Reinforcement Learning

The idea of “planning as graph search” is a good one, and does a great job of planning many kinds of tasks. There are times, though, when the problem is more complicated than that, where A* would run out of memory, or all actions are always legal. In these cases, finding a path through a graph may be too difficult.

In this event, it may be appropriate to use Reinforcement Learning. For an introduction of this topic, please see my dissertation, Chapter 1 (stopping at 1.2) and Chapter 2 (stopping at 2.1.2). After reading this, you should understand the following terms:

- Markov Decision Process
- Policy
- Value Function
- Bellman Equation

Value Iteration

It is usually impossible to calculate the value of any given state exactly, meaning we need ways to approximate the value of a policy \( V_\pi \). Value iteration is one such way.

We’ll assume the number of states is small, and that \( R \) and \( P \) are known. We start with an arbitrary value function vector \( \hat{V}_0 \) and perform the following update:

\[
\hat{V}_{k+1} = R + \gamma P \pi \hat{V}_k,
\]

until the difference between \( \hat{V}_{k+1} \) and \( \hat{V}_k \) becomes sufficiently small. This makes use of the fact that applying \( R + \gamma P \) (known as the Bellman operator) is a contraction, meaning that \( \hat{V}_{k+1} \) will be closer to the true value function \( V_\pi \) than \( \hat{V}_k \) was. For the mathematically inclined, we can say that \( \| (R + \gamma P \pi) \hat{V}_k - V_\pi \|_\infty \leq \gamma \| V - V \|_\infty \), where \( \| V \|_\infty = \max_i |V(s_i)| \) (called the max norm).

So, using Value Iteration, our approximate value function converges to the true value function of that policy.

Policy Iteration

Policy iteration is a way of using value function approximation to obtain an effective policy. It starts by defining an arbitrary policy \( \pi_0 \). It then alternates between two steps, policy evaluation, in which \( V_\pi \) is approximated, and policy improvement, in which a new greedy policy \( \pi_{i+1} \) is calculated, where

\[
\pi_{i+1}(s) = \arg\max_a \left[ R(s) + \gamma \sum_{s'} p(s'|s,a) V(s') \right].
\]

Policy iteration stops when the policy no longer changes.

If \( V \) is calculated exactly, every iteration of policy iteration can be shown to result in a better policy; with a finite state space, there is a finite number of policies. Therefore, policy iteration must arrive at the optimal policy.

Fitted Value Iteration

Value Iteration works well if the number of states is small. The reason for this is that we have to be able to represent our approximation \( \hat{V} \) as a single \( n \times 1 \) vector, where \( n \) is the number of states. Oftentimes, however, the number of states is large or infinite, making this infeasible or impossible.

In this case, we need a different way of representing our approximation \( \hat{V} \), which meets the following requirements:
The approximation should be able to be computed given a small subset of the total number of states.

The approximation should be easily extended to new states which were not part of the training set.

For many applications, we need this extension to be quickly computable.

One approach to this is to use a linear approximation, in which \( \hat{V}(s) \) is set equal to a weighted sum of features. There is some intuitiveness behind this idea. Imagine you’re riding a bicycle. Being upright tends to be good! So, choose “degrees from upright” as a feature, and give it a negative weight. Additionally, maybe velocity is a good thing, because you can get to the goal faster; make that a feature, and give it a positive weight. With the right features, and the right weights, you can make an approximation to the value function.

Suppose we have \( n \) sampled states and \( k \) features. We can arrange these feature values into a matrix, \( \Phi \), where

\[
\Phi = \begin{bmatrix}
\phi_1(s_1) & \phi_2(s_1) & \cdots & \phi_k(s_1) \\
\phi_1(s_2) & \phi_2(s_2) & \cdots & \phi_k(s_2) \\
\vdots & \vdots & \ddots & \vdots \\
\phi_1(s_n) & \phi_2(s_n) & \cdots & \phi_k(s_n)
\end{bmatrix},
\]

and the weights are stored in a \( k \times 1 \) vector \( w \):

\[
w = \begin{bmatrix}
w_1 \\
w_2 \\
\vdots \\
w_k
\end{bmatrix}
\]

Now, we can say that \( \hat{V} = \Phi w \), capturing that \( V(s) = \sum_{i=1}^{k} \phi_i(s)w_i \). Once you have \( w \), it’s very easy to find the value of new states; you just calculate the feature values of that new state (what the row of \( \Phi \) corresponding to that state would look like), and do the weighted sum.

But how do we find \( w \)? Suppose we have a vector \( b \) that we would like to approximate using a linear approximation, so that \( \Phi w \approx b \). Consider the following steps:

\[
\Phi w = b
\]

\[
\underbrace{\left( \Phi^T \Phi \right)^{-1} \Phi^T \Phi} = I
\]

\[
w = \left( \Phi^T \Phi \right)^{-1} \Phi^T b
\]

You’ll notice on the second step, the underbraced portion is noted as equal to I, the multiplicative identity matrix. This is because \( A^{-1}A = I \) for any invertible matrix \( A \).

You’ll notice on the third step that \( \left( \Phi^T \Phi \right)^{-1} \Phi^T \) is labelled as \( \Pi \), or the projection operator. This is because it projects \( b \) into the space defined by \( \Phi \). This is one way to calculate \( w \).

Returning to the idea of value function approximation, let us now assume the number of states is very large, and we do not know \( P \) or \( R \). What we have done, however, is sample from the MDP, so we have \( n \) samples of the form \((s, a, r, s')\), where \( s \) is the state we started in, \( a \) is the action taken, \( r \) is the reward received at that state, and \( s' \) is the state we ended up on after completing the action.

Remember that according to the Bellman equation, the value of a state is equal to the reward of that state plus the discounted, expected value at the next state. So, here, we can approximate the value with \( \Phi w \), where \( \Phi \) is the feature values of all our \( s \) components of our samples. The value at the NEXT state, is \( \Phi' w' \), where \( \Phi' \) is the feature values of all our \( s' \) components of our samples. Additionally, we’ll construct the vector \( R \), consisting of the \( r \) components of our samples.

So, we can now do Value Iteration using a linear approximation:

\[
\Phi w_{t+1} = R + \gamma \Phi' w_t
\]

\[
w_{t+1} = \Pi(R + \gamma \Phi' w_t)
\]

This is called Fitted Value Iteration.
Temporal Difference Learning

So, Fitted Value Iteration has taken our first approach (Policy Iteration with Value Iteration), and removed the constraint that the number of states be small, as well as the assumption that we know $P_\pi$ and $R$. That’s good!

However, empirically, FVI requires a lot of iterations, and a lot of samples. Is there something better?

Another algorithm for calculating a value function, when we start off knowing nothing at all is called Temporal Difference Learning, often abbreviated as TD(0).

We’ll start by assuming a small number of states, so that we can represent the vector $\hat{V}$ directly, without resorting to a linear approximation.

Suppose we have some approximation $\hat{V}$, and are at some state $s$. How can we tell if our approximation $\hat{V}(s)$ is correct? Well, we can take an action, resulting in a data point of $(s, a, r, s')$, meaning, “I was at state $s$, took action $a$, received reward $r$, and ended up at state $s'$.” Note that no knowledge of $P$ or $R$ is required to take this sample. So, is our value function good? Well, it would be perfect if for some $(s, a, r, s')$ sample, $\hat{V}(s) = r + \gamma V(s')$, right?

What if the right side of that equality was too big? Well, we should increase our approximation of $V(s)$, right? What if it’s too small? Well, then we should decrease it.

This makes for a very simple update rule (aside from the “makes sense” argument, there are more mathematically supported reasons for this as well, namely, that this moves down the slope of an error function, leading to a new approximation with lower error):

$$V(s_t) \leftarrow V(s_t) + \alpha \left[ r + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t) \right],$$

where $\alpha$ is a learning rate parameter, between 0 and 1. That makes the algorithm shown in Algorithm 1.

**Algorithm 1** TD(0)

```
Initialize $\hat{V}$ arbitrarily, and $\pi$ to the policy to be evaluated
for each sampling episode do
    for each step of the episode do
        $a \leftarrow \pi(s)$
        $(s, a, r, s') \leftarrow$ observed result of taking action $a$
        $\hat{V}(s) \leftarrow \hat{V}(s) + \alpha \left[ r + \gamma \hat{V}(s') - \hat{V}(s) \right]$
        $s \leftarrow s'$
    end for
end for
```

After enough episodes, and enough passes through each state, for a small enough $\alpha$, $\hat{V}$ will converge to the true value function $V_\pi$.

Notice this does depend on visiting every state a number of times. In many cases, this is unrealistic.

**Linear TD(0)** So, what if we remove the assumption that the number of states must be small? Well, again, we’ll use a linear approximation to represent our value function.

Now, we’ll change the update rule so that we’re not updating the value of an individual state, but rather, updating the entire weight vector. Now, when we discover that our approximation was off by $r - \gamma \hat{V}(s') - \hat{V}(s)$, how much should each weight be changed?

One way of looking at it is that the weights associated with the largest feature values are most responsible for the incorrectness of the value. Again, there are mathematical “moving down the gradient of an error function” reasons for this as well. That gives us the update rule

$$w \leftarrow w + \alpha \left[ r + \gamma \Phi(s')w - \Phi(s)w \right] \Phi^T(s),$$

where $\Phi(s)$ is the row vector of feature values for state $s$, and $\Phi^T(s)$ is the transpose of this vector (so it is now a column vector).

If $\alpha$ is steadily decreased as data is added in a principled way, then Linear TD will converge, though not necessarily to the best possible solution.
Online learning  This type of learning, where the approximation is gradually improved as the experiences occur (as opposed to afterwards, as in the case of Fitted Value Iteration), is known as *Online Learning*. Sometimes online learning fits your scenario (maybe, for instance, failure is quite expensive, and so repeatedly undergoing uninformed experiences is a bad idea), and sometimes it doesn’t.

Conclusions

As is our usual approach, we started assuming a number of things that made life easier, namely, that the number of states was small, and we knew $P$ and $R$. By the time we got to Linear TD, though, all those assumptions had been removed. Linear TD can be used to approximate a value function, and therefore generate a good policy, even when we know nothing at all about the environment, or the effects of our actions.

Temporal Difference Learning was introduced in 1988; in 1992 Linear TD was used by Gerald Tesauro to build a backgammon player (known as TD-Gammon). TD-Gammon’s features were very carefully chosen (Tesauro was himself an expert backgammon player), but otherwise it began with a completely arbitrary set of weights. TD-Gammon then played itself millions of times, beginning with a random policy, but improving its policy after each game.

After finishing, TD-Gammon was a better backgammon player than all but the very best backgammon players in the world. Because it had not been “taught,” but instead learned by random experimentation, its strategy was very different from any other backgammon player’s. Its success has greatly impacting the way the game is played by expert humans. This was the first major success for Reinforcement Learning.

Since then, RL has been applied to a wide range of complicated tasks; the more complicated the task, the more promising RL is as an approach. This is because it may be easier for the agent to learn how to accomplish something, rather than have it be hard-coded in.