CS234 Notes - Lecture 5
Value Function Approximation

Alex Jin, Emma Brunskill

March 20, 2018

we don't directly update V^{π} and Q^{π}. Instead, we update w and V^{π} and Q^{π} is subsequently updated because they are calculated from w.

7 Introduction

So far we have represented value function by a *lookup* table where each state has a corresponding entry, $V(s)$, or each state-action pair has an entry, $Q(s,a)$. However, this approach might not generalize well to problems with very large state and/or action spaces, or in other cases we might prefer quickly learning approximations over converged values of each state. A popular approach to address this problem is via function approximation:

$$v_\pi(s) \approx \hat{v}(s,\mathbf{w}) \qquad \text{or} \qquad q_\pi(s,a) \approx \hat{q}(s,a,\mathbf{w})$$

Here $\mathbf{w}$ is usually referred to as the parameter or weights of our function approximator. We list a few popular choices for function approximators:

- Linear combinations of features
- Neural networks
- Decision trees
- Nearest neighbors
- Fourier / wavelet bases

We will further explore two popular classes of **differentiable** function approximators: Linear feature representations and Neural networks. The reason for demanding a differentiable function is covered in the following section.

8 Linear Feature Representations

In linear function representations, we use a feature vector to represent a state:

$$x(s) = [x_1(s)\ x_2(s)\ \dots\ x_n(s)]$$

We then approximate our value functions using a linear combination of features:

$$\hat{v}(s,\mathbf{w}) = x(s)^T\mathbf{w} = \sum_{j=1}^{n} x_j(s)\mathbf{w}_j$$

We define the (quadratic) objective (also known as the loss) function to be:

$$J(\mathbf{w}) = \mathbb{E}_\pi\Big[(v_\pi(s) - \hat{v}(s,\mathbf{w}))^2\Big]$$

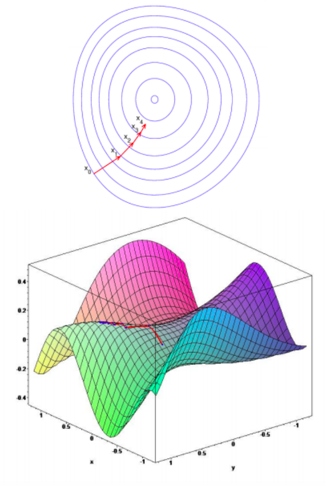

Figure 1: Visualization of Gradient Descent. We wish to reach the center point where our objective function is minimize. We do so by following the red arrows, which points in the negative direction of our evaluated gradient.

8.1 Gradient descent

A common technique to minimize the above objective function is called *Gradient Descent*. Figure 1 provides a visual illustration: we start at some particular spot x_0, corresponding to some initial value of our parameter $\mathbf{w}$; we then evaluate the gradient at x_0, which tells us the direction of the steepest increase in the objective function; to minimize our objective function, we take a step along the negative direction of the gradient vector and arrive at x_1; this process is repeated until we reach some convergence criteria.

Mathematically, this can be summarized as:

$$\begin{aligned}
\nabla_{\mathbf{w}} J(\mathbf{w}) &= \left[\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_2} \cdots \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \right] && \text{compute the gradient} \\
\Delta \mathbf{w} &= -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) && \text{compute an update step using gradient descent} \\
\mathbf{w} &\leftarrow \mathbf{w} + \Delta \mathbf{w} && \text{take a step towards the local minimum}
\end{aligned}$$

8.2 Stochastic gradient descent

In practice, Gradient Descent is not considered a sample efficient optimizer and stochastic gradient descent (**SGD**) is used more often. Although the original SGD algorithm referred to updating the weights using a single sample, due to the convenience of vectorization, people often refer to gradient descent on a minibatch of samples as SGD as well. In (minibatch) SGD, we sample a minibatch of past experiences, compute our objective function on that minibatch, and update our parameters using gradient descent on the minibatch. Let us now go back to several algorithms we covered in previous lectures and see how value function approximations can be incorporated.

8.3 Monte Carlo with linear VFA

Algorithm 1 is a modification of First-Visit Monte Carlo Policy Evaluation, and we replace our value function with our linear VFA. We also make a note that, although our return, $Return(s_t)$, is an

Algorithm 1 Monte Carlo Linear Value Function Approximation for Policy Evaluation

Evaluation only

```
Initialize $\mathbf{w} = 0$, $Returns(s) = 0\ \forall s$, $k = 1$
loop
    Sample k-th episode $(s_{k1}, a_{k1}, r_{k1}, s_{k2}, \dots, s_k, L_k)$ given $\pi$
    for $t = 1, \dots, L_k$ do
        if first visit to (s) in episode k then
            Append $\sum_{j=t}^{L_k} r_{kj}$ to $Return(s_t)$
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha(Return(s_t) - \hat{v}(s_t, \mathbf{w}))x(s_t)$
    $k = k + 1$
```

unbiased estimate, it is often very noisy.

8.4 Temporal Difference (TD(0)) with linear VFA

Recall that in the tabular setting, we approximate V^π via bootstrapping and sampling and update $V^\pi(s)$ by

$$V^\pi(s) = V^\pi(s) + \alpha(r + \gamma V^\pi(s') - V^\pi(s))$$

where $r + \gamma V^\pi(s')$ represents our *target*. Here we use the same VFA for evaluating both our target and value. In later lectures we will consider techniques for using different VFAs (such as in the control setting).

Using VFAs, we replace V^π with $\hat{V}^\pi$ and our update equation becomes:

$$\begin{aligned}\hat{V}^\pi(s, \mathbf{w}) &= \hat{V}^\pi(s, \mathbf{w}) + \alpha(r + \gamma \hat{V}^\pi(s', \mathbf{w}) - \hat{V}^\pi(s, \mathbf{w}))\nabla_{\mathbf{w}} \hat{V}^\pi(s, \mathbf{w}) \\ &= \hat{V}^\pi(s, \mathbf{w}) + \alpha(r + \gamma \hat{V}^\pi(s', \mathbf{w}) - \hat{V}^\pi(s, \mathbf{w}))\mathbf{x}(s)\end{aligned}$$

In value function approximation, although our target is a biased and approximated estimate of the true value $V^\pi(s)$, linear TD(0) will still converge (close) to the global optimum. We will prove this assertion now.

8.5 Convergence Guarantees for Linear VFA for Policy Evaluation

We define the mean squared error of a linear VFA for a particular policy π relative to the true value as:

$$MSVE(\mathbf{w}) = \sum_{s \in \mathbf{s}} \mathbf{d(s)}(\mathbf{v}^\pi(s) - \hat{\mathbf{v}}^\pi(s, \mathbf{w}))^2$$

where $\mathbf{d(s)}$ is the stationary distribution of states under policy π in the true decision process and $\hat{\mathbf{v}}^\pi(s, \mathbf{w}) = \mathbf{x}(s)^T \mathbf{w}$ is our linear VFA.

Theorem 8.1. *Monte Carlo policy evaluation with VFA converges to the weights $\mathbf{w}_{MC}$ with minimum mean squared error.*

$$MSVE(\mathbf{w}_{MC}) = \min_{\mathbf{w}} \sum_{s \in \mathbf{s}} \mathbf{d(s)}(\mathbf{v}^\pi(s) - \hat{\mathbf{v}}^\pi(s, \mathbf{w}))^2$$

Theorem 8.2. *TD(0) policy evaluation with VFA converges to the weights $\mathbf{w}_{TD}$ which is within a constant factor of the minimum mean squared error.*

$$MSVE(\mathbf{w}_{MC}) = \frac{1}{1-\gamma} \min_{\mathbf{w}} \sum_{s \in \mathbf{s}} \mathbf{d(s)}(\mathbf{v}^\pi(s) - \hat{\mathbf{v}}^\pi(s, \mathbf{w}))^2$$

We omit the proofs here and encourage interested readers to look at [1] for a more in-depth discussion.

Algorithm	Tabular	Linear VFA	Nonlinear VFA
→ Monte-Carlo Control	Yes	(Yes)	No
SARSA	Yes	(Yes)	No
Q-learning	Yes	No	No

Table 1: Summary of convergence of Control Methods with VFA. (Yes) means the result chatters around near-optimal value function.

8.6 Control using VFA

Control vs. Evaluation = $\hat{q}(s,a,w)$ v.s. $\hat{V}(s,a,w)$

Similar to VFAs, we can also use function approximators for action-values. That is, we let $\hat{q}(s,a,\mathbf{w}) \approx q_\pi(s,a)$. We may then interleave **policy evaluation**, by approximating using $\hat{q}(s,a,\mathbf{w})$, and **policy improvement**, by ϵ-greedy policy improvement. To be more concrete, let's write out this mathematically.

First, we define our objective function $J(\mathbf{w})$ as

$$J(\mathbf{w}) = \mathbb{E}_\pi[(q_\pi(s,a) - \hat{q}^\pi(s,a,\mathbf{w}))^2]$$

Similar to what we did earlier, we may then use either gradient descent or stochastic gradient descent to minimize the objective function. For example, for a linear action-value function approximator, this can be summarized as:

$$\begin{aligned}
x(s,a) &= [x_1(s,a)\ x_2(s,a)\ \ldots\ x_n(s,a)] && \text{state-action value features}\\
\hat{q}(s,a,\mathbf{w}) &= x(s,a)\mathbf{w} && \text{represent state-action value as linear combinations of features}\\
J(\mathbf{w}) &= \mathbb{E}_\pi[(q_\pi(s,a) - \hat{q}^\pi(s,a,\mathbf{w}))^2] && \text{define objective function}\\
-\frac{1}{2}\nabla_{\mathbf{w}} J(\mathbf{w}) &= \mathbb{E}_\pi[(q_\pi(s,a) - \hat{q}^\pi(s,a,\mathbf{w}))\nabla_{\mathbf{w}}\hat{q}^\pi(s,a,\mathbf{w})] && \text{compute the gradient}\\
&= (q_\pi(s,a) - \hat{q}^\pi(s,a,\mathbf{w}))x(s,a)\\
\Delta\mathbf{w} &= -\frac{1}{2}\alpha\nabla_{\mathbf{w}} J(\mathbf{w}) && \text{compute an update step using gradient descent}\\
&= \alpha(q_\pi(s,a) - \hat{q}^\pi(s,a,\mathbf{w}))x(s,a)\\
\mathbf{w} &\leftarrow \mathbf{w} + \Delta\mathbf{w} && \text{take a step towards the local minimum}
\end{aligned}$$

we approximate $q_\pi(s,a)$ w/ different things depending on method

For Monte Carlo methods, we substitute our target $q_\pi(s,a)$ with a return G_t. That is, our update becomes:

$$\Delta\mathbf{w} = \alpha(G_t - \hat{q}(s,a,\mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s,a,\mathbf{w})$$

For SARSA, we substitute our target with a TD target:

$$\Delta\mathbf{w} = \alpha(r + \gamma\hat{q}(s',a',\mathbf{w}) - \hat{q}(s,a,\mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s,a,\mathbf{w})$$

For Q-learning, we substitute our target with a max TD target:

$$\Delta\mathbf{w} = \alpha(r + \gamma\max_{a'}\hat{q}(s',a',\mathbf{w}) - \hat{q}(s,a,\mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s,a,\mathbf{w})$$

why $\nabla_w \hat{q}(s,a,w)$ only? $\gamma \max_{a'} \hat{q}(s',a',w)$ also depends on w. so this is an approximation.

We note that because our use of value function approximations, which can be expansions, to carry out our Bellman backup, convergence is not guaranteed. We refer users to look for Baird's Counterexample for a more concrete illustration. We summarize contraction guarantees in the table 1.

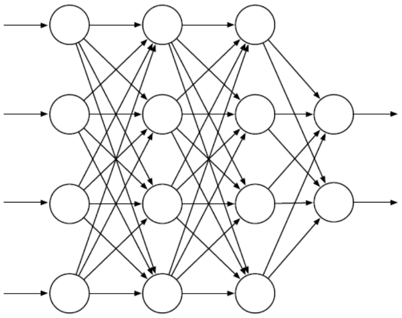

Figure 2: A generic feedforward neural network with four input units, two output units, and two hidden layers.

9 Neural Networks

Although linear VFAs often work well given the right set of features, it can also be difficult to hand-craft such feature set. Neural Networks provide a much richer function approximation class that is able to directly go from states without requiring an explicit specification of features.

Figure 2 shows a generic feedforward ANN. The network in the figure has an output layer consisting of two output units, an input layer with four input units, and two hidden layers: layers that are neither input nor output layers. A real-valued weight is associated with each link. The units are typically semi-linear, meaning that they compute a weighted sum of their input signals and then apply a nonlinear function to the result. This is usually referred to as activation function. In assignment 2, we studied how neural networks with a single hidden layer can have the "universal approximation" property, both experience and theory show that approximating the complex functions needed is made much easier with a hierarchical composition of multiple hidden layers.

In the next lecture, we will take a closer look at some theory and recent results of neural networks being applied to solve reinforcement learning problems.

References

[1] Tsitsiklis and Van Roy. An Analysis of Temporal-Difference Learning with Function Approximation. 1997.