CS 7642 Project 2: Lunar Lander

Ali Alrasheed - Ajar3@gatech.edu
Git hash: 44fd83bcd8feab53418e3d3aa2c4e02258f3ea38

Abstract

This paper aims to implement and train an agent that can successfully land the "Lunar Lander" spaceship. The implementation is simulated using the OpenAI gym environment "LunarLander-v2" [1]. The agent implementation is based on DQN (Deep Quality Network) method developed in [2]. It is shown that the trained agent was able to obtain an average reward over 200 for 100 independent testing episodes. Furthermore, the effect of four hyper-parameters on the performance of the agent were investigated. In particular, the agent was trained using different value of discounts rates, decay rates, learning rates, and Replay Memory size.

1. Introduction

In this paper, I present an implementation for solving "LunarLander-v2" provided by OpenAI gym environment.The trained agent should be able to successfully land the spaceship while achieving an average reward of more than 200 based on 100 independent testing episodes.

1.1. Problem Definition

The "Lunar lander" is a classical rocket trajectory optimization problem [1]. The aim of the problem is to land the spaceship without crashing the aircraft. There are 8 continuous states of the aircraft (x, y, $\dot{x}$, $\dot{y}$, θ, $\dot{\theta}$, Leg_L,Leg_R). x and y corresponds to the horizontal position and vertical position, $\dot{x}$ and $\dot{y}$ corresponds to the horizontal and vertical speed, θ and $\dot{\theta}$ correspond to the angle and angular rate, and finally Leg_L andLeg_R are Boolean states that correspond to whether or not the left and right leg is touching ground. In addition, there are four discrete actions: do noting, fire left orientation engine, fire main engine, fire right orientation engine.

Furthermore, the aim of the agent is to maximize rewards. The agent would get rewarded +100-140 points if the aircraft moved from the top of the screen to the pad while coming to rest. If the aircraft crashes, it received an rewards of -100 points. The agent get also penalized If the aircraft deviates from the pad. If the agent is able to successfully make the aircraft comes to rest, it receives additional +100 points . Furthermore, the award of either leg touching the ground is +10. Finally, the agent loses 0.03 points per frame every time the engine get fired. The problem is considered solved if the agent is able to obtain more than 200 points of rewards averaged over 100 independent testing episodes.

1.2. Q-Learning

Q-Learning is one of the most popular off-policy and model-free reinforcement learning algorithm. The algorithm aims to estimate the value of each state-action pair. The Q-learning algorithm uses experience to incrementally update its value. Given infinite experiences for each state-action pair, and small learning rate, Q-learning is guaranteed to converge to the optimal policy [7]. The update rule can be given as:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{\bar{a}} Q(\bar{s},\bar{a}) - Q(s,a)) \tag{1}$$

where α is the learning rate, r is the reward for taking the action, γ is the discount rate, and finally $\bar{s}$ next state given action a. The main idea here is that the Q value is updated towards the target value which the reward plus the maximum Q value of the new state.

However, It must be pointed out that since the traditional Q-learning algorithm uses look-up tables to store the state-action pairs, it is only practical when the number of states is not very large. For the context of our Lunar lander problem, the states are continuous which means that the states are infinite. Thus, approximating the Q value for each state-action pair is necessary. In this paper, I will focus on implementing a Deep Q-Network method for estimating the Q-values.

1.3. Deep Q-Network (DQN)

For continuous states-actions MDPs, it is impractical to use simple look-up table to store the Q-values. Thus, the Q-value needs to be estimated using linear or non-leaner methods. There are many ways to estimate the Q-value function. A popular way is using

Deep Q-Network (DQN) described in [3]. Using a Deep Neural networks to estimate Q-value function is relatively not a new idea. However, neural networks in Reinforcement Learning generally suffer from 2 issues: overfitting to the training data, and instability in the training. In particular, overfitting could be a major challenge in Reinforcement learning since the training data comes from consecutive experiences which are highly correlated. This results in both unstable training and overfitted model. [3] overcomes these challenges by using 2 techniques: Replay Memory, and a Target Network. The idea of the replay memory is to store experiences in First-In-First-Out buffer and then randomly sample mini-batch to train the target network using stochastic gradient decent (SGD).

The Q-target network can be trained to minimize the loss function which is usually the squared difference between the target and the estimated Q-value. The ideal loss function to update the target model is given as:

$$L = \sum_{s,a} (Q^{\star}(s,a) - Q_t(s,a,\theta))^2 \tag{2}$$

Where $Q^{\star}$ is the true Q-values, and Q_t is the estimated Q-value using the parameters θ. However, since the DQN is an off-policy algorithm, the true Q-value is not known. Thus, DQN relies on Bellman equation to estimate the TD target. Thus, the loss function will be given as follow:

$$L = \sum_{s,a} (r + \gamma \max_{\bar{a}} Q_t(\bar{s}, \bar{a}, \theta_{i-1}) - Q_t(s,a,\theta_i))^2 \tag{3}$$

Therefore, the TD target is estimated using the same Q_t network but using the weight from the previous iteration. However, since the weight of Q_t network is updated every iteration, the training is not very stable as the Q-network is chasing a moving target. For that reason, the second version of DQN was introduced in [2] which improved the stability of the training by having two identical network: Q-action, and Q-target. The Q-action is updated every iteration based SGD of the loss of the mini-batch sampled from the replay memory. The Q-target network is used to estimate the TD target, and it is only updated its weight to the Q-action weights every C steps. Therefore, the DQN algorithm can be illustrated as in algorithm 1.

Although there are more sophisticated algorithms that could be used to solve the lunar lander problem such as Double DQN, the DQN algorithm illustrated in pseudo-code 1 was chosen since it was proven to produce the required performance. The trained DQN agent was able to successfully land the lunar lander aircraft while achieving an average rewards over 200 points using 100 testing episodes.

Algorithm 1 Deep Q-Learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with weights θ
Initialize target-value function $\hat{Q}$ with weights θ^{-}
for $episode = 1, M$ **do**
  for $t = 1, T$ **do**
    Select a random action a_t with probability ε.
    Otherwise, select $a_t = \arg\max_a Q(s_t, a; \theta)$
    Execute action a_t, collect reward r_{t+1}
    Observe next state s_{t+1}
    Store the transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$
    Sample mini-batch of transitions from $\mathcal{D}$
    Set $y_j =$
    $\begin{cases} r_{j+1}, & \text{if } s_{j+1} \text{ is terminal} \\ r_{j+1} + \gamma \max_{a'} Q(s_{j+1}, a'; \theta^{-}), & \text{otherwise} \end{cases}$
    Perform a gradient descent step on
    $(y_j - Q(s_j, a_j; \theta))^2$
    Every C steps, set $\theta^{-} \leftarrow \theta$
  end for
end for

2. Experiments and Results

The trained DQN agent was based on 3 fully-connected layers: 2 hidden and the output layers. The input layers has 8 nodes which are the states of the aircraft (x, y, $\dot{x}$, $\dot{y}$, θ, $\dot{\theta}$, Leg_L,Leg_R) while the output layer has 4 nodes corresponding to the possible action by the aircraft which are: do noting, fire left orientation engine, fire main engine, fire right orientation engine. The number of nodes in the hidden layers are considered hyper-parameters that were explored in the training process.

Furthermore, the loss function used to train the agent was Mean Squared Error (MSE), and SDG optimizer was used to update the weight of the network.

2.1. Training

An essential step in the training is selecting decent parameters to train the agent. To facilitate the process, Optuna [4] framework was used to find an acceptable hyper-parameters. In each loop, the Optuna function trains an DQN agent then test it on 100 independent testing episodes. Then, the hyper-parameters that produce the highest average rewards in the testing is selected. The Optuna function has executed 20 trials, and the best hyper-parameters among the 20 trials were as illustrated in table 1.

Parameter Name	Value
Replay Memory Size	10000
Discount rate γ	0.99
Decay rate for ϵ	0.991
Update Frequency C	1000 steps
learning rate	0.00065
Number of nodes: layer 1	88
Number of nodes: layer 2	50

Table 1: Selected hyper-parameters for the trained agent

Figure 1 shows the rewards for each episodes in the training process. The orange line depicts the 50 moving average of the rewards. It can be seen from the graph the DQN agent was able to learn from experiences and obtain over 200 pints as moving average. It could be also noticed that training first 300 episodes was slow. This is because the DQN agent uses the ϵ-greedy technique to explore the environment using random actions, then as ϵ decay to every low value (0.05) the agent started exploiting the environment (after 300 episodes) as it used what it learned from the previous experiences.

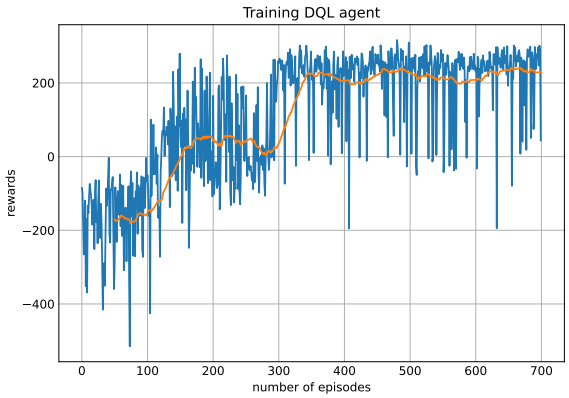

Figure 1: Training agent using 700 episodes

2.2. Testing

The trained DQN agent using the hyper-parameters described in table 1 was used to simulate the rewards for 100 independent testing episodes. The result of the test is given in figure 2.

The aim of this project is to train an RL agent that obtain an average rewards of over 200 points in the "LunarLundar-v2" simulator from OpenAI gym. As it can be seen from figure 2 the trained agent were able to obtain an average rewards of 250 points using 100 independent episodes.

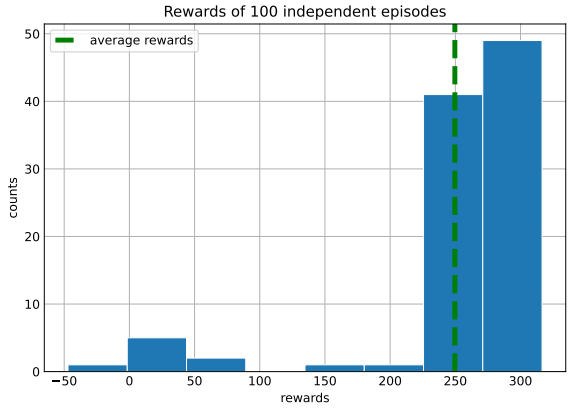

Figure 2: Rewards for 100 independent episodes

2.3. Effect of Hyper-parameters

Choosing appropriate hyperparameters is very crucial in order to train a well-behave agent. In this section, we will analyze the effect of 4 parameters which are learning rate for the SDG optimizer, discount rate for the rewards, decay rate for ϵ-greedy, and the replay memory size. Theses hyperparameters were chosen because they are the important parameters to tune in order to obtain a satisfactory performance. The hyper-parameters will be trained on 700 episodes, and will be evaluated based on the average of 100 independent episodes.

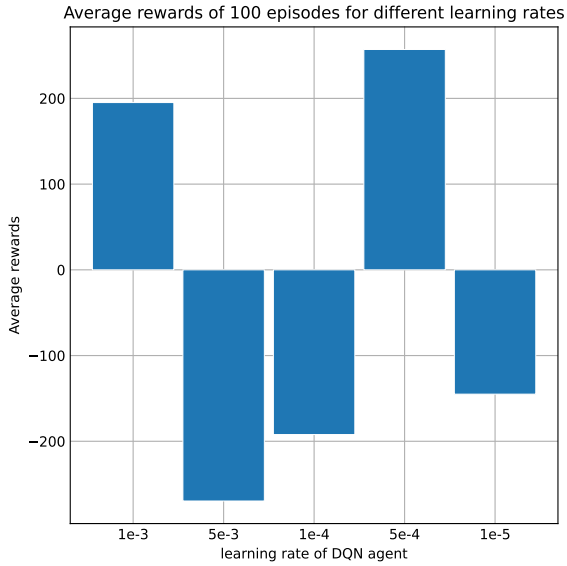

Figure 3: Average Rewards of 100 testing episodes using different value of learning rate

The learning rate is a crucial parameter that need to

be carefully tuned. Too large learning rate will results in very bad learning while too small will results in very slow learning. Thus, to evaluate the effect of the learning rate, the value of the parameters in table **??** were kept the same while training and testing with different value of learning rate.

Figure 3 shows average Rewards of 100 testing episodes using different value of learning rate. It can be seen from the figure that performance of the agent is very sensitive the learning rate. Figure 3 shows that it is not straightforward to choose the best learning rate. Having a value near 0.001 produced an acceptable result. However, decreasing the learning rate to 0.005 or 0.0001 seems to produce terrible results. However, a value near 0.0005 seems to work very well compares to the rest of the values. Therefore, experimenting with different ranges of learning rates is very important to obtain good result when training DQN agent. Another

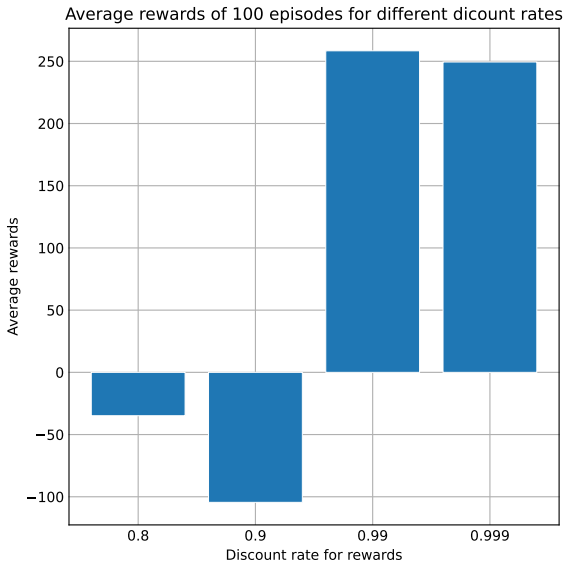

Figure 4: Average Rewards of 100 testing episodes using different value of discount rates

important parameter is the discount rate γ. The discount rate is used to reduce the importance of future rewards. It can be seen from figure 4 that having low discount rate less than 0.9 produced negative average rewards. This could be due to the fact that the rewards of landing successfully greatly discounted so the agent does not try to land but rather stay in the air for longer time. However, a value of 0.99 or 0.999 produce much better result with and average of near 250 points. This is because the value of high rewards at the end (successful landing) was not reduced significantly.

A third parameter that was evaluated is decay rate of ϵ-greedy. In ϵ-greedy, ϵ starts with a value of 1 and then multiplied by the decay rate every episodes in the training process. The ϵ-greedy determines the exploration-exploitation strategy of agent, a balance between the exploration and exploitation is crucial obtain a good estimate for the Q-value. Setting the decay rate to appropriate value depends also on how many episodes will be used to train the agent, higher number of episodes will allow for higher value of the decay rate. In figure 5, it is shown that a value of 0.9, 0.95, or 0.99 was enough for the agent to explore and exploit the environment. However, a value of 0.999 was not enough for the agent to finish exploring the environment. In fact, the ϵ value after 700 episodes with a decay rate of 0.999 will be approximately 0.496. Therefore, a decay rate of 0.999 will need more than 2500 episodes to produce good results.

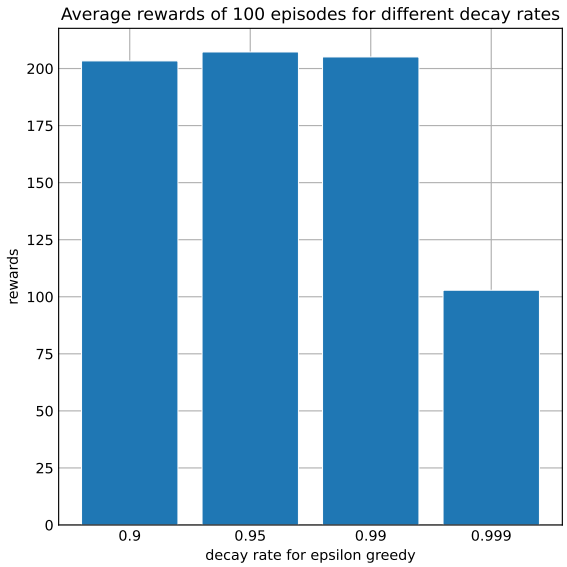

Figure 5: Average Rewards of 100 testing episodes using different value of decay rates

Finally, the Replay Memory size was also evaluated due to its high effect on the performance of the agent. As expected, the higher the Replay memory size the better the DQN agent can generalize to new episodes. This can be seen in figure 6. A Replay Memory size of 1000 was not enough for the agent to generalize well as the 1000 sample in the buffer would likely be correlated. A Replay Memory size of 5000 was much better than 1000 but still near 0 average rewards. However, a value of above 10000 sample in the Replay Memory was enough for the agent to learn and avoid overfitting. A size of 15000 were able to achieve an average of over 250 points of rewards.

To sum up, this subsection showed how important tuning and selecting the hyper-parameters of the DQN agent in order for the agent to generalize and learn well. Choosing unsuitable hyper-parameters could lead to unstable learning and not well-behave agent.

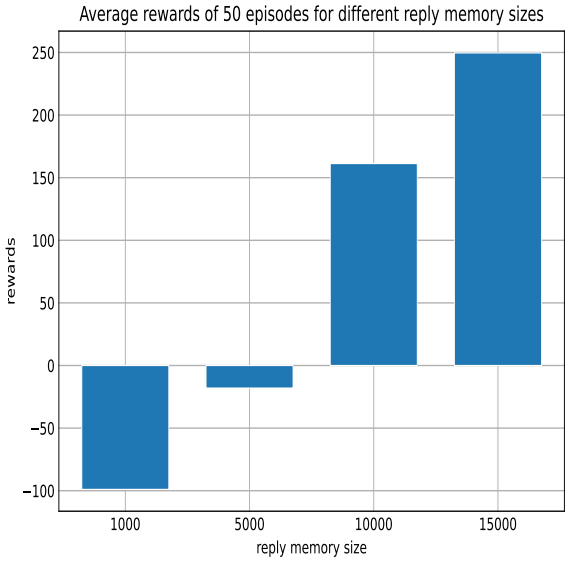

Figure 6: Average Rewards of 100 testing episodes using different sizes of Replay Memory

3. Future Improvement

It must be noted that the DQN method was chosen because it produced the required perfomance in this project. Thus, there was no need to implement more complicated methods. However, there are couple of ways to improve the agent performance of the lunar Lander. In this section, I will mention two ways to improve the current implementation. First, DQN suffers from overestimating the Q-value. Therefore, an improvement to the DQN is using Double DQN instead that was introduced in [6]. In Double DQN, the estimation of the Q-value is improved by decoupling the target and action networks where the action network is only used to choose the next action (ϵ-greedy), and the target network is only used to estimate the Q-value.

A second possible improvement is using a prioritized replay memory introduced in [5]. In prioritized replay memory, the mini-batch is not randomly selected but rather the sample are weighted based on the how large their TD error is. Samples with higher TD error get selected with higher probabilities.

4. Conclusion

In this paper, DQN agent was trained to successfully land the "Lunar Lander" spaceship with an average rewards over 200 points. Furthermore, the effect of hyper-parameters on the performance of the agent were investigated. It was shown that choosing an appropriate hyper-parameters for the DQN is crucial for stable training of a DQN agent.

References

[1] *LunarLander*. `https://www.gymlibrary.ml/environments/box2d/lunar_lander/`.

[2] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[3] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[4] *optuna*. `https://optuna.org/`.

[5] Tom Schaul et al. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[6] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[7] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3 (1992), pp. 279–292.