Machine Learning Engineer Nanodegree
Capstone Project

Scott Lilleboe
March 6, 2019

I. Definition

Project Overview

The capstone project involves the creation of an agent that will land the OpenAI gym Lunar Lander under established metrics that will determine a successful solution. The Lunar Lander simulation utilizes the OpenAI gym library which is a collection of environments to test reinforcement learning algorithms (OpenAI, Getting Started with Gym, 2018).

The inspiration for the project came from my interest with various NASA projects, especially with planetary exploration. This particular environment relates to recent Mars exploration such as the Insight mission (NASA, 2018).

Problem Statement

The project codes a simulated lunar lander module that attempts to land between two flags on a surface from a pre-determined height, so it doesn't crash and maximizes its reward score. The environment where the lunar lander resides is dynamically generated by the gym code base; however, the location of the "landing pad" is kept static. The better the landing, defined by the lunar lander code, the greater overall score it receives. The goal is to utilize Deep Reinforcement Learning using a Deep-Q Learner. The architecture will learn the environment and attempt to provide the highest chance of success with respect to the lander effectively landing.

Metrics

The benchmark and solution model will be evaluated over two metrics. The first metric will be the total average reward of a trained model over 100 iterations. This metric will give credit to the solution that had a better solution in terms defined by the Problem Statement. The second metric is the number of iterations needed to successfully train the model and reach a successful solution. The reason for the second metric is to give recognition to a solution that was successful and may have a substantially quicker time to train. A successfully defined solution is when a model during training is returned an average score of 200 or more for 5 consecutive training episodes.

II. Analysis

Data Exploration

The Lunar Lander problem comprises an 8-dimensional continuous state space and an action space of two real value vectors from -1 to +1. The landing pad is static at coordinates (0, 0) and the coordinates are the first two numbers in state vector. The reward for moving from the top of the screen to landing

pad with zero speed is between 100 to 140 points. The episode finishes if the lander crashes or comes to rest, receiving an additional +-100 points. Each leg to ground contact is worth an additional 10 points. Firing the main engine is worth -0.3 points each frame. The fuel for the lander is infinite. (OpenAI, LunarLanderContinuous-v2, 2018)

Exploratory Visualization

The lunar lander environment is visualized in Figure 1 below. In this environment, the surface terrain that the lander module can make a successful landing upon varies in height and slope with each episode the lander encounters. The only static surface with respect to location and height is the area between the two yellow flags. This allows the lander to learn its location after enough successful training iterations. The small red colored dots represent the thruster/engine activity. Dots displayed to the left and right of the lander are from the lander triggering side thrusters while the dots below the lander (the top left image) represent main engine movement. As can be seen in the bottom right image, the main engine is also activating while the lander itself has flipped its orientation, obviously not a good position to be in. These failed episodes are common as the lander begins to learn about its environment through repeated iterations.

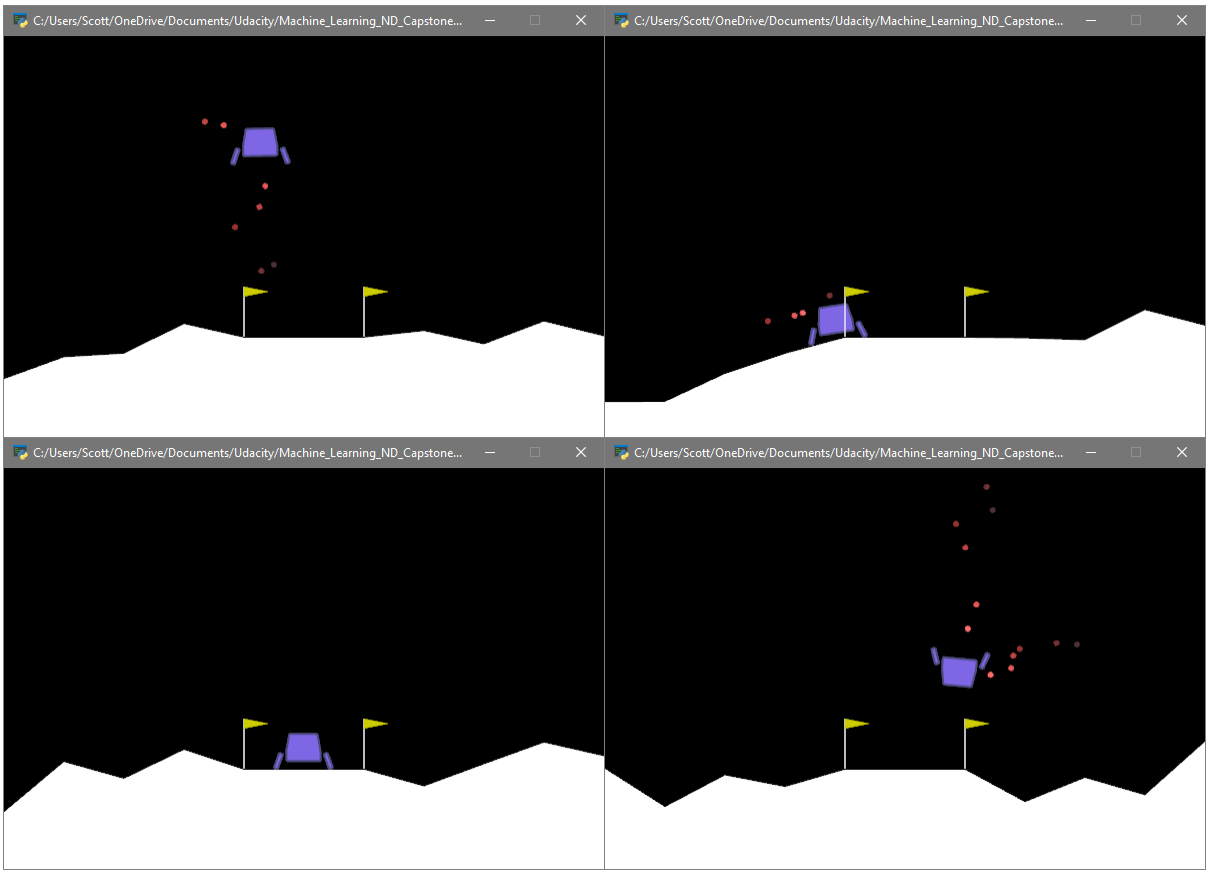

Figure 1. Lunar lander environment displaying various landing surfaces, module positions and orientation.

Algorithms and Techniques

A successfully defined solution requires the lunar lander to land with a total reward of 200 points averaged over 100 consecutive landing attempts. In order to accomplish this definition of project success, a deep Q-learning network (DQN) algorithm was utilized to accomplish the problem of the Lunar Lander. A DQN expands the Q-Learning algorithm by utilizing a deep neural network (DNN). The DNN is used to approximate the action-state function. Since the problem space is non-linear and continuous a DQN was the logical choice and was inspired by the Atari paper by Minh (Volodymyr Mnih, 2013)

The decision to use the neural network (NN) structures with the number of nodes and layers for both the benchmark and solution models was evaluated based on some preliminary tests. There was concern that having a larger NN would impact training time along with not providing adequate training iterations. Table 1 shows fours tests that were run over two NN architectures utilizing a set of default parameters. The larger and smaller architectures were run twice with varying training batch sizes. What can be seen is that the larger NN didn't impact overall training time. In fact it was quicker to train than the smaller NN. The theorized reason for the faster overall training was simply it found the optimal solution faster, not that the actual training of the NN was faster, which it was not per batch. The optimal solution in this case was reaching a training value of 200 for a single episode. Increasing the batch sizes further reduced the overall training time by decreasing the number of episodes before the solution was found. Increasing batch size has the effect of increasing the amount of data that the NN is trained on. The larger the batch the more samples are pulled from the replay memory and used per step per episode.

Dropout was applied between the hidden layers (set at 25%) to help with possible overfitting issues; however, the NN did not perform well. It could have been not enough episodes were run, but in the end it was deemed unnecessary to apply it as the non-dropout architecture was performing well.

The rationale for running these minor tests for architecture size was to reach a set of models that can be further tested by manipulating their respective parameter values. The size of the NN has a significant impact on training and testing times; therefore, only a cursory evaluation of them was performed. The focus of the project wasn't as much on finding the optimal node/layer combination but rather on how much parameter values impacted the overall architecture and impacted the learning of the lunar lander environment.

Table 1: Tests of various batch and network sizes.

NN - # of batches	Episodes		Time (m)
128 layer – 16	1200		147.95
128 layer – 24	1000		135.95
128/256 layer – 16	900		125.88
128/256 layer – 24	800		115.72

The final network architecture for the solution model can be seen in table 2. The benchmark model can be seen in table 4. The only difference is the solution model has a second layer of 256 nodes.

Table 2: Solution network architecture

Solution Neural Network Structure		
Layer (type)	**Output Shape**	**Parameters**
Dense_1 (Dense)	(None, 128)	1152
Dense_2 (Dense)	(None, 256)	33024
Dense_3 (Dense)	(None, 9)	2313
Total params: 36,489 Trainable params: 36,489 Non-trainable params: 0		

Both the solution and benchmark models were evaluated over various parameters of varying values. Table 3 shows the selected parameter values and their tested values.

Table 3. Parameter values utilized during testing of both benchmark and solution models.

Parameter	**Value 1**	**Value 2**	**Value 3**
Decay	0.01	0.001	0.0001
Epsilon	0.99	0.9	0.7
Gamma	0.7	0.9	0.99
Learning Rate	0.01	0.001	0.0001
Tau	1.0	0.1	0.01

Benchmark

The chosen bench mark model was a single 128 layer deep Q-learning network (DQN). The deep neural network (DNN) portion of the algorithm was compiled with a MSE loss parameter utilizing the Adam optimizer. Table 4 shows the neural network structure with a total number of trainable parameters set to 2,313.

Table 4: Benchmark network architecture

Benchmark Neural Network Structure		
Layer (type)	**Output Shape**	**Parameters**
Dense_1 (Dense)	(None, 128)	1152
Dense_2 (Dense)	(None, 9)	1161
Total params: 2,313 Trainable params: 2,313 Non-trainable params: 0		

III. Methodology

Data Preprocessing

The preprocessing that was performed in the environment was to limit the range of the action space. Since this was a continuous state space for actions, preprocessing the data so it became a discrete set was necessary in order to reduce the complexity of the underlying DQN.

The first real value vector was for controlling the main engine. This vector ranged from -1.0…0 off, 0…+1.0. The main engine's throttle varied from 50-100% power, it does not function with values less than 50%. The second real value vector was for the left and right engines where -1.0…-0.5 fired the left while +0.5…+1.0 fired the right engine.

The discrete space that was selected for the main engine was -1.0, 0.0, 1.0. This effectively limited the main engine to run at 100% or off. The space for the left and right engines was limited to the same set of values, -1.0, 0.0, 1.0. This limited the left/right engines to have either one fire or neither.

In the DQN class, the below code creates a dictionary of discrete values to use for input values and for looking up returned actions.

```
for i in [-1.0, 0.0, 1.0]:
    for j in [-1.0, 0.0, 1.0]:
        self.action_dict[key] = np.array([i, j])
        self.reverse_actions[i, j] = key
        key += 1
```

Implementation & Refinement

The code for the main part of the project is located in the capstone.py file. This file contains all the classes to run the algorithms, create the DQN and create multiple metric tests.

The code utilizes a soft update of the weight values (Timothy P. Lillicrap, 2016) In order to accomplish this there was two DNN's created. The first was the main online agent DNN that was trained at each batch step. The second was a DNN, named target agent, which was held static with respect to predicted values and only updated at the end of a batch update sequence. The update itself was governed by a parameter, tau, which gradually updated the weight values of the target DNN. This allowed for less variance in the predicted values which allowed the algorithm to learn with a smoother trajectory and with less training iterations.

The first attempt at designing the training algorithm had the epsilon update within the step loop. This proved to have two issues. First, it caused epsilon to decrease too rapidly without providing an extremely low decay value. Second, it penalized through lower exploration later steps than earlier ones by decreasing the epsilon as the episode progressed. The epsilon should remain locked for the extent of the episode and only be updating after all the steps are complete.

Another version of the algorithm had the weight updates of the online agent done in a single batch by sending a Numpy array of all the inputs and targets to Keras. This proved to greatly decrease the training times which would be expected as the updates are vectorized; however, it also showed inconsistent results when trying to successfully solve the lander problem. Having the updates run one at a time was substantially slower, but the results were better with regards to training the NN. This was most likely an issue with the way I coded the batch updates to Keras.

The tau value in the code governs how much influence the online DQN agent has over the target DQN agent, the one used to generate the target values for the neural network training. A value of 1.0 negates the soft updating of the target DQN by replacing all the weight values of the target with the online DQN after each batch training update. A value of 0.0 would effectively not update the target DQN with any values from the online agent. The particular parameter was one that was tested over various values to look for an optimal setting for both the benchmark and solution models.

The file graph_utils.py hosts a handful of metric graphing routines that the methods in the capstone.py file calls to graph data from the various training and evaluation functions.

IV. Results

Model Evaluation and Validation

Figures 2-11 show various tests that were done on a range of values for epsilon, epsilon decay, gamma, learning rates and tau parameters for both the benchmark and solution models. All the tests were run on the neural network architectures specified in Tables 2 & 4. A limitation to this approach was the hyper parameters that were not tested in each figure were held constant which limited the available space of parameters to search for an optimized solution. This limitation was due to time constraints as each test of each parameter could take multiple hours. The number of episodes and steps per episode were set to 1000 regardless if the trial was successful. This allowed for the episodes per trial to remain consistent. Setting to lower steps and episodes allowed for more tests to be run at the expense of a

possibly better parameter mixtures. Each of the figures show results of mean values over the previous 25 values. This was done to show a less variant result as plotting each episode's result proved to be too difficult to render a viable graph with good visual quality.

The ranges for each parameter was chosen to provide adequate variation in the underlying parameter. In almost all cases a low, median and high range was selected. Table 3 shows the values that were explored for each parameter during the training of the benchmark model.

The graph for epsilon values, Figure 2/3, showed a lot variance with each parameter value. The final results for each epsilon value ranged between 100 and 200; however, with further tests it wasn't as big of a factor as other parameters.

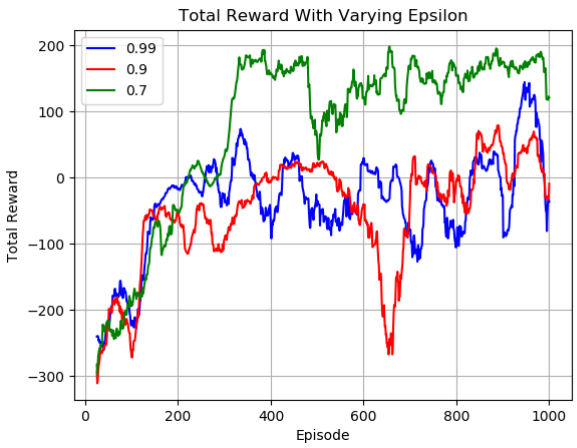

Figure 2: Benchmark varying epsilon values.

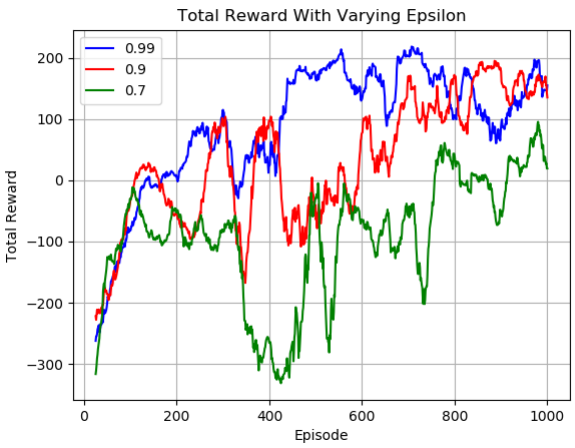

Figure 3: Solution varying epsilon values

The decay value for epsilon has a direct correlation with the underlying epsilon. This makes sense in that the decay is just the rate at which epsilon is reduced every episode. A high initial epsilon and large decay value (0.01) doesn't allow for as much exploration as a smaller decay for the same epsilon value. What can be seen with Figures 4/5 is a lower epsilon value provided for a higher total reward at the end of 1000 episodes. A high epsilon and lower decay provided for extensive exploration without impacting the number of episodes before convergence. Through observing the behavior of the lander through the render frames, poor exploration tended to keep the lander hovering far above the ground.

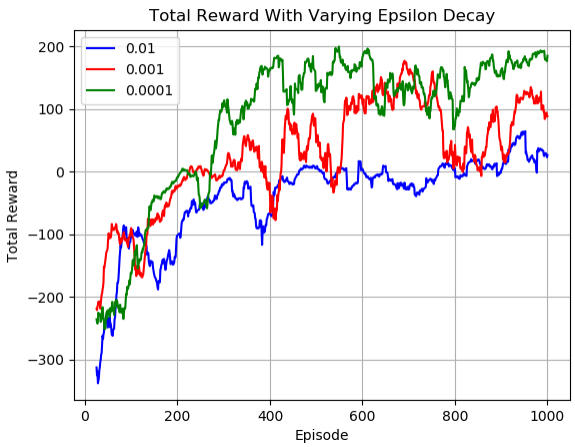

Figure 4: Benchmark varying epsilon decay values.

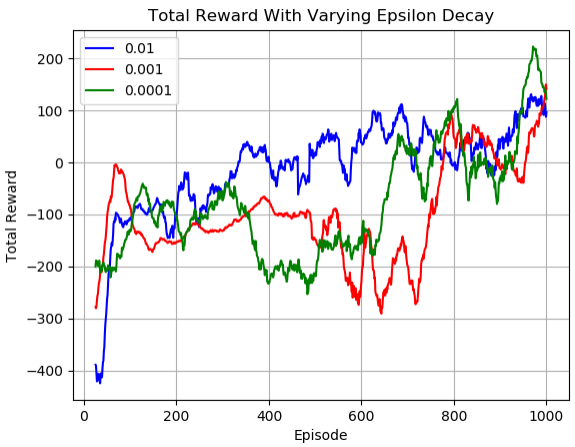

Figure 5: Solution varying epsilon decay values

The gamma values, Figures 6 & 7, show a large discrepancy between the largest value, 0.99, and the other values tested. This suggests that the environment performs better as the future states have more influence over previous steps. Based on the figures, a gamma value of 0.99 was the obvious choice for the final architecture for the benchmark model; however, 0.99 was the worst performer for the solution model. The particular parameter was intriguing, as the solution model clearly leads one to believe that 0.9 was a vastly superior value over 0.99, but this was not the case. I performed multiple training runs with a gamma of 0.9 with the other parameters at their optimized value, based on the figures, but the lander performed inadequately. Once the gamma was set to 0.99, the lander was noticeable more efficient. The thought is the testing of gamma that is seen in Figure 7 had issues around the 400 episode mark that caused it to become faulty in further episodes. The sudden drop in total reward at that spot couldn't not readily be explained. It could have been a mixture of other parameter values or just a batch of random unfortunate steps in the training.

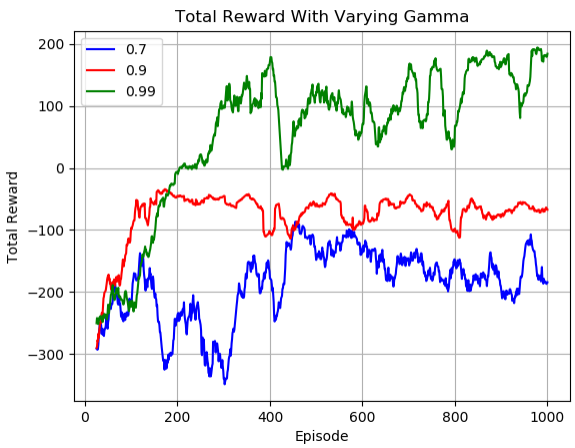

Figure 6: Benchmark varying gamma values

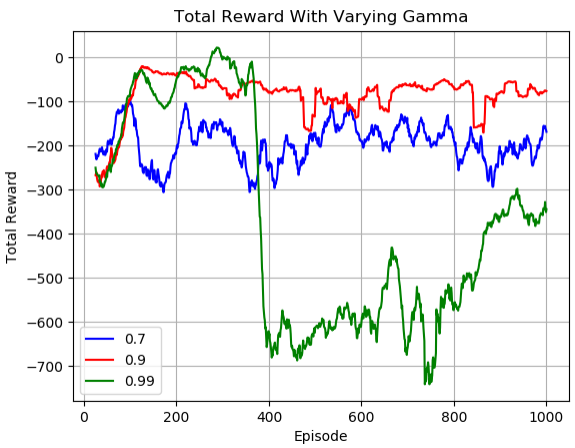

Figure 7: Solution varying gamma values

Figures 8 & 9 show various learning rates both the benchmark and solution models. Through those tests, a lower learning rate was desirable as the highest one, 0.01, looks to have possibly stuck the

neural network in some local minima and prevented it from learning effectively. Part of the specifications for the benchmark model was to keep the default learning rate of 0.001; however, I wanted to test that learning rate against a few others to see how well it performed. As Figure 8 shows, the default value performed the best overall. For the solution model, a learning rate of 0.0001 was chosen based on the figure because the final total reward was close to the 0.001 value; however, it had a slight uptick towards the end that pushed it past the total reward of the 0.001 value.

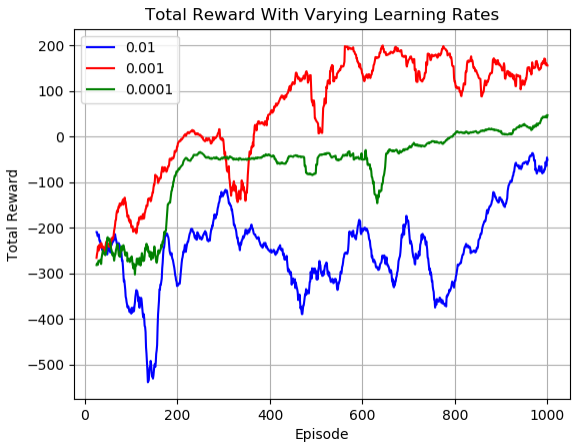

Figure 8: Benchmark varying learning rates

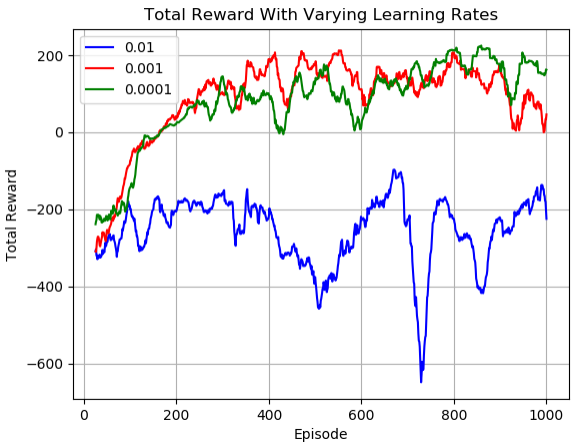

Figure 9: Solution varying learning rates

That last hyper parameter that was evaluated was the tau value which is used in the soft weight updates, figures 10 & 11. As previously stated, the tau value governs how much influence the online DQN agent has over the target DQN agent, and a value of 0.0 would effectively not update the target DQN with any values from the online agent. The values to be tested showed in both models that 0.01 was the worst value; however, just like the gamma setting for the solution model, this proved to be incorrect. A value of .001 for the benchmark and .0001 for the solution models were tested with the optimal values found for the other parameters; however the lander failed to perform well. In fact, the lander spent most of its time hovering vs exploring the environment. It was speculated that the combination of the other parameters caused the issue. A full exploration of possible parameter values was unfortunately difficult in this experiment due to the amount of time each model takes to test.

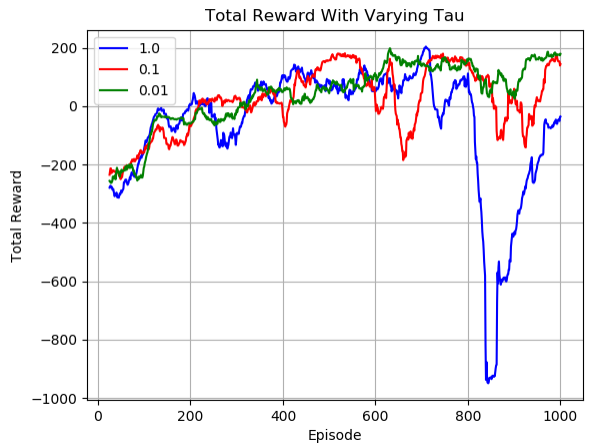

Figure 10: Benchmark varying tau values

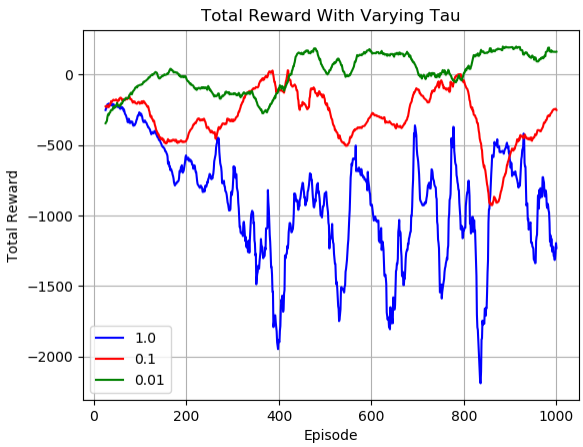

Figure 11: Solution varying tau values

Table 5 shows the final hyper parameter settings utilized in the optimized benchmark and solution models. Most of the parameters were tested and decided upon based on Figures 12-11. The rest of the parameters listed were chosen based on empirical observation. After watching enough rendered images of the lander, there becomes a point where you get a feel for how the agent is interacting with its environment. As unscientific as that was, replay buffer size, batch size and steps were determined in this manner. Given further time and resources, a better approach would have been to fully explore the domain of parameter combinations.

Table 5

	Benchmark	Solution
Parameter	Value	Value
Replay Buffer Size	50,000	50,000
Epsilon	0.7	0.99
Epsilon Floor	0.01	0.01
Epsilon Decay	0.0001	0.0001
Tau	0.01	0.01
Gamma	0.99	0.99
Learning Rate	0.001	0.0001
Batch Size	16	24
Steps	1000	1000

Justification

Figures 12-13 show the average reward over the previous 100 episodes for both the benchmark and solution models. The final values in the graph are when the algorithm proved to be successful and

converged as defined by the metrics section. A successfully defined solution is when a model during training is returned an average score of 200 or more for 5 consecutive training episodes.

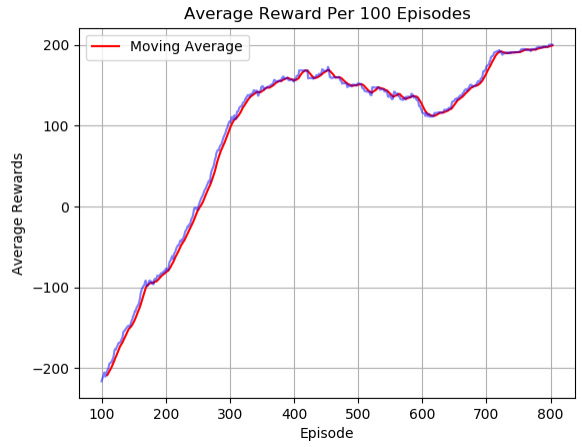

Figure 12: Benchmark average reward

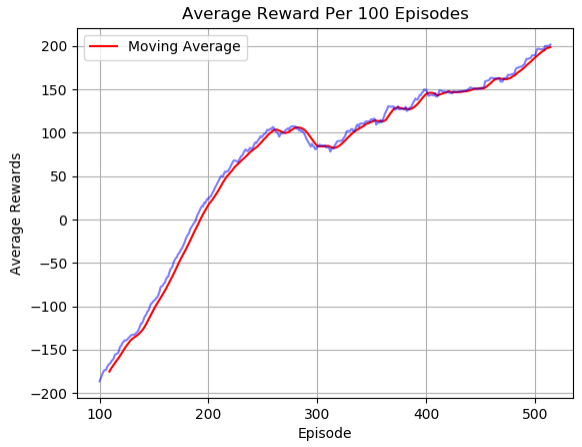

Figure 13: Solution average reward

Figures 14 & 15 show the per episode total reward for the optimized algorithm. It was observed that rewards did increase to about the optimal of 200 around the 175th episode for both models which directly correlates with the time the epsilon was close to its floor value (0.01) and the replay buffer was filled up and began dumping older values. The epsilon value had the greatest influence as the algorithm was using the max action 99% of the time after the 175th episode. There was other trials where epsilon was not degraded fast enough or it was degraded too fast and the performance of the lander was poor and did not converge after a large number of episodes.

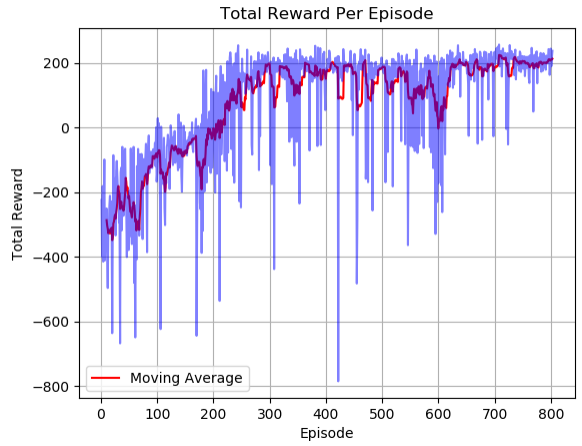

Figure 14: Benchmark total reward per episode during training

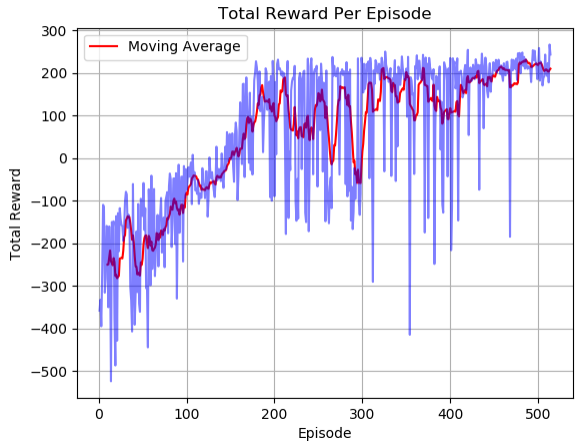

Figure 15: Solution total reward per episode during training

In Table 6 the time, in seconds, was recorded for the training until success of both the benchmark and solution models. The purpose of recording these values was to give an estimate on the speed of training of the models. That may or may not be a concern depending on the underlying needs and circumstances; however, it was a metric that was to be evaluated. It wasn't entirely surprising that the

solution model had a faster training time as it had less episodes run prior to success, but it wasn't proportional. For example, the differences in iterations before success was 500/800 (solution model/benchmark model) equating to 60% more episodes for the benchmark to complete vs the solution. The time increase until success for the benchmark was only 14% more than the solution model. This can be attributed to the greater complexity of the solution model's DQN taking longer to train.

Table 6

	Model Training Times	
Model	Benchmark	Solution
Time (s)	7930.06	6938.94

In Figures 16 & 17 the results of running the two models over 100 iterations, with no further training, is displayed. The figures clearly show that solution model performed better over the 100 iterations. The benchmark model had eight instances where it produced rewards well below zero which indicates the lunar lander crashed. Observing the 100 runs as they were rendered, the outliers were from some unusual landscapes in the environment itself. It was areas outside the yellow flags having odd shapes that caused the lander to topple over awkwardly. Another common outlier was sharp peaks where the lander landed on the peak which caused the lander to register a crash. Overall, the lander did a decent job considering the overall training and network size in both models. If given more training data and better parameter turning, the lander might have anticipated and/or handled outliers more gracefully. Overall, the solution model performed better in both the time/episodes during training and over the 100 episodes with the trained model. V. Conclusion

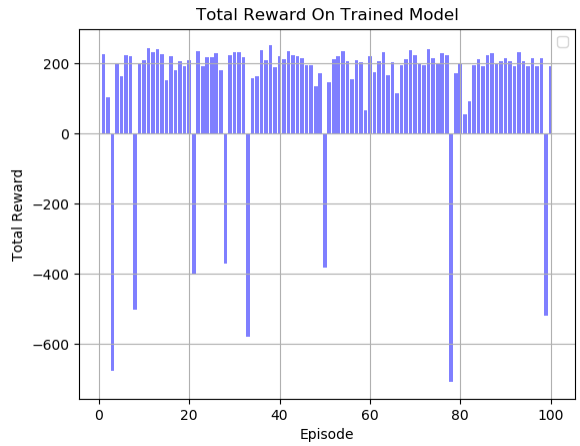

Figure 16: Benchmark test over trained network

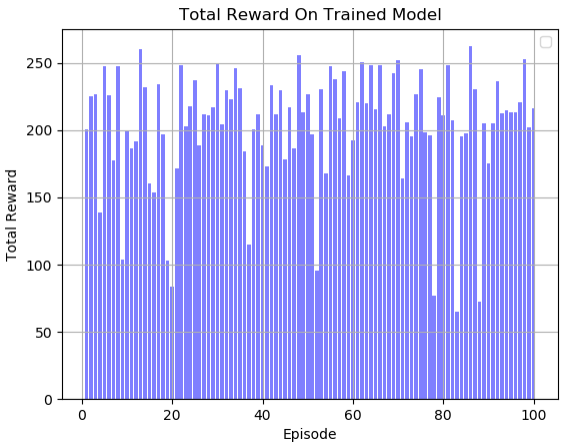

Figure 17: Solution model test over trained network

Free-Form Visualization

Replay buffer, step count and batch size all have a correlation with each other that took some time to finesse in order to find a solution that was successful. The replay buffer is filled up with each step. You need to provide enough steps per episode so the lander can potentially reach the landing area. So, if you have 1,000 steps per episode and a replay buffer size of only 2,000 you could potentially only be

storing the past 2 episodes in your buffer to sample from. You shouldn't just reduce the steps per episode in order to get more episodes in your buffer. What should be done is increase the replay buffer size and allow enough steps to permit proper exploration. Figure 7 shows the growth of the replay buffer over 500 episodes. As can be seen after about 125-150 episodes the replay buffer reached it max and would begin pushing out the oldest steps.

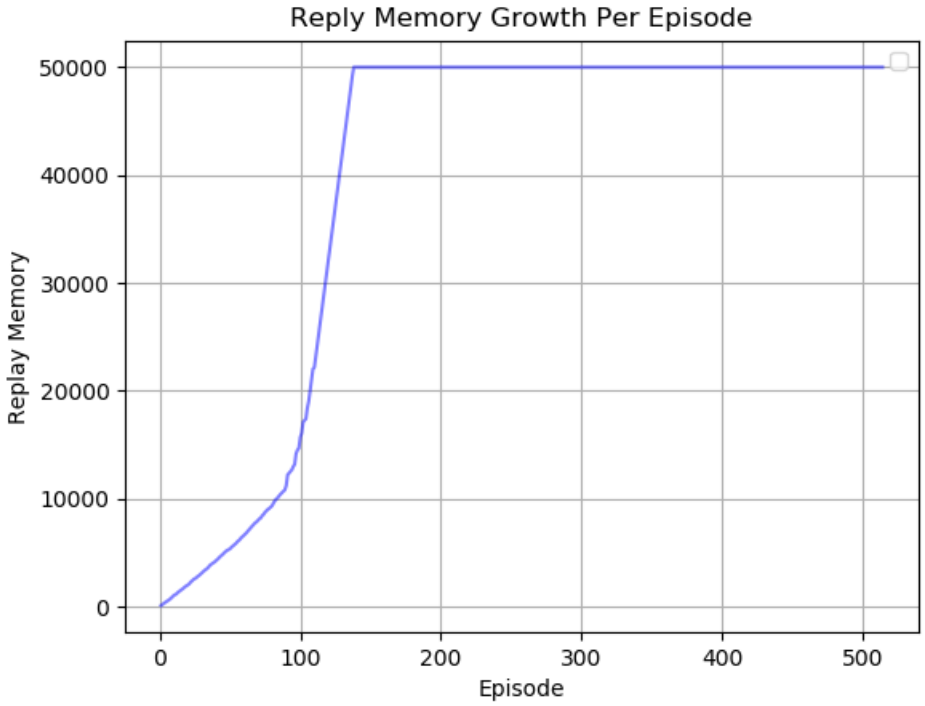

Figure 18 Replay memory growth per episode.

From the replay buffer, the batch size should allow enough randomly sampled data to train the neural network sufficiently. If the batch size is too small you may need far more episodes in order to train the neural network; however, you may run into an issue that I saw repeatedly. The replay buffer contains steps, assuming you have explored your state space sufficiently (see discussion about epsilon and epsilon decay) that has instances where the lander had a high reward such as landing between the flags. If your buffer is too small and/or you don't have a large enough batch value, you may find that these valuable steps could be removed from the buffer before the neural network can properly learn from them. This can happen if your exploration ends too soon and you begin taking the optimal action of your current policy. The lander may begin to hover in place based on this policy. As it hovers it is now filling up the replay buffer with steps that further remove any steps that had high reward due to exploration. Eventually, your lander only trains on samples of a small subset of the state space which is going to reinforce the NN on these suboptimal actions.

A replay buffer of 50,000 was sufficient to capture enough step history to successfully train both the benchmark and solution models. 1,000 steps per episode was used; however, as the episodes increased the steps per episode decreased. This intuitively makes sense since as episodes are run the policy

ideally should be improving which would require less steps to complete. There is a negative reward on main thruster use, so less steps potentially means less negative reward.

Reflection

One of the most difficult aspects of the project was the time it takes to train various models in order to evaluate their results. As previously discussed only a subset of parameter values could be tested in a reasonable amount of time. This does limit the robustness of the experiments; however, I do feel the overall results of the benchmark and solution do reflect well on the differences of the underlying model architecture.

As discussed in the free-form visualization section, the introduction of replay memory with the actor-critic model added a whole level of complexity that isn't normally present when training a NN. It was fascinating and frustrating at times trying to finesse a set of values that adequately preserved good training data but didn't bloat the experiments in terms of time and computational resources.

Improvement

It would have been nice to further investigate the other parameters in Table 3 that were not as thoroughly looked at as those in Figures 2-11. Testing time made it difficult to dive deeper into those values.

With Figures 2-11, it would have been more informative to have tested each parameter against variations of the other ones. Utilizing Scipy grid search would have been a solution I normally would have tried; however, the vast amount of combinations would not have been possible under the time constraints. Testing all the hyper parameter combinations against various neural network architectures would have possibly produced a more optimal solution, but that wouldn't have been feasible.

The limiting of the continuous space to the values selected would have been interesting to expound on as this could have influenced the NN models by introducing more trainable parameters. I am speculating that an increase in available values for the engines would have increased the time it would have taken to train the models which would have cascaded throughout the whole project with respect to overall time to accomplish the project.

It would have been advantageous to further investigate the issue of batch updates to the Keras training method vs the for loop method that was used. Sending the batch as a single parameter to Keras is substantially faster and can leverage GPU support better.

Once completed with the coding portion of the project, I came to realize I would have liked to refactor much of what was written to be far more modular. The overall code I found to be to linear and wasn't generalized as well as it could have been. What was difficult was making code adjustments that impacted training and having to rerun training and/or testing portions to evaluate those code adjustments. Even with running small subsets episodes and tests could take minutes to potentially hours when debugging changes.

References

NASA. (2018, December 19). *MARS Insight Mission*. Retrieved from NASA: https://mars.nasa.gov/insight/timeline/overview/

OpenAI. (2018, December 19). *Getting Started with Gym*. Retrieved from OpenAI: https://gym.openai.com/docs/

OpenAI. (2018, December 19). *LunarLanderContinuous-v2*. Retrieved from OpenAI Gym: https://gym.openai.com/envs/LunarLanderContinuous-v2/

Timothy P. Lillicrap, J. J. (2016). Continuous control with deep reinforcement learning. *arXiv:1509.02971*.

Volodymyr Mnih, K. K. (2013). Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602v1*.