Solving the OpenAI Lunar Lander Problem using Deep Q-Learning

Praveen Jawaharlal Ayyanathan[1]

[1]Aerospace Engineering, Auburn University

1 Introduction

The ability to land rockets softly is crucial for the potential future human exploration of other planets and moons, as it would enable spacecraft to land and take off again without being damaged, which would be necessary for conducting multiple missions. Additionally, For atmospheric flight problems, the ability to land rockets gently is important for several reasons, including reusability, which reduces the overall cost of space launches. Soft landings are also critical for preserving the integrity of payloads, such as scientific equipment, that may be attached to the rocket. It has been shown recently that Deep Neural Networks (DNNs) can be trained to learn optimal control during pinpoint landing (1). In (2), a DNN is introduced to improve the computational efficiency of the fuel-optimum lunar landing problem. Using a DNN it is possible to map a nonlinear relationship between input and output. For instance, instead of using the classical approach of forming the $Q-$tables, it is possible to get the $Q-$values by using a Neural Network. This project solves a simplified version of a lunar lander problem under OpenAI Gym environment using Deep Q-Network (DQN). Also, a study has been done to learn how the hyperparameters of the Neural Network affect the rewards obtained as the training progresses.

2 Problem Statement

The Gym is an open-source Python library with several environments that represent general Reinforcement Learning (RL) problems. Lunar Lander is a part of the Box2d environments and is a classic rocket trajectory optimization problem. The environment is shown in Figure (1). The agent in this environment aims to land the lander between the two flags. This is an interesting problem as it represents a simplified version of a landing problem but is complex enough to get an experience on how the tuning of hyperparameters involved in DQN results in a good reward. This problem has been solved with uncertainties in (3). However, uncertainties are not considered in this project.

The state vector for this problem is

$$X = [x, y, \dot{x}, \dot{y}, \omega, \dot{\omega}, b_1, b_2], \tag{1}$$

where x and y denotes the two-dimensional position, $\dot{x}$ and $\dot{y}$ denotes the linear velocities, ω and $\dot{\omega}$ represent the orientation in space and the angular velocity respectively, b_1 and b_2 are Boolean

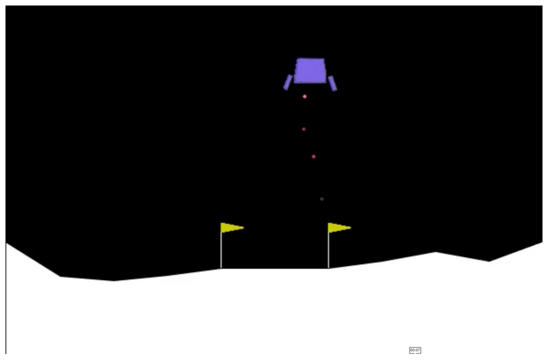

Figure 1: Environment of the Lunar Landing Problem.

that represent the left and the right legs touching the ground respectively. The x coordinate is 0 at the line connecting the center of the landing pad (in between the two flags) to the top of the screen and the y coordinate is 0 at the level of the landing pad. The action space consists of 4 discrete actions: 1) do nothing, 2) fire left engine, 3) fire right engine, and 4) fire main engine. By providing information about the current state and action to gym environment libraries we can get information regarding the next state and reward.

The reward for this problem is as follows:

- -100 points for crashing
- +100 points for coming to rest
- +100 to +140 points for moving from the top of the screen to the landing pad and coming to rest
- +10 points for each leg in ground contact
- -0.3 points for each frame for firing the main engine
- -0.03 points for each frame for firing the side engine

The problem is considered to be solved if the cumulative reward is 200 points or above. The starting point of the lander is from the top center with a random initial force applied to its center. An episode terminates if the lander crashes (i.e., its body touches the ground) or the lander gets outside the viewport (i.e. when the absolute value of x-coordinate becomes greater than 1). Additionally, the episode also stops if the lander does not move. The information regarding the termination of an episode can be obtained from the *done flag* of the environment library.

3 Deep Q-Learning

Q-Learning is a model-free classical RL algorithm that learns the Q-Function directly from experience. Classically, the Q-function is represented in a tabular form that is updated iteratively and this does not scale well when the state space is large. However, it is possible to use function approximation to represent the Q-Function. Therefore the Q-Function now becomes

$$Q(s,a) = \Phi_\theta(s,a) \tag{2}$$

where $Q(s,a)$ is the Q-Function, s and a denote the state and action respectively, $\Phi_\theta : S \times A \to \mathbb{R}$ is a function parameterized by $\theta \in \mathbb{R}^p$. Here S and A denote the state and action space respectively. One way to perform function approximation is to use Artificial Neural Networks (ANN). When using ANN, Φ is determined by the number of layers, the number of neurons, and the choice of activation functions, and θ are the weights of the network. The fitting process is then called training and is given as

$$\min_\theta \sum_{k=1}^{K} \left| Q\left(s_k, a_k\right) - Q^*\left(s_k, a_k\right) \right| = \sum_{k=1}^{K} \left| \Phi_\theta\left(s_k, a_k\right) - Q^*\left(s_k, a_k\right) \right| \tag{3}$$

where Q^* denotes the actual optimal Q-Function.

To let the agent explore every action and state it is important to consider exploration strategies. In this project, the ϵ-greedy strategy is used. An exploration probability term, β, is initially taken as 1. After each episode, it is multiplied by an exploration decay factor of 0.995, so that as the episode number increases, the actions are not randomly taken (like in the exploration step) but are exploitatively taken by considering the $Q-$values. This is done so that the agent explores the action and state space during the initial episodes of the training and as the episodes increase the agent's action will rely on the DQN. The $Q-$values for the network are obtained as

$$Q(s,a) = r + \gamma Q(s', a'), \tag{4}$$

where s and a are the current state and action respectively, s' and a' are the next state and action respectively, and γ is the discount factor.

4 Method

For this project, a DQN is implemented in python using Keras which is a high-level API for tensorflow. One of the important parameters to consider while implementing a DQN is the number of layers and the number of neurons in each layer. The **input size** will correspond to the number of states (8) and the **output layer size** will correspond to the number of actions (4). Keras offers a variety of optimizers that can be used during the training process. For this project the training is performed using the *Adam* optimizer along with the *mean squared error* loss function. While using the *Adam* optimizer, *learning rate* is a hyperparameter that determines the step size at each iteration while moving toward a minimum of the loss function. Also, the *batch size* should be fixed which affects the training process. Another important thing to consider is the activation function for each layer. There are several activation functions available and Rectified Linear Unit (ReLU) is the standard-to-go activation function as given in (4). Therefore for this project, ReLU activation function will be used for the hidden layers and a linear activation function will be used for the output layer.

4.1 DQN-1

As a first step, I am considering 4 hidden layers with 256 neurons in the first hidden layer, 128 neurons in the second hidden layer, 64 neurons in the third hidden layer and 32 neurons in the fourth hidden layer for the policy network. The values for the different hyperparameters are:

- learning rate - 0.001
- discount factor - 0.99
- batch size - 64

The plot of Reward for 1000 episodes is given in Figure (2). It can be seen that the average reward does not show much improvement. Also, the simulation took more than 11 hours to complete 1000 episodes.

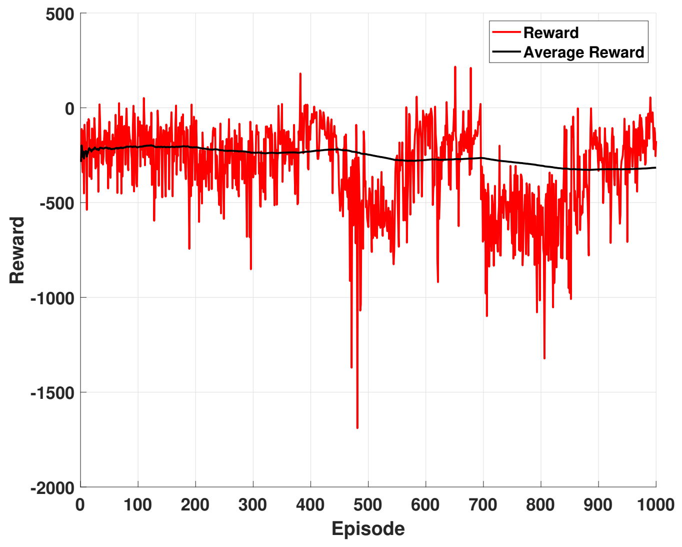

Figure 2: DQN-1: Episode vs Reward

4.2 DQN-2

As a second step, I considered 2 hidden layers with 64 neurons in each layer for the policy network. The values for the different hyperparameters are:

- learning rate - 0.001
- discount factor - 0.99
- batch size - 64

The plot of Reward for 274 episodes is given in Figure (3). It can be seen that the average reward does not show much improvement, but DQN-2 configuration was able to provide a better average reward than DQN-1 while considering up to 274 episodes.

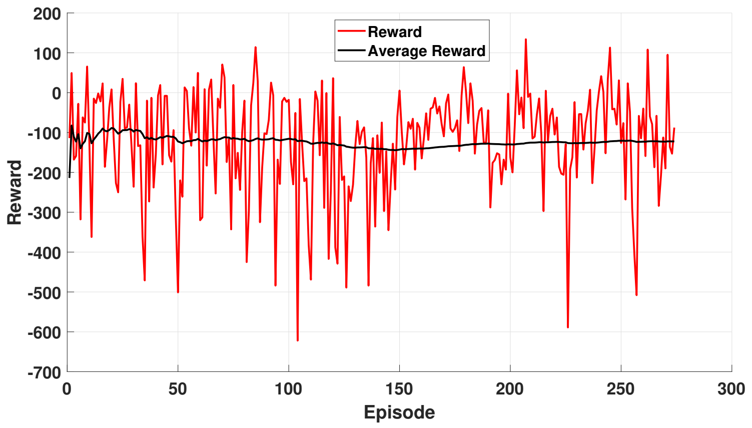

Figure 3: DQN-2: Episode vs Reward

4.3 DQN-3

To promote the improvement of average reward during the training process, the DQN-2 is implemented with *experience replay*, which helps to improve stability and efficiency during the training process. Without *experience replay*, consecutive experiences during training are highly correlated and this can lead to instabilities in learning. Briefly, in *experience replay* the past experiences are stored which are used to improve the $Q-$function. This is also how humans learn too - by recalling past experiences (slightly philosophical). The idea is to store an agent experience, e, at time t in a replay buffer as

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1}), \tag{5}$$

where r denotes the reward. As the training proceeds, a random sample of these experiences is used for updating the policy network. The size of the *experience replay* buffer is another hyperparameter that needs to be fixed.

Also, to reduce the temporal correlations between consecutive $Q-$value estimates and improve the stability of $Q-$target values, a separate target network is used to estimate $Q(s', a')$ along with the existing policy network. The target network weights, unlike the policy network, will be updated every N episode that is determined by the user. An overview of the algorithm used is given in Algorithm (1).

Algorithm 1 Algorithm for DQN-3 with experience replay and target $Q-$network.

```
Initialize all the hyperparameters
Initialize the policy network
Clone the policy network to create the target network
Populate the experience replay buffer with some initial experience
for 1 : episode do
    while not done do
        Obtain the starting state
        Select the action using exploration or explotitation
        Execute the action to obtain the reward, done, and next state
        Append the experience in the replay buffer by deleting older experiences
        Sample random batch from replay memory
        Pass the next states to the target network to get $Q(s', a')$ and evaluate Eq. (4)
        Gradient descent updates weights in the policy network to minimize loss
    end
    Update weights in the target network (to that of the policy network) every $N$ episode
    save the model file of the policy network
    save the reward as an array and find the average reward
    if average reward $\geq 190$ then
        break
    end
end
```

4.3.1 DQN-3a

As a first step, I tried DQN-3 with the following hyperparameters:

- learning rate - 0.001
- discount factor - 0.99
- batch size - 64
- replay buffer size - 10000
- N - 10

The plot of Reward is given in Figure (4). The average reward reaches about 190 at the 1500^{th} episode and several episodes gave a reward of greater than 250. The model files with rewards greater than 250 were stored and the lunar lander was tested using the model file. It was found that the lander was able to successfully land in between the flags.

4.3.2 DQN-3b

To compare how the average reward changes by changing the learning rate, all other hyperparameters were fixed except for the learning rate.

- discount factor - 0.99

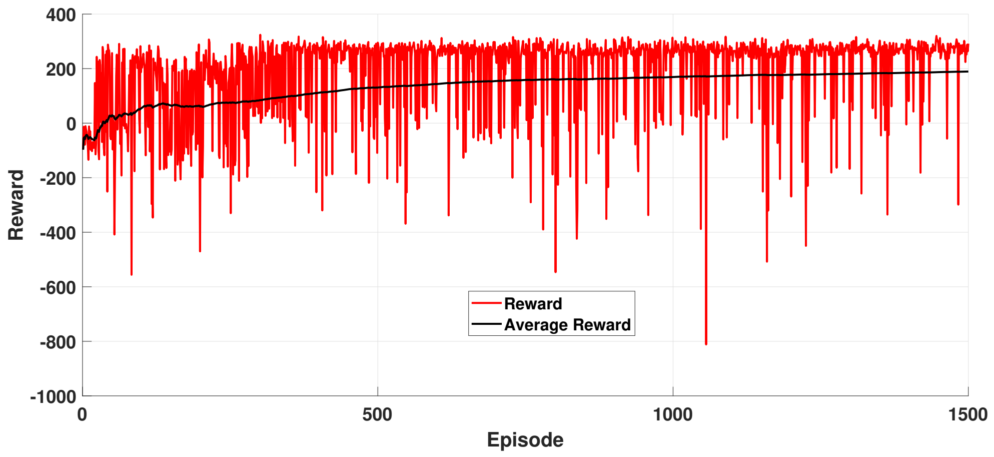

Figure 4: DQN-3a: Episode vs Reward

- batch size - 64
- replay buffer size - 10000
- N - 10

The training was done for different learning rates and the average reward plot is shown in Figure (5). The learning rate affects the average reward and a learning rate of 0.001 is good compared to 0.005 and 0.05 for this problem.

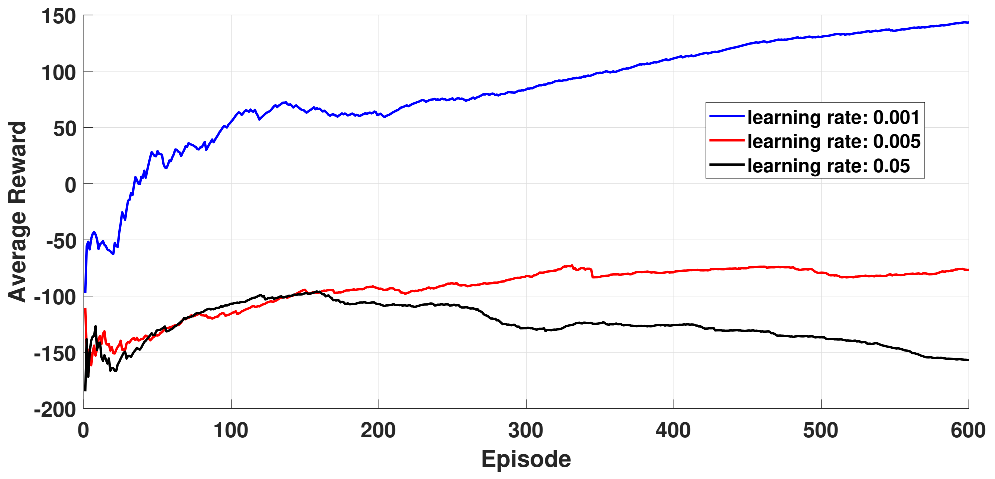

Figure 5: DQN-3b: Episode vs Average Reward for different learning rates.

4.3.3 DQN-3c

To compare how the average reward changes by changing the discount factor, all other hyperparameters were fixed except for the discount factor.

- learning rate - 0.001
- batch size - 64
- replay buffer size - 10000
- N - 10

The training was done for two different discount factors and the average reward plot is shown in Figure (6). Clearly for this problem, it is important to consider future rewards and therefore a higher discount factor is favorable.

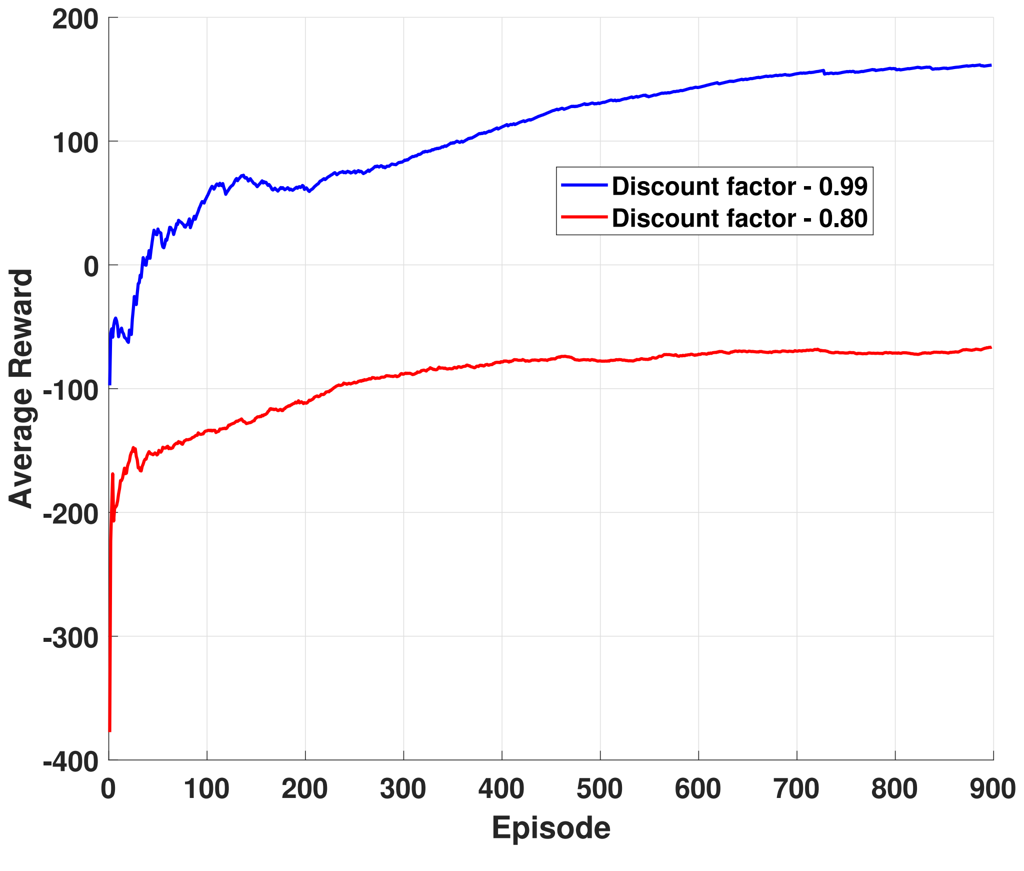

Figure 6: DQN-3c: Episode vs Average Reward for different discount factors.

4.3.4 DQN-3d

To compare how the average reward changes by changing the size of the replay buffer, all other hyperparameters were fixed except for the size of the replay buffer.

- learning rate - 0.001
- batch size - 64
- discount factor - 0.99
- N - 10

The training was done for two different replay buffer sizes and the average reward plot is shown in Figure (7). With a bigger replay buffer size, it is possible to store more experiences and the agent can learn better with a lot of options.

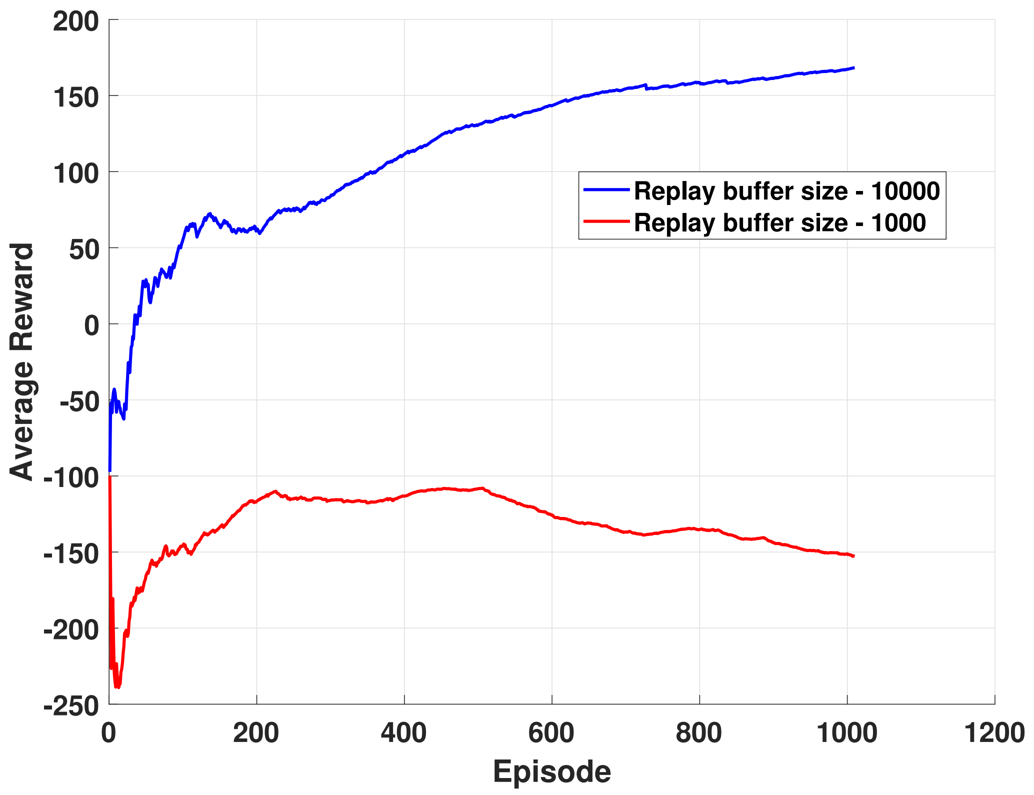

Figure 7: DQN-3d: Episode vs Average Reward for different replay buffer sizes.

5 Conclusion

In this project, the openAI lunar lander problem is solved using a DQN. Initially, I was unable to get a good average reward by using the structure of the hidden layers that I chose. But after including the techniques of experience replay and target network I was able to get a good model file after training along with better average rewards. I was also able to use this model to visualize the lander landing properly. The trained model file and a video of the lander landing are also uploaded.

One important fact that I learned was the importance of the choice of hyperparameters. They affect the rewards to a good extent. For this problem, using two hidden layers with 64 neurons in each hidden layer, the following choice of hyperparameters gave me the best result.

- learning rate - 0.001
- discount factor - 0.99
- batch size - 64
- replay buffer size - 10000
- N - 10

However, it should be noted that there could be some other combination of hyperparameters that can be better than what I got. One potential future work would be to try and solve more complex problems using DQN and other techniques.

References

[1] C. Sánchez-Sánchez and D. Izzo, "Real-time optimal control via deep neural networks: study on landing problems," *Journal of Guidance, Control, and Dynamics*, Vol. 41, No. 5, 2018, pp. 1122–1135.

[2] L. Zhu, J. Ma, and S. Wang, "Deep neural networks based real-time optimal control for lunar landing," *IOP Conference Series: Materials Science and Engineering*, Vol. 608, IOP Publishing, 2019, p. 012045.

[3] S. Gadgil, Y. Xin, and C. Xu, "Solving the lunar lander problem under uncertainty using reinforcement learning," *2020 SoutheastCon*, Vol. 2, IEEE, 2020, pp. 1–8.

[4] T. Szandała, "Review and comparison of commonly used activation functions for deep neural networks," *Bio-inspired neurocomputing*, 2021, pp. 203–224.