CS7642-RL Project 1 Report

Agnibho Chattarji
Github Commit Hash - 54ad45d32a599d727ae8f189ae9ed90cda1a13ff
Email: achattarji3@gatech.edu

***Abstract*—In an influential study, Sutton proposed a new way of making predictions using Temporal-Difference (TD) methods. In contrast to the traditional methods in use at the time, Sutton's method focused on how predictions change over time.**

I. Introduction

Sutton's paper[**?**] takes a close look at a new way of making predictions called Temporal-Difference (TD) methods. These methods are different from the usual way of predicting things because they focus on how predictions change over time. Instead of looking at the difference between what was predicted and what actually happened, TD methods look at the difference between successive predictions.

Sutton's TD methods offer two major advantages. First, they are simpler to use because they update predictions step by step. This means TD methods need less computing power and storage.

Second, TD methods are better at learning from past predictions. They get to the final prediction faster and the predictions are usually more accurate. The paper argues that because of these advantages, TD methods could be a better choice for making predictions than the usual supervised-learning methods.

Sutton focuses on problems that involve multi-step predictions. Supervised learning methods are ideally suited to simple one-step prediction problems, and don't differ from TD methods in those cases. But, for more complex problems that involve predicting a series of steps, TD methods show their strength and offer improvements over the traditional methods.

In this study, we delve into the intricacies of multi-step prediction problems, where we deal with sequences of observations leading to an outcome. Imagine a sequence like this: x_1, x_2, x_3, ..., x_m, z. Here, each x_t represents a set of observations at a specific time t in the sequence, and z is the final outcome of this sequence. We typically encounter many such sequences, each providing a snapshot of observations over time, culminating in an outcome.

Each observation, x_t, is a collection of real-valued data points or features, and the outcome, z, is a real-valued number. The goal for a learner in this scenario is to generate a series of predictions, P_1, P_2, P_3, ..., P_m, corresponding to each observation. These predictions are attempts to estimate the outcome, z. Generally, each prediction P_t could consider all previous observations in the sequence, but for simplicity, we'll focus on predictions that only consider the current observation x_t.

These predictions are informed by a set of modifiable parameters or weights, represented by w. The relationship between a prediction, an observation, and these weights can be expressed as $P(x_t, w)$, highlighting how each prediction depends on the current observation and the weights.

The $TD(\lambda)$ update procedure for the weight parameter ω is:

$$\omega \leftarrow \omega + \sum_{t=1}^{T} \Delta\omega_t \tag{1}$$

The process of learning in this context is defined by how we adjust the weights, w, based on the observations and outcomes we experience. Initially, we'll consider that the weights are updated once per sequence, meaning they remain constant throughout the sequence. After processing an entire sequence of observations and outcomes, the weights are updated based on the cumulative adjustments calculated from each observation in the sequence. This method lays the groundwork for understanding how we can iteratively refine our predictions in multi-step prediction scenarios.

A. Supervised Learning

In the supervised learning approach, we treat a series of observations and their final outcome as pairs. Each pair looks like this: (x_1, z), (x_2, z), and so on up to (x_m, z). When we make an update at time t, it's based on two things: how different the prediction P_t is from the actual outcome z, and how changing the weights w would affect P_t. We have a typical update formula in supervised learning:

$$\Delta w_t = \alpha(z - P_t)\nabla_w P_t \tag{2}$$

$$\Delta w_t = \alpha(z - w^T x_t)x_t \tag{3}$$

In this formula, α is a number that helps control how quickly we learn, and $\nabla_w P_t$ is a way of measuring how much a small change in each part of w would change P_t.

Take the case where P_t is just a straight-up multiplication of the observation at time t (x_t) by the weights w. Then P_t equals the sum of $w(i)$ times $x_t(i)$ for each i, where $w(i)$ and $x_t(i)$ are the corresponding parts of w and x_t. If this is how P_t works, then the $\nabla_w P_t$ is just x_t. This simplifies our update rule to what's called the Widrow-Hoff rule. It's a basic rule by Widrow and Hoff [**?**] for linear learning. It changes w in a way that brings P_t closer to z, so over time, the model gets better at predicting.

B. TD(λ) Learning

TD methods offer an alternative to traditional supervised learning approaches. Specifically, a TD method can produce the same result as the supervised learning rule (2), but it does so incrementally. This incremental computation is more efficient because it updates predictions based on the difference between successive predictions, rather than the final outcome. Mathematically, this can be expressed as:

$$z - P_t = \sum_{k=1}^{m} (P_{k+1} - P_k) \quad \text{where} \quad P_{m+1} \equiv z. \tag{4}$$

The update rule for the weight vector at time t in the TD method can then be written as:

$$\Delta w_t = \alpha (P_{t+1} - P_t) \nabla_w P_t. \tag{5}$$

Unlike the supervised learning update, which needs to wait until the end of the sequence to make a single large update, the TD update can be made at every step. This not only saves memory by not having to store all past values of the gradient but also distributes the computation more evenly over time. In essence, if M is the maximum sequence length, the TD method may require only $1/M$th of the memory needed by the supervised approach.

The Temporal-Difference (TD) learning methods are known for their sensitivity to changes in predictions over time, rather than just the overall error between predictions and outcomes. A TD method updates predictions incrementally, based on whether they increase or decrease the prediction for some or all of the preceding observation vectors $x_1, \ldots, x_t$. The TD(1) method is a specific case of this procedure, which treats all past predictions equally. In the TD(λ) family, predictions are weighted more heavily the more recent they are, using an exponential weighting scheme. This is expressed as:

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

The value of λ ranges between 0 and 1, providing flexibility in the learning process. When $\lambda = 1$, this reduces to the standard TD learning rule, which is similar to the update rule used in supervised learning. The advantage of using an exponential weighting scheme is that it can be computed incrementally, which is computationally efficient.

The error term e_t can be updated incrementally for each time step using current information, as shown in the following equation:

$$e_{t+1} = \nabla_w P_{t+1} + \lambda e_t. \tag{7}$$

TD(λ) can produce different weight changes compared to the traditional supervised learning methods, especially in the case of TD(0), where the weight update depends solely on the effect on the prediction associated with the most recent observation. This is formalized as:

$$\Delta w_t = \alpha (P_{t+1} - P_t) \nabla_w P_t. \tag{8}$$

This update rule is formally similar to the one used in supervised learning, but it incorporates the TD approach's advantage of incremental updates.

II. Random Walk Problem

To demonstrate the applications of TD methods, Sutton applies it to the bounded random walk problem.

A bounded random walk describes a process where a sequence of states is produced by taking random steps either to the right or left until a boundary is reached. Imagine a line of states from A to G, with D being the center state. Each step in the random walk moves to an adjacent state, choosing right or left with equal chances. This process begins from the middle state, D, and continues moving randomly step by step. The random walk comes to an end when it reaches one of the edge states, A or G.

In this study, we approached the bounded random walk problem using both linear supervised-learning and Temporal Difference (TD) methods in a direct manner. The outcome of a walk, denoted as z, was set to 0 if the walk ended on the left at state A, and 1 if it ended on the right at state G. This setup allows the learning methods to estimate the expected value of z, which, under this definition, corresponds to the probability of the walk terminating on the right side.

For every non-terminal state within the sequence, there's a designated observation vector x_i. If the walk is in state i at time t, then x_t = x_i. For instance, if a walk follows the path DCDEFG, the learning algorithm processes the sequence X_D, X_C, X_D, X_E, X_F, 1. The observation vectors x_i are defined as unit basis vectors of length 5, meaning they consist of four zeros and a single one, with the position of the one varying to represent different states (for example, X_D = (0, 0, 1, 0, 0). Consequently, if the state at time t places the one at the i^{th} component of its observation vector, the prediction P_t = w^T x_t becomes the value of the i^{th} component of w.

III. Implementation

In order to replicate the experiments from Sutton's paper [**?**] we need to generate sequences of random walks. The experiments require the generation of 100 training sets each consisting of 10 random walk sequences. Pseudo-code for the algorithm used to generate a sequence, outcome pair is shown in Algorithm 1. Sequences only contains non-terminal states, therefore we can have n = 5 non-terminal states. The sequence always begins at index = 2. The outcome of the sequence is 1 if it terminates on the right side and 0 if it terminates on the left side.

The `weight_updater` function is a procedure used in Temporal-Difference (TD) learning to update the weights, which are parameters that influence the predictions made by the model. The process is as follows:

1) The function takes in a sequence of states from a random walk, a learning rate (denoted as alpha), a discount factor

Fig. 1

Algorithm 1 Random Sequence Generator

```
function RandomSequenceGenerator(start_pos = 2, n = 5)
    seq ← [start_pos]
    outcome ← 0
    while start_pos > −1 and start_pos < n do
        if Random < 0.5 then
            start_pos ← start_pos − 1
        else
            start_pos ← start_pos + 1
        end if
        if start_pos = −1 or start_pos = n then
            if start_pos = n then
                outcome ← 1
            end if
            break
        end if
        seq.Append(start_pos)
    end while
    return seq, outcome
end function
```

(lambda), a vector of initial weights (w), the outcome of the walk, and the number of non-terminal states (n).

2) It starts by initializing a vector for the changes in weights, called delta_weights, and a vector for the error term, e_t, both filled with zeros.
3) For every state in the sequence:
   - A vector corresponding to the current state is created, with a 1 at the index of the current state and 0s elsewhere.
   - The error term e_t is updated by adding the current state vector to the previous error term, after scaling it by the discount factor lambda.
   - The current prediction, based on the current state and weights, is computed.
4) If the current state is the last one in the sequence, the function updates the delta_weights vector based on the difference between the outcome and the current prediction.
5) If not at the last state, the function computes the next prediction, and the delta_weights vector is updated based on the difference between the next and current predictions.
6) After iterating through all the states, the function returns the delta_weights vector, which represents the accumulated changes to the weights.

The pseudo code is shown in Algorithm 2.

Algorithm 2 Weight Updater Function

Require: seq (sequence of states), α (learning rate), λ (discount factor), w (weights vector), $outcome$ (outcome of the walk), n (number of non-terminal states)
Ensure: Δw (updated weights)

```
function WeightUpdater(seq, α, λ, w, outcome, n = 5)
    Δw ← zero vector of length n
    e_t ← zero vector of length n
    for t ∈ {0, ..., length of seq − 1} do
        seq_vector_t ← zero vector of length n
        seq_vector_t[seq[t]] ← 1
        e_t ← seq_vector_t + λ · e_t
        P_t ← w[seq[t]]                    ▷ Current Prediction
        if t = length of seq − 1 then
            Δw ← Δw + α · (outcome − P_t) · e_t
        else
            seq_vector_t_1 ← zero vector of length n
            seq_vector_t_1[seq[t + 1]] ← 1
            P_{t+1} ← w[seq[t + 1]]        ▷ Next Prediction
            Δw ← Δw + α · (P_{t+1} − P_t) · e_t
        end if
    end for
    return Δw
end function
```

IV. Experiments & Results

To achieve statistically robust outcomes, 100 training sets were compiled, each comprising 10 sequences, and were utilized across all learning algorithms. Weight updates were calculated in accordance with the Temporal-Difference (TD) method, specifically the TD(λ) variant as delineated by equation (6). Seven distinct λ values were tested, including $\lambda = 1$, aligning with the Widrow-Hoff supervised-learning approach, $\lambda = 0$, corresponding to the linear TD(0) model, and intermediary λ values of 0.1, 0.3, 0.5, 0.7, and 0.9. The true probabilities for right-side termination, also referred to as the

ideal predictions were calculated. These probabilities are $\frac{1}{6}$, $\frac{1}{3}$, $\frac{1}{2}$, $\frac{2}{3}$, and $\frac{5}{6}$ for states B, C, D, E, and F, respectively. To evaluate the efficacy of a learning procedure on a given training set, the root mean squared error(RMSE) between the procedure's asymptotic predictions and the ideal predictions was calculated.

A. Experiment 1

In the first experiment, contrary to the standard immediate update of the weight vector post each sequence as indicated by equation (1), the Δw's were aggregated across sequences. Initial weights are set to 0.5 for each non-terminal state. The accumulated updates were applied to the weight vector only after the entirety of a training set had been presented. This procedure was reiterated for each learning algorithm until there were no significant further alterations (less than 0.001) in the Euclidean norm of the weights vector compared to it's value prior to the previous iteration. The RMSE between the true weights and asymptotic predictions were averaged across the training sets. The pseudo code for the process is shown in Algorithm 3.

Algorithm 3 Training Procedure for Temporal-Difference Learning

Define $lambdas$ = [0, 0.1, 0.3, 0.5, 0.7, 0.9, 1]
Generate $training\_sets$, $outcomes$
$max\_iterations$ = 100000
$true\_weights$ = [1/6, 1/3, 1/2, 2/3, 5/6]
$errors$ to accumulate RMSE for each λ
for λ in $lambdas$ **do**
    for $training\_set$ in $training\_sets$ **do**
        Initialize weight vector w with 0.5 for each non-terminal state
        $iterations \leftarrow 0$
        while $iterations < max\_iterations$ **do**
            Initialize $\Delta weight$ as a zero vector of length 5
            $previous\_w \leftarrow$ copy of w
            for each seq in $training\_set$ **do**
                $\Delta weight \leftarrow \Delta weight +$ WEIGHTUPDATER(seq, α, λ, w, outcome of seq)
            end for
            $w \leftarrow w + \Delta weight$
            $iterations \leftarrow iterations + 1$
            if $\|previous\_w - w\| < min\_weight\_change$ **then**
                break
            end if
        end while
        $errors[\lambda] \leftarrow errors[\lambda] +$ RMSECALCULATOR(w, $true\_weights$)
    end for
end for

One of the crucial decisions in this experiment is the choice of the α value. Sutton mentions using a small α value in the paper. I first experimented with an α = 0.1. This often resulted in the predictions exiting probability bounds of [0-1]. The reason behind this is since we only update weights at the end of ten sequences, each sequence might produce one-sided delta weights in the case where majority of the sequences terminate on a single side. With a large α value the cumulative sum of the updates could cause predictions to overflow from the [0-1] range. I retried the experiment with an $\alpha = 0.01$. This produced results whose trend was in line with Sutton's as shown in Figure 2. My error values were lower than those of Sutton, peaking at 0.18 at $\lambda = 1$ versus 0.25 for Sutton. The discrepancy is likely due to variance in sequence generation. This was confirmed by rerunning the experiment with different training sets and observing similar trends, but varying error value ranges.

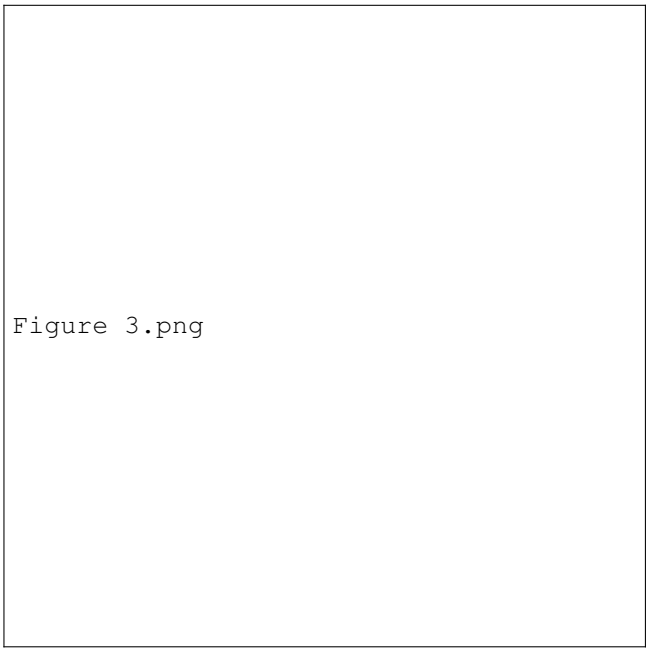

Fig. 2: Experiment 1 - Average error on the random-walk problem under repeated presentations

This result stands in contradiction to the conventional wisdom. It is widely acknowledged, as per Widrow and Stearns [**?**], that under repeated presentations, the Widrow-Hoff procedure is designed to minimize the RMSE between its predictions and the actual outcomes within the training set. The question then arises: How is it possible for this method, deemed optimal, to under perform compared to all the Temporal-Difference (TD) methods for $\lambda < 1$?

The answer lies in the distinction between minimizing error on the training set and minimizing error for future, unseen experiences. The Widrow-Hoff procedure excels in the former, optimizing prediction accuracy specifically for the training set. However, this does not necessarily translate to minimized error when encountering new data.

To better understand the importance of learning rate, I ran the experiment again with α = 0.001. The error graphs showed

similar trends to using α = 0.01. However, the computation time was significantly increased as the lower learning rate meant it took far more iterations to reach convergence.

B. Experiment 2

For the second experiment the same training sets were presented with the following changes. First, each training set was presented once to each procedure. Second, weight updates were performed after each sequence, as in equation (1), rather than after each complete training set. Third, each learning procedure was applied with a range of values for the learning-rate parameter a. Fourth, so that there was no bias either toward right-side or left-side terminations, all components of the weight vector were initially set to 0.5. λ values of 0, 0.3, 0.8, 1 were used. α values between 0-0.6 were used. The pseudo code for this implementation is described in Algorithm 4.

Algorithm 4 Experiment 2 Weight Update and Error Calculation

for each λ in $lambdas$ **do**
  for each α in $alphas$ **do**
    for $training\_set$ in range of $training\_sets$ **do**
      Initialize weight vector w with 0.5 for each non-terminal state
      for each seq in $training\_set$ **do**
        Calculate Δw using WEIGHTUPDATER($training\_set[seq]$, α, λ, w, $outcomes[training\_set][seq]$)
        Update $w \leftarrow w + \Delta w$
      end for
      Update $errors\_exp2[\lambda][\alpha]$ with the average RMSE
    end for
  end for
end for

The average error for each lambda, alpha combo is displayed in Figure 3. My results are similar to Sutton's with λ = 1 (Widrow-Hoff) performing the worst for lower α values. The value of α plays a significant role in the error. In fact I observed as α exceeded 0.4 the error for λ = 1 and λ = 0 grew exponentially and far exceeded the values for λ = 0.3 and λ = 0.8. This makes sense as with a very high learning rate it gets harder to converge as each learning-step may result in over corrections. For λ = 0 the propagation of learning is very slow along the weight vector as it can only influence one value per iteration. In order to produce a graph similar to Sutton I had to cap the Y-Axis ERROR to a limit of 0.7.

C. Experiment 3

The final experiment implements the same procedure as Experiment 2. λ values range from 0 to 1 with 0.1 increments. α values range from 0 to 0.6 with 0.05 increments. For each λ value I graph the RMSE on the best performing α value. Results are shown in Figure 4.

Figure 4.png

Fig. 3: Experiment 2 - Average error on random walk problem after experiencing 10 sequences. All data are from TD(λ) with different values of λ and α

Figure 5.png

Fig. 4: Experiment 3 - Average error at best α value on random-walk problem

Observed results align with Sutton's. The overall trend shows λ values around 0 to 0.3 perform the best. Sutton's best eprforming λ was 0.3 while mine was 0.2. Also the overall error or my graph was slightly lower with a min of around 0.085 versus 0.11 for Sutton. These minor differences can likely be attributed to the variance in generation of random sequences. $\lambda = 1$ (Wifrow-Hoff) is shown the be the worst performing confirming it's limitations.