Dyllon Dunton
Advanced Robotics
Final Project Report
Car-Racing-V3 Solution

My final project was a solution to the Car-Racing-V3 OpenAI Gymnasium game. This game consists of a car driving through a randomly generated track, where points are earned by making progress on the track. My solution to this problem will be a neural network Q-Learning approach. In addition, I will do a single step lookahead with a truncated rollout, which I will compare to the base policy generated by the Q-Learning approach. This report will cover the rules of the game, my methodology for solving it, and the results of the solution.

The setup for this game is a car on a track. The state is a 3 channel square image with a side length of 96 pixels. The action space has three continuous values. The first value is the steering, which is on a range from -1 to 1. The other two values are the gas and brake, which are on a range from 0 to 1. In order to make the state and action space more manageable, the state was grayscale and cropped, resulting in a single channel square image with side length of 80 pixels. In addition, the action space was discretized to values at every 0.5. This means that the steering has 5 discrete values, and the gas and brake each have 3 values. This creates 45 different combinations of actions. However, some actions are contradictory and have both gas and brake at the same time. While drifting could be a valid solution, I wanted a simple approach so I pruned the action space to just 25 different actions.

In order to solve this problem, two things are needed: A neural network base policy, and a way to generate and fit the neural network to the data. The base policy is a Convolutional Neural Network (CNN). The input is the state size, and it is convolved down to a 16 channel square image with a side length of 5 pixels over 4 convolutions. This is then flattened down to a single array with a length of 400. Next, 5 values are computed and concatenated to the array for a final length of 405. These 5 values are the speed, angular velocity, compass heading, and steering angle of the car, as well as whether or not the car is off road. Next, this is passed into 3 fully connected layers, reducing the length from 405, to 120, then to 80, and finally 25. These 25 values would be the predictions of the reward for taking that action given the input state. During inference time, the action that is picked is the action with the highest reward.

Next, in order to train the network, it is important to generate data that shows the reward, next state, and terminality, given a state-action pair. To generate this data, I use an epsilon-greedy algorithm that, in the beginning, takes random actions and records the corresponding reward, next state, and terminality in order to explore the state-space and learn which actions give higher rewards. Then, after 200 episodes, it will have learned the basic information about the environment, so it can start to take actions based off of its learned policy rather than just random. This change is gradual and the chance that the action is random rather than chosen is equal to 1 in the beginning, and is then multiplied by 0.995 each episode after episode 200. Every state-action pair taken is put into a replay buffer of length 10,000 along with the corresponding information about the reward, next state, and terminality. This buffer only holds

the most recent 10,000 records. After the buffer is full, a random set of 128 records is sampled from the buffer and used to fit the network to the data at a learning rate of 0.0001.

I had a lot of trouble with this algorithm at first and could not get the car to reliably drive down the track after 800 episodes. After trying many things, what worked was some very specific reward shaping. The default reward function was simply a -0.1 reward each action, and a 1000/N reward for each road tile reached, where N is the number of road tiles on the track. While this did give a goal of quickly reaching road tiles, it did not actually give any immediate penalty for going off of the track, or reward the agent for driving with a high speed directly. I reshaped the reward function to give a steep penalty for driving off road, and then gave an even steeper penalty and terminated the episode after being off road for 20 consecutive actions. I wrote the function to check if the car was off road by sampling the color in front of the car. If it was green, then the car was off road. If it was more gray, it was on the road. In addition, I gave a reward linearly proportional to the speed, capped at 3 for each action, which is about the same reward given for the average track tile reached. This not only served to directly teach the agent that driving off road was bad and encouraged high speed driving, but also restricted the replay buffer data to states that were on the road. This meant the car would spend more time learning about driving on the road than it would have if the data included the car quickly driving off road and collecting data there instead. This quickly learned how to stay on the track and quickly drive through turns. However, the policy is not perfect and after three or four successful turns, it has learned that stopping is the best way to safely keep a high reward rather than continuing through the track. Since I do not have much more time to retrain another model, this will be my final policy for the project, trained to 4000 episodes.

In addition, I was able to program a rollout algorithm that uses the learned base policy to get a substantial reward improvement and actually finish the track. To do this, I needed a method for simulating the future states of the environment. Since the Car-Racing-V3 environment had no way to duplicate the environment, I was forced to simply keep a list of the previous actions taken, generate a new environment of the same track, and run the game with the previous actions taken to reach a duplicate environment at the same state. At each point in the rollout algorithm, I do this 25 times (one for each action) and run the base policy for 20 additional actions. Then it recorded which action had the highest reward and then used that action in the main environment. After doing this over and over again for each action in the main environment, the rollout algorithm travels through the track with a much higher reward.

The Demo and code are attached to the submission. The Demo is also shown in the README on the github repository. In the demo, the left side is the base policy and the right side is the truncated rollout. It is clear that the rollout algorithm outperforms the base policy, which is to be expected due to rollouts guaranteed reward improvement. However, it can be noted that even though it can perform better than the base policy, it cannot solve problems that the base policy itself can’t solve. This is clearly seen in the final turn of the demo. The base policy was not accustomed to turning right and failed to make it through the turn. Therefore, since the base policy is the heuristic used in rollout to better predict the cost-to-go of each action, the rollout also fails.

The next time I do a similar project, I would make a few changes. Firstly, the Car-Racing-V3 environment has plenty of problems. Since it is an old environment, some of the functions are broken. For example, I could not change the winding direction of the track, which meant the agent learned much more about left turns than right turns, which is the exact reason that it fails so readily with right turns. In addition, most environments have a built-in method for copying the environment at a particular state. However, this one does not, which means I must take advantage of the determinism of the track to use the same random seed and keep a list of the actions taken, and duplicate the environment from the beginning and run the actions all the way to the present state in order to generate a copy. This makes the rollout procedure take an exponentially longer amount of time each step into the track. I would like to try this project again with a different environment, such as the multi-car-racing wrapper found here: https://github.com/igilitschenski/multi_car_racing. However, I am happy with this solution to this environment.