Project 2: Lunar Lander with DDQN

Trung Pham
Tpham328@gatech.edu
1142b98b1bf41034262c3bfd5e9bf2f43fbacb1c (Git hash)

***Abstract*—In this paper, we present and discuss solution to the Lunar Lander (LunarLander-v2) problem, an environment in OpenAI gym. The task was to safely land a spacecraft on the moon surface in 2-dimentional space, taking into account of 8 total dimensional state space (including position, speed, angle speed and landing leg). We solve the problem using Double Deep Q-Network (DDQN) algorithm as well as investigate the pivotal role of hyperparameters tuning, specifically the discount factor gamma, learning rate alpha, and epsilon-greedy. Our experiment results in the success of landing the lander to achieve 200 reward points or higher on average over 100 consecutive runs using DDQN agent, and also highlights the importance of hyperparameter selection in achieving optimal learning outcome.**

I. Introduction

The Lunar Lander problem from OpenAI gym library is a reinforcement challenge, tasking an agent to safely land a spacecraft on the moon by navigating through 8-dimensional state space (including horizontal and vertical position, horizontal and vertical speed, angle and angular speed of the lander, and the touching right and left leg of the lander) and making four discrete action at each time step. These action include do nothing, fire left orientation engine, fire right orientation engine and fire the main engine. The goal is to land safely and as closely to the landing pad as possible, gaining more reward points the closer to the pad as well as successful landing (+100 points). Crashing the lander receive -100 points, each leg ground contact is worth +10 points. Each firing engine action incur a penalty of -0.3 points and -0.03 points for main and orientation engine respectively. The problem is considered solved when achieving a score of 200 points or higher on average over 100 runs.

In the past, similar problem like the Atari games was solved using Deep Q-Network (DQN) algorithm. The performance of this algorithm heavily depends on the hyperparameters selection process, which including selecting the discount rate (gamma), learning rate (alpha) and the exploration probability (epsilon).

Despite its promising result, DQN suffers from overestimation of action values. We have tried to use DQN algorithm for the problem but our average reward points are only around 180 not passing the 200 target. To address this, we moved on to Double Deep Q-Network (DDQN), a modified approach of the DQN algorithm. The DDQN algorithm separate the process of selecting action and evaluating action between two networks, which helps reduce overestimations and lead to better results.

In this experiment, we apply the DDQN algorithm to solve the Lunar Lander problem. In addition, we also focus on selecting and studying the hyperparameters impacts on the efficiency and accuracy of the training agent. Our goal is that by carefully select hyperparameters in using DDQN algorithm, we can find solution to the Lunar lander problem (achieving on average 200 rewards points)

II. Methodology

Reinforcement learning is a branch of machine learning where an agent learns to make decision by performing actions in the environment to maximize future rewards. The agent learns through trial and error, using feedback from its action and reward to adjust its policy. This is a model free approach, where alpha is the learning rate of the agent, cumulative future rewards is discounted by a factor gamma. At the beginning of the learning process, the agent explore the game, playing randomly to gain as much experience and feedback as possible at an epsilon probability. Epsilon is decayed until a small number after the model has gain sufficient experience.

DQN combine traditional Q-learning with deep neural networks to handle high dimensional state spaces, making it's a suitable algorithm for this problem. As mentioned, DQN tend to overestimate the Q-values due to the max operation used in Q-value update and selection action rule. This maximization bias gets worse when we overestimate Q-value or get bad result during exploration.

Double Deep Q-Network (DDQN improve DQN by reducing overestimation bias in action value estimation. . This is achieved by separating the action selection process from action evaluation process, using a separate networks for each process: the policy network/policy online for choosing actions and the target network for evaluating their value. This separation provides more accurate Q-value estimate, leading to better result in our lander solution. DDQN algorithm select the best action using the policy network and evaluating this action using target network. The target Q-value is used to compute the loss against the predicted Q-value from policy network, with gradients used to update the network weights. The target network's weights are updated at a slower pace.

For our DDQN implementation, we employ a neural network with 2 hidden layers, each consists of 128 neurons. We did try it with 64 and 256 and see not much different in solution. The network takes the state as input, processes through each layers with reLU activation function, and outputs Q-values for each possible action. A single layer also was tried but produce worse results.

In addition, we also focus on three important hyperparameters in DDQN algorithm: the discount factor gamma, which affect future rewards, the learning rate alpha, which control how fast network weights update, and the exploration probability epsilon, which determine probability of random actions or exploration during learning process.

1142b98b1bf41034262c3bfd5e9bf2f43fbacb1c

The training process involves iteratively interacting with the environment, storing experiences in buffer memories, and sample mini batches for network update. When buffer is full, oldest memory is overwritten first, we use batch size of 64. Each game terminates if the lander crashed or landed and reward is calculated, this is considered a training episode. We trained the agent using 600 episodes, and used the trained agent to play another 100 games without further learning to report the average reward points. After many trials, we choose the following hyperparameters for our DDQN:

Gamma = 0.99

Alpha = 0.0003

Starting epsilon = 100%, decaying by 0.1% linearly after each time step until reaching ultimate epsilon of 1%. Meaning the agent start exploring randomly at the beginning but eventually prioritized maximizing rewards.

III. Result of Lunar Lander Solution using DDQN

Running to train the agent for 600 episodes, the resulting reward points start converging above 200 points after 200-300 episodes. Using DQN, it is around 400-500 episodes before converge. After 300 episodes, the agent started to achieve 200 reward points on average of the last 100 episodes. The red line in the graph is the 100 episodes moving average. This line is stable and stay above 200 after 300 episodes. This line serves as a smoothing mechanism, highlighting the underlying trend in our model performance.

Figure 1: Reward points on 600 training episodes using DDQN algorithm.

Significant reward results stay above 200 points, but there is present of some reward points result near 0 or negative points. These episodes of poor performance are indicative of challenges within the training process, attribute the several potential factors.

The ultimate epsilon value of 1% with decay rate of 0.1% could contribute to insufficient exploration in later stages of training leading to missing discovering more optimal strategies. Decision in certain state is not optimal due to low frequency of encounter.

Another potential factor could be overfitting, where the model becomes too closely attuned to specific characteristics of the training environment, reduce its ability to generalize varied scenarios within same environment. It also leads to degradation in performance resulting in unexpected low score.

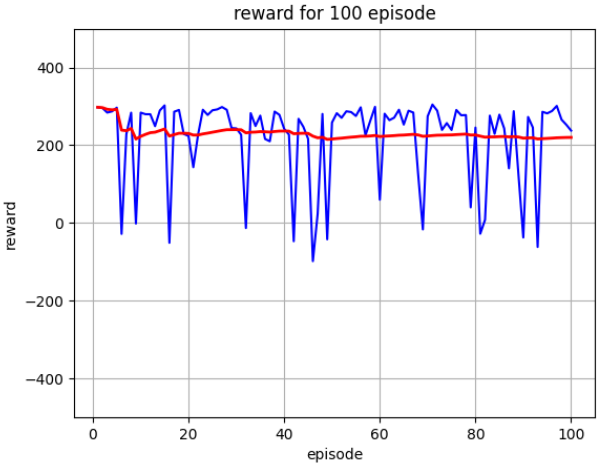


Figure 2: Reward points on 100 games, playing by the trained agent.

Figure 2 is the reward result of playing 100 games using trained agent, our red moving average line stay above 200 points throughout the graph. Despite some small number of bad plays, we successfully solve the Lunar lander problem, achieving 215 reward points on average. This success demonstrates the model capability to navigate the complexities of the environment and ability of DDQN algorithm in solving this particular problem.

There are about 13% of the results that are bad play, achieving less than 180 points. The reason is probably due to overfitting and exploration factor that has been mentioned in the previous section.

IV. Hyperparameter Tuning and Analysis

The performance of DDQN algorithm for solving Lunar Lander problem is significantly influenced by the choice of hyperparameters. This section will focus on our systematic approach to hyper parameter tuning, and the impact of following key hyperparameters: gamma, alpha, and epsilon – and provides a comparative analysis of different settings.

Based on our initial choosing of hyperparameter for the first experiment above. We increase the value of each hyperparameter while keeping everything else unchanged. Each parameter was varied across four values, with each subsequent value being couple times higher than previous one. This exponential progression was chosen to cover a broad range of potential values while observing the nonlinear impact these parameter might have on the model learning process as well as performance. For gamma, we train the model using gamma of 0.99, 0.90, 0.75 and 0.50. For alpha, we train the model using alpha of 0.0003, 0.001, 0.01, and 0.1. For epsilon, we train the model using the ultimate epsilon of 0.01, 0.05, 0.2, 0.5.

1142b98b1bf41034262c3bfd5e9bf2f43fbacb1c

For each value of hyperparameter we chosen, we train the agent for 500 episodes, and produce the reward performance graph for each hyperparameter value. We also summarize the moving average reward lines for each hyperparameter value into one graph. With the y – axis represent the reward points, any graph that going horizontally near the end indicating the model convergence or optimal policy is achieved. The moving average line pointing up indicating high score or good performance. An optimal policy or convergence but achieve low score is not a good policy, so we want to have a policy that goes up then flat.

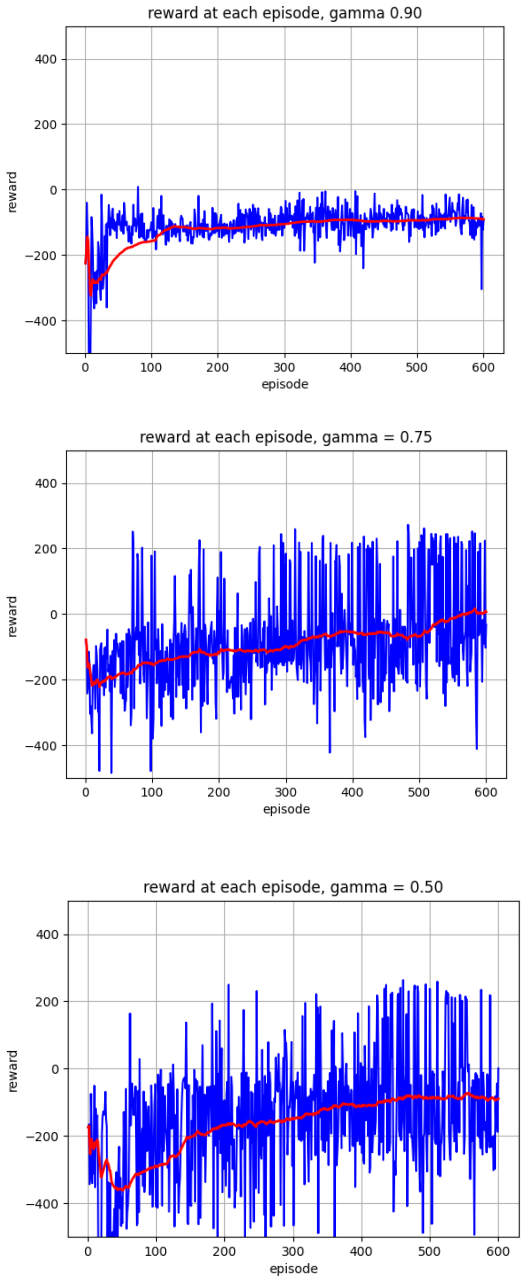


Figure 3: Rewards based on training episodes of varied gamma.

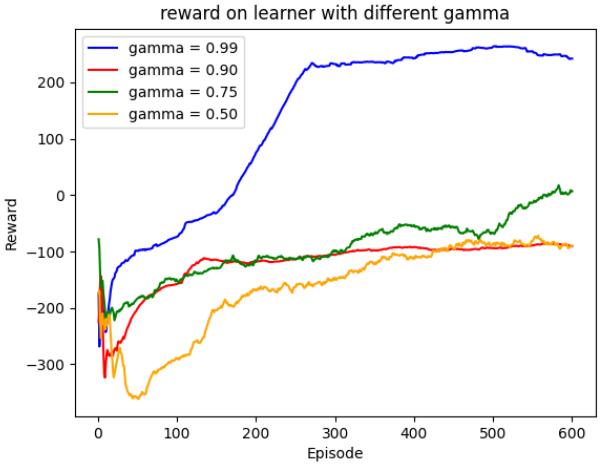
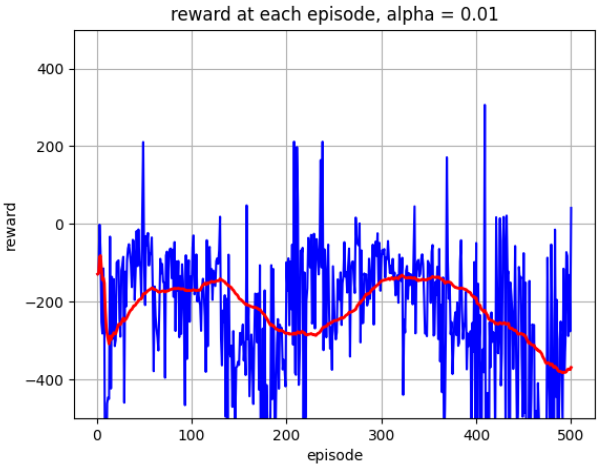


Figure 4: Summary of model performance under varied gamma

Based on figure 4, we achieve best result with gamma = 0.99. Model with gamma = 0.90 or 0.5 results in a converge or policy but not a good policy as it fails to get average reward above 100. Model with gamma = 0.75 seem not to converge yet so more episodes of training need to be done.

Gamma plays pivotal role in how future rewards are valued. A low gamma prioritize immediate rewards, while a higher gamma encourage the agent to consider long-term reward more significantly. We observed that with low gamma, the model seem to be shortsighted, leading to choosing action that result in immediate reward but bad in the long run. With high gamma, the model tend to looks for solution to land the lander successfully for more reward points. This makes sense in term of the problem. As we know, leg ground contact is worth +10 points, a short term reward. Landing the model successfully (+100 points) as well as landing on the landing pad, achieve much higher point. This far outweigh the penalty of firing the engine. As you can see with low gamma, the model optimal policy is landing as fast as possible, getting +10 points for leg touching but resulting in -100 for crashing the model, both line red and orange show a convergence near -100 points.

Here is the result for learning rate alpha test:



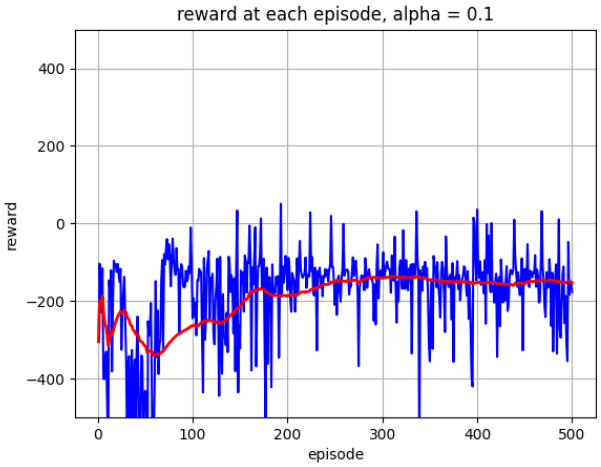


Figure 5: Rewards based on training episodes of varied alpha

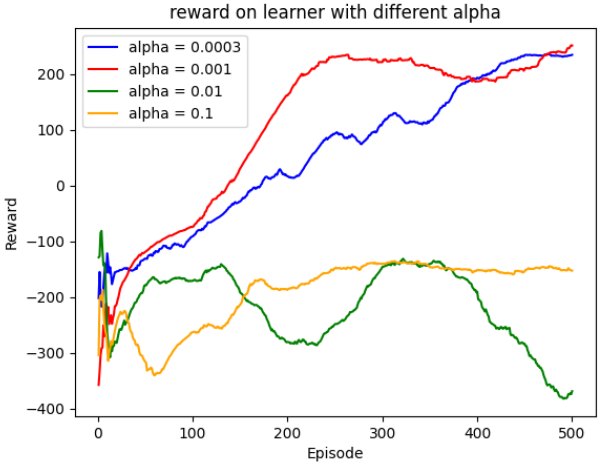


Figure 6: Summary of model performance under varied alpha

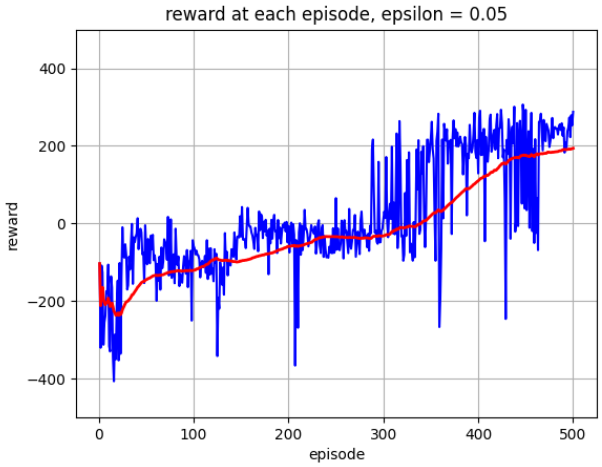
Based on figure 6, we can see that alpha learning rate = 0.0003 and 0.001 is a wining model here. They both solve the problem of achieve 200 average reward points. While alpha = 0.1 converged to negative score and 0.05 not converging.

The learning rate alpha determines how much the neural network's weights are adjusted in response to the estimated error each time the model is updated. That means a low learning rate resulted in excessively slow convergence, as the model was unable to significantly adjust its policy based on received feedback. A high learning rate either led to instability or even divergence. We can see even oscillation in alpha = 0.01 due to aggressive update. The goal of choosing alpha learning rate is facilitating a quick convergence but stable and efficient.

As seeing in figure 6, our choosing of alpha = 0.0003 is good, but inefficient, we could have achieved a faster convergence by choosing a little higher learning rate like alpha = 0.001, which showed converged after 200 episodes, while alpha = 0.0003 takes 400 episodes.

Epsilon dictates the balance between exploration (taking random actions) and exploitation (taking best value action). There are other aspect of epsilon like starting epsilon value and the decay rate of epsilon (how long until model prioritize best action over exploration). For simplicity, we only test on the ultimate epsilon rate of the model. We initiated training with epsilon of 1.00, meaning 100% exploring at the beginning of the training process. Since this is model-free approach, the model need to randomly play the game to get initial feedback or information needed. After each time step, epsilon started decaying by flat 0.1% until reaching an ultimate value. On this test, we tested the trained model under ultimate epsilon value of 0.01, 0.05, 0.2, and 0.5.

Here is result of epsilon test:



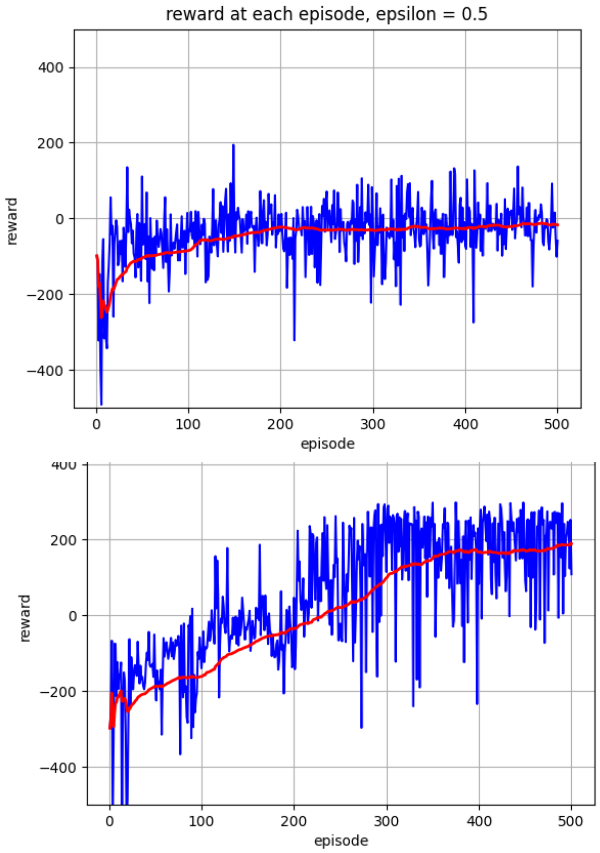


Figure 7: : Rewards based on training episodes of varied epsilon

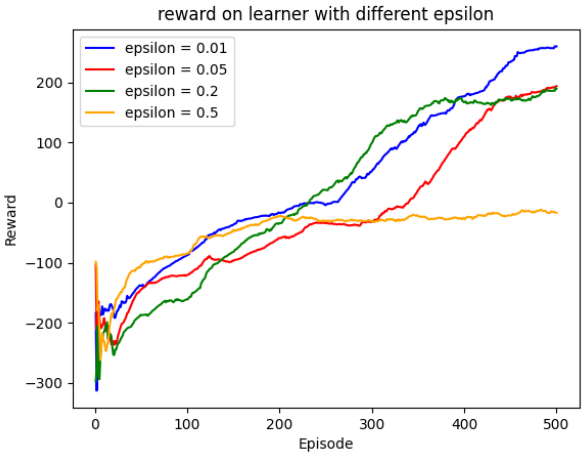


Figure 8: Summary of model performance under varied epsilon

Based on figure 8, epsilon = 0.5 is not good because it wander off too much from its main task, maximizing reward points. The other epsilon value of 0.2 to 0.01 are both good enough. Lunar Lander is a complex problem that require much exploration. It seems epsilon = 0.01 produce the best model among other epsilon values. This seem to be the case as the problem focus more on maximizing points or land the lander successfully . A better strategy would be to decay epsilon slower, give more chance of exploration initially before reduce exploration rate to 1%.

V. DRAWBACK

One drawback of our current experiment is that we are limited in processing power to expanding hyperparameter search. Our model could take advantage of utilizing more advanced hyperparameter optimization method.

We are using a linear decay epsilon strategy but can also opt for a more complex exploration strategy. Ultimate epsilon of 1% might have limited exploration in the later stages of training, contributing to episodes of low performance. Given more time, we can integrate more adaptive exploration strategy based on the agent's performance.

Observation of bad episodes, potentially indicate overfitting in the model. Incorporating regularization method like dropout can reduce overfitting by encouraging simpler models.

VI. CONCLUSION

Lunar Lander is a complex reinforcement challenge that can be solved using Double Deep Q Network (DDQN) algorithm. The application of DDQN significantly improved the agent ability to make decision to safely land the lander, prioritize the landing goal of the problem. In addition, our investigation and sensitivity testing of the hyperparameter revealed the significant impact of hyperparameter tuning on the agent's learning performance and efficiency. Solution to Lunar Lander problem cannot be achieved without correctly select hyperparameter. Optimal settings for discount factor identify the needs to prioritize long term goal. Choosing learning rate help improve performance as well as efficiency and selecting epsilon to balance exploration-exploitation trade off. The right parameter settings significantly improve the agent learning performance, avoiding pitfall like overfitting, underfitting, instability and inefficiency.

Given more time, we could study and employ a systematic approach to tuning, or optimization for hyperparameter, potentially automating and accelerating the tuning process. We can also expending the model to other environment to better understanding, testing robustness or finding generalization of selecting parameters.

REFERENCES

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, "Playing Atari with Deep Reinforcement Learning, December 2013.

[2] Havo van Hasselt, Arthur Guez, David Silver, Deep Reinforcement Learning with Double Q-Learning, Dec 2015.

[3] Adam Paszke, "Reinforcement Learning (DQN) Tutorial"," in Pytorch.org. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html.

[4] Richard Sutton, Andrew Barto, "Reinforcement Learning, second edition", Nov 2018