Project #2 Lunar Lander

Hui Xia (Hxia40)

Georgia Institute of Technology

Git hash: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*Abstract*— In this project, we aim to implement and train a Deep Q-learning (DQN) agent to solve the “Lunar Lander" environment in OpenAI gym. Here we firstly describe our implementations, then investigate how various hyperparameters will affect the performance of the agent.

# Introduction

Reinforcement learning is a major category in machine learning. In 1988, Richard Sutton described a reinforcement learning method, which is temporal difference (TD) [1]. The TD method is now well-received by the computer science community for solving multi-step Markov Decision Process (MDP), which involves a decision-making process of to taking various actions among various states. Compared with conventional supervised-learning methods, the TD method is superior, as it could utilize the information that is delivered during the development of a temporal sequences in a multi-step perdition problems.

The TD method has been developed into on-policy TD methods such as State-Action-Reward-State-Action (SARSA), and off-policy TD methods such as Q-learning [refs]. While these methods have been well-used, they suffer from a major draw-back of requiring discrete states and actions: these TD methods require the algorithm to build and maintain a Q-table, which updates the value of every given state-action combination. Such method is difficult to deal with large and/or continuous state and/or action space. For a problem that has infinite continuous state space, while it is possible to discretize the states, the performance of such algorithm will usually be negatively affected. This is because that discretizing continuous states into several state-action combinations often cannot precisely model the problem, while discretizing continuous states into many state-action combinations will result in the need of maintaining and updating a large Q-table, which become computational-expensive.

To solve the limitation, Q-learning have been coupled with neuron network (NN), and the combined methods is referred as deep Q-learning (DQN). Instead of consistently recording and updating all state-action Q-values in a Q-table, DQN use NN to directly calculate for the preferred actions from the input of states. The calculation of actions still uses Q-learning’s algorithm, which uses a ε-greedy algorithm that choose the argmax action among all actions, based on their respective Q-values. In this report, we implement a DQN algorithm to solve the OpenAI gym Lunar Lander problem, which has continuous state and discrete action. Beyond the algorithm implementation and problem solving, we further investigate how various hyperparameters affect the performance of the DQN algorithm.

# Method

## The OpenAI Gym Lunar Lander Environment

The Lunar Lander problem is from OpenAI gym’s LunarLanderv2 environment, which simulates the process of landing a lunar lander onto a desired site in a two-dimensional world [ref]. The state of this environment is an 8-dimensional vector:

In the vector above, () represents the agent’s position, () represents the agent’s horizontal and vertical velocity, represents the agent’s orientation, represents the agent’s angular velocity, and represents whether the left or the right leg of the lunar lander agent touches the ground.

This environment enables four discrete actions: do nothing, fire left engine, fire right engine, and fire main engine. The reinforcement learning agent should decide what action to take based on the state vector.

For each episode, the agent is expected to land the lunar lander onto the landing pad, which is always located at (0,0). To encourage the agent to land the lunar lander in a timely and efficient manner, the agent receives a small negative reward every time it acts. Furthermore, each firing of the main engine will result a -0.3-point penalty. The total reward for moving the lander from the top of the screen to the landing pad ranges from 100 to 140 points varying on the lander placement on the pad. Each leg touches the ground will enable a 10 points bonus. If the lander crashes lands, the episode is considered complete and it will be receiving additional -100 or +100 points depending on the outcome.

In this project, we aim to solve this environment. Solving this environment is defined as achieving a score of 200 points or higher on average over 100 consecutive episodes.

## Implementing the DQN Algorithm

Q-Learning is an off-policy TD algorithm that does not require an explicit definition of an MDP. It trains an agent to choose and execute the optimal action based on the state of a dynamic environment. “Optimal action” in calculated using a combination of long-term reward and immediate reward. The calculation process is refereed as Q-function, and the respective A screenshot of a cell phone

Description automatically generatedA close up of a device

Description automatically generatedvalue of executing an action under a given state is referred as Q-score. For each step in the MDP, the Q-score is updated using the equation below:

In the equation above, stands for the overall Q-value estimation in the step of the episode, given the state and action of , and is the immediate reward at step t. Alpha ( is the learning rate of the algorithm, which controls how fast the algorithm learns from new experience. To maintain the stability of the algorithm, is usually kept below the value of 0.1. Gamma ( is the discount factor, which controls how much weight it gives to future rewards in the algorithm. Considering that a good landing will grant the agent a bonus of 100-140 points, and a total of 200 points is already considered as problem solved, we would expect a high value of (i.e. very close to 1) will grant us solution to the Lunar Lander environment.

Another set of hyperparameters we implemented in algorithm is ε. Epsilon (ε) is the probability when the algorithm does not choose the “Optimal action” (that as the highest Q-value) but choose a random action. The purpose of having ε is to encourage the algorithm to perform exploration, which is especially important at the beginning of each experiment. While the agent experiences episodes and has accumulates experience for the environment, ε is supposed to slowly decay after each episode under an epsilon decay rate. The decreasing value of ε will move the focus of the algorithm from exploration of the environment to the exploitation of the accumulated experience. To ensure the algorithm have some reasonable ability to explore alone the whole experiment, an epsilon minimum is used to ensure that ε does no decrease below certain value.

As we have mentioned in the **Introduction** section, conventional Q-learning is limited in dealing with MDP that has a continuous state space. To solve such limitation, Q-learning have been coupled with NN and evolved into the algorithm of DQN [ref]. Instead of creating and updating a Q-table, DQN consider the Q-function as a parameterized function which take input states and return actions. A loss function is implemented to measure how well the Q-function performs, and train/optimize the Q-function by calculating gradients of the loss function, until the Q-function converge. In this project, we implemented mean squared error (MSE) as our loss function [ref].

Using DQN, we denote the eight vectors of the states as the input of the NN, and the four discrete actions as the output. The structure of NN could be described as width (i.e. how large is a hidden layer) and depth (i.e. how many hidden layers). In this project, we implement NN using Karas Sequential, which models a linear stack of NN layers, with the optimizer Adam for the compiler. Karas is a NN API, written in Python and is supported by TensorFlow.

One common method to stabilize the DQN is experience replay. If the DQN is only trained using the latest state, it might be trapped in local optima. To solve this problem, a replay memory pool Is used to contain a larger number, e.g. 100000 recent states, which may come from different episodes. After each step or each number of steps, a batch of records of certain size (e.g. 100) from the memory is used to train the NN.

## Experiments

In last section, we discussed several hyperparameters that can be investigated in this DQN model, including and from the Q-learner, ε in the ε-greedy exploration-exploitation strategy, the width and depth of the NN, and the size of memory in the experience replay. In this project, we will firstly investigate the width and depth of NN as well as the size of the memory by training and testing the DQN algorithm, then investigate , , and ε by the DQN algorithm.

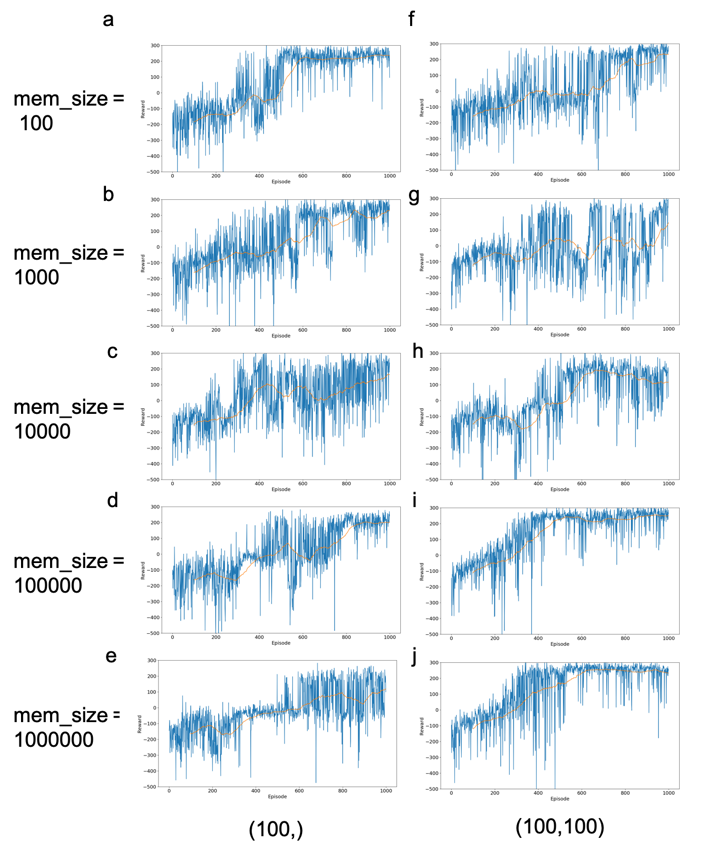
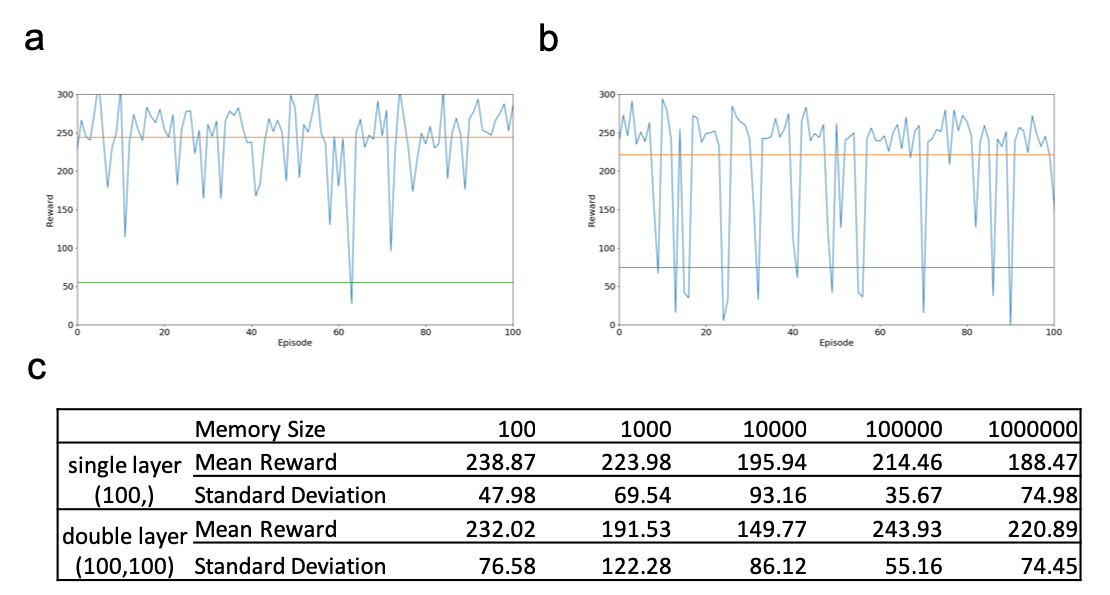
# Results and Discussion

## Neural Network Size Affect DQN Performance

We firstly investigate on how the size (i.e. width and depth) of the NN will affect the training efficiency of the DQN. We tested four one-layer networks of (10,), (100,), (1000,), and (10000,) hidden units, and four two-layer networks of (10,10), (100,100), (1000,1000), and (10000,10000) hidden units. All layers in the network are using RELU activation, except the last layer, which uses linear activation. All layers are dense. As shown in **Figure 1**, for a DQN using one hidden layer in the network, the trained model achieves best performance in about 500 episodes when using single hidden layer of 1000 units. Using single hidden layer of 100 units and train for 1000 episodes can also bring the average rolling reward of the model higher than 200. However, it takes more (about 850) episodes to do so, and the rolling mean rewards is not as high as using the 1000-unit hidden layer. Using a hidden layer that is too large (10000 units) or too small (10 units) cannot grant preferable performance.

As shown in **Figure 2**, for a DQN using two hidden layers in the network, the trained model achieves best performance in about 500 episodes when using two hidden layers of (100,100) units. Using double hidden layers of (1000,1000) units and train for 1000 episodes can also bring the average rolling reward of the model higher than 200. However, it takes more (about 650) episodes to do so, and the performance diverges once more episode has been used for training the model. Using a hidden layer that is too large (10000, 10000), the time-cost is beyond reasonable scope (we speculate this will take about 24 hours using a PC, while all other experiments on par takes less than 30 minutes). Using a hidden layer that is too small (10, 10) cannot grant preferable performance.

Out of our expectation, these experiment results suggest that larger hidden layer(s) are not necessarily advantageous. Rather, the total trainable parameters of the NN is the key to obtain high model training performance. For example, the total trainable parameters of the single hidden layer NN with the layer size of (10,), (100,), (1000,), and (10000,) are 134, 1304, 13004, and 130004, respectively. On the other hand, the total trainable parameters of the double hidden layer NN with the layer size of (10,10), (100,100), (1000,1000), and (10000,10000) are 244, 11404, 1014004, and 100140004, respectively. The best training performance achieved by single and double hidden layer networks are (1000,) and (100, 100), and their total trainable parameters are of comparable size (13004 and 11404, respectively). This observation brings out a new question: is that the complexity of the problem just favors the neuron network of the size of ~10000 total trainable parameters, or it is rather the total size of the experience replay memory favors the neuron network of the size of ~10000? While it is rather unfeasible to modify the complexify of the Lunar Lander problem in this project, we will try to answer this question by scanning through various experience replay memory size.



## Experience Replay Memory Size Affect DQN Performance

As shown in **Figure 3**, neuron networks of single hidden layer (100,) and double hidden layer (100,100) are trained under various experience replay memory size. The figure suggests that the NN that has a smaller number of total trainable parameters do prefer smaller replay memory size. In **Figure 3a-e**, we can see that the training on smaller, single layer NN can solve the lunar lander problem in about 600 episodes using the replay memory size of merely 100. Note that the memory batch size is always 100 for the whole **Figure 3**, this indicates that smaller NN does not need a larger experience replay to solve this problem. On the other hand, in **Figure 3f-j,** the larger double layer (100,100) NN would prefer larger replay memory. Using memory size of 100000, DQN using the double layer NN solved the problem in about 500 episodes. Even larger memory size will result in a slower solution with less mean reward.

The training performance demonstrated in **Figure 3** is confirmed by the testing performance shown in **Figure 4**. That is, NN that has a smaller total trainable parameters outperforms when using a smaller memory size. DQN using single hidden layer (100,) NN reaches best mean reward and lowest standard deviation when using a replay memory size of 100. On the other hand, DQN using double hidden layer (100,100) reaches best mean reward and lowest standard deviation when using a replay memory size of 100000. However, it is notable that 100000 and 100 are also the sub-optimal memory size for the single and double layer NN, respectively. It is likely that no matter what the NN size is, there are certain “stable islands” exist in terms of experience replay memory size. A more thorough investigation using more sizes of NN over various memory size would be beneficial for understanding this problem. Due to the limitation of this project report, we have to stop here. Nonetheless, based on these results, we can choose the NN with double hidden layer (100,100) and memory size of 100000 as optimal for experiments on other parameters.

## Alpha, Gamma, and Epsilon Affect DQN Performance

As discussed in the **Methods Section**, alpha, gamma, and epsilon are widely-used hyperparameters for Q-learner.

convergence. As Sutton did not describe in his paper on how the weight is converged, here we make **assumption (1)**, about

the definition of convergence: if the root mean square error between last and this update (of the weight vectors) is smaller than a parameter delta, we would consider the weight vectors are converged. To justify this assumption, we implemented TD() with various values of delta, trying to reproduce the corresponding experiment results of Sutton. Our **assumption (2)** is that the learning rate,, obviously should be a value between 0 and 1, should ensure the convergence of the algorithm. The learning rate will decide how large steps the algorithm will take to perform updates. Larger value of learning rate could potentially make the algorithm approach to desired values faster, but if the ‘step size’ is too large, the algorithms may never converge.

The difference between the second experiment and the first experiment is, in the second experiment we no longer pursue convergence of weight vectors - each of the random-walk sequence will be presented to the algorithm only once, and the weight vectors will be updated after each of the sequence presentation (compared with the accumulated weight vector update in experiment 1). By updating the weight vectors only once per sequence, we can observe under what parameter condition of and shall the algorithm converge faster. Thus our assumption (1) will be invalid for the second experiment, while assumption (2) remains reasonable.

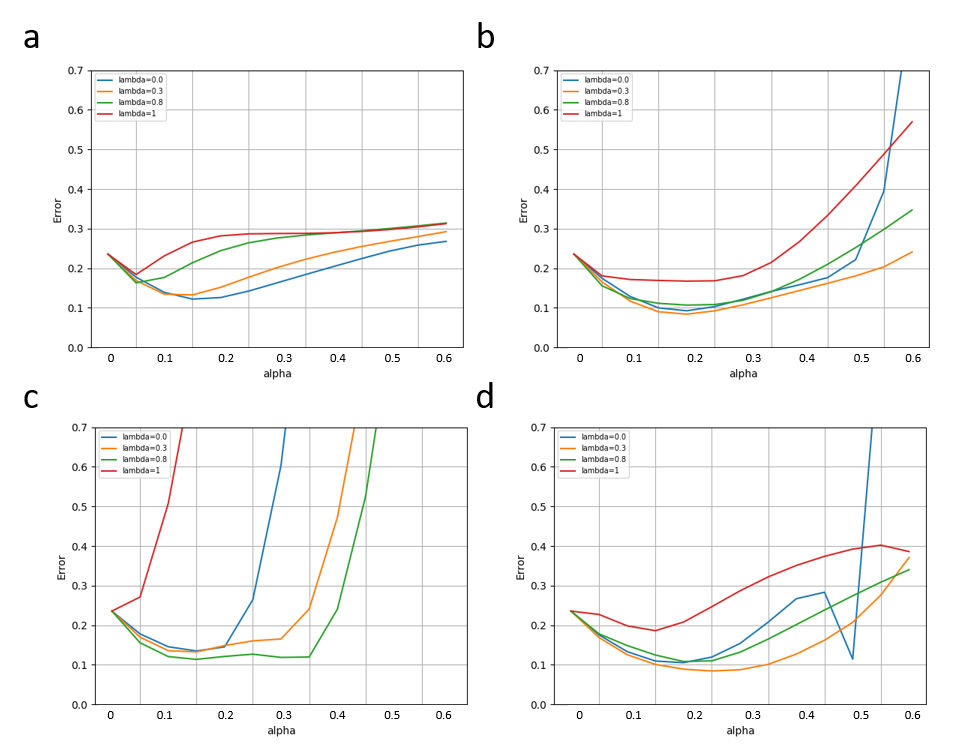
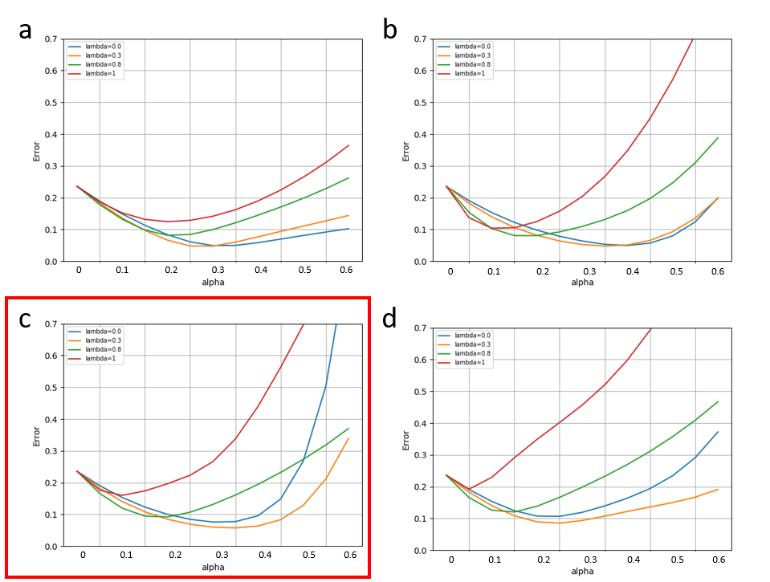
As each sequence will be only presented once (rather than being presented until convergence), the total updates that applied on the weight vectors are extremely limited. For such a small sample pool, outlier case scenarios, such as an overly long random-walk sequence, could potentially bring unstable performance to the algorithm. For example, if an outlier case of the agent randomly wandered around states B and C fifty times, but ended up in edge state G, the weight vectors will be updated with a wrong value. This is especially true when we are using higher value of , such as =1, which treat all historical states equally. In such scenario, the deteriorative effect of data outlier will be more pronounced. Thus, we make **assumption (3)** that to reveal the performance difference among various values of TD() curves, and reproduce Figure 4 of Sutton, a dataset that has a proper amount of data outlier is needed. We will discuss this topic further in the Results and Discussion section. For methods taken here, various numbers of sequences per dataset, as well as various maximum sequence length, have been investigated to find datasets have proper appearance of data outliers. We could justify this assumption if we can find such proper dataset that reproduces Figure 4 of Sutton.

# Results and Discussion

## Reproducing the First Experiment

To justify our assumptions for experiment 1, we implemented the TD() algorithm with various values of alpha and delta. As illustrated in **Figures 1** **and 2**, all combinations of parameters show a range of RMS error between 0.15 and 0.3, similar to the error range shown in Sutton Figure 3, which is 0.19-0.25. For graphs which has an alpha/delta ratio that is ≤ 1, i.e. **Figure 1a** and **Figures 2a, 2b**,the lowest error happens on a point where . For other parameter combinations when alpha/delta > 1, the lowest error on the curve happens on the point where .

The observation that TD(0) does not always have the lowest RMS error is reasonable. In his paper, Sutton has reasoned that although linear TD(0) provide optimal predictions for finite training sets, it is expensive to perform such computation. That is, TD(0) is rather slow at propagating the prediction back alone the sequence, one state at a time. In scenarios when delta is relatively large, e.g. alpha/delta ratio ≤ 1 for our cases, it is very possible that the algorithm consider itself ‘converged’, and stopped weight updating prematurely for the points where is relatively small. Thus, we can justify our assumptions (1) and (2), as using proper RMS error delta and proper , such as alpha = 0.01 and delta = 0.001, we could reproduce the same trend of the curve shown in Figure 3 of Sutton’s paper. Thus, we reproduced Sutton’s discovery that the performance of the TD algorithm improved rapidly as was reduced below 1, and reaches best at = 0.

It is noticeable that while we have reproduced the trend of the curve, it is however hard to reproduce the exact error range of the curve in Sutton’s Figure 3, in which the error ranged between 0.19-0.25. We reasoned that this is due to the randomness of the dataset sample. As the random-walk datasets are generated, well, randomly, using such dataset to predict the true probability of reaching edge state G from each state could result in different errors. Using same parameters of α = 0.01 and delta = 0.001, we investigated the error range over the curve using different random seeds. While all of the resulted curves show the same trend that the performance of TD(0) is superior, the error ranges could vary. When the random seed values of 1, 2, 3, 4, and 5, the corresponding RMS error ranges between 0.12-0.32, 0.04-0.08, 0.12-0.24, 0.07-0.1, and 0.08-0.28, respectively. Thus, without knowing the exact dataset Sutton used to train his TD algorithm, it is well beyond the scope of this project report to reproduce the exact error range of Sutton’s Figure 3.

## C:\Users\hxia3\Pictures\Figure 5.pngReproducing the Second Experiment

We further implemented the procedures described by Sutton for the second experiment. Compared with the first experiment, the second experiment focus on how quickly can a TD() algorithm learn. This feature is useful in many real-world problems, as data for real-world problems are usually limited, and/or expensive to obtain. To simulate this case, the data we are using here is also limited – only 100 datasets are available, and each sequence will only be presented to the algorithm once. Because the data is limited, it is important to ensure a proper representation of data outliers appearing in the datasets.

The purposes of doing so is, the dataset to be used here should have a proper number of data outliers to demonstrate the difference between curves with higher and lower values. The curves that are drawn with higher values are easier to be negatively affected by data outliers (due to the nature of high TD() of distributing higher weight for older observation). Thus, the number of outliers that we use can neither too low to demonstrate no extreme error among all curves, nor too high to make all curves demonstrate high errors. We noticed that if we set the maximum threshold of the sequences length to be 20 steps, most (if not all) of the outliers in the datasets will be removed, featured by the relatively low error in all curves, as shown in **Figure 3a and 4a**. On the other hand, if we do not set limitation for maximum sequence length, high errors will appear in all λ curves in an incomprehensible manner, as shown in **Figures 3d** and **4d**.

After scanning through various sequence length thresholds and sequence number per dataset, our best deliverable to reproduce Sutton Figure 4 is shown in **Figure 4c** in the red frame, which use seven sequences per dataset, and has a maximum sequence length of 50 states. Thus, our assumption (3) that “to reveal the performance difference among various values of TD() curves, and reproduce Figure 4 of Sutton, a dataset that has a proper amount of data outlier is needed” could be justified. More importantly, although the curves plotted using ten sequences per dataset (which is the what Sutton has done, plots in **Figure 3**) appear different compared with Sutton Figure 4, this result still agree with Sutton’s conclusions drawn from his Figure 4. Which is, first, the best results obtained with intermediate values of α; second, for all values, the TD(1) procedure produced the worst estimates. Thus, we can conclude that our reproducing work on Sutton Figure 4 is reasonable.

Compared with Sutton’s Figure 4, which is largely dependent on the appearance frequency of data outliers, Sutton’s Figure 5 is easier to reproduce, as more certainty is involved. **Figure 5** demonstrate curves using various maximum sequence length. All of them show similar trend as Sutton’s Figure 5, in which an intermediate value have the best performance. In Sutton’s Figure 5 (which uses 10 sequences per dataset), such value is somewhere near 0.3. Similar to what happened in reproducing Sutton’s Figure 4, our closest reproduction is still the plot drawn using 7 sequences per dataset, which has a best value between 0.3 and 0.4.

##### Conclusions

Via working on this project report, we gained understanding of TD algorithm, by justifying our three assumptions and reproducing Sutton’s Figures. The major pitfalls that we have encountered in this project is realizing what a reasonable assumptions are, learning to realize how randomness can affect the figures (especially for Sutton’s Figure 4), and learning to focus on the major conclusions.

##### References

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