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 [2]. 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 [3]. 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 [4]. 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 loss function.

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.

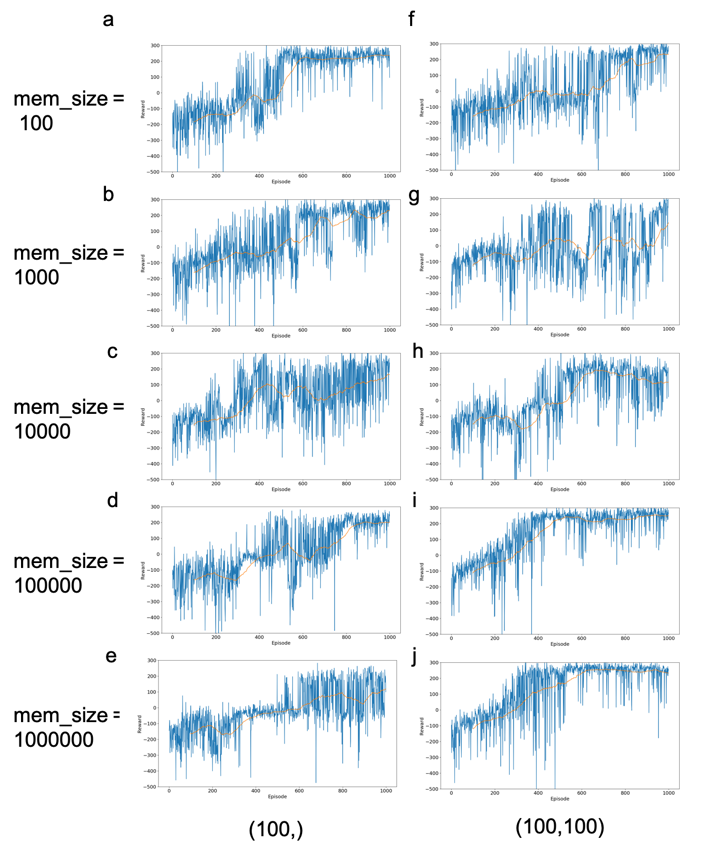
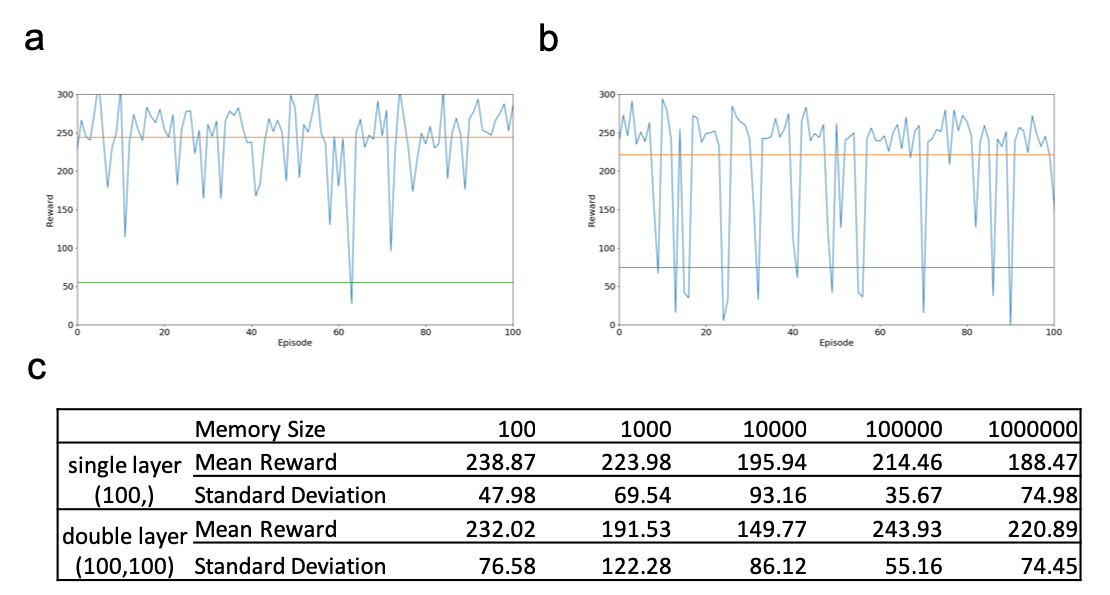
# Results and Discussion

## 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 parameter 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, Epsilon Decay Affect DQN Performance

As discussed in the **Methods Section**, alpha, gamma, and epsilon are widely-used hyperparameters for Q-learner. 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, should be kept low. As shown in **Figure 5**, the DQN training achieves best performance when . Higher value of makes the algorithm too ‘rigid’ can is not able to make delicate adjustments. On the other hand, lower value of updates the Q values too slow and cannot converge the algorithm in best performance within 1000 episodes.

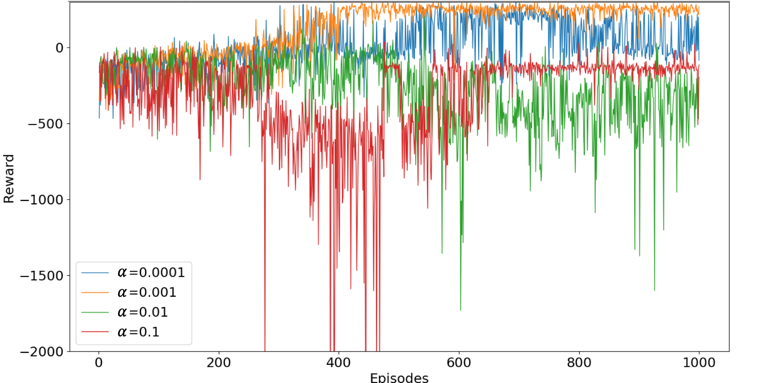


Figure 5. Reward for each episode while training the DQN agent using various alpha for 1000 episodes. NN network using double hidden layers of (100,100). epsilon = 1.0, epsilon\_decay = 0.995, gamma = 0.99, memory\_size = 100000, memory batch size = 100.

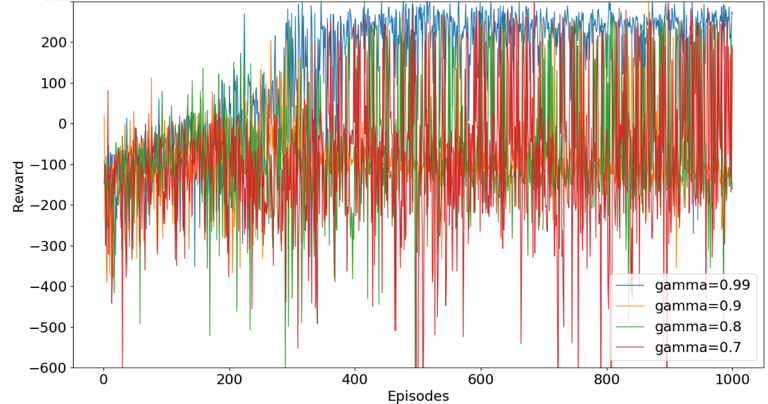


Figure 6. Reward for each episode while training the DQN agent using various gamma for 1000 episodes. NN network using double hidden layers of (100,100). alpha = 0.001, epsilon = 1.0, epsilon\_decay = 0.995, memory\_size = 100000, memory batch size = 100.

Gamma ( is the discount factor, which controls how much DQN algorithm weights future rewards compared with immediate rewards. As shown in **Figure 6**, 𝛾 = 0.99 will result in preferable performance. Other lower values of gamma will decay the long-term reward too much, thus is not favorable when solving this problem. After all, successful landing grants most of the reward points in this problem. On the other hand, the training performance using 𝛾 = 0.999 achieves sub-optimal performance, indicating that the model also needs to take immediate reward into consideration.

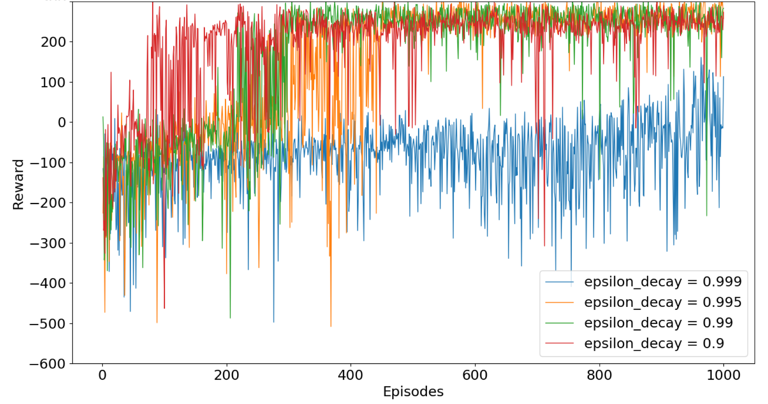


Figure 7. Reward for each episode while training the DQN agent using various epsilon decay for 1000 episodes. NN network using double hidden layers of (100,100). alpha = 0.001, gamma = 0.99, epsilon = 1.0, memory\_size = 100000, memory batch size = 100.

Similarly, the DQN algorithm using epsilon decay within a range of 0.99 to 0.995 will result in preferable training performance. Using epsilon decay of 0.9 prematurely disable the exploration, thus although the algorithm is able to reach higher reward, its performance become less stable. On the other hand, a higher value of epsilon decay of 0.999 overly encourages exploration, making the DQN not able to converge to optimal performance within 1000 episodes.

Overall speaking, the performance of DQN when adjusting alpha, gamma, and epsilon decay is similar to conventional Q-learning algorithm. That is, although DQN and Q-learning themselves are model-free algorithms, there is a model-dependent optimal for the hyperparameters.

##### Conclusions

Via working on this project report, we gained understanding of DQN, by investigating on hyperparameters such as size of neuron network, size of experience replay memory, as well as ‘conventional’ Q-learning hyperparameters such as alpha, gamma, and epsilon decay. By adjusting the hyperparameters, we solved the Lunar Lander environment, and investigated on optimizing the solution. From working on this project, our major findings is, it is very possible that to solve a given MDP environment using DQN, the size of the neuron network should be decided by the mutual effect of the complexity of the MDP and the size of the experience replay memory.

##### References

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

[2] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning,* vol. 8, no. 3-4, pp. 279-292, 1992.

[3] (2020). *LunarLander-v2*. Available: <https://gym.openai.com/envs/LunarLander-v2/>

[4] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature,* vol. 518, no. 7540, pp. 529-533, 2015.