Eq. (11.10.1), there are many size metrics that can be utilized.

$$\max_{\Delta\theta} J(\theta + \Delta\theta) \qquad s.t. \qquad \|\Delta\theta\| \leq \epsilon \tag{11.10.1}$$

The gradient descent algorithm comes about when we choose the metric $\|\cdot\|$ to be the l_2 norm over the parameters ($\sqrt{\Delta\theta^\top \Delta\theta}$). In policy gradient methods such as REINFORCE, this definition of the metric can cause the algorithm to fail, if utilizing highly correlated features. This is due to the fact that the l_2 norm defines a "small" change in the gradient direction as depending on the cumulative sum in parameter change, which may have varying degrees of correlation with actual policy change. Instead, we would like to define the size metric such that the notion of "small" encompasses changes in the parameterized *policy*, not simply the changes in the parameters themselves. This leads to two questions.

Q1) What does steepest descent look like given other metrics?

Q2) What metric captures the fact that we would like our metric to be tied to the difference between the $\pi_\theta(a|s)$ and $\pi_{\theta+\Delta\theta}(a|s)$, and not just θ and $\theta + \Delta\theta$?

Q1 – What does steepest descent look like under other metrics?

For small changes in the parameters, we can think of the metric as some quadratic function of the parameters, as evidenced by the Taylor expansion. The steepest descent optimization problem then becomes

$$\max_{\Delta\theta} J(\theta + \Delta\theta) \qquad s.t. \qquad \Delta\theta^\top G(\theta)\Delta\theta \leq \epsilon \tag{11.10.2}$$

where $G(\theta)$ defines the specific metric. In general, G is a distance metric and thus is symmetric positive semi-definite[19]. This matrix defines the notion of distance in the parameter space locally around θ and, in some cases, can be constant; if this is true, the metric is referred to as flat. Intuitively, a flat metric entails that distance is measured the same everywhere in the parameter space. While a flat metric can be helpful, in the general case it will not accurately capture the true notion of distance on the parameter manifold.

[19] Being pedantic, it is actually a pseudo-metric if it has nontrivial nullspace

However, we don't always want to use flat metrics because it does not always precisely reflect what does "small" means in our particular situations. For example, one change of parameters $\Delta\theta$ at θ_1 can result in a very minor change of our policy, while the same $\Delta\theta$ can result in a large change at θ_2. We want our metric $G(\theta)$ to reflect that.

We can solve this new optimization problem (Eq. (11.10.2)) for the parameters using the technique of Lagrange multipliers. This converts the constrained optimization problem (11.10.2) to unconstrained optimization problem with respect to the *Lagrangian* of the system,

$$\max_{\Delta\theta} \mathcal{L}(\Delta\theta, \lambda) = J(\theta + \Delta\theta) - \lambda \left[\Delta\theta^\top G(\theta)\Delta\theta - \epsilon\right], \tag{11.10.3}$$

where $\lambda \geq 0$ is the *Lagrange multiplier*.

The theory says that there exists a choice of $\lambda \geq 0$ such that the constraint optimization problem (11.10.2) and the unconstrained optimization problem

(11.10.3) has the same solution. To begin with, we could use only the direction and simple take λ to be a fixed scalar and solve for $\Delta\theta$. However, it's been demonstrated in practice that parameterizing in terms of ϵ is actually a good way to control step-size. Note that this is straightforward to compute since we can simply normalize $\delta theta$ to ϵ norm in the G metric (that is, it's easy to explicitly compute the correct lagrange multiplier)! [20]

[20] This is the dominant benefit of the "TRPO" method over naive implementations of the natural gradient.

Because we are only considering small steps in $\Delta\theta$, we can approximate (11.10.3) by using the first-order Taylor expansion of J:

$$\mathcal{L}(\Delta\theta, \lambda) \approx J(\theta) + \Delta\theta^\top \nabla_\theta J - \lambda \left[\Delta\theta^\top G(\theta)\Delta\theta - \epsilon\right] \doteq \tilde{\mathcal{L}}_\lambda(\Delta\theta). \tag{11.10.4}$$

Here we use the notation $\tilde{\mathcal{L}}_\lambda(\Delta\theta)$ to emphasize that we are taking λ as a constant and hence $\tilde{\mathcal{L}}_\lambda$ is a function of $\Delta\theta$.

Note that the approximated Lagrangian $\tilde{\mathcal{L}}_\lambda$ is quadratic in $\Delta\theta$. To find the solution, we can simply take the partial derivative of the approximated Lagrangian $\tilde{\mathcal{L}}_\lambda$ with respect to the change in parameters and set it to zero:

$$\frac{\partial \tilde{\mathcal{L}}_\lambda}{\partial \Delta\theta} = \nabla_\theta J - 2\,\lambda\, G(\theta)\,\Delta\theta = 0. \tag{11.10.5}$$

If $G(\theta)$ is nonsingular, the solution to the above equation is thus:

$$\Delta\theta = \frac{1}{2\,\lambda}\, G^{-1}(\theta)\, \nabla_\theta J. \tag{11.10.6}$$

Intuitively, we are taking the gradient and multiplying it by the inverse of the metric that defines what it means to be large, and then taking a step in that direction. However, it may still be the case that $G(\theta)$ is singular, or very close to singular, due to two features being very highly correlated. For example, if we are using two features that are exactly the same, the metric should look something like

$$G(\theta) = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

In this case, if we make change $\Delta\theta = [\Delta\theta_1\ \Delta\theta_2]^\top$ to the parameters, the size of this change measured by metric $G(\theta)$ is thus,

$$\begin{aligned} \Delta\theta^\top G(\theta)\Delta\theta &= \begin{bmatrix} \Delta\theta_1 & \Delta\theta_2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix} \\ &= \Delta\theta_1^2 + 2\Delta\theta_1\Delta\theta_2 + \Delta\theta_2^2 \\ &= (\Delta\theta_1 + \Delta\theta_2)^2 \end{aligned}$$

This means that changes in any of the features, or any combination of the features should be the same if they add up to be the same because they effectively act on the same feature. Because this matrix is singular, there exists a space in which we can move, and it will not change the policy at all (the nullspace of $G(\theta)$). For example, we can add δ to the first parameter and subtract the second by δ, and the policy is still the same. In this case, the most natural thing to do is use the pseudo-inverse, denoted as $G^\dagger(\theta)$, in place of the inverse, which means that we not trying to do anything in the nullspace, only the space in which we can actually affect things.

Q2 – What metric do we want to use for policy gradients?

Despite now knowing how to change and solve the optimization problem for different metrics, we are still left with the question of what the metric should be. It turns out that there is a canonical answer for probability distributions, given by Chentsov's theorem. This theorem effectively says that there is a unique metric such that distance is invariant to a class of changes to the problem, such as label switching, for parametric family of distributions; this metric is known as the *Fisher Information Metric* (Eq. 11.10.7).

$$G(\theta) = E_{p_\theta}\left[\nabla_\theta \log(p_\theta)\, \nabla_\theta \log(p_\theta)^\top\right] \tag{11.10.7}$$

Another way to come to this same result is to consider the Kullback–Leibler divergence, or K-L divergence, of two probability distributions. Given two probability distributions p and q,

$$\mathcal{KL}(p\|q) = \sum_{x\in\mathbb{X}} p(x) \log\left(\frac{p(x)}{q(x)}\right). \tag{11.10.8}$$

It turns out that the change in parameters measured by the Fisher Information Metric is exactly the second order approximation of the K-L divergence of the probability distributions before and after the change,

$$\begin{aligned} \mathcal{KL}(p_{\theta+\Delta\theta}\|p_\theta) &\approx \Delta\theta^\top G(\theta)\Delta\theta, \\ \mathcal{KL}(p_\theta\|p_{\theta+\Delta\theta}) &\approx \Delta\theta^\top G(\theta)\Delta\theta. \end{aligned} \tag{11.10.9}$$

In general, the second-order approximation of "obvious" metrics on probability distributions will result in the Fisher Information Metric.

For the specific problem of policy optimization, we take the Fisher Information Metric on trajectories as our metric (Eq. 11.10.10). This is because we want to essentially measure the distance between trajectories (distributions of states) given changes in parameters.

$$G(\theta) = E_{d^{\pi_\theta}(s),\pi_\theta(a|s)}\left[\nabla_\theta \log \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)^\top\right], \tag{11.10.10}$$

Recall that $d^{\pi_\theta}(s)$ is the distribution of states, or the fraction of time spent in states, under policy π_θ.

In practice, $G(\theta)$ can be estimated as a running average of the states experienced (Eq, 11.10.11), and its inclusion makes an enormous difference in the success of algorithms such as REINFORCE. [21]

$$\tilde{G}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\nabla_\theta \log \pi_\theta(a_i|s_i) \nabla_\theta \log \pi_\theta(a_i|s_i)^\top\right] \tag{11.10.11}$$

Intuitively, from a Machine Learning perspective, this algorithm is attempting to move in the direction that improves the performance the most, subject to changing the distribution of input examples as little as possible. This is also very similar to whitening of data, a natural normalization technique in Machine Learning.

In the general case, where we are just doing steepest descent with a distance metric, the algorithm is referred to as the covariant gradient method. In the special case shown above when you are measuring is distance between probability distributions, the algorithm is known as the natural gradient method.

[21] J. A. Bagnell and J. Schneider. Covariant policy search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2003

Then, we can combine this estimated policy gradient with the natural gradient method, which gives us the update rule,

$$\Delta\theta = \frac{1}{2\lambda}\,\tilde{G}^{-1}(\theta)\,\widetilde{\nabla}_\theta J. \tag{11.10.12}$$

This is known as the *Natural Policy Gradient* method. Note that Eq. (11.10.12) requires inverting the estimated Fisher information matrix, which can be computationally expensive when the number of parameters is large. One solution is to solve for Eq. (11.10.12) through iterative methods, e.g., Conjugate Gradient method, and terminate early. This in practice gives us reasonably good estimates of the natural policy gradient.

11.11 Conservative Policy Iteration

REINFORCE is essentially like a soft policy iteration, trying to change the probability of actions so that they are correlated with things that have high Q values. However, REINFORCE does not suffer from the disadvantages of policy iteration, because it makes small changes.

We can modify approximate policy iteration to avoid the problems caused by making big changes at each time step. We can make the policy iteration stochastic, by choosing to follow the old policy with probability α, and taking action $\text{argmax}_a \tilde{Q}(s,a)$ with probability $1-\alpha$. This algorithm, known as *conservative policy iteration*, essentially makes a small change to the probability distribution over trajectories, but by choosing actions to go the steepest direction uphill.

11.12 Related Reading

[1] McNamara, A., Treuille, A., Popović, Z. and Stam, J., *Fluid control using the adjoint method*, ACM Transactions On Graphics (TOG) 2004.

[2] Krizhevsky, A., Sutskever, I. and Hinton, G.E., *ImageNet Classification with Deep Convolutional Neural Networks*, NIPS 2012.

[3] Le, Quoc V, *Building high-level features using large scale unsupervised learning*, Acoustics, Speech and Signal Processing (ICASSP), 2013.

[4] Bagnell, J.A. and Schneider, J. *Covariant policy search*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2003.