Solving Lunar Landing with Deep Q-Learning

Jessica Manzione
Georgia Institute of Technology
Ellicott City, USA
jmanzione3@gatech.edu
Git Hash: 1c53e9147db77e1d5f752f00302ec7e7de931c7e

***Abstract*—The purpose of this paper is to solve the lunar landing game from OpenAI Gym using Deep Q-Learning. There are various possible methods that could have been chosen. I will discuss my architecture and selections and provide results to show their effectiveness.**

I. Introduction

Lunar Landing is a grid environment that contains a rocket that is trying to land on the designated landing pad. The rocket always starts at the top of the screen and the landing pad is at (0,0). There are four discrete actions the rocket can take in any state which include do nothing, fire the left orientation engine, fire the main engine, or fire the right orientation engine. The state space is 8-dimensional with six continuous state variables and 2 discrete state variables presented as:

$$(x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, leg_L, leg_R) \tag{1}$$

For each state this shows the coordinates of the rocket as x and y, the horizontal and vertical speed as $\dot{x}$ and $\dot{y}$, the angle and angular speed as θ and $\dot{\theta}$, and whether the left leg is touching the ground or right leg is touching the ground as binary variables as leg_L and leg_R.

The rocket also known as the agent starts out knowing nothing about the environment and needs to learn in order to find the landing pad. As the rocket takes any of the discrete actions and moves to and from different state spaces, reward points will be earned or taken away. The reward points accumulated help the rocket to eventually learn which actions to take in each state to achieve higher rewards and achieve the goal of landing successfully on the landing pad. The reward structure includes penalties for the rocket moving away for the landing pad that matches what it would have earned moving towards the landing pad, -100 points for crashing, +100 for coming to rest, each leg with ground contact is +10, firing the main engine is -0.3, and firing the side engine is -0.03. An episode is terminated if the rocket comes to rest, crashes, or leaves the viewpoint. Fuel is infinite so the rocket is capable of exploring. The game is solved when the rocket reaches 200 or more points on average over 100 consecutive episodes.

To solve this lunar landing problem, I will be using Deep Q-Learning which includes multiple neural networks and the Bellman equation.

II. Method

As usual with reinforcement learning, the agent, the rocket, is going to use value based learning known as Q Learning. However, the number of state spaces is infinite and creating a Q table would be difficult. As a result, I used Deep Q Learning. Instead of a Q table, I am using a Neural Network to approximate the next action to take and to perform a gradient step to update the weights. Neural Networks are designed with layers that can extract features from the input and utilize weights to determine the significance of those features. Neural Networks can have different numbers of layers and neurons depending on the complexity of the network. For my Neural Network, I used 3 linear layers with ReLU activation layers in between. The initial input size is 8 and the hidden layers each have 32 neurons. In addition, I used Mean Squared Error for my loss function and Adam as my optimizer.

There are a few other key components used in this Deep Q Learning approach. First, frequent Q updates still occur incrementally. Two Neural Networks are created that are initially the same. One is used to consistently train throughout each step and the other is more stable and used as the target, only getting updated every so often. With each Q update, we update the Q value for the current state and selected action with the reward plus the action with the maximum value predicted using the next state and the target model shown in equation 2. Unless the episode is over, then it is just the reward.

$$Q[state][action] = r + \gamma \max Q(s', a', \theta_i^- \tag{2}$$

Second, as previously mentioned, two models are created that are initially the same. The main model that is trained and incrementally updated and the target model that is not updated with each step in order to keep it more stable. The target model is also used to determine the max action value predicted for the next state that is a part of the Q update. It is updated to have the same model parameters as the main model after every 100 steps.

Third, a loss function and backwards propagation is utilized. The loss function is calculated using the mean squared error shown in equation 3. The Q value for the current state and action based on the update is used as the target ground truth value. The prediction is calculation using the model with the current state as input to estimate the optimal action to take. The difference squared is the estimated error and used in backpropagation. Backpropagation through the Neural Network will bring the model closer to the target model and update the weights.

$$loss \ = \ ((r + \gamma \max Q(s', a', \theta_i^-)) - Q(s, a, \theta_i))^2 \tag{3}$$

Fourth, memory replay is used to create less dependence on the sequence of episodes. Each combination of state, action, next state, and whether the episode is done or not that occurs is recorded in the replay memory. A batch size in this case of 32 is used to randomly sample from that replay memory and perform the Q update as well as loss calculation. The first time the replay memory is used to train the model is after the number of instances equal the batch size so there is enough data to begin training the model. Also, the entire replay memory is not used to sample from each time and just the last 10,000 steps in order to prevent steps that occurred a long time ago to not consistently get used in the back propagation and instead use more recent steps. This also allows more training to occur and each step could be used multiple times. Neural Networks need a lot of data to train so this will help the Neural Network to train faster and sooner.

Finally, an epsilon greedy approach is used. With so many possible states even though few actions, a randomized approach is needed in the beginning. This will force the rocket to explore and try different actions in each state. Initially, epsilon is set to 1 so it is always taking a a random action. Over time, this is decayed so it begins to use the max action value predicted by the Neural Network model for that state. Eventually, epsilon will reach 0.01 because we want the actions to be determined by the model as the model is trained and updated so the better actions are selected.

III. Experiments Results

A. Training the Model

The Neural Network model and target model are initialized with random weights for all features. Each episode in training has various steps that continue to occur until the episode is terminated by the rocket crashing or landing. During each step, an action is chosen either randomly or based on the action values predicted from the model. The rocket performs that action to get to the next state. The previous state, action, next state, and whether or not the episode is terminated is recorded in the replay memory. A replay is performed which updates the Q value for the current state and action based on equation 2. Then the loss is calculated based on equation 3. This is backpropagated to update the weights of the model. After every 100 steps, the target model is updated to have the same weights currently as the model. Also, after every 10 episodes, epsilon is decayed to make the actions selected less random and more based on the model.

Shown in Figure 1, training was performed over 1,000+ episodes. Initially, the total reward per episode is negative and oscillates over a wide range. This is because the agent knows nothing about the environment and is taking random actions. As a result, the model is able to accumulate data and start learning. The reward per episode increases as the number of episodes increases and then eventually levels out. The randomness with the larger value for epsilon allows the rocket to explore causing the initial increase as well as the large oscillation. As the model learns it starts to utilize the model to predict the actions. This creates less variation in the reward values and it levels out. However, there are random extreme reward values in the positive and negative direction. This is still due to some randomness that can occur.

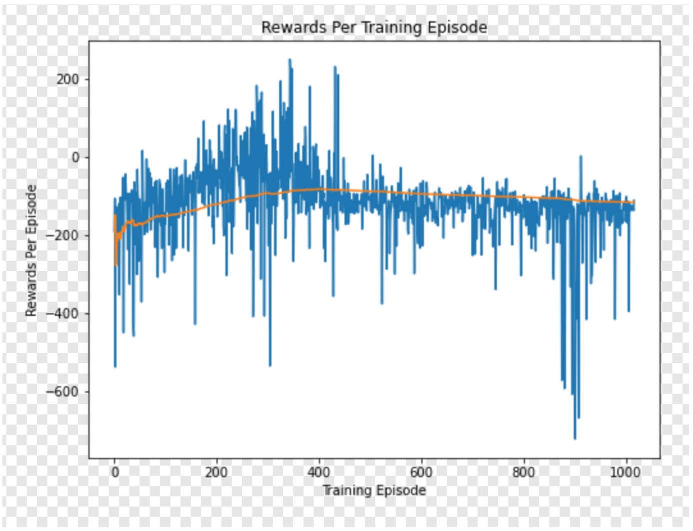


Fig. 1. Reward per episode during training

Parameters:

- batch size = 32
- input size = 8
- layer 1 size = 32
- layer 2 size = 32
- output size = 4
- learning rate = 0.001
- gamma = 0.9
- initial epsilon = 1
- epsilon decay = 0.9 every 20 episodes starting after 21 episodes
- update target model every 100 steps

You can see that the model was unable to reach the average of 200 reward points over 100 episodes. This is likely due to the combination of parameters used. I tried various combinations but still did not find the best combination. I could have also considered a different loss function or optimizer or the architecture of my neural network. Another potential hindrance could be the number of episodes that I allowed to run because it could have needed more to tune the model.

B. Testing on Trained Model

Once the model is trained, I used the trained model to run 100 consecutive episodes to calculate the total reward per episode shown in Figure 2.

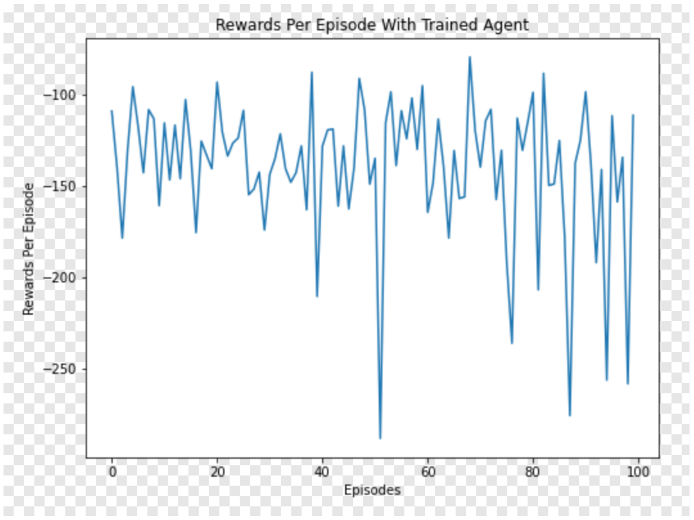


Fig. 2. Reward per episode over 100 episodes using trained model

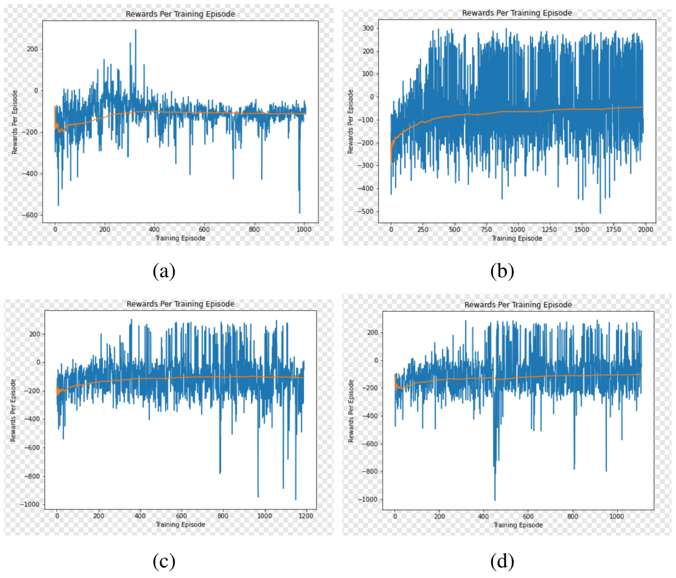


Fig. 3. (a) Gamma of 0.9 (b) Gamma of 0.8 (c) Gamma of 0.7 (d) Gamma of 0.6

For most of the episodes the reward values are relatively similar but there are some episodes that are extreme. I expected to see less oscillation and more stability since the model is trained and there is not randomness to the selection of the action for each step. Each step is based on the prediction of the target model for the optimal action. However, my trained model to did not reach an average of 200 reward points over 100 episodes as seen in Figure 1. However, even if my model was optimal, I would still expect some variation due to the infinite state spaces and continuous environment. It would be difficult for the trained model to have experienced every variation and there might be some paths that it is not optimal.

C. Hyperparameter Tuning

Choosing the parameters that allowed program to reach over 200 points on average for the last 100 consecutive episodes was challenging. Potential parameters to tune included number of neurons in each hidden layer for the Neural Network, number of layers in Neural Network, gamma, learning rate, batch size, epsilon decay, frequency to update the target model, frequency to replay memory, and how far back in replay memory to use to sample batches.

The first parameter I selected to tune was gamma. Gamma is the discount factor used to determine the impact the predicted value of the next state has on updating the Q value. Smaller gamma values means less impact in each update and larger gamma means a larger impact in each update. The values I used were 0.9, 0.8, 0.7, and 0.6 as shown in Figure 3.

There is not a significant different in the rolling average rewards for each gamma value and there is an upward trend. However, the range for the rewards varies for each gamma value. For gamma of 0.9 the range is tighter and the reward has less variation per episode. Gamma of 0.8 is very different and has a wide range and a lot of oscillation with the reward per episode. Gamma of 0.6 and 0.7 have more variation than 0.9 but less than 0.8 but some extreme reward values for random episodes. This shows that there is not a significant impact on the general trend of the total reward value per episode but will effect the training time for the model. The more oscillation creates more fluctuation seen with lower values of gamma and especially with gamma of 0.8. Using a gamma of 0.9 is more stable as a larger amount of the predicted max value of the next state impacts the Q update.

The second parameter I tuned was epsilon. Epsilon determines when a random action is selected (when the random number selected is less than epsilon) versus when the action is selected by the model based on the current state. Often, a value for epsilon is selected and used throughout the entire training process. However, I decided to decay the epsilon value over time in order to help the rocket to learn and experiment more in the beginning and rely more on the model as time progresses. I started with an epsilon value of 1 and decayed it over time to a minimum of 0.01. The first decrease occurs after 21 episodes and then every 20 episodes after that. The decay rate was 0.9. For other variations, I changed the number of episodes between each decrease but left the other values constant. The different variations of episodes included 30, 40, and 50 as shown in Figure 4.

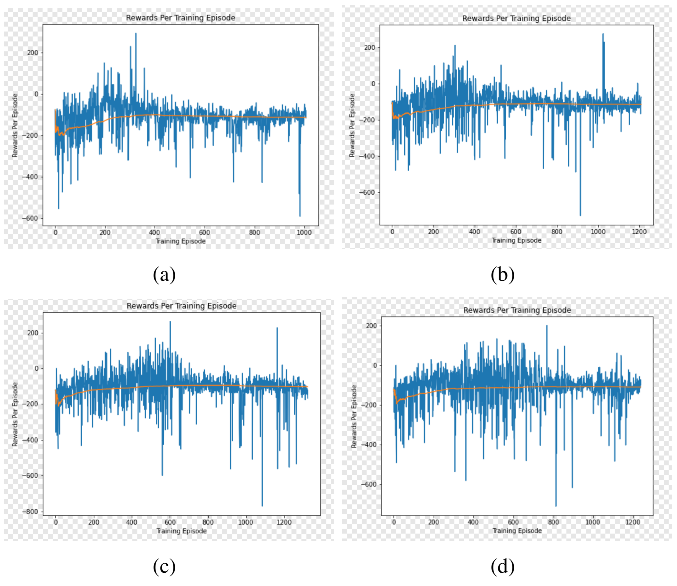


Fig. 4. (a) 20 episodes (b) 30 episodes (c) 40 episodes (d) 50 episodes

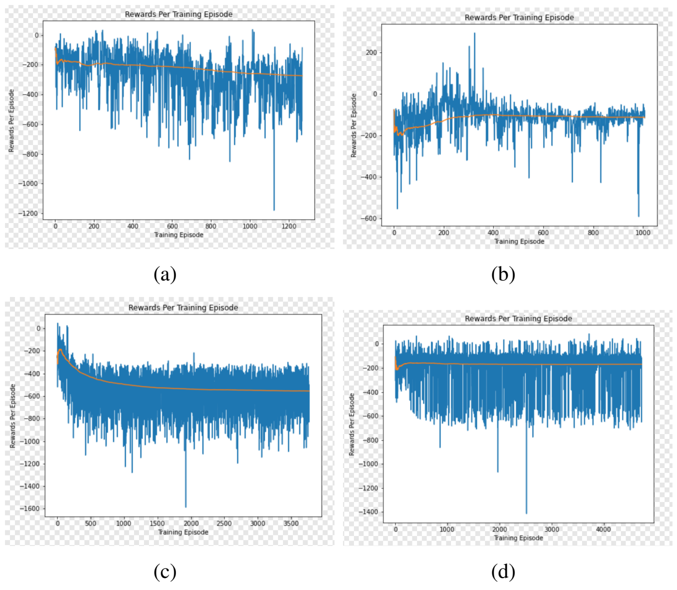


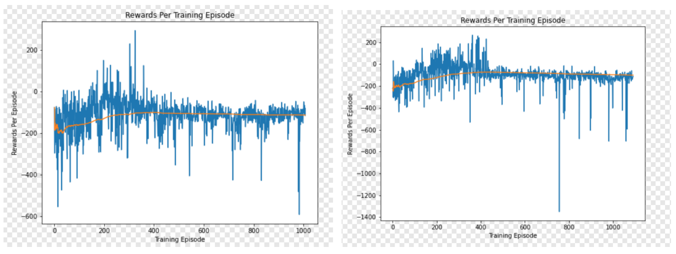
Fig. 5. (a) learning rate of 0.01 (b) learning rate of 0.001 (c) learning rate of 0.0001 (d) learning rate of 0.00001

The faster decline in epsilon with the smaller number of episodes between each decrease in epsilon show a faster initial increase in the total rewards per episode. Shown in (a) and (b) for Figure 4 between 0 and 200 episodes the rewards per episodes increase faster. The randomness in selecting the actions is decreasing and instead relying more on the model. Although the model is still in its initial training, this is showing that is making more optimal choices than using randomness. However, for all of the graphs it seems to level off as epsilon is very small and there is barely any randomness. This is becomes it is becoming stable and there is minimal to no exploration by the rocket.

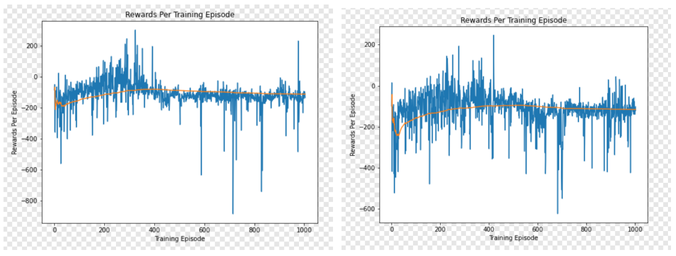
The third parameter I tuned was the learning rate. The learning rate determines the impact the loss or estimated error has on the weights for each update during backpropagation and is a value between 0 and 1. A smaller learning rate means a smaller impact and a larger learning rate means a larger impact on the weights update. Smaller learning rates could also take longer to train due to the small changes during backpropagation from the estimated error. Larger learning rates, on the other hand, can train faster with the larger steps but can converge to a suboptimal solution due to the large steps. The learning rates used included 0.01, 0.001, 0.0001, and 0.00001 shown in Figure 5.

The largest learning rate tested, 0.01, in (a) shows a gradual decrease in the average reward per episode. This is likely due to large steps and impact of the estimated error causing a suboptimal solution being determined. The smallest learning rate tested, 0.00001, in (d) shows a steady trend for the average reward per episode. With such a small learning rate, there is barely any change per step. As a result, it is going to take a lot longer to train the model. A learning rate of 0.001 shown in (b) initially has an increase in average reward per episode and then levels off. Being a larger learning rate, the weights are updated faster but it is not too large to overshoot the optimal weight values.

The fourth parameter I tuned was the number of steps between each update to the target model that copies the parameters of the main model being trained to the target model. The frequency of this update affects the stability of the target model and the loss calculation. Updating the target model more frequently can cause the target model to oscillate more. The main model is still being trained so there is a lot more randomness and this will create the same instability in the target model. However, waiting longer to update the target model can cause it to be behind and based on older steps that are less relevant. The frequencies used for the target model updates include 100 steps, 90 steps, 70 steps, and 50 steps shown in Figure 6.



(a) (b)



(c) (d)

Fig. 6. (a) Update target model every 100 steps (b) Update target model every 90 steps (c) Update target model every 70 steps (d) Update target model every 50 steps

As you can see in Figure 6, (d) which updates the target model every 50 steps shows the most variation in the total rewards per episode. This is because the target model is less stable since it is update more frequently as the main model is being trained. Interestingly, on the other end, with the target model getting updated every 100 steps there is more consistent variation than 90 or 70 steps. This could be because it is waiting a lot longer to update the target model and as a result the target model is behind and relying on old steps that are less relevant while the main model has been trained on the more recent steps.

IV. Conclusion

Overall, Deep Q Learning is effective to learn an environment with infinite state spaces. Although I was unable to get the optimal model, it is clear to see the effectiveness as well as the impact of the different parameters. Future considerations could be continuing to tune the parameters to find the optimal model. Also, could add more parameters such as setting max number of steps per episode, using other optimizers, and finding other ways to force the rocket to try new paths besides epsilon.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, p. 529, 2015.

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* 2nd Ed. MIT press, 2020. url: http://incompleteideas.net/book/the-book-2nd.html.