Project 2: Lunar Lander v2 (Continuous)

Beom Jin An

ban37@gatech.edu

git: 944775e9f8a69a1fce0b26a2ba5dd71eda90c0c5

I. Introduction

The purpose of this project is to solve the OpenAI Gym's Lunar Lander v2. This paper will explain the reinforcement learning method that was used to solve the Lunar Lander problem. The Lunar Lander v2 is a continuous state space MDP where a lander's goal is to safely land on the landing pad, which is located at (0,0). The lander will receive between 100-140 points when it successfully lands on the pad. The lander will lose points if it moves away from the landing pad. Also, the lander will lose 100 points if it crashes and will gain 100 points if it comes to rest safely on the surface. Each leg ground contact is 10 points, and firing the main engine costs 0.3 points per step. The solved is 200 points or higher.

Within these constraints, a reinforcement learning agent will explore and exploit the MDP to figure out a good set of policies to achieve 200 points or higher. For this project, the Double Deep Q-Learning with experience replay will be used to approximate the optimal policies.

II. Background

A. Q-Learning

Q-Learning is known to overestimate action values under certain conditions [1]. It sometimes overestimates because it has a maximization step over estimated action values, which elicits unrealistically high action values.

B. Deep Q Network (DQN)

A Deep Q Network is a multi-layered neural network that uses a target network and experience replay, which allows the agent to sample memories to improve performance [2].

C. Double Q-Learning

Double Q-Learning has two value functions that are built by assigning each experience randomly to update one of the two value functions [3]. It uses greedy policy estimation to make the first set of weights, but the second set of weights can fairly evaluate the policies and update symmetrically by switching the two weights.

D. Double Deep Q Network (DDQN)

The purpose of the DDQN is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation [4]. Its goal is to get most of the benefit of Double Q-Learning while keeping the rest of the DQN algorithm intact for a fair comparison and with minimal computational overhead.

III. Implementation

For this experiment, a good understanding of DDQN is crucial because even the smallest tweaking of the hyperparameters will impact the experiment greatly. Firstly, the agent will be a DQN with experience replay. Experience replay is designed to replicate how learning in our brain works, but it is much simpler than how brain actually works. The agent will look at the batch with a set amount of recent experiences, and it will update the actions values in the model. Since we are using double Q-learning, there will be two functions that are learning by assigning every experience randomly for update of any one of the two functions, and there will be two sets of weights. These will be denoted as targets in the algorithm. This technique will smooth out any updates in the distribution, and lessen the risk of overstimation by decoupling the correlation between the action values, the Q values, and the target. Therefore, selecting a good batch size for the experience replay and a good learning rate for distribution of data will impact the agent's performance greatly.

In summary, the algorithm will require these essential functions: create model, memorize, and replay. The model will be created with two or more hidden layers with preferably using relu activation. The memorize function will simply store every experience in an array, and the replay function will utilize the double Q-learning technique to ensure smooth updates of the action values. Thus, the agent is expected to learn the environment smoothly without a high risk of overestimation.

IV. LunarLander-v2 (Continuous)

A. State and Action

There are 8 dimensions in a state, and there are 4 different actions available to the agent. The agent reads the state with the following tuple:

The state variables x and y are the horizontal and vertical position, $\dot{x}$ and $\dot{y}$ are the horizontal and vertical speed, and θ and $\dot{\theta}$ are the angle and angular speed of the lander. Finally, $legL$ and $legR$ are binary values to indicate whether the left leg or right leg of the lunar lander is touching the ground.

V. Experiment

A. Method

The experiment will use the Double Deep Q-Learning to find the optimal policies for the LunarLander-v2 and will achieve an average score of 200 points per 100 episodes. The experiment will also explore how the hyperparameters used in DDQN influence the proficiency of the agent. The hyperparameters are expected to heavily influence the learning by the agent.

B. DDQN Set Up

First, a basis for the algorithm must be established. For this experiment, we will be using two hidden layers with 256 and 128 nodes and with relu activations for both, respectively. These numbers were not derived from extensive experimentations, and the basis of these numbers was derived from a number range that was replicated in others' experiments. However, a cpuple numbers were tried by myself to see which numbers would elicit the best results for this specific experiment.

For the hyperparameters, I guessed that the learning rate α to be somewhere in the 0.001~0.0001. I thought the agent would need to learn much slowly in order to make sure enough samples were given to smooth out predictions. I also guessed that the discount rate λ should be somewhere in 0.9~0.99. The discount rate is important to make the agent weigh the current state more than the future states. This would prevent the agent from thinking that hovering is better than trying to land.

In terms of the batch size for the experience replay, I set the batch size to 10. The number needed to be small enough so that the computer was able to finish the algorithm within a realistic time. A higher batch number might've given better results, but the time it would take to calculate was unrealistic.

Lastly, the epsilon value must be established. I set the initial epsilon value to be 1.0. Also, the epsilon decay will determine how fast the epsilon value will drop. This is important when trying to control the amount of random exploration before starting to learn for real. I wanted to give the agent more time to explore randomly to learn more about the environment; however, I also did not want the agent to learn too late. I predicted that if I wanted the agent to finish learning around 1000 episodes, I had to drop the epsilon value all the way down to 0.01 starting around 300~400 episodes. Thus, I had to select a decay value that will drop the epsilon reasonably fast which would be around 0.99~0.999.

C. Algorithm

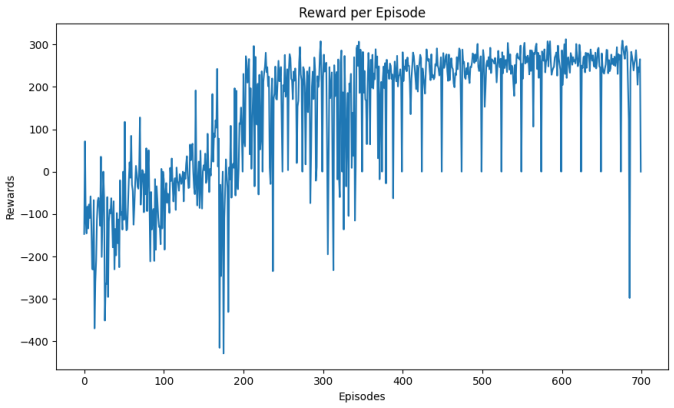
The algorithm takes in the Lunar Lander environment and the hyperparameters, and it will output a trained Q-value function that has the near-optimal policies. The algorithm will first initialize with a Q-value function with all zeros and will initialize the agent class that creates the DDQN model with the memory replay functions. For every episode, the agent will explore the environment until reaching the terminal state or reaching 10000 steps. During every step the lunar lander moves, the agent will store information and replay memory with a batch size of 10. The algorithm will end once there are three accounts of the average score higher than 200, but otherwise, the algorithm will terminate at 2000 episodes.

D. Results

By using the hyperparameters that I deemed good enough, I was able to solve the Lunar Lander problem. Thus, I tried to alter the hyperparameters in order to see how they affect the performance of the agent.

As a result, the average reward per 100 episode was near 250. The agent reached the average score of 200 within 600 episodes. The algorithm saved the weights of the model every 100 episodes, so I was able to test the trained agent. When I loaded the trained agent, I witnessed that the agent became very proficient at landing on the landing pad; however, I observed that the lunar lander was landing on the right side of the landing pad most of the time. You can see that the lander sometimes scores very low.

Result using Epsilon Decay of 0.99, gamma of 0.99, alpha of 0.0001

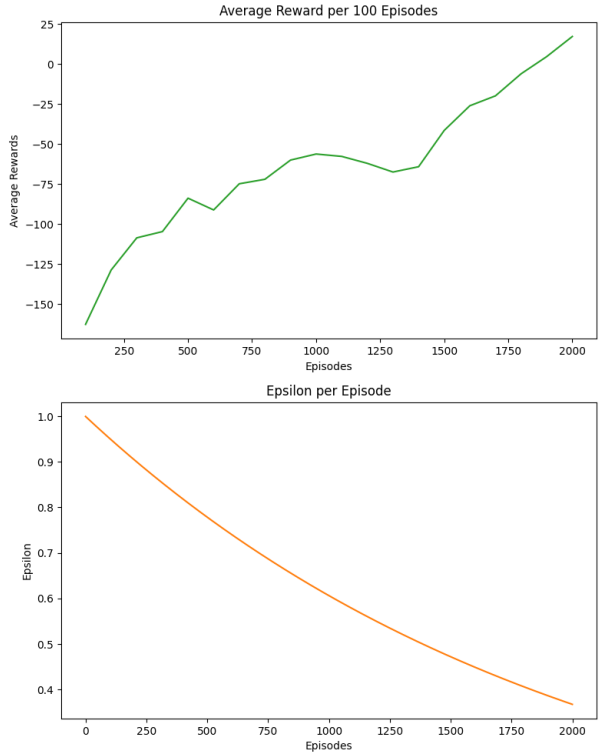


Most of the time, the lander will land between the flags, but if there was a slope on the right side of the flag, the lander sometimes gets stuck there. I cannot explain exactly why the lander learned to land on the right side of the landing pad, but overall, the lander was able to learn how to safely land onto the pad. If I had ran the algorithm for a couple hundred more epsiodes, I think that the agent would've fixed this minor problem. The test run of this model can be found in lunartest.py on github.

1) Altering the Epsilon Decay: I believe that the epsilon decay is perhaps the most important hyperparameter because it influences the computational time and learning in general. My goal was to find the decay value that helps the agent to learn the fastest while not falling into local optimum. If the decay is too fast, the agent won't explore much of the environment, and if the decay is too slow, the agent won't exploit the information gathered to find the best policies.

For this experiment, I ran the same algorithm with the decay of 0.9995 and 0.99, respectively.

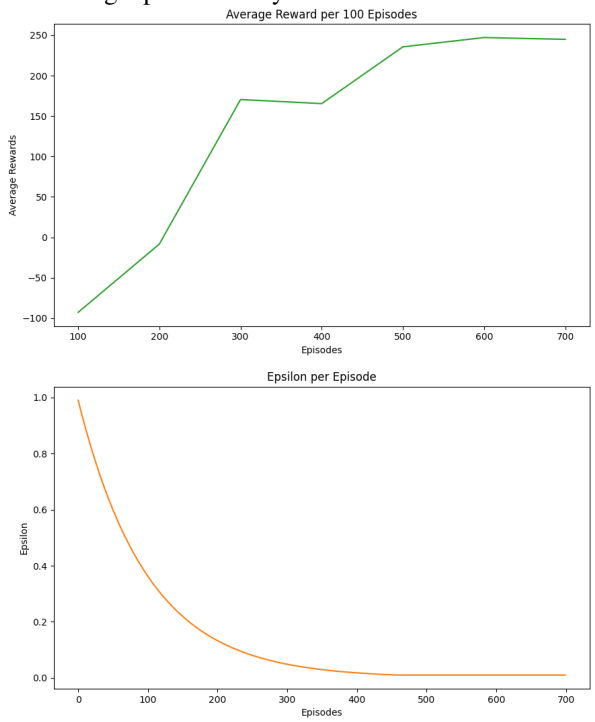
Result using Epsilon Decay of 0.9995:



The decay of 0.9995 was too slow, and the agent did not learn the optimal policy by episode 2000. However, the average scores did improve, but it would've taken a few thousands more episodes to find good policies.

Next, the agent used the 0.99 epsilon decay. The decay is much faster than that of 0.9995, and I predicted that it would be just right amount for the agent to explore and exploit.

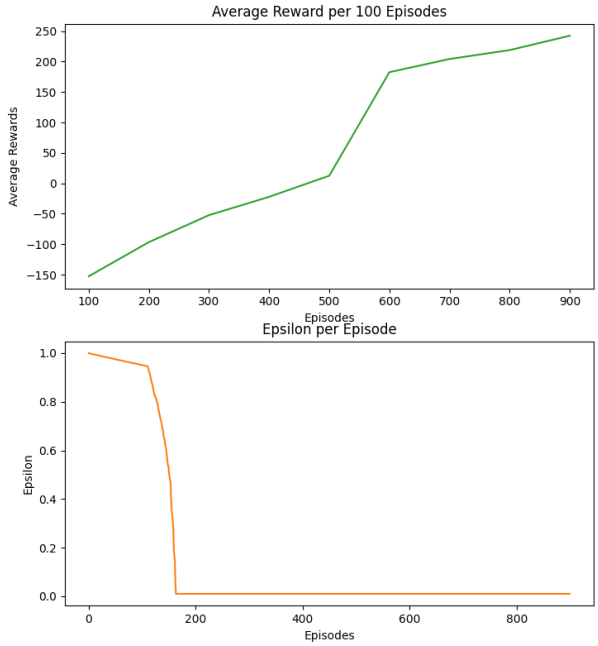
Result using Epsilon Decay of 0.99



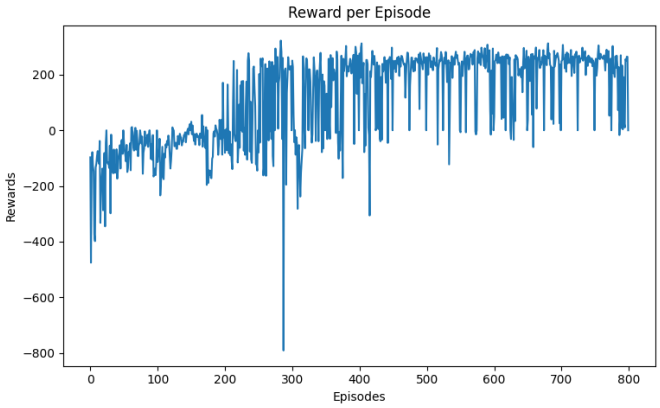
This run used a decay value of 0.99 which allowed the epsilon to converge a lot faster. The agent was able to master the landing in only 600 episodes, and this presented to be the best run in this experiment. The decay was just slow enough for the agent ro explore but just fast enough to exploit the explored information.

For additional testing, I tried to control the behavior of the epsilon decay by adding a linear decay before the geometric decay. The shape of the decay graph below will show how the decay was controlled.

Result using linear decay for 10000 steps:



The linear decay was only for the first 10000 steps that the agent took. The linear decay was about 1/10000 every step. After that, the decay was switched off to the geometric decay of 0.99. This experiment also showed success as the average reward was well above 200.

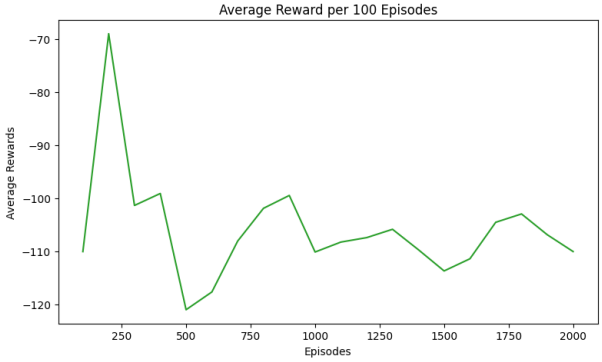


This reward graph has shown to be very similar to that of the reward graph in the Result section 5.4. I can conclude that the linear decay had no significant impact on the outcome of the learning.

2) Altering Discount Rate γ: The best discount rate for the agent was around 0.99; thus, I wanted to see what happens if the discount rate is 0.9. I predicted that this would prevent the

agent from recognizing the future rewards, which ultimately is the reward for landing safely. The high discount rate may trap the agent into thinking that hovering and not crashing is better than even trying to land.

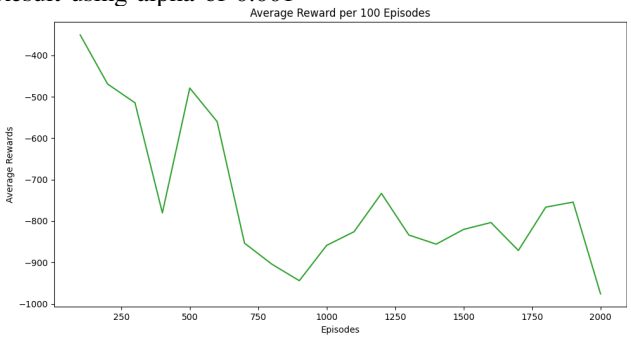
Result using discount rate of 0.9



The average rewards show that the agent consistently crashed or hovered until the time is done. The agent never learns that landing safely on the landing pad will yield greater reward.

3) Learning Rate α: For the next experiment, I wanted to see what happens if I increase the learning rate by a factor of 10. The original learning is set to 0.0001 which is low. This is how much the model will change everytime it learns. Since the model does not change quickly, it gives the agent more time to explore and search for optimal policy. Therefore, I believe that increasing the learning by a factor 10 will force the agent to learn too quickly while it is still exploring randomly. Thus, the agent will not learn the optimal policy.

Result using alpha of 0.001



Looking at the average rewards, the values are consistently negative. The rewards also decreases more as time passes. This means that the agent learned from bad experiences and continued to exaggerate the bad experiences as time passed.

E. Pitfalls

The hardest part of this experiment was to set up the tensorflow environment and to find reasonable hyperparameters to test the algorithm.

First, using the tensorflow required me to research how to run the algorithm efficiently. I had to install CUDA and had to understand how different versions of TF on different operating systems worked. Once I had the right version of tensorflow installed, I was able to use my personal computer to its near-fullest capacity. This allowed me to save a lot of time when computing.

Secondly, I had a lot of trouble selecting batch size and how the agent would reply memories. At first, I set up the algorithm so that the batch size was large (128), and the agent would replay every episode. The computation time was so much quicker, but the result was bad. I further researched the paper on DDQN and found out that I was supposed to replay every step instead of every episode. This meant that the computational time was going to increase exponentially. Thus, I had to decrease the batch size to 10 in order to arrive at a reasonable computational time. I also attempted to see if I can train every 10 steps instead of 1 step. However, the calculation time was faster, but the result was sub-optimal. Luckily, the batch size of 10 was enough for the agent to utilize the experience replay to improve. I believe that if I had a stronger and faster computer, I would've been able to compute using 32~128 batch size to get a better result.

For the other hyperparameters, I had no choice but to try out differently values to come to a conclusion. However, using intuition, I was able to select a good starting values to work from. The time spent of research saved a lot of trouble with the hyperparameters.

F. Future Improvement

For the future, I plan to use cloud services to run these tests. Sometimes, our personal computer is not strong enough to compute more complex reinforcement learning algorithms. If I were to use my PC, I would have to plan out how I am going to conduct experiment thoroughly in order to save a lot of time. I did not have the luxury to try out many different hyperparameters and techniques. Maybe with hosted services, I might've been able to conduct more experiments.

I believe that the DDQN was a good pick for the Lunar Lander v2 since it was on a continuous space that called for fluid and smooth updates on the weights, but I want to take more time to discover which types of algorithm would fit best into the different types of environment. Overall, this project gave me a lot insight onto how the mathematical formulas in the Reinforcement Learning textbook translate into real life examples.

VI. Conclusion

The Lunar Lander problem allowed me to explore deep reinforcement learning. There are many different techniques and algorithms that can be used to solve the problem. DDQN gave me a new perspective on reinforcement learning. I learned that since the purpose of the reinforcement learning is improve over time, the impact of hyperparameters also amplifies over time. Thus, the hyperparameters must be set correctly to yield the best result. I also learned that it is hard to predict or pre-calculate the result since the learning is somewhat random. Therefore, an extensive experimenting is required to build a good model.

References

[1] H. van Hasselt, A. Guez, D. Silver, "Deep Reinforcement Learning with Double Q-Learning", pp. 1, 2015.
[2] H. van Hasselt, A. Guez, D. Silver, "Deep Reinforcement Learning with Double Q-Learning", pp. 2, 2015.

[3] H. van Hasselt, A. Guez, D. Silver, “Deep Reinforcement Learning with Double Q-Learning”, pp. 2, 2015.
[4] H. van Hasselt, A. Guez, D. Silver, “Deep Reinforcement Learning with Double Q-Learning”, pp. 4, 2015.

https://github.gatech.edu/gt-omscs-rldm/7642Fall2020ban37/tree/master/Project_2