CS7642 Project 1

TD(λ)

Wei En Chen (Git hash 13fe31e124614949b8ef6ce182d3f6791529af89)

***Abstract* – This paper discusses the concept and implementation of Temporal Difference (TD) learning to replicate results provided by Sutton's paper [**Error! Reference source not found.**]. It is successfully shown very close results, and it demonstrates how TD can converge faster with α and λ parameters.**

I. INTRODUCTION

In Sutton's paper [**Error! Reference source not found.**], he introduced an algorithm called temporal differences (TD) to learn to predict future behavior by using past experience while having limited information on the system or environment. With traditional supervised learning, the algorithm learns by reducing the error between predicted and the ground truths. On the other hand, TD is an incremental learning procedure that learns from error between temporally predictions. Sutton's paper claims TD is easier to compute because it is more incremental than traditional supervised learning. Also, it claims TD converges faster and outputs more accurate predictions. With incremental learning procedure, TD is supposedly more suitable in multi-step prediction, such as incoming data with chronological order. In this paper, we will use a learning procedure called TD(λ) to attempt to replicate Figures 3, 4, and 5 from Sutton's paper and discuss implementation details, assumptions, and any discrepancy in the figures.

II. TEMPORAL DIFFERENCE

To further understand TD, we will compare/contrast it with traditional supervised learning. A typical linear supervised learning update rule can be expressed as the following:

$$\Delta\omega_t = \alpha(z - \omega^T x_t)x_t \quad (1)$$

$\Delta\omega_t$ is dependent on z, which is the ground truth outcome of the sequence. Without having the entire sequence, z is not known and $\Delta\omega_t$ cannot be calculated.

On the other hand, a TD learning procedure can be computed incrementally. The error z - P_t is represented as the sum of changes in predictions shown below:

$$z - P_t = \sum_{k=t}^{m}(P_{k+1} - P_k) where P_{m+1} \equiv z \quad (2)$$

With the equation above, TD update rule can be expressed as below:

$$\Delta\omega_t = \alpha(P_{t+1} - P_t)\sum_{k=1}^{t}\nabla_\omega P_k \quad (3)$$

As one can see, each non-terminal $\Delta\omega_t$ update is not dependent on z, the ground truth outcome only known at the end of the sequence. This can save memory space because it is not necessary to load entire sequence to learn. For computational speed, it may or may not be an advantage because on one hand the computation is simpler, but they need to be done in sequence.

A modified TD learning procedure, called TD(λ), adds exponential weighting to the TD update rule equation (3):

$$\Delta\omega_t = \alpha(P_{t+1} - P_t)\sum_{k=1}^{t}\lambda^{t-k}\nabla_\omega P_k \quad where\ 0 \le \lambda \le 1 \quad (4)$$

The summation portion (error) can be computed incrementally:

$$e_{t+1} = \sum_{k=1}^{t+1}\lambda^{t+1-k}\nabla_\omega P_k = \nabla_\omega P_{t+1} + \lambda e_t \quad (5)$$

Equations (4) and (5) are the heart of this paper's discussion. An example of random walk is used in the next section to analyze and understand TD(λ) in depth.

III. IMPLEMENTATION

The bounded random walk example uses an environment shown in Figure 1 [**Error! Reference source not found.**]. Every random sequence starts at state D, and terminates at A or G, with 50% chance of moving to the right or left state from the current state.

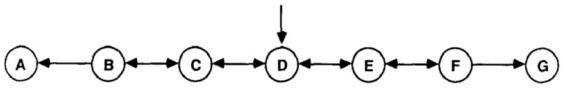


Figure 1 - Environment used for bounded random walks. All sequences start with state D, and there is 50% chance of transitioning to the left or right neighbor state. When the sequence reaches either A or G, it terminates.

In Sutton's implementation, z, the ground truth outcome is expressed as either 0 or 1, z=0 for reaching state A, and z=1 for reaching state Z. Any non-terminal state is expressed as a vector $\{x_t\}$ (column vector). For example, if a random walk sequence is DCBA, then the whole sequence is expressed as a 2D matrix:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Each column represents a state. The first column is $\{x_D\}$. Note that the terminal state is not modeled as part of the matrix because in Sutton's paper [**Error! Reference source not found.**], each non-terminal vector has only 5 elements representing state locations from B to F. In my implementation, to make the algorithm easier to execute, I increased the vector elements from 5 to 7, to include states A and G positions. Also terminal state is included in the 2D matrix. The same sequence above, DCBA, is now expressed:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This matrix is to be used in the $\Delta\omega_t$ update rule. First, we need to generate the necessary training sets. In Sutton's paper, there are 100 train sets and each set consists of 10 sequences. The algorithm below will give us the whole training data in one variable.

Next is the $\Delta\omega_t$ update algorithm. It basically combines Equations (4) and (5). The subscript t is 'shifted' in order to merge two equations.

$$\Delta\omega_t = \alpha(P_{t+1} - P_t)(\nabla_\omega P_t + \lambda e_{t-1}) \qquad (6)$$

Where $\nabla_\omega P_t = x_t$, and $e_{t-1} = 0$ at the beginning of the sequence.

Pseudocode 1: Generate Training Sets

```
n_seq is number of sequences in a training set
n_sets is number of train sets in total
func seq_maker(n_seq, n_sets)
  start = [0,0,0,1,0,0,0]
  train_sets = []
  for i from 0 to n_sets do
      set = []
      for i from 0 to n_sets do
          seq = start
          while seq is not terminal do
              if randint(2)  is zero then
                        current_seq = seq[-1] move
                  location of 1 by -1 index.
              else
                        current_seq = seq[-1] move
                  location of 1 by +1 index.
              end if
              seq.append(current_seq)
          end while
          set.append(seq)
      end for
      train_sets.append(set)
  end for
  return train_sets
end func
```

Pseudocode 2: Update $\Delta\omega_t$ within a sequence

```
func update_dw(α, λ, ω, seq)
  initialize Δω and ε as zero arrays
  for each state (x) in given seq do
      P_{t+1} = x_{t+1} ● ω
      P_t = x_t ● ω
      ε ← x_t + λ x ε
      Δω ← Δω + α x (P_{t+1} - P_t) x ε
  end for
  return Δω
end func
```

For the two experiments mentioned in Sutton's paper [**Error! Reference source not found.**], the first experiment, called repeated presentations, requires the $\Delta\omega$ to accumulate till the end of each training set (in this case 10 sequences), and then use it to update the weight vector, ω. Each training set is presented repeatedly to the learning algorithm until negligible change is observed in the ω. Sutton did not mention

what threshold was used. In my experiment in the next section, I will provide the number I use. For learning rate, α, as Sutton suggested, a small value will always allow ω to converge to the same final value. I will provide the value of α in the next section. The algorithm, repeated presentations, is shown below.

Pseudocode 3: Repeated Presentations

```
func exp1(α, λ, train_sets)
  e_list = []
  n = len(sequence per set)
  n_set = len(train_sets)
  for each set in train_sets do
      ω = [0,0.5,0.5,0.5,0.5,0.5,1]
      while ||Δω|| is > threshold do
          Δω = zero array
          for each seq in set do
                  Δω ← Δω + update_dw(α, λ, ω, seq) / n
          end for
          ω ← ω + Δω
      end while
      e_list.append(RMS(ω, ω_target))
  end for
  e_μ = mean(e_list)
  e_σ = standard deviation(e_list) / n_set
  return [e_μ, e_σ]
end func
```

One should note that the initial ω can be any values for the non-terminal states as long as the first element is 0, and the last element is 1. The update will allow the last element of 1 to flow to the rest until it converges towards the theoretical values, ω_{target}, $[0, \frac{1}{6}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{5}{6}, 1]$ as Sutton described in his paper [**Error! Reference source not found.**].

In pseudocode 3, I accumulate each Δω by dividing it with number of sequences. This has the same effect as having a small α. The advantage here is if the number of sequences is increased, I do not need to manually adjust the α value for convergence. Lastly, RMS in pseudocode 3 calculates the root mean square value of the difference of the two values, ω and ω_{target}.

For experiment 2, each training set is presented once to the learning procedure with various α and λ values to produce Figure 4 from Sutton's paper. In addition, ω is updated after each sequence is presented. Lastly, all non-terminal ω elements are initialized with 0.5. The algorithm is shown in pseudocode 4.

Pseudocode 4: Single Presentation

```
func exp2(α, λ, train_sets)
  e_list = []
  n_set = len(train_sets)
  for each set in train_sets do
      ω = [0,0.5,0.5,0.5,0.5,0.5,1]
      for each seq in set do
          ω ← ω + update_dw(α, λ, ω, seq)
      end for
      e_list.append(RMS(ω, ω_target))
  end for
  e_μ = mean(e_list)
  e_σ = standard deviation(e_list) / n_set
  return [e_μ, e_σ]
end func
```

Note that the weight update occurs at the end of every sequence, and each sequence is only presented once, so there is no need to divide Δω update by number of sequences per set.

IV. EXPERIMENT AND RESULTS

For both experiments, a random seed of 71394 is used so the same randomness can be easily compared and contrast in the result.

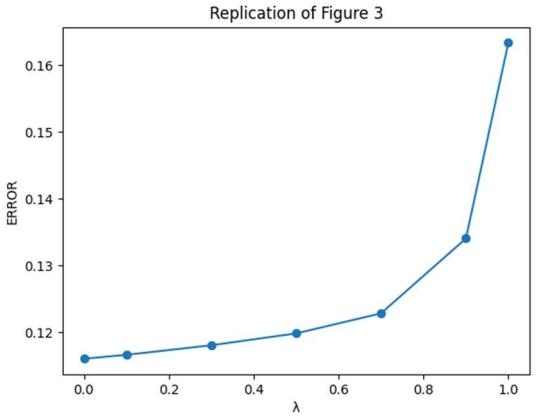


Figure 2 - Replication of Figure 3 from Sutton's paper. Average error under repeated presentations. The standard error from my experiment is < 0.01.

For replicating Figure 3 from Sutton's paper [**Error! Reference source not found.**], experiment 1, repeated presentations, is needed. Pseudocode 3 runs with one scalar value of λ. To produce the necessary data points to match the paper, I use a for loop with a list of λ = [0, 0.1, 0.3, 0.5, 0.7, 0.9, 1]. For α, I used 0.2. I also tested with other values of α. As long as the algorithm does

not generate a data overflowing warning, it always converges to the same value, as pointed out by Sutton [**Error! Reference source not found.**]. For the convergence method, first order norm of the $\Delta\omega$ vector is calculated and compared with the threshold, 10^{-6}. Second order norm and slightly different threshold are also tested and reveals no difference in the final convergence plot. The replicated plot is shown in Figure 2 (my paper).

My plot has the same curve shape as Sutton's result, but the error values are different. It seems like all my data points are shifted lower by 0.08. The standard errors on all data points are less than 0.01 which is the same as or better than Sutton's result. The possible reason that the two plots are different is the difference in training set (randomness). When I used different random seeds, slightly different plots are produced where the shape of the curve stayed the same and the error values may go up or down. However, error values only shifted by a small amount compared to 0.08. Another possible reason could be Sutton's Figure 3 was analyzed with one training set of fewer sequences than 10. I will explain the reasons in experiment 2 since replicated Figure 4 is required for the explanation.

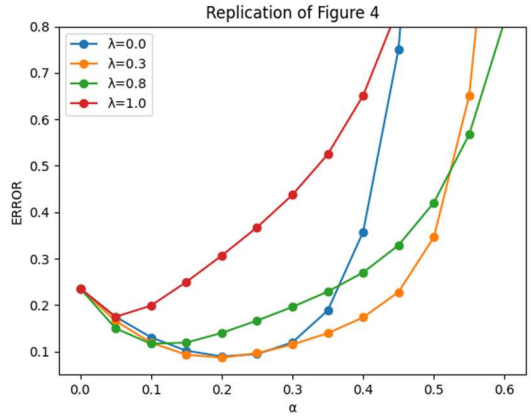


Figure 3 - Replication of Figure 4 from Sutton's paper. Average error after seeing 10 sequences once.

In experiment 2, to replicate Figure 4 from Sutton's paper, a list of α and λ are passed into a nested loop to run pseudocode 4. α = [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6]. λ = [0.0, 0.3, 0.8, 1.0]. Comparing with Sutton's result, all curves start at error around 0.23 when $\alpha = 0$. This makes sense since if $\alpha = 0$, then $\Delta\omega$ is always zero so the error is simply RMSE of the initialized weights and theoretical weights. All curves start to decrease in error as α departs in positive direction from zero. Curve of $\lambda = 1$ is the first starts to reverse the trend. In equation 6, if $\lambda = 1$, entire e_{t-1} is added to the $\Delta\omega$. As α increases, the $\Delta\omega$ likely overshoots and produces more error. As α increases more, $\lambda = 0.8$ reverses the trend, and then $\lambda = 0$, and lastly $\lambda = 0.3$. $\lambda = 0$ is not the last curve to reverse trend because it loses the ability to use e_t term to fine tune $\Delta\omega$ update. From the plot we can see that by playing around with α and λ values we can reduce computation cost to converge faster on the same dataset. The replicated figure is very close in comparison with the original Figure 4 from Sutton's paper. They don't have exactly same values, likely due to the difference in training sets.

To complete the earlier discussion on Figure 2 (my paper) and Figure 3 (Sutton's paper) discrepancy, I'd look at data points: $\lambda = 0.3$ from Figure 2 and Figure 3 (my paper). From Figure 2, it yields a data point of error around 0.12. From Figure 3, it is a yellow curve and the lowest error is around 0.1. The two numbers should not be far from each other if number of sequences in a training set is sufficient. If the number of sequences is low, it's natural for repeated representation algorithm to converge to values with higher errors, similar to how estimation with fewer sampling from a Gaussian distribution will lead to higher error. I tested a configuration of a training set of 5 and the result is very close to Sutton's. They are included in Appendix A. For the rest of the discussion a set of 10 is used.

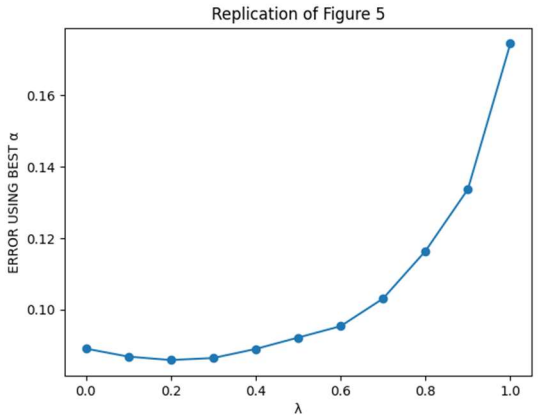


Figure 4 - Replication of Figure 5 from Sutton's paper. Average error of best α value for every λ value. The list of α values is the same in previous figure.

The last figure to replicate is Figure 5 in Sutton's paper. To produce the plot, α is the same as experiment 2 and λ = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0] are used in a nested loop to run pseudocode 4 (single presentation). Then, for every λ value in the list, use the

best (minimum) error produced by one of the α in the list. The replicated Figure 5 is shown in Figure 4 in my paper. Table 1 shows the best α values that created the plot.

Table 1 - Lambda values with their corresponding best alpha values. This table aids Figure 4 in my paper.

λ	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Best α	0.2	0.2	0.2	0.2	0.2	0.15	0.15	0.15	0.1	0.1	0.05

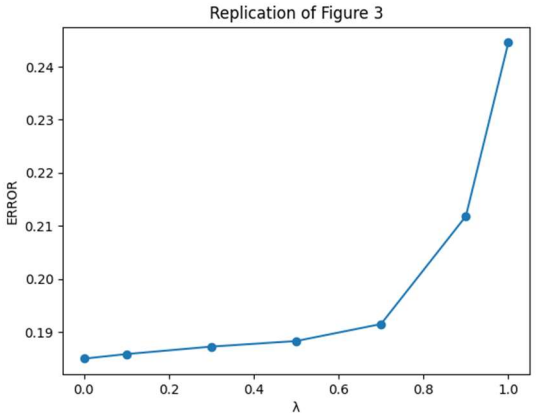
We can see that the plot takes the same shape as Figure 5 from Sutton's paper. The lowest error in my plot is at λ = 0.2. Sutton's result is around λ = 0.3. This is likely due to again the difference in training set. When I changed the random seed I did get curves with minimum error at slightly different λ.

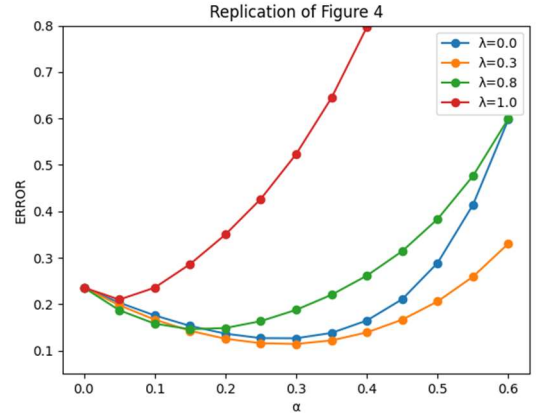
V. CONCLUSION

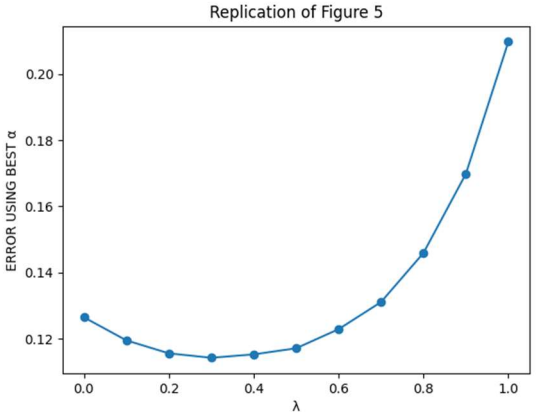
It's shown in this paper that by following mathematical equations and logic in Sutton's paper, algorithms are successfully created to generate the training sets and run necessary experiments. Figures 3 to 5 in Sutton's paper are successfully replicated, with some discrepancies in Figure 3, which can be reconciled in Appendix A. The rest of the figures take very similar shapes. Small differences are likely due to randomness in training sets. If the amount of data is increased by 10 or 100 folds, it's likely they will have the same results between mine and Sutton's papers. It is also successfully shown that a certain combination of α and λ values can quickly allow the algorithm to converge, which can save computation time.

VI. APPENDIX A

A training set of 5 sequences instead of 10 is used. In addition, α = 0.1 instead of 0.2 is used to avoid data overflow. The standard error in replicated Figure here is around 0.011, close to 0.01 reported by Sutton. The 3 plots are very close to Sutton's. There is a good chance Sutton used 5 sequences in his paper.







VII. REFERENCES

1. R. S. Sutton, "Learning to predict by the methods of temporal differences," Machine learning, vol. 3, no. 1, pp. 9–44, 1988.