Randomly Walking with Temporal Difference Learning [TD(λ)]

Tomberlin, Gabriel
Reinforcement Learning CS7642
Georgia Institute of Technology
Atlanta, United States
jtomberlin6@gatech.edu

I. Introduction

Reinforcement Learning has been around since the mid-20th century, but it hasn't started making much noise until this past decade. Considered one of the founding fathers of reinforcement learning, Richard Sutton has been a crucial driving force for the field. There are many different methods for reinforcement learning, but I will be looking closely at one example of the use of Sutton's *Temporal Difference Learning* – in particular, the *Random Walk* scenario, covered in Sutton's 1988 paper *Learning to Predict by the Methods of Temporal Differences*. I will attempt to replicate results in this paper as best as I can, and discuss the results, struggles, and possible improvements.

II. Reinforcement Learning Primer

A. What is it?

Reinforcement learning a type of machine learning that deals with goal-oriented algorithms. I like to think of it in terms of Pavlov's Bell: there's a dog which is the *agent* in an *environment*, and the dog receives a *reward* for certain *actions* or ending up in certain *states* via actions – in this case, the dog receives treats for ringing the bell. Through *reinforcement*, the dog is able to learn and understand what actions yield rewards. Notice the emphasis on the keywords there: *agent*, *environment*, *reward*, *actions*, and *states*. These are the ingredients for reinforcement learning in a nutshell, without getting too deep.

The goal of reinforcement learning algorithms is for the agents to learn the best actions to take in an environment to maximize reward. When I refer to reward, actions can yield negative rewards as well as positive. The negative reward is there to help the agent understand the best actions. When you touch a hot pan, after feeling that pain (negative reward), you're less likely to do it again, leaving your total reward unchanged in the negative direction – you don't receive positive reward for not touching the pan, but you also don't receive negative reward, leaving you with overall higher cumulative reward if you were to not touch it.

That is the general idea behind reinforcement learning, and I'm going to cover a specific learning method called *Temporal Difference Learning*.

III. Temporal Difference Learning

Temporal difference methods are concerned with *learning to predict*, in the case of *time* – hence *temporal*. The term *difference* refers to the difference in error of predictions in succession. To describe the "learning" process in terms of temporal differences in the words of Sutton himself, "…learning occurs whenever there is a change in prediction in time." [1] Throughout this discussion, I will reference *Temporal Difference* as simply "TD" for short.

The goal of TD learning is to predict future rewards in time so the agent can make better action decisions to best maximize expected reward. There are a few components to calculating error and future rewards that I want to cover prior to diving deeper into TD methods. Some components have very profound effects to the calculation of reward and error, and they are very important to understand.

A. Value Function

In the computation involved in TD learning, there is the presence of a *value function* V, which stores the value estimates for each state after taking a certain action and receiving a reward for that action, remembering that rewards can be negative *or* positive. For a given state *s* for example, the value estimate for that state would be given as V(s) – the value estimate of arbitrary state *s*. Throughout computation through time, these values are updated as learning improves.

B. Eligibility Trace

When computing and updating values in the value function as time goes on, you need a way to assign credit to particular states for time steps. The eligibility trace, denoted *z* by Sutton, is a sort of memory structure. I say memory because that is what it resembles most to me. Why? The eligibility trace keeps track of how many times a state has been visited and how long ago it has been visited.

In Sutton's book, he states that the "algorithmic mechanism" offered by eligibility traces is "a short-term memory vector." [2] The idea he gives in regards to how the eligibility trace works is when an estimated value is computed, the corresponding z_t (eligibility at time *t*) for that value is "bumped up and then begins to fade away." The fade away is important part – this corresponds to how it understands how long ago a state was visited and how many times. The more an eligibility z has faded, or decayed, in value, the longer it has been since that state was last visited.

In summary, eligibility traces can be seen as a type of memory structure, or "history book", of which states have been visited in the past and how often they have been visited, ultimately affecting the future rewards that are received.

C. Lambda

I discussed previously that eligibility traces are a way of keeping memory of what states were visited and the frequency of their visits. I also stated that these traces were a method to deal with credit assignment, but they are missing a very crucial component that happens to be in the name of the TD learning method we are using – the lambda variable.

Lambda (λ) can be seen as the speed factor at which the eligibility trace decays. Remember that eligibility traces are "bumped up", as Sutton states, and they then decay, or fade away. Lambda controls the speed of this decay.

The changing value of lambda and its implications will be discussed further once the rest of the groundwork is laid out.

D. Gamma

Gamma (γ) is what's called the *discount factor*. This component determines how important future rewards will be for the current state and is between values 0 and 1. If gamma is close to 0, only immediate rewards will be considered more often. On the other hand, if gamma is closer to 1, future rewards will be considered more often, causing the agent to delay reward until future receipt if the reward in the future is higher.

In our implementation of Sutton's work, gamma will be defaulted to a value of 1 and will remain so throughout. Remember, this means that the agent will consider future rewards more than immediate rewards.

E. Alpha

Alpha (α) is the *learning rate* of the TD learning method. You can think if it almost literally as that – the rate at which learning happens. Alpha takes on a value between 0 and 1, inclusive. The higher the alpha value, the "faster" the learning process. The higher alpha is, the more aggressive the algorithm learns and adjusts. Conversely, the lower the alpha, the slower and more conservative the learning.

TD methods seem to be very sensitive to the learning rate, and it can be difficult to narrow down an optimal alpha value. If alpha is too high, meaning our learning rate is fast, then our errors might never converge to an optimal value and would actually diverge instead. If the alpha value is too low, meaning our learning rate is slow, then the errors again might not converge because the learning rate is too conservative and not fast enough to find the optimal value.

Alpha is one of the hyper-parameters that requires careful tuning and experimentation and can control convergence rather heavily.

F. Delta

Delta (δ) is likely more commonly known to general mathematics and is simply the change or difference of a value. For example, if the value estimate of state s1, V(s1), is initially 10 at one time step, then is 15 the next time step, then the delta would be 5 – the change or difference in value from one moment to the next.

Now that the major components to the learning method have been identified, there are three different algorithms I want to cover that lead into the bigger picture of TD learning: TD(1), TD(0), and TD(λ). Lastly, I'll cover the algorithm I used in attempt to replicate Sutton's results.

A. TD(1)

The first algorithm, TD(1), updates the value estimates in the value function using what's called the accumulated sum of discounted rewards (G_t), which can be seen below:

$$G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T$$

As you can see in the above equation, you're just taking the sum of rewards at each time step, with the rewards following the first being discounted by gamma. Now, this also means that TD(1) updates all of its states at the end of each episode (an episode is equivalent to a complete walk sequence ending in a terminal state). Therefore, it requires the accumulated sum of all discounted rewards. Gamma in our case is always going to be 1, so gamma multiplied with the rewards is just the rewards, so all that is happening is really addition of all the rewards. Below, you will see where G_t fits into the value function update:

$$V(S_t) = V(S_t) + \alpha(G_t - V(S_t))$$

The value at state *t* is updated by adding the accumulated sum of discounted rewards minus the current value estimate. The subtraction of the G_t term and the current value estimate is called the learning step, which is multiplied by the learning rate alpha to determine how aggressive, or not, learning should be.

B. TD(0)

There is a lot of similarity between TD(1) and TD(0). The main difference is that you aren't using the accumulated sum

of discounted rewards (G_t). See the replacement of this term in the equation below:

$$V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

TD(0) is only updating the state that you just left, rather than *all* of the states, like TD(1) does. Now, because you're not updating all the states, you're looking at the *immediate* reward plus the discounted estimated value of *one step ahead* (S_{t+1}). This is why TD(0) is also called the one-step estimator – you're looking one step ahead, rather than using estimates from an entire episode like TD(1).

C. TD(λ)

Now, how can we combine TD(1) and TD(0)? By simply substituting 1 and 0 with a variable – lambda (λ). If you recall covering lambda on page 2 above, it is the speed factor at which the eligibility trace decays. So, lambda controls how fast the traces decay.

What does TD(1) and TD(0) have in relation to TD(λ)? Quite literally, lambda. TD(1) can be converted to TD(0) by multiplying the eligibility trace by lambda, and setting lambda to 0, essentially removing the eligibility trace from the equation, which turns it back into TD(0) – a one-step lookahead. Vice versa, if you set lambda to 1, you're multiplying the eligibility trace by 1, just giving you the trace itself, which is essentially just TD(1).

The lambda can be used to create TD(1) or TD(0), but it is also used to look at intermediate error values where lambda is *between* 0 and 1. With that, we can examine the possibility of lambda values that could produce lower error values than 0 or 1.

D. Semi-Gradient TD(λ) – The Algorithm I Used

The algorithm I decided to use in my implementation and attempted replication of results was Sutton's Semi-Gradient TD(λ)[2]. The pseudocode from the Sutton's book is as follows:

Semi-gradient TD(λ) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

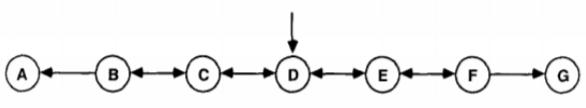
Loop for each episode:
  Initialize S
  $\mathbf{z} \leftarrow \mathbf{0}$ (a d-dimensional vector)
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \nabla\hat{v}(S,\mathbf{w})$
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
    $S \leftarrow S'$
  until S' is terminal

In this pseudocode, *z* represents the eligibility trace. For each episode, you initialize the starting state *S*, initialize the eligibility trace *z* to zeros, and then loop through each step of the episode. For each step of the episode, you choose an action from your policy, observe the resulting reward and next state from taking that action, update the eligibility trace, update delta, update the weights (or values, as I use them synonymously) using the learning rate, delta, and eligibility trace, then finally set the current state to the next state. You repeat this process until a terminal state is found.

IV. Experimentation and [Attempted] Replication

The following sections will be covering my attempts to replicate the graphs and results from Sutton's 1988 paper as best as possible. First, I'll go over what the experiments were.

The graphs and results in the following sections were created by simulating *bounded random walks*. Bounded meaning there are terminal states to end the episodes (walks). In this case, there are seven states that are connected, meaning they can be traversed to, with the states on both ends of the walk being terminal states. This can be visualized with the picture below from Sutton's paper:



You always start at state D, then take random actions (move left, or move right – 50% chance) for each state other than terminal states, where the simulation ends, ending an episode. For this example, all non-terminal state rewards are 0, state A has a reward of -1, and state G has a reward of +1.

Each experiment that will be covered consisted of 100 training sets (episodes), with 10 sequences (walks) each. I will first go over my issues or struggles, then I will discuss my results and graphs.

A. Sutton 1988 – Experiment 1, Figure 3

For this experiment, the goal is to average the errors from the random walks, analyzing the error values at lambdas ranging from 0.0 to 1.0, covering TD(0) and TD(1), and intermediate tenth values covering the intermediate procedures.

It was slightly difficult to produce similar results at first. I needed to understand how to implement the pseudocode as actual code, which didn't seem difficult at first, but it required careful examination of detail in the wording and description of the experiment, not just the pseudocode. Why? Because there were certain things you needed to know for the code to work. For example, the biggest thing I found out was the requirement of *resetting* particular variables. Without resetting variables, you would accumulate values that you shouldn't have been accumulating, causing your errors to be way off, and your graph to look way off. So, you needed to know when and where to reset, which took some tinkering.

The next problems that came were understanding when to calculate the error and where you needed to update your weights/values. For the error, it took some trial and error, and rereading the paper to understand where he was calculating the error in the process. The more enlightening issue was the update location, which solved a lot of the errors in my graph and values once I figured it out. In the pseudocode, it shows that you're supposed to update the weights with delta after calculating delta. However, this is not the case in order to produce correct results. Hinted at from my fellow classmates and colleagues, you needed to be careful of the update placement, and notice that the placement is actually different between figure 3 and figure 4. I began to test update placement in my code, realizing that once I place the update to occur at the end of each training set, rather than each sequence of steps, my graph then began taking a better looking shape, but still wasn't how it should have been.

To get the graph to its final shape adjustment, there was one thing I hadn't tried yet – adjusting the alpha hyper parameter, or, the learning rate. But this was more trivial, as I tested a range of different values spanning exponentially 0.1, 0.01, 0.001, 0.0001. The closest shape came from 0.001, so I then began adjusting from that value, increasing slowly from the hundredths place, until I came across the shape I have now using the alpha value of 0.005.

For the figure, I managed to get the same general shape as Sutton's, but the error values are oddly higher than his. When I got to this shape as a result, I was glad that it looked almost exactly like his, except for the fact that I realized my errors were shifted up – they were higher, but maintained scale, allowing the shape of the graph to be very similar. This leads to me to believe that my issue is something likely very small if I'm still obtaining the same shape. I started altering the update steps in the inner loop for sequences, thinking maybe I wasn't understanding how it was done properly, but no matter the variation I changed it to, things were getting worse rather than improving. I also tinkered with resetting variables in different locations, or not at all, but to no avail. It's possible I could be calculating my error incorrectly, since my issue is with error values, but I'm not sure if this is the case, nor how I would correctly fix it if it *was* the problem. Unfortunately, I was not able to correct for this mistake, so my errors remain incorrect.

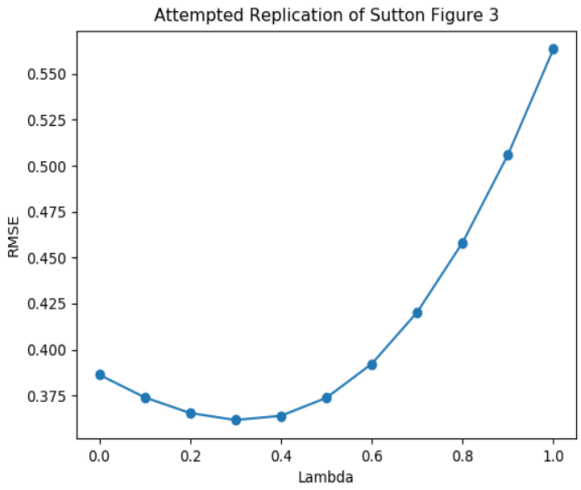


Fig. 1. Average error (RMSE) for each lambda value, using alpha 0.005, for the random walk experiment.

B. Sutton 1988 – Experiment 2, Figure 4

For experiment 2, things were fairly the same, however, instead of measuring error with respect to the lambda along the x-axis of the graph, we're looking at the effect of different alpha values for multiple lambda values. The lambdas that Sutton evaluated on were 0.0, 0.3, 0.8, and 1.0. After examining Sutton's graph, it looked like he used alphas spanning from 0.0 to 0.6, spaced linearly with eleven different points. So, we're still running the 100 episodes, and 10 sequences per episode with all lambda values, but we're now testing each alpha value *with* each lambda value, increasing our time complexity slightly.

This graph was a little more difficult to produce, and it gave me the hardest time. This difficulty was present, *despite* the code being fairly similar. Again, the biggest difference was the addition of complexity caused by having to run through each lambda using each alpha before moving to the next lambda value. Quite trivial, I would think. However, when running the experiment, there were obvious issues with replicating the shape of the curves.

One of the issues that caused some disparate error values at times, usually skyrocketing them, was the fact that if sequences became too long in length, it would cause errors to explode in size. My hunch for this is that as the sequences get longer, there is more room for error and an increasingly difficult time propagating the error through temporally with this experiment setup. By simply limiting the sequence lengths, the shapes immediately improved because you were truncating the sequences before the errors exploded. Decent lengths seemed to be within the range of 10-14. I stuck with 10 for my experiment.

Another issue was similar to the first experiment, but a different variation. This experiment required a different location of updating the value function than the first. Once I figured out where Sutton was updating properly, the graph began to take his general shape. However, if you notice

(comparing to Sutton's), it appears that my graph looks like it is cut off before the curves can finish their shapes. You can see that they are taking the general correct shape, but they don't finish – they get cut off too early. Again, I was happy to see the shapes take form, but it is odd and difficult to understand why they stopped short. I attempted to play with sequence length, as well as different locations for resetting variables. I also attempted to change how exactly I "truncated" the sequences, as I thought there was something I wasn't doing properly syntactically with the code, but there was no improvement with the alterations I attempted. but I was unsuccessful in determining the root cause of the issue.

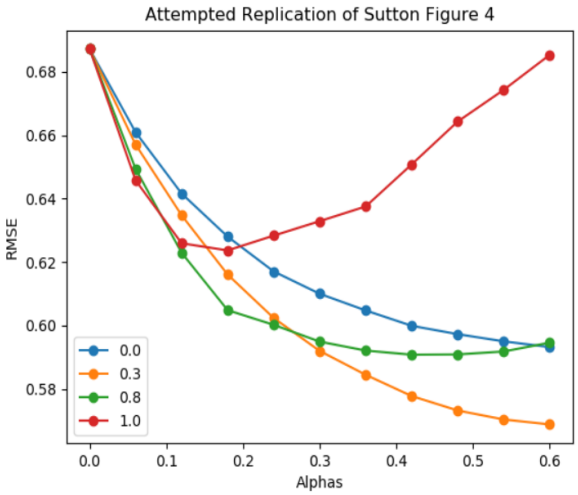


Fig. 2. Average error (RMSE) for different alphas of 4 different lambdas for the random walk experiment.

C. *Sutton 1988 – Experiment 3, Figure 5*

For experiment 3, it was a little easier than the first two. This experiment required graphing the error values for each lambda as experiment 1 but using the best alpha from experiment 2. Despite not having correct results for experiment 2, you can still see that the best alpha was around 0.6 in conjunction with lambda of 0.3 to produce the lowest error. With that, it was trivial to use similar code for both the first two experiments, ensuring correct update placement and the alpha value of 0.6 for *each* lambda, producing what looks to be a very similar curve to the one in the first experiment. This coincides with the results of Sutton, with his experiment 1 and 3 curves looking alike.

Again, the shape is correct, but the error values appear to be slightly higher than Sutton's, just like my first experiment. This still leads me to believe that the issue causing this is quite small, but I was unable to locate what exactly was causing the error level to be slightly higher.

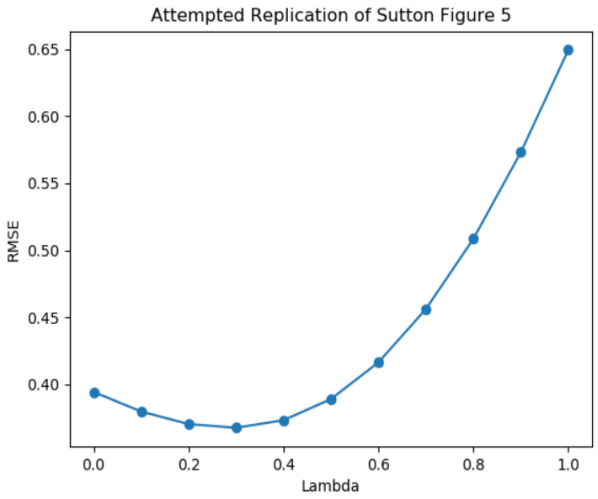


Fig. 3. Average error (RMSE) for each lambda 0.0-1.0 using the best *observed* alpha value from the second experiment.

V. Conclusion

To conclude, Temporal Difference Learning is a very powerful method of Reinforcement Learning when time is involved and is crucial to the prediction of rewards. This was just a small portion of what temporal difference is incorporated with, and there are many other algorithms that use it to this day. As a short primer, Sutton's 1988 paper helped me better understand the underlying mechanisms of TD with regards to lambda, error calculation via value/weight update, and the use of the eligibility trace and how exactly it impacts the calculations. Lastly, attempting to reproduce this paper, despite my lack of accuracy with concluding results of each experiment, allowed me to get a better understanding of TD, and how to better analyze papers for the future.

References

[1] S. Richard, "Learning to Predict by the Methods of Temporal Differences", 1988, p. 10

[2] S. Richard, B. Andrew, "Reinforcement Learning: An Introduction", 2018, p. 287-293