TD(λ) for Multi-step prediction problems

1. INTRODUCTION

In this project, I have implemented the Temporal Difference Learning algorithm described in Richard Sutton's 1988 paper Learning to Predict by the Methods of Temporal Differences. I have also conducted the experiments as explained by Sutton and replicated the results for these experiments.

2. PROBLEM DESCRIPTION

Here, we are going to formulate the pseudocode for the Temporal Difference algorithm from the equations and the theorems provided by Sutton, infer the missing parameter values through our understanding of how the algorithm works and also through trial and error. We finally use the procedure to conduct the two experiments using the Random walk example and produce the three figures (Figure 3, 4, 5) illustrated in the paper.

2.1. Temporal Difference Learning

We are framing a **prediction** problem as a **reinforcement** learning problem to be solved, where the agent does not have complete knowledge about its environment and learns from its experiences to predict future actions and outcomes. In Sutton's paper, he describes a particular kind of learning algorithm called temporal-difference learning procedures. The TD or Temporal Difference learning procedures differ from the traditional counterparts in a way that the model learns with each prediction it makes, leading to the outcome. That is, the TD methods have an **incremental** approach towards learning from their own predictions.

Sutton claims that the TD methods perform better than supervised learning methods for **multi-step** prediction problems, where little information about the outcome is revealed over multiple steps rather than as a single input-outcome pair.

2.2 Random Walk Example

Sutton considers a simple prediction problem to describe his findings. A random walk is a simple model, where an agent has a dynamic chance of walking in either right or left side. The walk starts at state D, and the walk moves right or left with equal probabilities through states B,C,D,E,F until the walk leads to either the left terminal state A or the right terminal state G. The outcome value of z = 0 is obtained on ending in terminal state A and z = 1 on ending in state G. This value is considered as the probability of ending in state G.

For this random walk problem, we use the TD algorithm to predict the probability of a right-side termination for each state. We will generate such random walks and measure the prediction computed by the algorithm.

3. TD(λ) ALGORITHM:

The equations for the TD(λ) algorithm can be derived from the paper as follows,
We take only the non-terminal states for this problem, since we already know the predictions for the terminal states.
Let us consider each state as an **observation** vector $\mathbf{x_t}$, represented with a value for that state.
For example, for state B, x_B = {1,0,0,0,0} and so on.
A **sequence** is one random walk from the starting state ending in either terminal states. A sequence will consist of the order of the state vectors. For example, Sequence 1 = {D,E,F,G} and so on.

From equation (4) stated in the paper,
We compute delta weight as,

$$\Delta w_t = \alpha (P_{t+1} - P_t) \Sigma^t_{k=1} \lambda^{t-k} * \text{gradient} \quad \text{Eqn (4)}$$

Also, from the paper,

$$e_{t+1} = \Sigma^{t+1} \lambda^{t+1-k} * \text{gradient}$$

Therefore, we compute e_t as,

$$e_t = \Sigma^{t} \lambda^{t-k} * \text{gradient} \quad \text{Eqn (5)}$$

From equation 4 & 5, we can write delta weight as,

$$\mathbf{\Delta w_t = \alpha (P_{t+1} - P_t) e_t} \quad \text{Eqn (6)}$$

Also, from the paper,

$$\mathbf{e_{t+1} = gradient + \lambda e_t} \quad \text{Eqn (7)}$$

where, P_t is the weight/probability prediction of the current state.
P_{t+1} is the weight/probability prediction of the next state we will transition to.
α is the learning rate of the algorithm, 0 <= α <= 1
λ is the decay rate, 0 <= λ <= 1

4. EXPERIMENTS:

4.1. Training Data

As the first step to conducting our experiments we generate training data as described in the paper. We generate **100 training sets**, where each set consists of **10 sequences**.

Pseudocode for generating training data:

```
For each Training set (1 to 100):
   For each Sequence (1 to 10):
      Initialize sequence
      Starting state = D
      Add starting state to sequence
      Repeat until terminal state:
         Randomly decide to move to the right or left (with 50% probability)
         Add next state to the sequence
```

4.2 Experiment 1

For experiment 1, Sutton computes the error difference in the predictions for various λ values from 0 to 1.

- Weights for the states were **initialized** to **0.5** for each training set.
- A learning rate **α** value of **0.001** was used.
- Each training set was presented to the learning procedure until the weights **converged**. (each training set was sent to the procedure with the previous run's weight predictions
- Convergence was measured to be achieved when the difference between the weights of successive runs were less than **0.001**.
- The weights were **updated** only **after** completing a **training set**.
- Ideal weight predictions as given in the paper, {B,C,D,E,F} = {**1/6, 1/3, 1/2, 2/3, 5/6** }
- The Root mean squared error (**RMSE**) values were computed for each training set using the ideal and predicted weights given as,

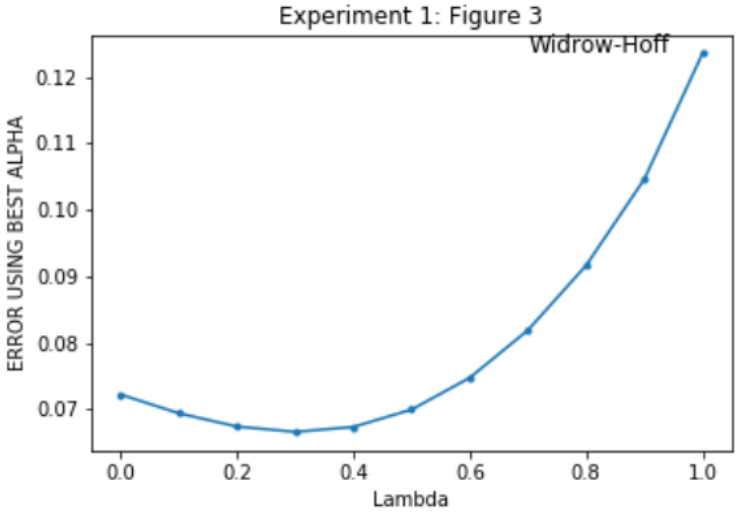
$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (Predicted_i - Actual_i)^2}{N}}$$

- The **average** RMSE value was taken over the 100 training sets.
- This was done for each λ = 0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0

4.2.1 Pseudocode for experiment 1:

TD(λ):

For each Sequence in Training set:
  For each State in Sequence:
    Calculate P_t using vector x_t and weights
    If current state is either B or F, P_{t+1} = 0 and 1 respectively
    Else, calculate P_{t+1} using vector x_{t+1} and weights
    Calculate e_t using eqn(7)
    Calculate delta_weights using eqn(6)
    Accumulate the delta_weights for that sequence/walk
Update the weights at the end of training set as, weights = weights + sequence_weights



4.2.2 Observations:

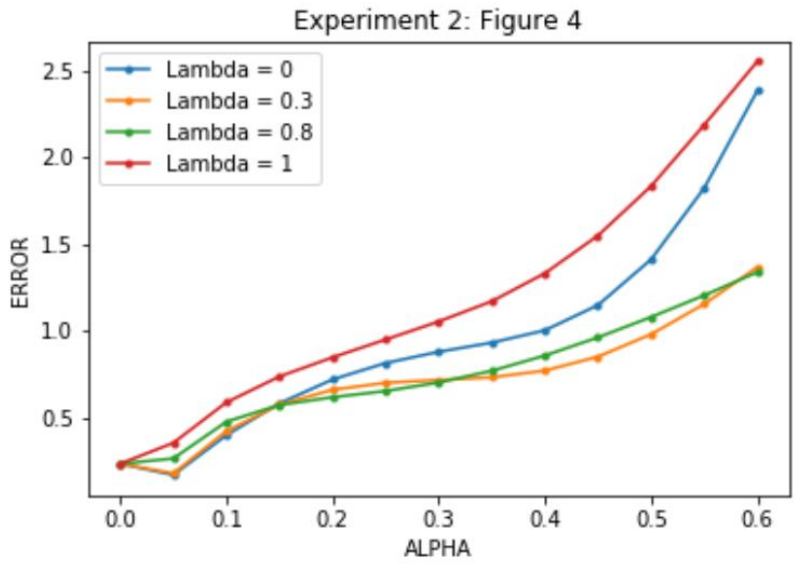
From the figure 3 above, we can see that **λ = 1,** yields the **maximum** RMSE value and also is actually the same as the **Widrow-Hoff** (supervised) learning procedure. My figure 3 differs from the one in the paper where the error seems to decrease through λ = 0.1, before increasing after λ = 0.3 following a similar curve like in the paper. The reason could be that the randomly generated training data may contain **sequences** wherein the **walk oscillates** between **repeated states** and we accumulate wrong weight values at λ = 0, so the error is not at its minimal like in the paper. The reason for error decreasing through 0.1 to 0.3 might be that the λ **balances the eligibility** we allocate to the states. The best λ depends on the nature of the data we are actually training the model with. For the rest of the curve, it follows a similar path to the paper, the error increases because λ is being unevenly distributed to weigh repeated useless state transitions, thereby assigning wrong weights to the states. The RMSE values here are much better from the paper, a number of factors like the **convergence criteria**, **learning rate** and even the **seed** used to generate data play a role.

4.3 Experiment 2

Experiment 2 follows the same pseudocode in 4.2.1 except in the following steps,
Each Training set was presented to the learning procedure only once.
Weights were updated after every sequence/walk.

4.3.1 Figure 4

RMSE values were calculated on the predictions for each λ = 0, 0.3, 0.8, 1 with various 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 values.

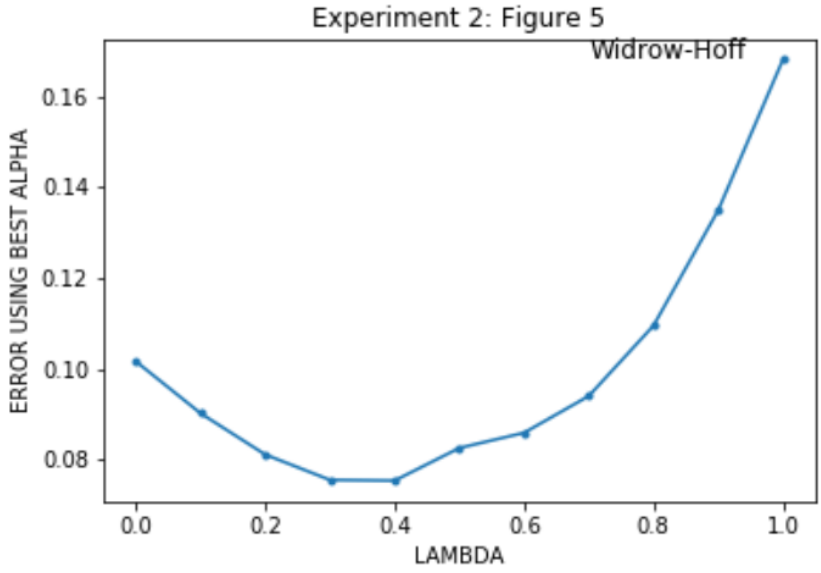


4.3.2 Observations:

Figure 4 has curves for different λ values which follow similar patterns like in the paper. Even in this experiment the widrow-hoff procedure with λ = 1 produces the highest error. We can also note here that RMSE values in this figure are way higher than the ones from figure 3. This is mainly because **high learning rates** caused **bigger errors** and the fact that **weights** are **updated** after each **sequence**, so the procedure might be overfitting the actual weight values. Maybe a few random walks where states are repeated makes the procedure to assign more value to those states than they really deserve.

4.3.3 Figure 5

Following the assumptions for experiment 2, RMSE values were calculated for various λ values 0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0 using the best α value between 0.0 to 0.6.



4.3.4 Observations

From figure 5, we can see that it follows the **same curve** as in the paper. λ = 1 even with the best α yields the highest error here as well. The error is the lowest **at λ = 0.3** similar to the paper. We can note here that the RMSE values are significantly lower than that of figure 4, since we get to choose **very low learning rate** values. It is interesting to note here that higher learning rates tend to lead to higher error because the procedure is learning something that is wrong, like there might be **long sequences of useless walks** between repetitive states.

REFERENCES

"Temporal Difference Learning." Wikipedia, Wikimedia Foundation, 30 Apr. 2019, en.wikipedia.org/wiki/Temporal_difference_learning.

Ziad, and Ziad SALLOUM. "TD in Reinforcement Learning, the Easy Way." Towards Data Science, Towards Data Science, 28 Nov. 2018, towardsdatascience.com/td-in-reinforcement-learning-the-easy-way-f92ecfa9f3ce.

"Numpy.nonzero¶." Numpy.nonzero - NumPy v1.16 Manual, docs.scipy.org/doc/numpy/reference/generated/numpy.nonzero.html.

Root-Mean-Square Error (RMSE) | Machine Learning, www.includehelp.com/ml-ai/root-mean-square error-rmse.aspx.