Temporal Difference Learning and why it's better than Supervised Learning

Replication of Sutton,1988

Adarsh Gouda
Mechanical Engineer, AI Practitioner
Schlumberger
Edmonton, Canada
adhi.pesit@gmail.com

***Abstract*—This paper replicates the experiments described in the "Learning to Predict by the Methods of Temporal Differences" by Richard Sutton (1988). Specifically, this paper describes the random walk problem, discusses the theory behind solving the problem, provides pseudo-codes where required and contextualizes the similarities and differences in the results while mentioning the pitfalls and challenges faced in replicating the results.**

I. Introduction

In the paper "Learning to Predict by the Methods of Temporal Differences", Sutton introduces a whole new aspect of incremental learning method primarily focusing on prediction problems. The conventional method for prediction, widely known as Supervised Learning, are based on differences between predicted outcome and the ground truth. The agents (or, models) are expected to associate input-target variable pair, provided to them as a training set, and produce an inferred function which can be used to predict the target when new inputs are provided while honoring the pipelines used during the training. The paper proposes the concept of Temporal Difference (TD) learning. The basic idea of such a method is to pair consecutive predictions by exploiting the temporal relationships between states rather than pairing the predicted target to the ground truth. This approach of incremental learning has two major advantages over the conventional Supervised Learning. First, the TD methods are easier to compute because of the incremental nature in learning and therefore consume less memory. Second, TD methods exploit experience and therefore they converge faster and produce better results.

II. Temporal Difference Method

To establish the premise of TD learning, Sutton distinguishes between *single-step prediction problems* and *multi-step prediction problems*. While the former revels all information about the correctness of each prediction all at once, the later reveals only partial information relevant to its correctness at each step. TD methods can be distinguished from supervised-learning methods only in multi-step prediction problems.

A. Supervised Learning Method

To show that TD procedure can be implemented incrementally with much less computation power while it produces the same weight changes as the supervised-learning methods, Sutton considers a type of multi-step prediction problem in which experience comes in observation-outcome sequences of the form:

$$x_1, x_2, x_3, ..., x_m, z$$

where, each x_t is a vector of observations available at time t and z is the outcome of the sequence.The model produces a sequence of predictions $P_1, P_2, P_3, ..., P_m$ for a given observation-outcome sequence, each of which is an estimate of z. While each P_t can be a function of all the preceding observation vectors, it is assumed that P_t is a function of x_t for simplicity. The predictions are also based on a vector of modifiable weights, w. Therefore, P_t can be explicitly expressed as:

$$P_t = f(w, x_t) = w^T x_t = \sum_i w(i) x_t(i) \tag{1}$$

where, $x_t(i)$ and $w(i)$ are i^{th} dimension of x_t and w.

Fundamentally, all learning methods are rules to update w. In supervised-learning methods, the weights w are updated only after a complete cycle of observation-outcome sequence. For each observation, an increment Δw_t is determined. After a complete sequence, the weights are updated as follows:

$$w = w + \sum_{t=1}^{m} \Delta w_t \tag{2}$$

The typical update procedure for supervised-learning method is expressed in the following equation,

$$\Delta w_t = \alpha(z - P_t)\nabla_w P_t \tag{3}$$

Here, α is the learning rate and the term $\nabla_w P_t$ is the gradient. Substituting (1) in (3) we get:

$$\Delta w_t = \alpha(z - w^T x_t)x_t \tag{4}$$

An important point to be noted here is that Δw_t in (3) cannot be computed unless z is known. Therefore, all predictions made during a sequence is to be stored until z is revealed.

B. TD Approach

In (3), the term $(z - P_t)$ represents the error which is the difference between the actual outcome and the predicted outcome. With TD approach, this error term can be broken down into smaller error terms as the difference between consecutive predictions.

$$(z - P_t) = \sum_{k=t}^{m} (P_{k+1} - P_k) \tag{5}$$

where, $P_{m+1} = z$ and using (2), (3) and (5) we can arrive at,

$$\Delta w_t = \alpha (P_{t+1} - P_t) \sum_{k=1}^{t} \nabla_w P_k \tag{6}$$

The advantage of (6) over (3) is clear from the fact that the weight updates do not have to wait for z to be revealed. The learning occurs incrementally at each non-terminal state. This makes TD learning effectively memory-less since only the most recent weight updates is all that is required to be remembered. This also speeds-up the learning process drastically and helps in early convergence. It is also worth pointing out that $Theorem1$ Sutton's paper states that linear $TD(1)$ procedure produces the same per-sequence weight changes as the that of Widrow-Hoff procedure.

C. TD(λ) Approach

The TD procedure given by (6) is a special case in which each prediction carries same weights Δw_t as t changes. In the paper, Sutton introduces a class of TD procedures that biases the updates to most recent predictions. Specifically, the predictions are updated with exponential weighing with recency.

$$\Delta w_t = \alpha (P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k \tag{7}$$

Here, λ^k for $0 \leq \lambda \leq 1$ is the weighing factor applied to the predictions of observation vectors occurring k steps in the past. Therefore, for $\lambda = 1$, (7) reduces to (6). This proves $Theorem1$ given in the paper.

Although there are several ways in which the predictions can be altered, the exponential form in (7) is advantageous because we can compute this incrementally.

$$e_{t+1} = \sum_{k=1}^{t+1} \lambda^{t+1-k} \nabla_w P_k = \nabla_w P_{t+1} + \lambda e_t \tag{8}$$

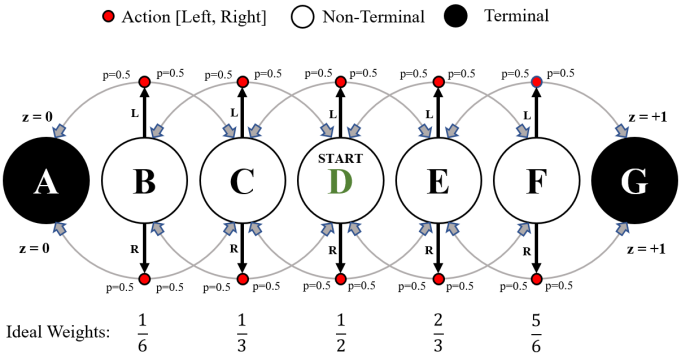

Fig. 1. A generator of random walks - all walks start at state D. Once an action is taken, either left or right, the action has $p = 0.5$ chance that it will go in the opposite direction. The walk terminates when state A or G is encountered.

III. Random-Walk Experiments

To illustrate that TD methods learn more efficiently than supervised-learning methods, Sutton proposed a simple dynamical system that generates *bounded random walks* - a state sequence generated by taking random steps to right or to the left until a terminal state is reached.

As shown in Figure 1, the walk begins at D and ends at either A or G. If the walk ends in A, the reward is $z = 0$ else if it ends in G, the reward is $z = +1$. A typical walk could be $D \rightarrow E \rightarrow D \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G$. Sutton proposed that a sequence would be represented by column-staking each individual non-terminal states present in the sequence. For the sequence mentioned above, with states $X_D, X_E, X_D, X_C, X_D, X_E, X_F$, the sequence would be represented in *one-hot encoding* format as follows:

$$\begin{matrix} X_D & X_E & X_D & X_C & X_D & X_E & X_F \end{matrix}$$
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Since the end state is G, $z = +1$.

For simplicity in code implementation, I have row-stacked each individual state instead. This helped me tremendously in simplifying the code by not having to transpose every row vector to column. Everything works fine as long as we ensure that the vector representation of states and weights are consistent across code implementation.

A. Generating a Single Random-walk Sequence

Algorithm 1 describes a function that return a single, one-hot encoded random-walk row-stacked matrix. The algorithm randomly chooses between +1 and -1 and it is added to the current state index which gives the index of the next state. The loop terminates once state reaches either A (hole state) or G (goal state).

B. Generating Training Sets

Algorithm 2 describes a function that takes number of sequences in each set and number of such sets that would

Algorithm 1: Random Walk Sequence Generator

```
def oneSequence():
    Initialize: start = 3, goal = 6, hole = 0, reward =
                0, index = start, sequence =[ ],
                sequenceOneHot = [ ]
    while True do
        sequence.append index
        index = index + randint.choice(-1,1)
        if index == goal then
            reward = 1
            break
        if index == hole then
            reward = 0
            break
    for each sequence do
        sequenceOneHot.append ← encode sequence
    return sequenceOneHot, reward
```

constitute the entire training set. Sutton used 10 sequences each in 100 sets for both the experiments. In this paper, I will start with (100,10) to replicate Sutton's graphs and then I will further explore with different training set sizes.

Algorithm 2: Training Set Generator

```
def generateTrainingSet(sets, sequences):
    Initialize: trainingSet, reward
    for i ← 0 to sets do
        for j ← 0 to sequences do
            allSets[i][j], reward[i][j] = oneSequence()
    return trainingSet, reward
```

Algorithm 3: Repeated Representation Experiment 1

```
def repeatRepresent(singleSet, setReward, λ, α, ϵ):
    Initialize: w = [0.5, 0.5, 0.5, 0.5, 0.5]
    while changes in w < ϵ do
        for each Sequence in singleSet do
            Initialize: e, Δw
            for each state in Sequence do
                compute e and Δw
            accumulate Δw
        update w with accumulated Δw
    return w
```

C. Determining True Weights, w_{true}

For the setup described in Figure 1, we can calculate the expected rewards at each states which is in fact the ideal weight vector.

$$\mathbb{E}[R(s_D)] = \frac{1}{2}(0+1) = \frac{1}{2}$$

$$\mathbb{E}[R(s_F)] = \frac{1}{2}(\mathbb{E}[R(s_E)] + \mathbb{E}[R(s_G)])$$

$$= \frac{1}{2}(\frac{1}{2}(\mathbb{E}[R(s_D)] + \mathbb{E}[R(s_G)]) + 1)$$

$$= \frac{5}{8} + \frac{1}{4}\,\mathbb{E}[R(s_E)] = \frac{5}{6}$$

$$\mathbb{E}[R(s_E)] = \frac{1}{2}(\mathbb{E}[R(s_D)] + \mathbb{E}[R(s_F)])$$

$$= \frac{1}{2}\left(\frac{1}{2} + \frac{5}{6}\right) = \frac{2}{3}$$

$$\mathbb{E}[R(s_B)] = 1 - \mathbb{E}[R(s_F)] = \frac{1}{6}$$

$$\mathbb{E}[R(s_c)] = 1 - \mathbb{E}[R(s_E)] = \frac{1}{3}$$

Therefore, the optimal or true weights of the states are:

$$w_{true} = \left[\frac{1}{6}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{5}{6}\right]$$

Next we will discuss which $TD(\lambda)$, including the Widrow-Hoff $TD(0)$, will show better performance in terms of weight updates. We will use Root Mean Squared Error between the estimated weight vector w and true weight vector w_{true} to gauge the performance for various λ and α values. Note that the w vector is a row vector in my implementation and not a column vector as used by Sutton.

Algorithm 4: RMSE Repeated Representation

```
def exp1RMSE(trainingSet, rewardSet, λ, α, ϵ):
    Initialize: trueWeights = [1/6, 1/3, 1/2, 2/3, 5/6]
    Initialize: error = 0, w = 0
    for each singleSet in traningSet do
        w = repeatRepresent(singleSet, setReward, λ,
          α, ϵ)
        error = error + sqrt(mean((w_true − w)^2))
    error = error averaged over len(trainingSet)
    return error
```

D. Repeated Representation Experiment

As mentioned before, Sutton used (100,10) training set. In Repeated Representation experiment a set of 10 sequences is repeatedly presented to the learner until convergence. Algorithm 3 illustrates how this is accomplished. The weights are initialized to [0.5, 0.5, 0.5, 0.5, 0.5]. The learner accumulates Δw for 10 sequences and only updates the weights when all sequences are completed and the converged weight vector w is returned. Algorithm 4 takes these converged weight vector updated per 10 sequences and computes the RMSE. This is repeated for all 100 sets. The accumulated RMSE is

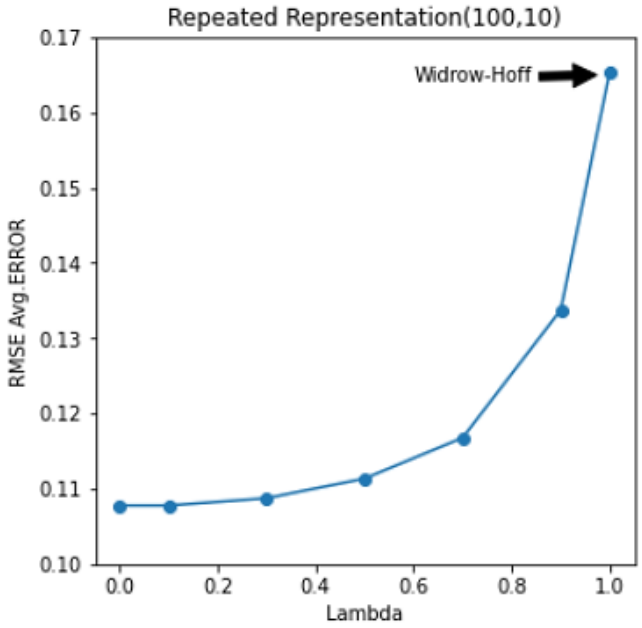

Fig. 2. Average RMSE error for different values of λ plotted for Repeated Representation experiment. This plot corresponds to Figure 3 in Sutton, 1988

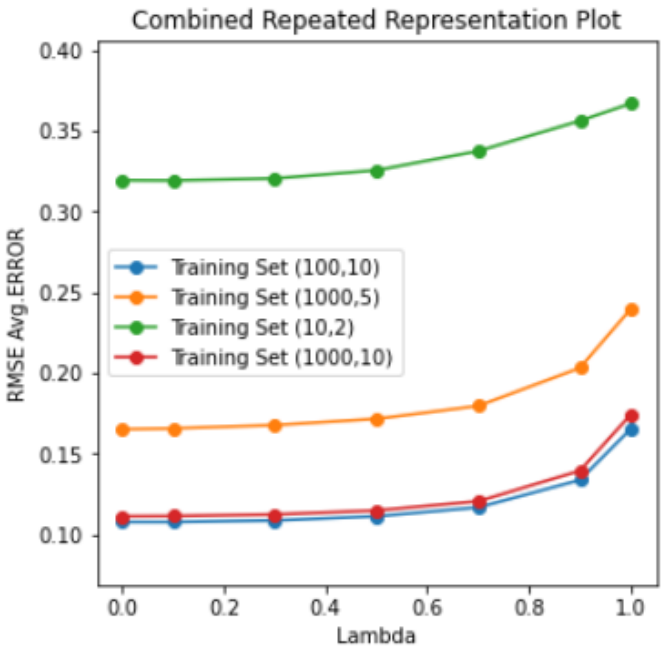

Fig. 3. Combined Average RMSE error for 4 different training set sizes, over different values of λ plotted for Repeated Representation experiment.

then averaged to arrive at the final RMSE for a give α, λ combination.

Shown in Figure 2 is my replication of Sutton's Figure 3 based on Algorithm 3 and Algorithm 4. The parameters used in my experiment are, learning-rate of $alpha = 0.01$ and convergence threshold of $\epsilon = 0.0001$. The plot shows that $TD(1)$, which is Widrow-Hoff method, has the worst performance of all the λ values. In general, the replication plot is remarkably similar to that of Sutton's Figure 3 in that the shape of the curve suggests a exponentially increasing average RMSE.

The point worth noting here is that the average RMSE in Sutton's graph is slightly higher. There can be several reasons why Sutton's RMSE are higher - choice of α and ϵ values, overall floating point precision, the inherent stochasticity in the training set. However, after repeating the experiment with a new training set of size (1000,10) shown in Figure 3, it was observed that stochasticity is not a likely cause.

Regardless of the differences in the RMSE values, Sutton's findings that $TD(0)$ outperforms $TD(1)$ by a significant margin, still holds true.

To investigate how the size of the training set influences the above finding, I setup 3 additional Repeated Representation experiments with different sizes of training set - (1000,5), (10, 2), (1000,10). Figure 3 shows, a combined error plot. Again, $TD(0)$ outperformed $TD(1)$ on all training sets. This contradicts the conventional notion that Widrow-Hoff procedure is an optimal method for prediction problems, it's not.

Algorithm 5: Single Representation Experiment

def *singleRepresent(singleSet, setReward, λ, α)*:
  Initialize: w = [0.5, 0.5, 0.5, 0.5, 0.5]
  for *each Sequence in singleSet* **do**
    Initialize: e, Δw
    for *each state in Sequence* **do**
      compute e and Δw
    update w with Δw
  return w

Algorithm 6: RMSE Single Representation

def *exp1RMSE(trainingSet, rewardSet, λ, α)*:
  Initialize: trueWeights = [1/6, 1/3, 1/2, 2/3, 5/6]
  Initialize: error = 0, w = 0
  for *each singleSet in traningSet* **do**
    w = singleRepresent(singleSet, setReward, λ, α)
    error = error + $\sqrt{mean((w_{true} - w)^2)}$
  error = error averaged over len(trainingSet)
  return error

E. Single Representation Experiment

The same (100,10) training set used in Repeated Representation experiment was also used for this experiment. The weights were initialized to [0.5, 0.5, 0.5, 0.5, 0.5]. The major difference between Repeated Representation and Single Representation is that the weights are updated after each sequence instead of waiting to complete an epoch. The Algorithms 5 and 6 details the experiment and the RMSE computation. This process is repeated on the entire training set and the average RMSE values are computed. Several combinations of λ and α are iterated over to study the impact of learning rate on the performance. Note that there is no ϵ in this experiment because the learner has just one shot at learning a sequence, there is no concept of convergence.

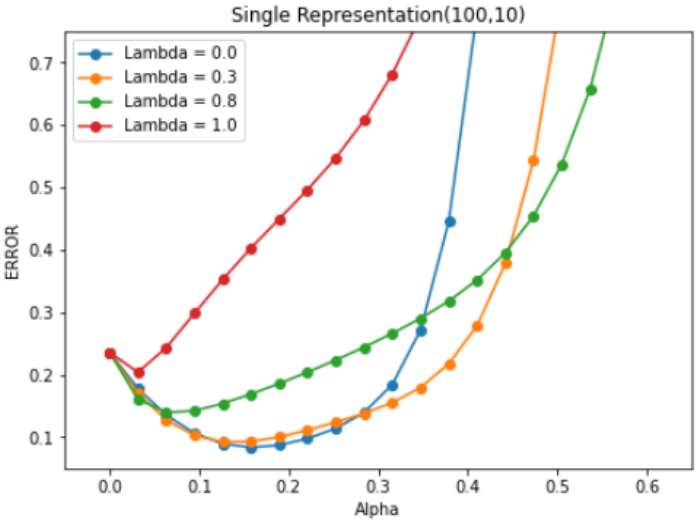

Fig. 4. Average RMSE error on random walk after experiencing 10 sequences. Replicating Figure 4 of Sutton, 1988.

Figure 4 shows, the average RMSE values for various combination of λ and α. It is clear that the learning rate has a significant impact on the performance. The lowest errors corresponds to α values in the range of 0.1 to 0.3. Therefore, α values around 0.2 can early convergence while computing the best weights. Also, the red series line corresponding to $TD(1)$ performs the worst for any given α.

The replicated plot in Figure 4 is closely similar to Figure 4 in Sutton's paper. They all are similar in shape while maintaining the proportions in RMSE. For $\alpha = 0$, the match is perfect. However, there are some deviations which could be due to the inherent stochasticity in generating the training set.

Figure 5 shows the Single Representation experiment results on 1000 sets of 5 sequences each. This too agrees with the assessment provided above for Figure 4. And it appears that Sutton's Figure 4 agrees more with (1000,5) training set than with (100,10) pointing towards lower precision of per sequence weight updates, used in Sutton's experiments, being be a major reason for discrepancies in the curves.

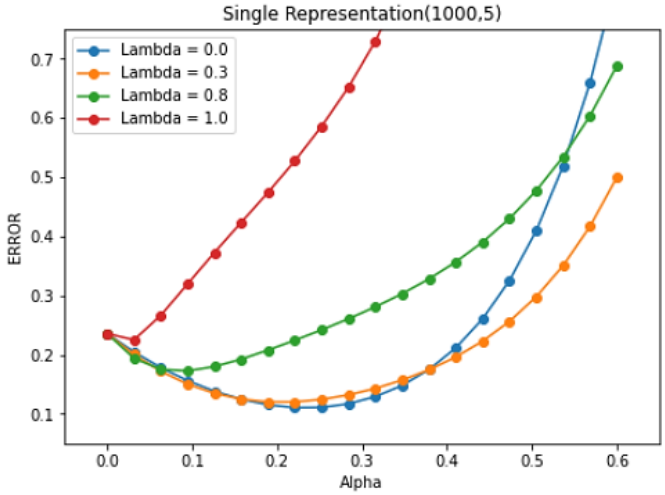

Fig. 5. Average RMSE error on random walk after experiencing 5 sequences for a 1000 sets.

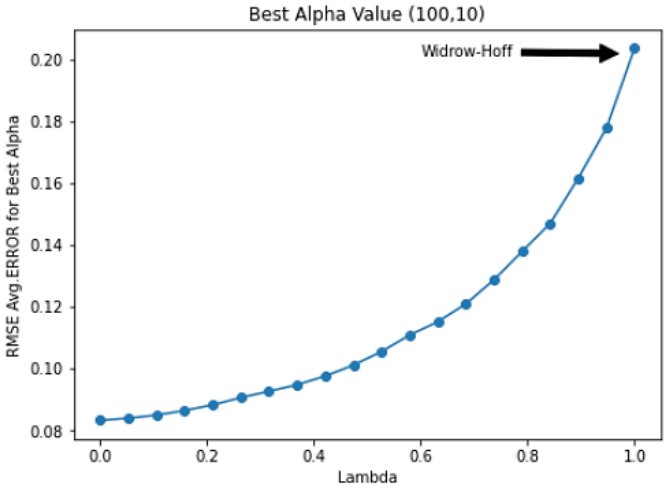

Fig. 6. Average RMSE error at best alpha value on random walk after experiencing 10 sequences for a 100 sets.

The replicated Figure 6 of Sutton's Figure 5 is strongly similar. However, the error values are lower than Sutton's errors for best α. Since there is no factor of convergence, the only explanation could be difference in the floating point precision used in running the experiment. Also, the stochasticity in generating the training set could've played a role.

IV. Conclusion

In this paper, I have discussed the details on how to setup a bounded random-walk process as described in Sutton, 1988. The applicable equations, theory and the description of the random-walk environment is discussed in Section II. The implementation on how to generate the training set is presented in Section III. The equations discussed in Sutton's paper have been used to develop algorithms for three types of learning procedures - Supervised Learning (Widrow-Hoff), Incremental Learning $TD(1)$ and the showstopper $TD(\lambda)$. Section III

also discussed the replicated plots of Sutton's Figure 3, 4 and 5. Both the experiments, Repeated Representation and Single Representation, are elaborately discussed and the results are presented succinctly. In this paper, I have described how $TD(\lambda)$ works and discussed the importance of hyper-parameters λ and the learning-rate α on the performance of Temporal Difference learning procedure. When implementing TD learning for prediction problems, a balanced learning-rate has to be selected for faster and stable convergence. A good λ value will not only provide the best performance but also ensures accuracy of the prediction.

REFERENCES

[1] R. S. Sutton, Learning to predict by the methods of temporal differences, Machine Learning, vol. 3, no. 1, pp. 944, 1988.