Random Walk Problem Using Temporal Differences

Jessica Manzione
Georgia Institute of Technology
Ellicott City, USA
jmanzione3@gatech.edu

Git hash: 9f72d91b72c294e434c41338351d2a16b1783a41

***Abstract*—This paper is a replication and further analysis of Richard S. Sutton's article on Learning to Predict by Methods of Temporal Differences [1]. Here I demonstrate multiple experiments on how to use temporal differences to solve the random walk problem instead of supervised learning. Each experiment will utilize multiple lambdas and alphas as well as other parameters and methods to evaluate the error calculated for each combination or determine the best value over 100 training sets. A comparison of my results to Sutton's as well as an overall analysis will be provided for each experiment.**

I. Background

In this paper, I am focused on the random walk problem. The environment consists of seven states, A, B, C, D, E, F, and G, in a horizontal line in that order as seen in Fig. 1. A and G are terminal states with rewards of 0 and 1, respectively. All other states do not have a reward. Each sequence will always begin at state D and each subsequent step of the sequence has equal probability (50%) of moving left or right. Due to the randomness in each step, this is a stochastic process. The sequence will continue until it reaches a terminal state and receives the corresponding reward. An example sequence is D, E, F, E, F, G with a reward of 1. The goal is to determine the probability of reaching terminal state, G, from any of the provided states.

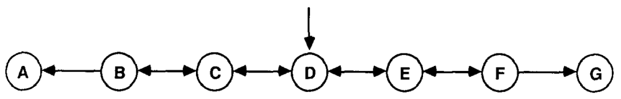

Fig. 1. Random Walk problem [1]. Start at state D and have equal probability of going left or right for each step. Sequence ends when reach either terminal state, A or G.

This problem requires us to learn to predict without using a model and without knowing the environment. A common approach would be to use supervised learning which would involve running a sequence and comparing the actual result to what was predicted. This approach may not be as useful in this problem due to the stochasticity in each step causing a lot of variations in sequences and not knowing the outcome. Instead, this paper utilizes temporal differences which is an incremental learning method. Temporal differences knows nothing about the environment and is unsupervised. When running, it appears to look like trial and error. Past states and rewards are remembered and could impact the weights with each incremental step depending on the value of lambda. Overall, instead of trying to do a complex calculation to determine the total future reward, it calculates incremental differences between the current reward and next predicted reward at that moment in time. Learning will be faster and start sooner. The most common temporal difference (TD) algorithms are TD(1), TD(0), and TD(λ) which vary in lambda used and when the weight update occurs.

A. TD(1)

TD(1) is similar to Monte Carlo and with a lambda of 1, it gives equal credit assignment to all previous states and rewards. The weight update will take place at the end of the sequence and the cumulative reward will be subtracted from the prior estimate to get the TD error. This is then multiplied by a learning rate and added to the prior estimate. Then this would repeat with a new sequence and updated weights.

B. TD(0)

Different from TD(1), TD(0) with a lambda of 0 will use the immediate reward added to the difference between the estimate value one step ahead and the current value. This is then multiplied by a learning rate and added to the prior estimate. TD(0) updates the weights at the end of each episode in a sequence instead of waiting until the end of the sequence to make an update. It can lead to a lower variance than TD(1) and also rely less on needing a terminal state.

C. TD(λ)

Finally, TD(λ) varies the lambda used as it assigns credit to previous and forward states. A variation of this is what is used in this paper. Future states will be discounted more and valued less. It will make updates to the weights before the end of the sequence unlike TD(1) but is not forced to update after every step like TD(0). Instead, it could look two steps ahead or three or more instead of just one.

Now, with an understanding of temporal differences (TD), why might this be optimal over supervised learning or the Monte Carlo method? First, supervised learning requires you to wait until the end of the game or session to start gaining any insight. Temporal differences allows you to gain information along the way. As a result, learning starts sooner and is faster. This can be crucial if the period of time for the whole game is long or you have no previous data to use and do not want to wait until you have enough data accumulated to be able to make predictions or modifications. Second, TD does not require a complete sequence and can be used in continuous problems. Since TD does not wait until the final state and reward and updates upon each step, having an end is not required. Third, it has less variance due to its smaller and incremental steps instead of comparing to the final state and reward to find the difference. Finally, it is more computational efficient due to the smaller and incremental calculations.

II. Experiments

Each state is a vector of 5 items. Four of the items are 0 and one is 1. For example, D is [0,0,1,0,0]. Weight, w, is also a vector of 5 items and will hold the predictions learned for each state. Initially, w will be 0 for each state. There will be 100 training sets and each training set consists of 10 sequences. For the final results, each experiment will use the root mean squared error and compare the estimated probabilities to the actual probabilities of 1/6, 1/3, 1/2, 2/3, and 5/6 for B, C, D, E, and F, respectively. All of the experiments performed use the same equation and hyperparameters, lambda and alpha, as shown below.

A. Equations

To calculate the change in w after each episode (step):

$$\Delta w = \alpha(P_{t+1} - P_t)\sum_{k=1}^{t}\lambda^{t-k}\nabla_w P_k \tag{1}$$

This equation shows how Δw is calculated which is the change in weight. Each sequence is a vector of m steps, x_1 to x_m. x_t is the current state at time t. P_t is the estimate at time t which is the same as $w * x_t$ or the current weights times the state at time t. P_{t+1} is the future estimate at time t+1 which is the same as $w * x_{t+1}$ and x_{t+1} is the next state. The difference between these two estimates is a major part of temporal differences as it gives the incremental difference between steps and provides the TD error. This value is multiplied by alpha, the learning rate, which determines how much influence this error will have on the change in weight. $\nabla_w P_k$ is the partial derivatives of P at time k with respect to w, also known as the gradient. This actually is the same at x_k which is the state at time k. Each state at k from time 1 to t is multiplied by lambda raised to the t-k power and then summed. This is also known as the eligibility trace and gives a higher impact to the weights for the states that occurred more recently than in the past. It will show which states will have their weights updated at each step. Finally, it is multiplied by alpha times the TD error to give the change in weight for each state.

To update the weights for each state, w, after a sequence or full training set:

$$w = w + \sum \Delta w \tag{2}$$

Essentially, the estimates for the probability of reaching G for each state, w, is updated with the sum of the change in w, Δw.

B. Parameters

The primary parameters used and varied within the experiments for this paper are alpha and lambda. See below for definitions:

Alpha (α) - Learning rate that has a value between 0 and 1 and is multiplied by the TD error. It determines how much of an effect the error should have on the weight. Larger alpha means larger adjustments and could cause fluctuation and prevent the results from converging. However, smaller alpha will be smaller adjustments but could take a lot longer to converge.

Lambda (λ) - Credit assignment with value between 0 and 1 to determine the influence of recent and past states on the weights. Higher value of lambda will allow past states to have a greater impact on the change in weights for each step.

C. Setup

These experiments were implemented in a Jupyter Notebook that can be found here with instructions on how to run in the README.md file. First, the random walk environment was created with all the states, A through G, and initial weights. There is a function to play the game which will start at D and randomly choose to left or right until it reaches a terminal state, A or G, and record that sequence and reward. For each training set, play is run 10 times to accumulate 10 sequences. Using the created training sets, the experiments are run and details of each are specified below.

D. Experiment 1

For the first experiment, I generated 10 sequences that started at state D and finished in a terminated state, A or G, that had a reward, z, of -1 or 1, respectively. Initially, weights are set to 0 for every state. These 10 sequences are for one training set. For each sequence, the Δ w is calculated. After the 10 sequences, that Δw is summed and averaged and inspected to determine if below the set threshold of 0.02 in order to find when the weights have converged and basically are stable and no longer changing. At this time, the training set is complete and the weights are updated by adding the Δw to weights vector. The same procedure is repeated for 100 total training sets.

The result of the first experiment is plotted in a line graph in Fig. 2 to show the error for each lambda value using a constant alpha. Lambda values of 0.1, 0.3, 0.5, 0.7, and 0.9 were used. For each value, the weights were determined as described above using an alpha of 0.1 and then compared to the actual probabilities using the root mean squared error for each training set and then averaged.

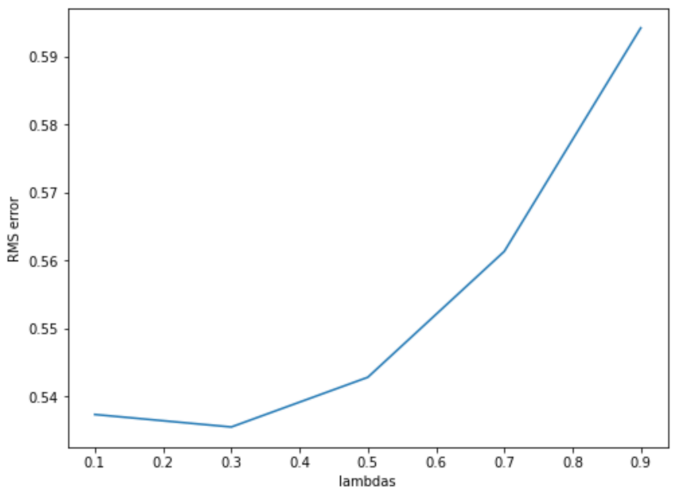

Fig. 2. Error using constant alpha for each value of lambda.

Fig. 2 shows that initially at lambda of 0.1 the error is low and continues to decrease as lambda reaches 0.3. However, as lambda continues to increase from 0.3 to 0.9, the error value increases.

E. Experiment 1 Analysis

Since the purpose of this experiment was to plot the different lambdas and their associated errors, I wanted to find the best alpha and threshold for convergence to be used. For convergence of the 10 sequences, the accumulated Δw needs to change less than the threshold after the 10 sequences. A higher alpha will mean the error in each step will have a greater impact on the weight resulting in larger changes and could converge faster or possibly never converge. A smaller alpha will have less impact from the TD error on the weights in each step and smaller changes. This might take longer to converge but has a higher likelihood of converging. I tried a few different alphas. When I used an alpha of greater than 0.1, the algorithm just continued to run and appeared to not be converging. When I tried an alpha below 0.1, I started to see higher errors for the lower lambdas than higher lambdas which is likely due to the TD error having almost no impact on the weights since the alpha is so small.

For the convergence threshold, I ended up using 0.02 which was used to compare to the average change in weights for all the states over the 10 sequences. Using a higher threshold, provided the same results for each lambda until reaching about 3 which resulted in a flat line and all the same errors for each lambda. This is because each training set is likely converging very quickly since the the threshold value for the change in w is so high. As a result, there is less learning occurring which could increase or decrease the error for each lambda depending on the sequences used in that initial training. Using a lower threshold caused the program to need a lot longer time to run. A smaller alpha makes it more likely to converge with the small threshold since the incremental impact is smaller but it will take longer for the weights to reach that small threshold. The weights will have to be very stable and barely changing over the 10 sequences. Also, there are 100 training sets that will need to reach this convergence.

I also experimented using a gamma value multiplied by the eligibility trace and also multiplied by the expected value for the next step. This will provide an additional discount and decrease the overall increments. However, I found a gamma of above 0.6 did not make a significant different on the errors. If I used a smaller gamma, then the errors would switch and become higher for the lower lambdas and lower for the higher lambdas. A lower gamma leads to a higher discount and this dramatically changes the errors. In the end, I decided to not use a gamma value for my final results.

With the updates to the weights only occurring after the 10 sequences converge, I included some averaging over the 10 sequences to the change in weights in order to prevent the change in weights from being over inflated. I tried a few different methods including no averaging, summing all the changes in weights for all sequences and dividing by 10 since 10 sequences, and dividing each change in weight for each step in a sequence by 10 times the number of steps in that sequence. In the end, I use the last option because it allowed for a faster convergence but similar errors.

Overall, Fig. 2 matches the general findings of Sutton. Lambda values less than 1 perform better since this puts a greater emphasis on the more recent states to influence the weights than past states. My results do have higher error values in general which is likely due to my combination of choices for threshold, alpha, gamma, and averaging. In addition, discussed further in problems and pitfalls, more computation power and time to run with a lower threshold or alpha to find convergence could have lead to lower error values.

F. Experiment 2

For experiment two, I again generated 10 sequences. This time, after each sequence, the weights were updated with the change in weights. Also, for each training set, the 10 sequences would only be evaluated once instead of until convergence. This is again repeated for 100 training sets. The initial weights in this experiment are set to 0.5 instead of 0 for each state to give equal probability for all states.

For each lambda value, various alpha values are used and plotted with the corresponding error determined. The lambda values used consisted of 0, 0.3, 0.8 and 1. The alpha values used consisted of a range of 0 to 0.6. Again, the error is calculated using the root mean squared error comparing the estimated probabilities to the actual probabilities for each state and then averaged over the 100 training sets.

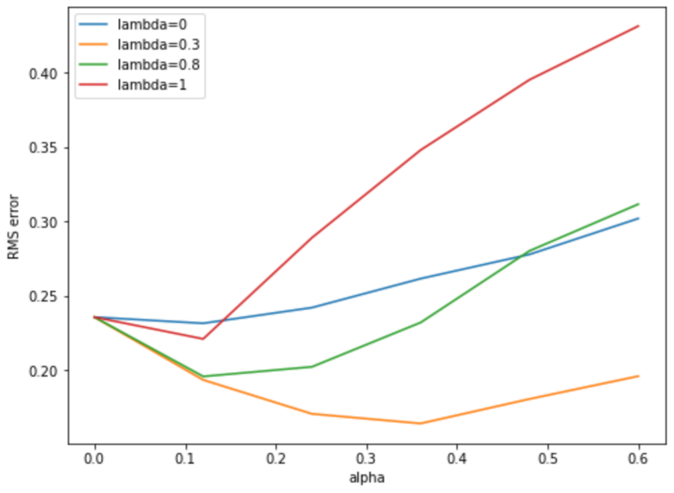

Fig. 3. Error for each alpha and lambda combination.

As shown in Fig. 3, all lambda values at alpha 0 have the same error. As alpha increase, the error values for each lambda diverge. Lambda of 1 consistently has the highest error and as alpha reaches 0.3 to 0.6, the error gap between the other lambdas becomes significant. Lambda of 0.8 error decrease as alpha approaches 0.25 but then begins to increase steeply. Lambda of 0.3 has the overall lowest error decreasing as alpha reaches 0.35 and only increase slightly after that. Lambda of 0 decrease slightly as alpha reaches 0.15 but then increase slightly as alpha increases.

G. Experiment 2 Analysis

With the removal of convergence for each training set, this experiment was simplified and required less complexity. I chose a similar averaging method for the change in weights to experiment 1 to remain consistent. However, for this experiment I used the hyperparameter gamma and set it to 0.7. I chose to include this additional discount because without it lambda of 0 was very inflated and it did not have a significant impact on the other values of lambda.

Similar to Sutton, it is clear that the lower lambdas values for any alpha produce a lower error just like seen in Fig. 2. Lambda value of 0.3 performed the best overall. Again, this is due to the greater impact from more recent states than past states especially since this is a stochastic process and past states do not have a role in determining the next state. Also, similarly, alpha values between 0.1 to 0.35 were the optimal alphas for each of the lambda value showing that a lower alpha value produced a lower error. The lower alpha values allow smaller changes due to the lower impact of the TD error on the change in weights. Although close, one slight difference from Sutton is the lambda value of 0 showing a higher value for lower alphas here than Sutton. This could be due to the choice of averaging as well as gamma.

H. Experiment 3

Using the same implementation as Experiment 2, a different graph seen in Fig. 4 was also created to highlight the error for each lambda value using the best alpha determined previously.

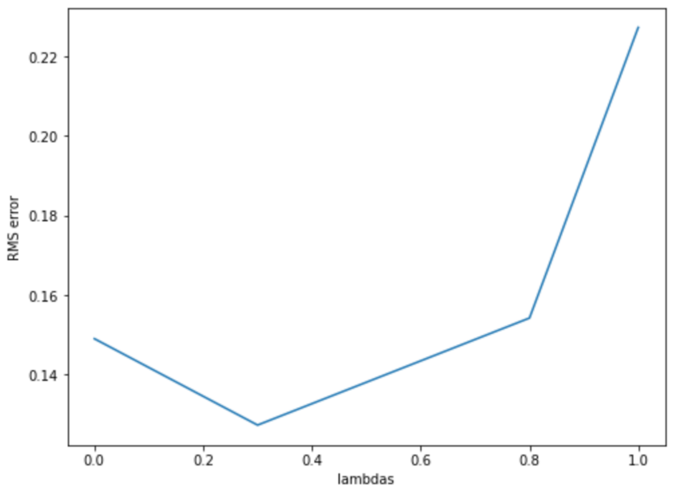

Fig. 4. Error using best alpha for each lambda.

I. Experiment 3 Analysis

The alpha values used for each lambda, 0, 0.3, 0.8, and 1 were selected based on the results shows in Fig. 3. These were 0.2, 0.35, 0.15, and 0.15, respectively. The alpha values for lambda of 0 and 0.3 were slightly higher than 0.8 and 1. This means the TD error impacting the change in weight is greater for the lower lambdas that have most of their influence coming from the most recent states.

Overall, this is similar to Sutton with the errors being lower for the lower values of alpha and then increasing as alpha increasing with alpha of 0.3 being the best.

III. Additional Analysis

An underlying assumption made that wax different than Sutton is using a reward value of -1 for state A instead of 0. I chose this because A is a terminal state that we do not want to end up in because this means you have zero chance of reaching terminal state G. Having A with a reward of 0 makes it equal to all the in between states that do have some possibility of still be able to reach state G. This change could have resulted in slightly different error values for the lambdas and alphas but overall still provided the same outcome that lambda less than one is optimal and an alpha of about 0.3 is optimal.

Another variation I attempted but decided not to use was including weight values for states A and G. A had a weight of 0 since once you reach A, then you can never reach G. G had a weight of 1 since you will 100% reach G because you are already there. I found that this skewed my results for the errors for each lambda to be more jagged and not follow expectations. As a result, I did not use it in the end results.

IV. Problems/Pitfalls

While performing these experiments, I ran into a few obstacles. First, for experiment 1 the convergence required a lot of computational power. This made it difficult to run a lot of variations since it could take a while to run. Also, I likely could have received better results if I was a able to run the program longer or faster when using a lower threshold or alpha. I tried to overcome this by finding values that were computational efficient but still provided accurate results as well as use Google Colab which gave me access to GPU. Second, some variables and methods such as gamma, alpha, and averaging were specifically stated in Sutton's paper. As a result, I had to do additional experimentation to find the appropriate value and method to use as stated in the experiments section.

V. Conclusion

Performing these experiments demonstrates the effectiveness and flexibility of temporal differences when learning to predict. I was able to incrementally determine the best prediction for each state and its probability of reaching terminal state G. Similar to Sutton, it was determined that lambda values less than 1 are optimal and alpha of 0.3 was optimal.

References

[1] Richard Sutton. "Learning to Predict by the Method of Temporal Differences". In: Machine Learning. 3 (Aug. 1988), pp. 9–44.

[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* 2nd Ed. MIT press, 2020. url: http://incompleteideas.net/book/the-book-2nd.html.