CS 622 Final Project Write Up

Korben DiArchangel

December 20, 2022

1 Introduction

Box2D's Lunar Lander is a complex toy problem that involves landing a ship on a surface. The center of this surface will remain flat and constant, but the surrounding terrain will change with every run. The user controlling the ship can adjust the ship's velocity and rotation by taking one of four actions: firing the left thrusters, firing the right thrusters, firing the bottom thrusters, or not using any thrusters at all. A success is achieved when the ship lands on the surface and doesn't move or use any thrusters. A failure happens when the ship crashes or runs out of time. The ship must land upright on its two legs without touching the surface in order to land correctly.

We will be using Q Learning, a reinforcement learning algorithm, to train a model that can consistently land the ship. The Q-Table model uses a matrix of discrete states to determine the correct action for a particular state. Q-Learning trains this model by putting it in the environment, and giving actions higher values if they lead to more successes, and giving action lower values if they lead to more failures. This will eventually lead to the model knowing the correct action for each state, and the model will be able to consistently succeed at the task.

2 Lunar Lander Train

This code will train a specified amount of Q-Table models through a reinforcement learning process. This process will run for a set amount of episodes (an attempt to successfully land the ship), and will adjust the model according to the rewards/punishments it receives. It will favor actions that lead to a higher reward, and will not take actions that will cause a punishment. Eventually, with the right hyperparameters, it will learn how to land the ship consistently.

The main function will run multiple algorithms in parallel to allow for more general results. Each parallel process will have its own seed so that the resulting Q-Tables will be different for all processes. Passing the same seed will return consistent results.

2.1 Parameters

There are a large amount of hyperparameters that the programmer can set before runtime. SEED is the random seed set at the start of the program; setting the same seed will result in the same Q-Table being produced. NUM_PROCESSES is the amount of Q-Tables to be produced; they will be trained in parallel. LEARNING_RATE is how fast the model learns; a lower learning rate means that it will learn less from mistakes, but a higher learning rate means that it will only consider recent episodes. EPSILON is a random chance that the algorithm will pick a random action instead of most optimal one. It will start at EPSILON_INIT, be reduced multiplicatively by EPSILON_REDUCTION, and will have a minimum of EPSILON_MIN. Lowering the epsilon like this will allow the algorithm to explore states, and to exploit the actions that will lead to success. DISCOUNT is the amount of future reward the algorithm considers when adjusting the table. Higher discount means that long term reward is valued over short term reward, and since the largest amount of reward is given at the end, this is typically set to something very high. EPISODES is the amount of times to run the environment; higher EPISODES means that the model will be trained more, but will take longer to train. RENDER_EPISODE sets when to display the current episode, RENDER determines if to display the current episode at all, and EPISODE_CHECK determines when to print the progress of a process on the console. DISCRETE_OS_SIZE determines the bin size of the Q-Table; higher sizes will be more precise but will require more training. FOCUSED_OS is the specific range in the observation space that will contain the majority of bins. Q_INIT is what to set the Q-Table values to initially; it will choose random values between the ones set.

2.2 Get State From Observation

This function will calculate the discrete state from a continuous observation. If the value of a dimension is not within the focused observation space, it will classify it as either 0 or the number of bins specified for that dimension minus one (the highest and lowest values). Otherwise, it will return the value that corresponds with the continuous value in the focused space.

2.3 Get Action From State

This function will calculate the appropriate action given a state. It will return the argument of the highest value of the Q-Table on a certain axis; this argument represents the action to take. For example, if state S has Q-Values of [0 6 3 2], it will return 1, as it is the index of 6, and the learner will take action 1.

2.4 Q Learning

This function will train and return a Q-Table using the Q-Learning algorithm, and it will also return the reward gathered in each of the training episodes. For each episode, it will create an environment, and it will iterate through each state in the environment by getting the discrete state from the observation (using the function), and then either picking the best action (using get_action_from_state), or picking a random action, depending on epsilon. It will then take the action, calculate the future reward, and use that to calculate the new value for the previous action taken on the Q-Table. It repeats this until the ship lands, crashes, or times out, and then will repeat the steps for a specified number of episodes. It will return the final Q-Table as well as the reward obtained in each episode. The reward obtained throughout all processes will be averaged and put on a table.

3 Lunar Lander Test

This code will read a directory of Q-Table models generated by the training function, and will test them against the lunar lander environment for a set amount of episodes. It will then output the average reward earned for each Q-Table and the number of successes it achieved. This function uses variables and functions from lunarlandertrain.py, including get_state_from_observation, get_action_from_state, and the observation space min/max variables.

Similar to the training code, the main function will run the code in parallel; it will create a process to run for each Q-Table in the input directory. This will help the testing process run faster and produce results quickly.

3.1 Parameters

This function only has two parameters; SEED and NUM_EPISODES. SEED will set the random seed of the function, similar to the seed variable in the training code. This allows for consistency and reproducibility. The other parameter, NUM_EPISODES, allows you to decide how many times the table will be tested against the environment. A higher amount of episodes will produce more accurate measurements, but will take longer.

3.2 Test Model

This function will take in a Q-Tables (along with an ID) and will return the reward the Q-Table has obtained through the lunar lander environment. It works similarly to the training function, but does not modify the Q-Table. It will read the environment, take the best action according to the Q-Table, and update the environment based on its action. It will record the amount of reward it receives while the environment is running. It will then repeat this process for a specified amount of times (NUM_EPISODES), and use the data it gathered to output the average reward and the number of times the Q-Table was successfully able to land the ship.

4 Results

The model was eventually able to learn to consistently land the ship on the surface. A reward of 200 or higher meant that the landing was successful. Higher values above 200 mean that less fuel was used when landing. Values less than 200 but above 100 meant that the ship touched down on the surface, but wasn't able to stop its engines.

4.1 Training

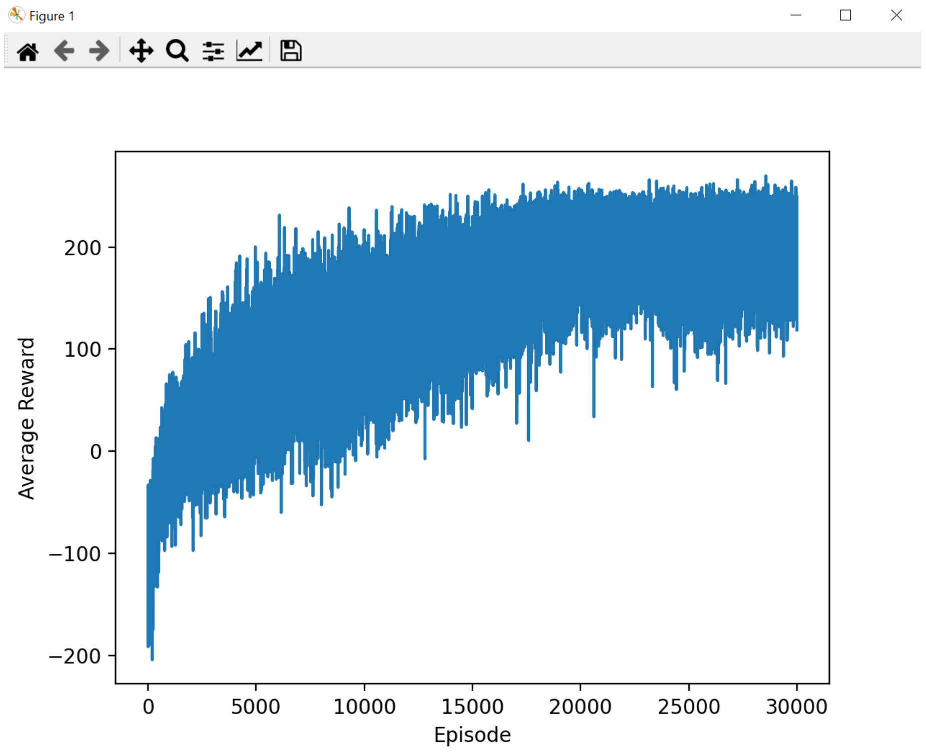
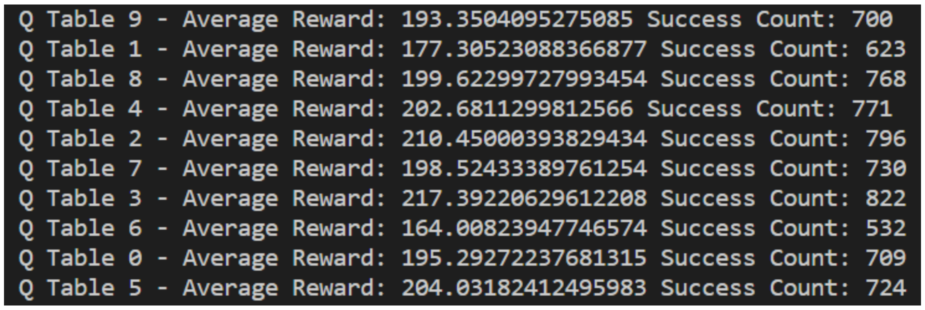


Figure 1: The average reward of all ten Q-Tables throughout training.

Figure 1 shows the reward over time of the model. The training took 30,000 episodes and around one hour to learn the environment. It started off not doing too well, as it wasn't familiar with the environment, and it would frequently take random actions. It would then learn which actions worked and which didn't, and started to succeed more consistently. It eventually learned how to land, with the ship rarely crashing and being successful more often. The video attached also shows how the ship learned to land over time.

4.2 Testing



```
Q Table 9 - Average Reward: 193.3504095275085 Success Count: 700
Q Table 1 - Average Reward: 177.30523088366877 Success Count: 623
Q Table 8 - Average Reward: 199.62299727993454 Success Count: 768
Q Table 4 - Average Reward: 202.6811299812566 Success Count: 771
Q Table 2 - Average Reward: 210.45000393829434 Success Count: 796
Q Table 7 - Average Reward: 198.52433389761254 Success Count: 730
Q Table 3 - Average Reward: 217.39220629612208 Success Count: 822
Q Table 6 - Average Reward: 164.00823947746574 Success Count: 532
Q Table 0 - Average Reward: 195.29272237681315 Success Count: 709
Q Table 5 - Average Reward: 204.03182412495983 Success Count: 724
```

Figure 2: The average reward and number of successes of each Q-Table across 1000 episodes.

As shown in **Figure 2**, the trained models frequently succeeded at landing the ship. The average reward across all trained models was 196.2, and the average success rate was 71.2% (the models were tested across 1000 episodes). The highest values obtained was an average reward of 217.4 and a success rate of 82.2%.