Emily Broom Git Hash: 44f6624ca4a33ae22a1a5dc13c2995c2e07980c2

Lunar Lander Solver with DQN

I. Lunar Lander

Lunar Lander is an environment in the gym package, which is a Python package for testing reinforcement learning (RL) algorithms. It's similar to the Atari Lunar Lander game. The goal is to land safely on a pad located at (0,0) between two flags. The spacecraft itself is initialized at a random point at the top of the screen. There are four actions that control the spacecraft: fire the right, left, or main thruster or doing nothing. The agent can only pick one action at a time. Fuel is infinite, and the lander can land outside the landing pad.

As is the case with RL, there are a set of rewards that the agent can earn or lose. The reward for moving from the top of the screen to the landing pad with zero speed is about 100 to 140 points. If the spacecraft moves away from the pad, the reward is lost (so the lander doesn't gain extra points for going away and coming back). The episode ends when the lander either crashes or lands safely, gaining -100 or 100 points respectively. Each leg that touches the ground receives 10 points. Firing the main engine is -0.3 points each frame. "Solved" is considered 200 points.

The input space contains 8 parameters: x, y, vx, vy, θ, vθ, left-leg, and right-leg. The first six, continuous parameters detail the position (x and y), the velocity (vx and vy), the angle (θ), and the angular velocity of the lander (vθ). The last two, discrete parameters give information on whether the legs are touching the ground or not.

II. Why DQN?

Being more familiar with Q-learning (due to lectures and homework problems), I wanted to solve the problem in a similar fashion. However, since the input has continuous parameters, normal Q-learning was not going to work. Either I needed to be able to discretize the input or use a function approximator. Function approximation is a well-known application of neural networks. Otherwise, if I were to try to discretize the input, how big should the steps be? Is x = 1.1 different enough from x = 1.2 or is this not granular enough? If the space is too granular, the amount of space needed in order to solve the problem would be enormous because the Q table depends on the input/state space.

A similar problem had already been solved by Minh et al. in the paper, *Playing Atari with Deep Reinforcement Learning* [1]. The paper explains how they used the same RL algorithm, Deep Q-Network (DQN), to get the agent to learn how to play 7 Atari games: Breakout, Pong, Enduro, Q*bert, Seaquest, Space Invaders, and Beam Rider with human-like or better success. These games were so similar to mine (continuous input space and discrete actions) that I decided to implement the DQN algorithm for the Lunar Lander.

III. DQN

DQN is model-free. DQN samples from the environment but does not model the environment. It does not predict the next state or next reward. Rather, it just finds the optimal action for whatever state it is in (same as normal Q-learning). DQN is also off policy; it uses a ε-greedy strategy:

$a = \max_a Q(s,a; \theta)$ with a probability of 1- ε and chooses a random action with a probability of ε.

DQN is different from Q-learning in a few ways. Firstly, it uses a neural network for approximating the state to action value function. Second, it uses experience replay and thirdly, it uses a target network to compute actions.

A. The Neural Network

Minh used a convolutional neural network (CNN) for their algorithm on DQN. This is because their input needed to be preprocessed due to their input being actual pixel values from the Atari games themselves. The Lunar Lander challenge gives the input as the values of position, velocity, and whether it has landed or not. There isn't any preprocessing needed. A CNN seemed overkill for this project and from reading on Piazza (#748) the instructors agreed that a DNN should be used.

B. Experience Replay

Experience replay happens when as the episodes are being shown, the agent stores information about the transition in memory: state, action, rewards, and next state. When the agent is ready to learn, it takes a batch, or a random sample, from these experiences to update the DNN. Doing this reduces correlation between experiences, increases the learning speed, and avoids catastrophic forgetting by reusing past transitions [2]. Using experience replay, the behavior distribution is averaged over many previous states rather than giving more weight to recent transitions, avoiding catastrophic divergence or getting stuck in a local minimum/maximum [1].

C. Target Network

The online network of the DNN is constantly being updated, for this reason there is a network called a target network which lags behind the online network. The target network is used in the Q-learning algorithm:

$$Q(s, a) = r + \gamma \max_a Q(s, a; \theta)$$

The updated Q is then used to train the DNN in the online network. Sporadically the target network is updated from the weights used in the online network.

The reasons for the target network and experience replay are very similar. Using a target network helps alleviate extreme data points from causing the network to deviate before it can correct itself. For example, if the lunar lander does a complete 360 and manages to land perfectly on the landing pad, this is obviously an outlier and not what we want the agent to learn. Over time, the network would realize this isn't optimal and the high reward was probably a fluke as most other times if it tried this, it would probably crash. Therefore, with a target network it is able to correct itself before the target network is updated with the online network. However, if the program was on policy, the weights would be updated right away, and the program could diverge before it realizes its mistake.

D. Other Techniques Used for Atari

There are other techniques that Minh used for the Atari DQN, clipping rewards and skipping frames, that I did not use for this problem. Minh clipped the rewards to be between -1 and 1 for all games so that the algorithm would be able to be generalized to learn any of the Atari video games, since different video games have different ranges of scores. Since

I was only concerned about the Lunar Lander game, I did not have to clip the rewards.

Minh also skipped frames. The agent would see and select an action every k^{th} frame. Every non-k^{th} frame, the agent would repeat the action it last chose. The reason for this was to decrease computation and therefore decrease runtime.

I tried to recreate this in the Lunar Lander project without success. Firstly, I am not sure what the interval is for every step. If I give the environment an action, is the next information I get the next frame, or do they already implement frame skipping? It's possible these intervals are large enough they should not be skipped. Second, if one compares the Lunar Lander game with one like Space Invaders, one can clearly see that time is more important in the Lunar Lander where the user controls the spacecraft every second, and the game is a much faster pace. Space Invaders, on the other hand, if a user sees an alien ship shoot at them, they have quite a bit of time comparatively to react. They definitely have more than 3 or 4 frames worth. However, I did not spend much time on this aspect of the project, and this could be an area to look more into.

IV. Implementation

A. The Algorithm

The pseudocode for the DQN algorithm used for the Atari games was given in the Minh paper (Figure 1).

Figure 1

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

The Q function here is the DNN where the agent inputs a state and the DNN returns the state-action value function approximation. The algorithm for solving the Lunar Lander is only slightly different. First, it needs no preprocessing, so the state can be sent directly into the Q function. Secondly, I perform an epsilon decay after every episode. In the paper it does mention that the epsilon was annealed linearly until 0.1, but this was somehow left off the algorithm. For the gradient descent, I train the DNN on a loss function of mean squared error with an Adam optimizer which is a version of stochastic gradient descent.

B. Starting Hyperparameters

Finding good hyperparameters was a challenge. The algorithm used in the Minh paper was different enough that some parameters given didn't work, like annealing epsilon linearly over the first million frames. What is a frame in our context? Also, thinking conceptually, our games probably tend to be a lot shorter than the Atari games, so a smaller number is probably better. Other parameters such as gamma and alpha were never given in the paper. While the number of layers and neurons per layer were not given, they wouldn't have been useful anyway as I am using a different architecture for the neural network.

I started thinking conceptually about some of the parameters. Gamma would have to be high as there are many state transitions before a reward is given for landing and I want to be sure that the reward for landing is correctly propagated through to the beginning states. For exploration, the epsilon decay would have to be high as well (where the algorithm is ε = ε * ε_decay). If the epsilon decay was too low, then epsilon would get to the minimum value, 0.1, very quickly and we would be stuck with whatever action the program thinks is optimal without having explored the state space enough to be sure that this really is optimal.

For the DNN I knew the input neurons were 8, due to the inputs given by the problem statement and 4 output neurons corresponding to the number of actions. However, I did not know how many hidden layers or how many neurons there were for the hidden layers of the DNN. My first attempts, at finding these values did not succeed, so I looked to Reyes' writeup on the same problem [3]. Since he was using the same environment as I was, his parameters would likely be a good starting place and if I could recreate his results, then I could start to tweak the other hyperparameters. He used a neural network with two dense hidden layers with 128 and 64 neurons respectively (many NN papers have this pattern with using powers of two and halving for subsequent layers such as [4]). He used a ReLU activation function on the first hidden layer and a linear activation function on the output layer. To start, I chose the alpha with .01 (which didn't work well) and varied it in different experiments which will be detailed in a later section. I decided on gamma starting at .9 and epsilon decay to be .998. I already knew from the Minh paper that the minimum epsilon would be 0.1. Some other parameters were the amount of memory, the minimum memory before training, and the batch size. Minibatches of 32 are common for this type of problem and is what Minh used. For memory, 65536 was used as being a large value of a power of two. A minimum memory was selected as 64. These values came from Reyes' writeup of his Lunar Lander.

I did a little bit of experimenting with the DNN before using Reyes' architecture. I tried using 64 and 32 neurons for the hidden layers. I also tried only having one hidden layer of 128 neurons. Both of these architectures did not succeed, meaning they were not complete after 1600 episodes (which was my cut off). The one with 64 and 32 neurons was promising at first but would reach around an average of 80 and start oscillating, getting worse and then getting better up to around 80 before getting worse again.

So, I went with Reyes' architecture which was given previously. Given more time, I would like to play with the DNN architecture, however this assignment was more focused on the RL, Q-learning algorithm, so I decided to pay more attention to that aspect of the problem.

V. Best Results

With the hyperparameters being: alpha = .0001, gamma = .999, epsilon decay = .998, minimum epsilon = 0.1, maximum memory = 65536, minimum memory = 64, and batch size = 32, the agent was able to solve the Lunar Lander challenge in 983 episodes.

Figure 2 shows the agent learning with reward on the y axis and episode on the x axis. The reward of 200 is shown

Figure 2

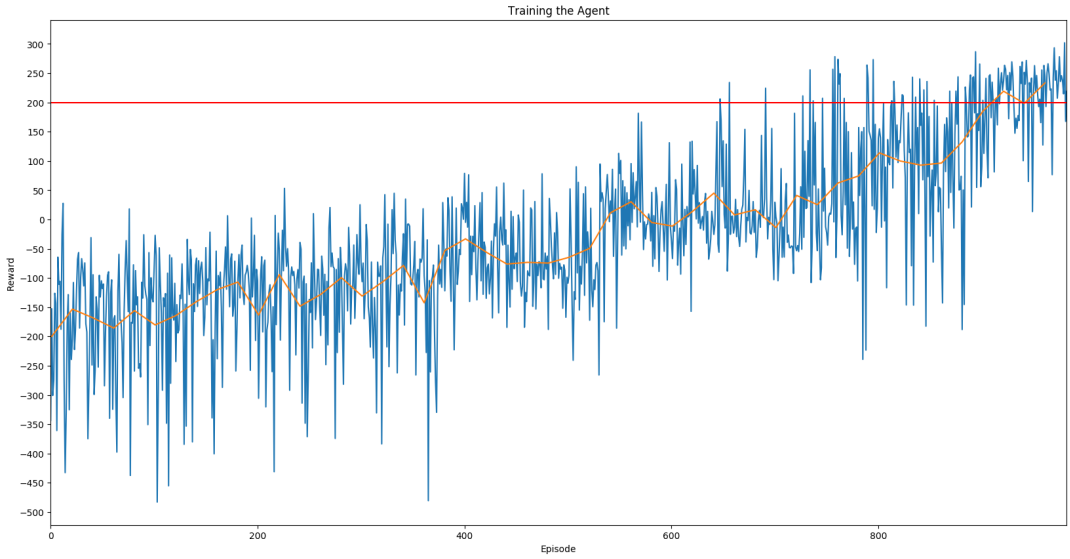
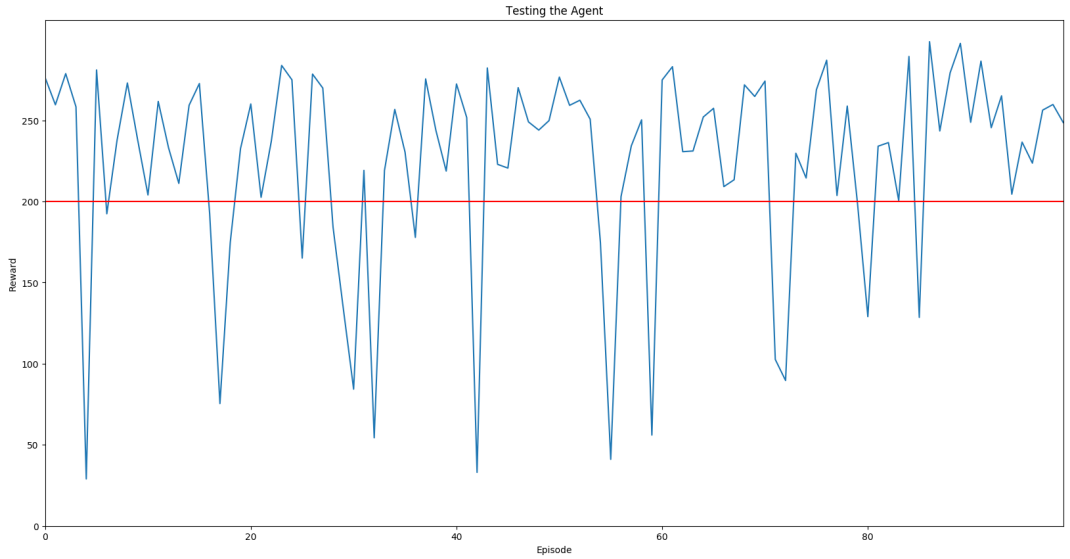


Figure 3



as the red horizontal line and is what the agent wants to surpass.

The agent steadily learns over time, eventually hitting the threshold of 200 and then getting an average of 200 for 100 episodes which was the goal of the project. The orange line is the moving average of 20 episodes, and it is easier to see the upward trend on this line. Even though there were some unlucky/bad episodes resulting in many dips, the agent was able to successfully learn and perform better. At the beginning there were lots of crashes and failed attempts. The standard deviation (just from looking at the graph) gets smaller after about 400 episodes. This is probably where the lander starts learning to stay close to the pad. The standard deviation grows at around 700 to 900 episodes, which there were either just unlucky episodes or maybe the lander was trying to learn how to actually land on the pad without firing the main engine too much but also without crashing. It looks like it crashed several times before learning how to land successfully.

After the training, the trained agent was taken and used on 100 more episodes where the reward per episode was plotted as well (Figure 3). Most of the time, the lander succeeds at the test. When reviewing the video there were several times where the lunar was close to the landing pad, having one leg on the pad, but wasn't fully on the pad. Other bad times it landed but without either leg on the pad. Probably if I let the agent keep going for a little while longer, it would have been able to do even better. However, it never has a catastrophic crash which is a big success comparing how the agent was doing in the beginning.

VI. Effects of Hyperparameters

There are lots of hyperparameters that one could change in this experiment: batch size, memory size, how often to update the target network, the alpha used for the DNN, gamma, minimum epsilon, epsilon decay and probably a couple more, especially when including the parameters of the architecture in the DNN. I focused on alpha, gamma, and minimum epsilon.

A. Gamma

In Figure 4, a moving average over 20 episodes was used because plotting all the data points of each line over one another made the graph too messy to deduce any information.

Figure 4

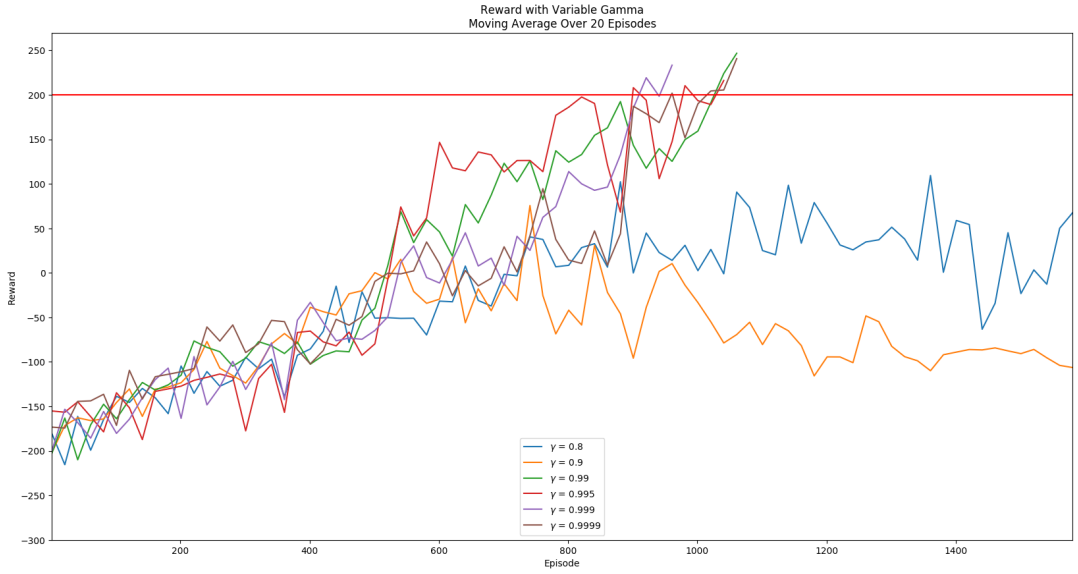


Figure 5

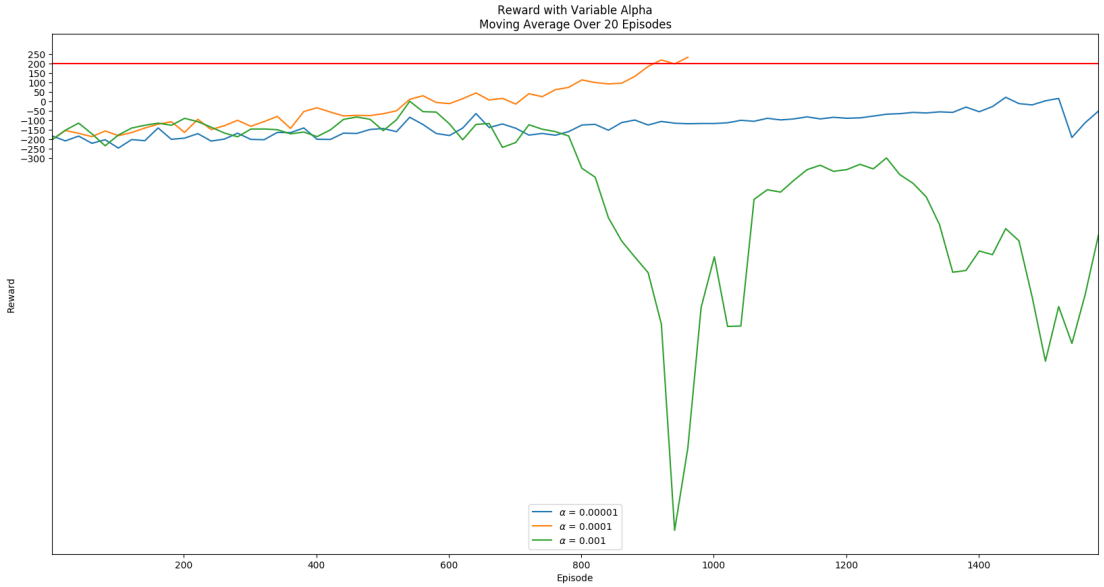
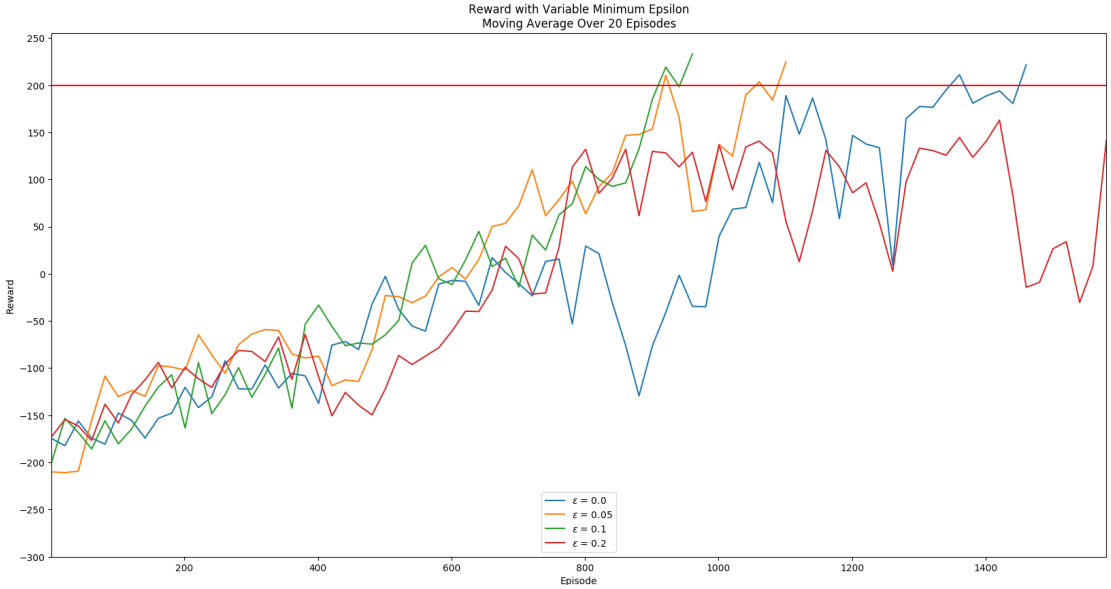


Figure 6



In this figure, all hyperparameters were kept the same except for gamma. Shown here, gamma of 0.8 and 0.9 never make it to an average (over 20 episodes) of 200. However, gamma of 0.8 actually does better than 0.9. Gamma of 0.9 seems to have had quite a few bad episodes and isn't able to come back from it. Perhaps 0.8 does better because it gives more reward to last minute decisions. The actions the lunar takes at the start of the episode don't matter as much as those at the end, like saving itself from collisions. Gamma of 0.9 may be at the spot where good rewards aren't propagated enough to prevent collisions from a very early state but also isn't given enough gratification for immediate rewards that would save it from colliding at the last minute.

Gammas of 0.99, 0.995, 0.999, and 0.9999 all do pretty well. Gamma over or equal to 0.99 seems to have enough reward propagation to keep the lander learning until it hits the goal. 0.999 does the best as it seems to have hit the sweet spot of reward propagation and decaying rewards. 0.9999 goes back down which tells me that it is probably not decaying rewards enough.

B. *Alpha*

Figure 5 shows the effects of changing the alpha value. Originally, I had also plotted alpha with .01 since that was my original estimate, but it was so catastrophic that it dwarfed the other lines. .001 is even catastrophic comparing to the alpha of .0001 and .00001. An alpha of .00001 learns a little bit but doesn't even come close to the 200-reward line. The alpha of .001 learns too fast and puts too much emphasis on recency. The alpha of .00001 is just extremely slow, but possibly could keep learning if the agent was allowed to keep learning as it doesn't seem to have quite plateaued just yet (although it has some rather unfortunate episodes at the end).

C. *Minimum Epsilon*

The minimum epsilon is the value where the epsilon no longer decays. The epsilon is used during the greedy policy to deal with the exploration vs exploitation dilemma. After the state space has been explored enough, the epsilon stops decreasing and stays constant.

Figure 6 shows the different values of minimum epsilon used. Epsilon with 0.1 was the most successful, which is no surprise given Minh's research. 0.05 is very close to being as successful, and it's probably just some unluckiness in the given episodes that made it dip, but it was able to come back. Epsilon of 0.0 also worked but was a bit slower. Epsilon of 0.2 was even slower but I think given a bit more time it would have succeeded as well. It's giving more importance to the exploration instead of exploitation, and perhaps if it was used with more and more episodes, the expected value of winning during testing would be more stable and have less standard deviation as more of the state space has been explored.

Epsilon starting at 1.0, ending at 0.1, along with the decay allows the agent to explore for a long period of time before the exploitation takes precedence, however, the agent always has an option to take a random route, even if it is small. Keeping the epsilon small, but not zero is a good amount of exploration vs exploitation for this problem.

VII. Summary

In summary, DQN did very well and was able to solve the problem. Given that I did a lot of research before starting the project, I did not attempt other algorithms. However, I knew right away normal Q-learning was not going to work because of the continuous input space. I have briefly read some other articles on DDQN and Duel-DDQN, which I think would be good candidates for this problem. I think it is very likely that one or both would be able to surpass the results found here. If I had more time, this is what I would mainly focus on, implementing the same problem with these algorithms and see whether the performance improves.

The biggest pitfalls were time and knowledge. My lack of knowledge/experience on what hyperparameters to start with really drew me back for a while until I found Reyes' paper. Firstly, I thought one hidden layer would be enough for the DNN. After I implemented the same layers and neurons as Reyes, I was stuck on the hyperparameters for a while (because I wanted to see if I could figure it out before looking at what someone else had used). I thought I was using a small enough alpha (0.01 which was absolutely catastrophic) and a large enough gamma (.9 or .99), which clearly wouldn't have worked. Time was also a hindrance as it took a long time to run each trial and I did not have infinite time to try dozens or hundreds of combinations of hyperparameters, so when the first several trials did not work out and I didn't know what direction to go in, that's when I turned to finding more information. This was difficult to do because most of the papers did not mention what they had used for their actual hyperparameters.

For the hyperparameters themselves I would like to know exactly why some did and didn't work. I have a good idea for some of the hyperparameters like gamma and alpha, but what makes a good minimum epsilon and why, and the architecture of the neural network are still fuzzy to me. If I had more time, I'd also want to learn more about this.

VIII. References

[1] Minh, V. *Playing Atari with Deep Reinforcement Learning*. https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

[2] Seno. T. (Oct 2017) *Welcome to Deep Reinforcement Learning Part 1: DQN.* https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b

[3] Reyes, A. *Lunar Lander.* (Jul 2017) https://allan.reyes.sh/projects/gt-rl-lunar-lander/

[4] Arulkumaran, K. (Jun 2017) *Classifying Options for Deep Reinforcement Learning.* https://arxiv.org/pdf/1604.08153.pdf