Chapter 8

Introduction

In the first part, we presented the fundamentals of model-based and model-free RL:

- the agent-environment formulation
- the MDP assumption for E
- GPI via policy iteration using asymptotic Eval and greedy Impr
- GPI via value iteration replacing the asymptotic Eval ($\rightarrow v_\pi$ as $n \rightarrow \infty$) with approximate EvalTrunc (V_π)
- building on the above, we further relax the need for p to *calculate* V or Q, and admit the use of stochastic sampling of *episodic traces* to *estimate* V and Q (MC)
- building on the above, we further relax the need for full episodes, and admit the use of n-step samples with bootstrapping to complex the approximation of the return (TD(n))

An agent requires knowledge of the environment (the problem space) in order to properly select actions to solve the control problem. The salient operational aspects of this knowledge are contained in V and Q as we've seen; with Q the agent is able to select an appropriate state-feedback action based on it's knowledge of the efficacy (value) of that action.

Tabular methods store this knowledge in a table data structure with size of order $|S| \times |A|$ ($|S|$ and $|A|$ denote the sizes of the state space and action spaces, respectively). This is clearly only practical for moderately-sized state and action spaces. Thus we are confronted with the issue of scale: in order to apply our RL techniques to problems with large state and action spaces, we require a better alternative to represent our functions.

There are generally three means of representing functions:

- tables of values

- analytic expressions
- function approximations

We've already seen that tabular methods do not scale, and we can imagine that the very notion of having expressions for our value functions is fantastic at best: if we truly had such analytic expressions, there are probably better methods of solving our control problem than RL (e.g., control theory, or classical computer science). We're thus left with function approximation methods; fortunately, the field of supervised learning provides us with an effective and general toolset — neural networks.

We consider, in this the second part of the course, how to use function approximation techniques to represent the functions of interest in RL: value fucntions and policies.

Chapter 9

Function Approximation

Our motivation in looking at function approximation techniques is that we wish to have a method to represent our objects of interest (value functions, at present; eventually, policies) that scales better than tabular ones. We provide here a primer on function approximation methods based on parameterized functions tuned via gradient descent methods; these are workhorses of adaptive systems and have been profitably used in a variety of engineering contexts for decades. Our development is founded on linear regression, which has been used since the 1800s (at least), and is taught in high school; bear this in mind — this is not a complex subject.

9.1 Functions

Functions are mathematical objects that map elements of a set, called the domain, to another set, called the range; in the equation below:

$$f : D \to R \tag{9.1}$$

indicates that f is a mapping from set D to set R, and:

$$f : x \mapsto y \tag{9.2}$$

indicates that f maps the value $x \subset D$ to $y \subset R$.

9.2 Adaptive Functions and Gradient Descent

Let P be some phenomenon of interest in the real world, that takes some input and produces an output. It is generally useful to be able to predict what the output would be for a given input, i.e., to develop an input-output model for P. Suppose that the mapping from input to output could be modelled by a function, and further suppose we do not know what f is. Instead we have a data set of input/output pairs experimentally sampled from P:

$$DS = \{(x^{(i)}, t^{(i)})\}_{i=1}^{N} \tag{9.3}$$

that shows how f maps some element $x^{(i)} \subset D$ to $t^{(i)}$. Note: in supervised learning, $x^{(i)}$ is termed the "data", $t^{(i)}$ the "target"; we use the superscript (i) to denote the i-th element of our dataset. We wish to approximate the unknown f with an approximation $\hat{f}$ so we can make predictions; we will use information gained from the dataset to construct this approximation.

We will propose that $\hat{f}$ is some[1] parameterized function, $\hat{f}_\theta$, that can map D to R. Our problem is now to find the parameter θ^* such that:

$$\hat{f}_{\theta^*} : x^{(i)} \mapsto y^{(i)} \approx t^{(i)}, \forall i \in [1, N] \tag{9.4}$$

With this parameterization of $\hat{f}_{\theta^*}$, we will have an approximation for f that fits the data set (i.e., our available information).

To make progress in finding this θ^* we need to structure our search, based on quantitative metrics. Let us measure the per-prediction error (i.e., the error of our prediction of the i-th target) via the squared-error function:

$$PE = \frac{1}{2}(y^{(i)} - t^{(i)})^2 \tag{9.5}$$

With this we can also assess the aggregate error of our predictions over the entire dataset by averaging:

$$\begin{aligned} E &= \frac{1}{N}\sum_{\forall i} PE_i \\ &= \frac{1}{N}\sum_{\forall i} \frac{1}{2}(\hat{f}_\theta(x^{(i)}) - t^{(i)})^2 \end{aligned} \tag{9.6}$$

The aggregate error, E, is parameterized only by θ since all other components in this expression are constants. Thus $E(\theta)$ defines a hyper-surface over the space of θ parameters (θ is generally a vector of several parameters as we'll see). This hyper-surface shows the prediction error for each choice of θ — our problem is now to traverse this performance hyper-surface to find the minimum error setting for θ. Fortunately, we have an ancient means to do this automatically: gradient descent.

Gradient descent is based on the simple idea that we can start from an arbitrary $\theta = \theta_0$ and update our estimates in the direction that descends the gradient ("slope") of E_θ, i.e., $-\nabla_\theta E$. Thus algorithm 1 can be used to search the performance surface for a local minimum. Note the parameters to the algorithm:

- $\alpha > 0$ controlling the rate of update on θ (this is necessary since the above is a disretization of a fundamentally continuous process; setting the α too high may cause instability or oscillation, while setting α too low may impact the time to converge)

[1]We'll propose the forms of some common parameterized functions later.

- k_E sets the threshold on the acceptable aggregate error on our predictor
- k_∇ sets the threshold on our acceptable criterion for "flatness": i.e., when we're in the vicinity of a local minimum, how close to the minimum should we be (in terms of flatness of the hypersurface) prior to termination

Algorithm 1: Gradient Descent

```
Initialize $\theta \leftarrow \theta_0$;
while TRUE do
    compute $E$;
    compute $-\nabla_\theta E$;
    if ($E < k_E$) or ($\nabla_\theta E < k_\nabla$) then
        break;
    else
        $\theta \leftarrow \theta - \alpha \nabla_\theta E$;
```

This is the gradient descent algorithm, which can be used to tune a parameterized function to fit a general dataset. Now, does this mean that we can provide *any* parameterized function and be assured of getting an appropriate parameterization (i.e. θ)? Absolutely not! We must have sufficient degrees-of-freedom (i.e., dimensions of θ) to fit a dataset; this is a design setting that the engineer must select. Is there a means of computing the acceptable dimensions for θ (i.e. the degrees of freedom, DOFs)? No! This is called a hyper-parameter[2], and is set, open-loop, by the engineer.

Is there a guaranty that once the DOFs (i.e., the dimensions of θ) is set, we will get the global optimum? No! There is no guaranty of a global optimum existing, first of all, and if we hit a minimum, it may be a local one whose location is sensitive to the initial condition, θ_0.

Given all of the above caveats, this technique — gradient descent — is the most prominent technique for tuning general (nonlinear) differentiable parameterized functions.

9.2.1 Stochastic Gradient Descent

For problems with very large datasets, repeated computation of E with the full data set is often impractical. To address this, stochastic gradient descent is used where we, for each iteration, select a random subset of the dataset to employ in the computation of E. The size of this subset can be decreased right down to using singleton elements of the dataset, randomly selected. This will impact the convergence time, but will will be usable.

[2]Versus a parameter, e.g., θ, which is something that is solved for via the gradient descent optimizer.

9.3 Parameterized Functions

9.3.1 Linear

The elementary parameterized function is the adaptive linear combiner (ALC) of linear regression:

$$\hat{f} : x \mapsto w^T x + b \tag{9.7}$$

where $x, w \in R^N$, $b \in R$, and $\cdot^T$ denotes the transpose operator. This may be written as:

$$\hat{f} : x \mapsto b + \sum_{i=1}^{N} w_i x_i \tag{9.8}$$

There will be N update laws for the N components of w, plus one for b. As an exercise, compute these update laws.

9.3.2 Polynomial

General functions are not linear and so will require more expressiveness than the adaptive linear combiner model. Fortunately there is a simple means of transforming linear regression to enable polynomial (nonlinear) regression.

With N components for x, there are an infinite number of polynomial relationships possible of the form $x_1^{p_1} x_2^{p_2} \dots x_N^{p_N}$, where $p_i \in [0, \infty)$, that the function approximation can use as a basis. We can construct a feature mapping function, *phi*, that transforms $x \in R^N$ to $\phi(x) \in R^L$ in terms of L of these desired basis functions:

$$\phi : x \mapsto \begin{pmatrix} \Pi_{i=1}^{N} x_i^{p_i^1} \\ \cdots \\ \Pi_{i=1}^{N} x_i^{p_i^L} \end{pmatrix} \tag{9.9}$$

For example, for $x \in R^2$ as possible feature mapping is:

$$\phi : x \mapsto \begin{pmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \\ x_1^3 \end{pmatrix} \tag{9.10}$$

The feature mapping is something that the engineer selects based on their knowledge of the problem space.

One can then apply linear regression on the transformed variable, now with $w \in R^L$.

The advantages of this technique is that, *if* we can identify an appropriate ϕ, the nonlinear regression problem is merely a standard application of linear regression on $\phi(x)$.

As the ancient Spartan once said: "if"!

9.3.3 General Nonlinear

If we can not identify an appropriate ϕ — and recall, there are infinite ϕ available — then we require a technique to which we can resort[3]. The neural network is this general technique[4].

The idea is to cascade multiple adaptive functions; the motivation behind this can be seen with polynomial regression. With proper choice of ϕ we were able to convert nonlinear polynomial regression to linear regression. Why not cascade a tunable function with another, with the idea that the gradient descent process will tune the first function into a feature function like the ϕ above and the second one into a regressor (as above). And if this is valid, why not expand the capabilities of this structure by cascading multiple such adaptive functions.

Let's work out some details.

Suppose you had a $x \in R^N$. Applying one adaptive linear combiner (ALC) to x you would have:

$$h_1 = w_1^T x + b_1 \tag{9.11}$$

where $w_1 \in R^N$ and $b_1 \in R$. Let's say you applied a second ALC to x to produce h_2:

$$h_2 = w_2^T x + b_2 \tag{9.12}$$

and you repeated this for a total of H_1 times:

$$h_{H_1} = w_{H_1}^T x + b_{H_1} \tag{9.13}$$

You can imagine a schematic where you have the N components of x entering at the left, and flowing through these H_1 ALCs to produce H_1 components of h on the right. Now you can see this block as a general transformer block that takes A inputs on one side and produces B outputs on the other (where in this case $A = N$ and $B = H_1$). Suppose you were to apply another one of these transformers on the H_1 components of h to produce H_2 outputs. You can repeatedly apply these transformers to create a very large network (say with H transformers), ending with a final ALC that produced the $y \in R$ output.

Now, we know enough linear algebra to conclude that the above is an exercise in futility. Being linear functions, there is no fundamental difference between doing the above and having a single ALC from x to y.

Suppose, however, we add a nonlinearity σ to the ALC's output as follows:

$$\begin{aligned} z_1 &= w_1^T x + b_1 \\ h_1 &= \sigma(z_1) \end{aligned} \tag{9.14}$$

[3]Since it is not solid engineering practice to rely on methods that have no synthesis methodology other than creativity or inspiration. Moreover, our basic premise in RL is that we don't have access to much knowledge about the environment, and hence may lack the basis to propose an adequate feature function.

[4]The neural network is of course not magical; it is "general" in terms of saving us from requiring feature functions, and decoupling us from mere polynomial bases. There are still a large number of structural hyper-parameters that require the engineer to set in an open-loop matter.

and repeat this for a total of H_1 times as above. The the transformers are nonlinear and there *is* a fundamental difference between our nonlinear network (or function composition) of transformers, and a simple ALC from x to y. This network of nonlinear transformers is called a neural network; when $H >> 1$ it's called a deep neural network.

Regardless of what it's called, if σ is differentiable, then we can compute the gradient as per the requirement of gradient descent, and use this as our object for nonlinear function approximation. There are a large number of DOFs; we can scale this up arbitrarily to fit arbitrarily complex functional relationships.

The above network is, mathematically, a composition of several functions. Computing the derivative will require the chain rule; trivial, though time consuming. Fortunately there are graph-based accounting tricks that make this application of the chain rule relatively painless; this is called *backpropagation*, and is nothing more than a technique to organize the application of chain rule on a large composition of functions. Even better, there are computer science algorithms that allow the automatic computation of these derivatives given an arbitrary network. This enables gradient descent on complex compositions of nonlinear adaptive functions (neural networks), and is one reason why neural networks are an important toolbox method for the engineer. They are often effective and convenient to use.

As an exercise create a simple neural network with $H = 1$ and $H_1 = 2$ from $x \in R^3$ to $y \in R$, and compute the gradient descent update laws for all of the tunable coefficients.

9.4 Summary

We've covered the basic theory of supervised machine learning. The formulation we've presented is based on fitting a parameterized function to a given dataset. In moving this formulation to be appropriate to the RL problem, we'll look at how we map the data we have in RL — which is not a dataset — to the above formulation.

As we move forward, keep in mind the philosophy we encountered with value iteration: the profit that comes from considering approximations to impractical ideals. This motivated the relaxations in MC (i.e., no need for p), the relaxations in TD (i.e., no need for full episodes), and will continue to motivate our use of estimates and approximations in this part of the course.