**Project 2: Lunar Lander DQN**
**Prepared by: Sam Farren**

The lunar lander environment is a continuous state space environment consisting of an 8 dimensional input feature vector. The feature vector had 6 continuous states and two binary states and was as follows: [x coordinate, y coordinate, x velocity, y velocity, lander angle, angular velocity, right leg down flag, left leg down flag]. In addition, this environment has a discrete 4 state action space. The environment was considered solved if a score of 200 or greater was reached. Out of a 100 episode test run on my final model using a Deep Q-learning agent with a Keras neural network, the agent was able to solve the environment 5 times, and score between 100-200 73 times. While most of the problems so far have been discrete state space problems, this was the first continuous state space which posed for a lot of problems and workarounds.

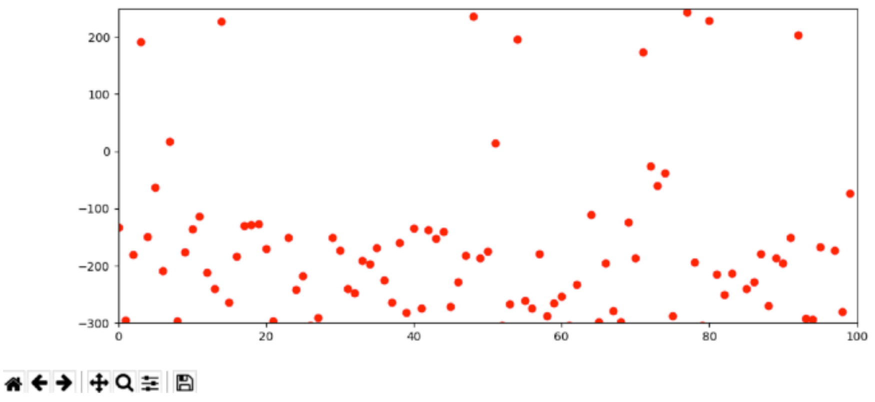
My initial thought at implementing an agent to solve this problem was to do a binned DQN approach to try and account for the continuity of the environment. The idea was to create bins for each continuous state in the state space in order to reduce the complexity of the environment into a more discrete form. There were many pitfalls to this however. After trying to fix these issues and not getting anywhere, I then decided to switch and try a Fully connected Feed Forward Neural Network with Keras and Tensor Flow where learning was based off of Q learning updates with various epsilon, gamma, and alpha values. An experience replay buffer was also used to further train the agent after an episode was completed.

The biggest challenge with implementing a neural network to solve this problem was determining the best parameters, hyper-parameters, loss functions, activation functions, and optimizers to capture the dynamic nature of the environment, but also not overfit and get the lander stuck in a local optimum. The base architecture of the neural network was an input and output layer where the input layer had 8 nodes representing the 8 states in the state space feature vector, and 4 nodes in the output layer to represent the 4 possible actions that could be taken. These two layers did not change throughout the experimentation process. What did change however was the number of hidden layers and nodes within the hidden layers in the network topology. So in summary, while it was very easy to set up the network topology and train the network with the environment, it was very, very difficult to get the correct loss function, activation function, optimizer, epsilon, learning rate, gamma, and other parameters that could also be tuned within each of the optimizers and loss functions. In particular, picking a learning rate and epsilon for determining how much you want the lander to explore the environment was also difficult. Initially, epsilon was set to 1, because you wanted the lander to explore as much as possible, then as training progressed, it was decreased by a factor between .95 and .99 after each full training training episode, since more information was available to the agent, so it could make better decisions. Another problem encountered was determining how much you wanted the agent to remember and replay for training. This ranged anywhere from 10 randomly selected states in the episode, to using every state within the episode. I ended up using the full memory of each episode to retrain the agent as I found the performance and learning to be better this way.

**Failed implementations:**
Many failed implementations were made in attempting to solve this environment, and consistently, each one got stuck in the same local optimum. In each situation, the agent would learn to hover above the ground and not get close to the ground (particularly y=0 in the state

space), which was due to the continual crashing and getting -100 points whenever it hit the ground at higher speeds. Therefore, for most of the models after training, the agent would just be seen hovering in the same position towards the top left or top right of the environment avoiding the ground at all costs, even though continually hovering led to horrible reward values, even those worse than just crashing into the ground without performing any action. Hidden layers with 32,64, and 128 nodes performed the worst in my case out of all implementations. In addition, having more than 2 hidden layers also decreased performance a ton, leading to suboptimal decisions, so I kept the network topology very simplified and tuned parameters based off of 2 hidden layers and 4 nodes in each. Below is a graph of the very first neural net utilizing Adam and Nesterov accelerator optimizers. It produces a very sparse and inconsistent score when testing the final models. All other optimizers, produced results yielding very similar inconsistent graphs.

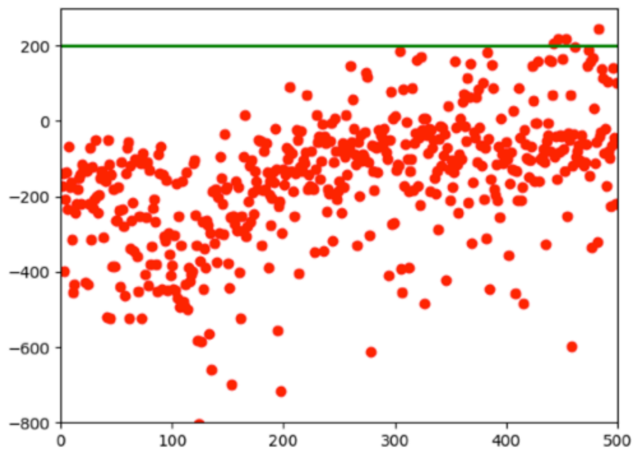
**100 Episode Test on Adam Optimizer**



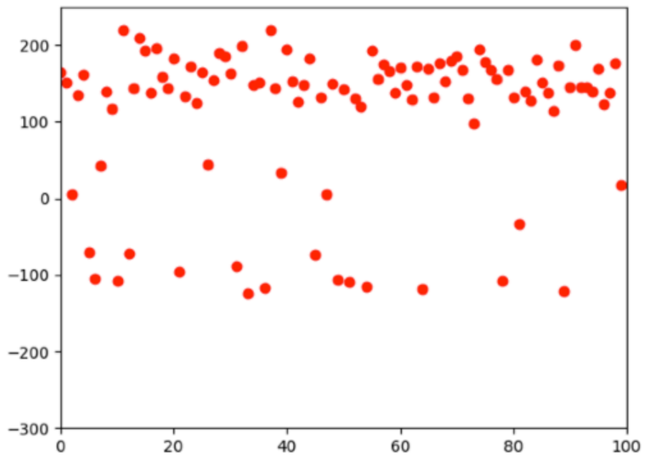
**Best Implementation:**

The best neural net topology consisted of two hidden layers with four nodes in each hidden layer. This was a surprise to me, because I didn't think this was enough structure to capture the complexity of the environment, but as I found, the more complex the network got, the worse the performance became, most likely due to overfitting. Below(on the left) you can see the score the agent received during 500 training episodes. You will notice an increase in performance after around 200 episodes, where the agent started to get in the 100's, with more gradual learning up to around 425 where the agent finally "solved" the environment and scored above 200. The figure on the right shows the scores from running the final model on 100 test episodes. The final mode used a RMSprop optimizer, similar to Adagrad, with rho=0.9, and zero decay. This optimizer was recommended for recurrent neural nets, and since the episodic state experience was very repeatable and recurrent, this ended up being the best choice. In addition, for the Q learning implementation in choosing the best action, epsilon was set to 1 initially with decay, gamma was 0.95, and alpha was .001. The loss function was kept as Mean Squared Error since this was seen in Project 1, and is recommended for many models. The last problem I couldn't seem to get passed was getting the agent to only use it's main thruster when absolutely necessary. Instead, it seemed to always have it's main thruster on which would rack up a ton of small negative rewards and make the final score just below 200, around 150-180. This stemmed from it learning to not crash, but the best way to learn not to crash is either to not get near the ground or approach the ground very very slowly. Finding the maximum velocity that the lunar could hit the ground and still not crash was the ideal find for this last problem, but I couldn't find a way to make this happen.

**Score for 500 Training episodes**



**100 Episode Test on Final Model**



**Conclusion:**
The Lunar Lander environment was the first continuous state space that we were exposed to. Many challenges were encountered, mostly with experimentation in tuning the many many parameters and hyper parameters that are in the Deep Q Learning model. I think the utilization of Keras was a very valuable technology that will be used in many more projects down the road. While the final model didn’t produce consistent environment solving, it did learn to land almost every time with at least one leg between the flags.

Works Cited:

*Ruder, S. (2018, March 03). An overview of gradient descent optimization algorithms. Retrieved March 19, 2018, from http://ruder.io/optimizing-gradient-descent/index.html#rmsprop*