Policy Gradients

In this lecture, we will continue to consider the problem of directly learning a policy from sampled information of costs and rewards. We will focus on *policy gradient methods*, that sample a noisy gradient from the environment and update the policy.

Before specifying the details of this approach, we will review back-propagation and its use in neural networks and controls.

Back-propagation

Many systems are composed of interconnected modules. This makes it easier to organize a complex system and debug the system by unit testing individual components. For example, robotics has the "sense, act, plan" paradigm, where each component is often studied and optized seperately. However, we make the paradoxical observation that modifying a single module can influence the overall system performance in a complicated way due to the inter-correlation 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 makes it possible to solve a large group of problems that are previously computationally intractable. One of the most notorious applications is training neural networks, which results in a lot of recent advances in computer vision such as the Google cats paper [3] and the GPU-accelerated ImageNet classifier [1]. However, there is a common misunderstanding that back-propagation is specific to neural network training. In fact, back-propagation can be used to compute gradients for any differentiable function. In particular, the very same idea has been used widely in optimal control, known as *the adjoint method*[7]. Just as back-propagation's success in training neural nets, the ajoint method has enabled researchers to tackle complex control problems with millions of control inputs. One 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.

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

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. A good reference for backpropogation canbe found here: `https://www.deeplearningbook.org/contents/mlp.html` Section 6.5.

The Chain Rule

Before diving in and solving complicated problems with 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 number. 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,

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{0.0.56}$$

The chain rule can further generalize to the case when $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$ 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,

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{0.0.57}$$

In vector notation, we rewrite the above equation as,

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y z, \tag{0.0.58}$$

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

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

is the *Jacobian matrix* of the function g.

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

[8] In fact, the chain rule can be generalized to the case of tensors. See `https://www.deeplearningbook.org/contents/mlp.html` for more discussions.

variables that are inputs to/outputs of the operations. For example, the system considered in the previous section can be represented as Figure 0.0.21.

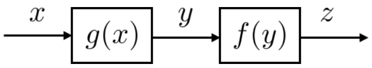

Figure 0.0.21: The block diagram representation of the simple example.

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 $Children(x)$.

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 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 o is changed when we change a variable x in the diagram, which is precisely the gradient $\nabla_x o$. By the chain rule, we have,

$$\nabla_x J = \sum_{y \in Children(x)} \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y J \tag{0.0.59}$$

Examples

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

- *Linear*

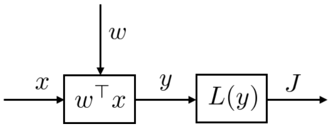

Figure 0.0.22: The block diagram of the linear module.

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

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

[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.

overall output $J = L(y)$. Then, we have,

$$\left(\frac{\partial y}{\partial x}\right)^\top = w, \qquad \left(\frac{\partial y}{\partial w}\right)^\top = x \tag{0.0.60}$$

$$\nabla_x J = \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y J = \frac{dL}{dy} w \tag{0.0.61}$$

$$\nabla_w J = \left(\frac{\partial y}{\partial w}\right)^\top \nabla_y J = \frac{dL}{dy} x \tag{0.0.62}$$

- *Squared Loss*

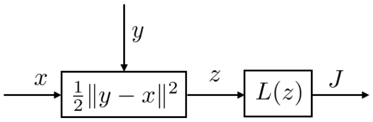

Figure 0.0.23: The block diagram of the squared loss module.

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,

$$\left(\frac{\partial z}{\partial x}\right)^\top = x - y, \qquad \left(\frac{\partial z}{\partial y}\right)^\top = y - x \tag{0.0.63}$$

$$\nabla_x J = \left(\frac{\partial z}{\partial x}\right)^\top \nabla_z J = \frac{dL}{dz}(x-y) \tag{0.0.64}$$

$$\nabla_y J = \left(\frac{\partial z}{\partial y}\right)^\top \nabla_z J = \frac{dL}{dz}(y-x) \tag{0.0.65}$$

- *Branch*

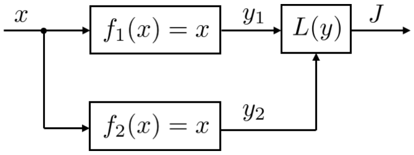

Figure 0.0.24: The block diagram of the branch module.

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,

$$\left(\frac{\partial y_1}{\partial x}\right)^\top = \left(\frac{\partial y_2}{\partial x}\right)^\top = I \tag{0.0.66}$$

$$\nabla_x J = \left(\frac{\partial y_1}{\partial x}\right)^\top \nabla_{y_1} J + \left(\frac{\partial y_2}{\partial y}\right)^\top \nabla_{y_2} J = \nabla_{y_1} J + \nabla_{y_2} J \tag{0.0.67}$$

- *Addition*

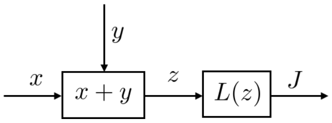

Figure 0.0.25: The block diagram of the plus module.

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,

$$\left(\frac{\partial z}{\partial x}\right)^\top = \left(\frac{\partial z}{\partial y}\right)^\top = I \tag{0.0.68}$$

$$\nabla_x J = \left(\frac{\partial z}{\partial x}\right)^\top \nabla_z J = \nabla_z J \tag{0.0.69}$$

$$\nabla_y J = \left(\frac{\partial z}{\partial y}\right)^\top \nabla_z J = \nabla_z J \tag{0.0.70}$$

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 (0.0.61) and (0.0.62), 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.

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.,

$$\{w_i^*\}_i = \arg\min_{\{w_i\}_i} J \tag{0.0.71}$$

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,

$$w_i^{k+1} = w_i^k - \alpha \nabla_{w_i} J. \tag{0.0.72}$$

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.

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 $o = \frac{1}{2}(z-y)^\top(z-y)$. 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.

TODO: change the figure to make the notation consistent

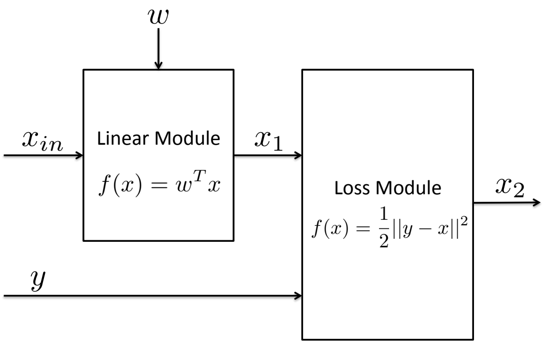

Figure 0.0.26: Linear regression represented as a cascade of modules.

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.

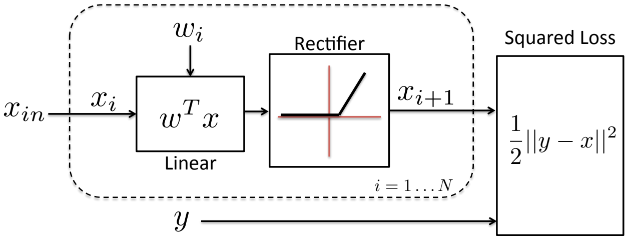

Figure 0.0.27: A neural net.

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

$$\nabla_{x_{N+1}} J = x_{N+1} - y. \qquad (0.0.73)$$

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

$$\nabla_{x_i} J = \left(\frac{\partial x_{i+1}}{\partial x_i}\right)^\top \nabla_{x_{i+1}} J \tag{0.0.74}$$

$$\nabla_{w_i} J = \left(\frac{\partial x_{i+1}}{\partial w_i}\right)^\top \nabla_{x_{i+1}} J \tag{0.0.75}$$

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.

Rederiving LQR with Back-propagation

By now, we have seen two "backward" algorithms in this class, the back-propagation algorithm that we just saw and the Riccati difference equation for the LQR problem. One natural question one may ask is whether there are some connections between the two, the answer is – indeed! Actually, we will see that we can derive the Riccati difference equation from back-propagation!

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

$$\min_{u_0, \dots, u_T} \quad \sum_{t=0}^{T-1} \left(x_t{}^\top Q x_t + u_t{}^\top R u_t\right) \tag{0.0.76}$$

$$\text{s.t.} \quad x_{t+1} = Ax_t + Bu_t, \quad \forall t = 0, \dots, T \tag{0.0.77}$$

where $x_{t+1} = Ax_t + Bu_t$ 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 of all, let us rewrite the LQR problem into a block diagram. The block diagram of the LQR problem is shown in Figure 0.0.28. Here we introduce a quadratic cost module at each time step and aggregate them into a total cost J.

First, we have,

$$\nabla_{u_T} J = 2R\, u_T, \tag{0.0.78}$$

$$\nabla_{x_T} J = 2Q\, x_T. \tag{0.0.79}$$

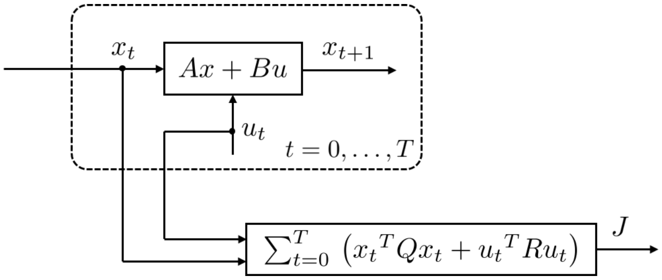

Figure 0.0.28: Finite horizon LQR realized by a block diagram.

By the chain rule, for any $t = 0, \ldots, T-1$, we have,

$$\begin{aligned} \nabla_{x_t} J &= \left(\frac{\partial J}{\partial x_t}\right)^\top + \left(\frac{\partial x_{t+1}}{\partial x_t}\right)^\top \nabla_{x_{t+1}} J \\ &= 2Q\, x_t + A^\top \nabla_{x_{t+1}} J \qquad (0.0.80) \\ \nabla_{u_t} J &= \left(\frac{\partial J}{\partial u_t}\right)^\top + \left(\frac{\partial x_{t+1}}{\partial u_t}\right)^\top \nabla_{x_{t+1}} J \\ &= 2R\, u_t + B^\top \nabla_{x_{t+1}} J \qquad (0.0.81) \end{aligned}$$

With the gradient we get from back-propagation, one can certainly run gradient descent for $\{u_t\}_{t=0}^{T-1}$. The gradient descent process does not require a matrix inversion as we saw earlier, but in return, it requires possibly many gradient descent steps. This can also be viewed as a policy search approach to LQR.

Note, however, that we can also *solve* for optimal input using these gradients since we know that the problem is convex – we can just set the gradients as zero!

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

$$2R\, u_T = 0 \qquad \Rightarrow \qquad u_T = 0. \qquad (0.0.82)$$

Let $V_T = Q$, we have,

$$\nabla_{x_T} J = 2Q\, x_T \doteq 2V_T\, x_T. \qquad (0.0.83)$$

- At time step $T-1$, we have,

$$\begin{aligned} \nabla_{u_{T-1}} J &= 2R\, u_{T-1} + B^\top \nabla_{x_T} J \\ &= 2R\, u_{T-1} + 2B^\top V_T\, x_T \\ &= 2R\, u_{T-1} + 2B^\top V_T\, (A\, x_{T-1} + B\, u_{T-1}) \\ &= 2(R + B^\top V_T\, B)\, u_{T-1} + 2B^\top V_T\, A\, x_{T-1} \end{aligned} \qquad (0.0.84)$$

By setting $\nabla_{u_{T-1}} J = 0$, we have,

$$u_{T-1} = -(R + B^\top V_T\, B)^{-1} B^\top V_T\, A\, x_{T-1} \doteq K_{T-1}\, x_{T-1}. \qquad (0.0.85)$$

Meanwhile,

$$\begin{aligned}\nabla_{x_{T-1}} J &= 2Q\, x_{T-1} + 2A^\top \nabla_{x_T} J \\ &= 2Q\, x_{T-1} + 2A^\top V_T\, (A + B\, K_{T-1})\, x_{T-1} \\ &= 2(Q + (A + B\, K_{T-1})^\top V_T\, (A + B\, K_{T-1}) \\ &\qquad - K_{T-1}^\top B^\top V_T\, (A + B\, K_{T-1}))\, x_{T-1} \\ &= 2(Q + (A + B\, K_{T-1})^\top V_T\, (A + B\, K_{T-1}) + K_{T-1}^\top R_T\, K_{T-1})\, x_{T-1} \\ &\doteq 2V_{T-1}\, x_{T-1}.\end{aligned} \tag{0.0.86}$$

- By repeating the process, we get,

$$\begin{aligned}K_{t-1} &= -(R + B^\top V_t\, B)^{-1} B^\top V_t\, A \\ V_{t-1} &= Q + (A + B\, K_{t-1})^\top V_t\, (A + B\, K_{t-1}) + K_{t-1}^\top R_t\, K_{t-1}\end{aligned} \tag{0.0.87}$$

This is precisely the Riccati difference equation!