Implement a Basic Driving Agent

To begin, your only task is to get the smartcab to move around in the environment. At this point, you will not be concerned with any sort of optimal driving policy. Note that the driving agent is given the following information at each intersection:

• The next waypoint location relative to its current location and heading.

• The state of the traffic light at the intersection and the presence of oncoming vehicles from other directions.

• The current time left from the allotted deadline. To complete this task, simply have your driving agent choose a random action from the set of possible actions (None, 'forward', 'left', 'right') at each intersection, disregarding the input information above. Set the simulation deadline enforcement, enforce_deadline to False and observe how it performs.

QUESTION: Observe what you see with the agent's behavior as it takes random actions. Does the smartcab eventually make it to the destination? Are there any other interesting observations to note?

ANSWER: The agent just takes random actions. The goal is reached; however, this happens in an *arbitrary amount of time. The agent can be in the previous waypoint but just take a different direction than the one of the goal. Deadline goes to 0 and then to negative values, and reduction in the deadline value does not stop until goal is reached and the board is reset.*

Inform the Driving Agent

Now that your driving agent is capable of moving around in the environment, your next task is to identify a set of states that are appropriate for modeling the smartcab and environment. The main source of state variables are the current inputs at the intersection, but not all may require representation. You may choose to explicitly define states, or use some combination of inputs as an implicit state. At each time step, process the inputs and update the agent's current state using the self.state variable. Continue with the simulation deadline enforcement enforce_deadline being set to

False, and observe how your driving agent now reports the change in state as the simulation progresses.

QUESTION: What states have you identified that are appropriate for modeling the smartcab and environment? Why do you believe each of these states to be appropriate for this problem? OPTIONAL: How many states in total exist for the smartcab in this environment? Does this number *seem reasonable given that the goal of Q-Learning is to learn and make informed decisions about each state? Why or why not?*

ANSWER: The states selected {inputs['light'], inputs['oncoming'], inputs['right'], inputs['left'], self.next_waypoint}. I chose them because I believe this way we can represent all the information we have with regards to the current state. To my understanding, since we do not have our global location, the taxi determines its state by what the traffic light is, where the planner says should be the next waypoint, and what the intended move is, for vehicles that are oncoming, or coming from the left or the right side of the intersection.

Thus, if I have not misunderstood, the taxi does not take into consideration at which intersection it is (the taxi does not know), and treats all points where 'light', 'oncoming', 'right', 'left', and 'waypoint' are the same, as one. For all these points, if there is no random selection chosen, the same action will be chosen (while, of course, Q(S,a) remains the same).

*'light' can take 2 values, 'oncoming', 'right', and 'left' can take 4 each, and 'waypoint' can take 3. Thus, the possible states in total are 2*4*4*4*3 = 384 .*

The deadline also provides information, however *has many values, and thus it would increase the number of possible states significantly. (For 100 different values we would then have 3840 states). So I chose not to include it in the state.*

A number of 384 states, in my opinion is reasonable. In the specific application that we work on, since the Q table is not reset for every new destination, we will probably have evaluated al state/action pairs before the program exits.

Finally, representing the inputs in the state separately (instead of together like in the previous submission), makes the number of the possible states significantly lower than the 384 mentioned above.

All 'None' values in the state are grouped together to one 'None' by Python. Therefore, all states that have the same non-'None' inputs are the same.

*Thus, 'None' in the states is filtered, which reduces the number of possible states to 2*3*3*3*3 = 162.*

This is even more reasonable than the 384 we had previously, and it is possible to evaluate all of them before the program execution ends.

Implement a Q-Learning Driving Agent

With your driving agent being capable of interpreting the input information and having a mapping of environmental states, your next task is to implement the Q-Learning algorithm for your driving agent to choose the *best* action at each time step, based on the Q- values for the current state and action. Each action taken by the smartcab will produce a reward which depends on the state of the environment. The Q-Learning driving agent will need to consider these rewards when updating the Q-values. Once implemented, set the simulation deadline enforcement enforce_deadline to True. Run the simulation and observe how the smartcab moves about the environment in each trial.

The formulas for updating Q-values can be found in this video.

QUESTION: What changes do you notice in the agent's behavior when compared to the basic driving agent when random actions were always taken? Why is this behavior occurring?

ANSWER: I believe that the agent now does not explore as much as before. Some actions tend to repeat themselves when the agent finds itself in the same intersections, and sometimes the agent tends to seclude itself in a smaller area of the grid, while before it seemed to traverse more waypoints. I believe that this occurs because the agent was given some reward for some states that is high, so being in the same state, the agent tends to repeat the action that yielded this high *reward.*

Improve the Q-Learning Driving Agent

Your final task for this project is to enhance your driving agent so that, after sufficient training, the smartcab is able to reach the destination within the allotted time safely and efficiently. Parameters in the Q-Learning algorithm, such as the learning rate (alpha), the discount factor (gamma) and the exploration rate (epsilon) all contribute to the driving agent's ability to learn the best action for each state. To improve on the success of your smartcab:

• Set the number of trials, n_trials, in the simulation to 100. • Run the simulation with the deadline enforcement

enforce_deadline set to True (you will need to reduce the update delay update_delay and set the display to False).

• Observe the driving agent's learning and **smartcab's** success rate, particularly during the later trials.

• Adjust one or several of the above parameters and iterate this process.

This task is complete once you have arrived at what you determine is the best combination of parameters required for your driving agent to learn successfully.

QUESTION: Report the different values for the parameters tuned in your basic implementation of Q-Learning. For which set of parameters does the agent perform best? How well does the final driving agent perform?

ANSWER: I run the program 100 times (with 100 simulations each), for various alpha, gamma and initial epsilon (I used epsilon decay) combinations. I modified the code to do this automatically, export the results to an excel file, and print out the best result.

The range of alpha and gamma was (0 to 1 – 1 included) with steps of 0.1, and the range of epsilon was 0.0 to 0.15 with a step of 0.05.

Epsilon was reduced during each update by (1/number_of_times_update_occurred). 0.0, 0.05, 0.10, 0.15 were only the initial values.

Its value was reset during each call of the agents reset method (number of times update occurred was reset, so epsilon took its original value)

So for each apha-gamma-epsilon combination, the program run 100 times, and each time a destination and a deadline was set 100 times. (In total 10,000 destinations)

Measured were the number of times the taxi *reached its goal, the rewards collected, the number of steps taken each time until the goal was reached, the average rewards per step.*

Additionally, it was observed whether the cab was reaching its goal during the last 10 times of each run, during which it had to find the target 100 times.

The performance metric considered was the number of successes (reaching the goal in time).

Tie breaker (or second metric) was the average rewards per step, and finally the number of moves

The best result was observed for alpha = 0.2, gamma = 1.0, and epsilon = 0.0.

With this combination, the smartcab reached the target on time 9817 out of the 10000 times. The average rewards per step were also the second highest one, and the number of moves per successful run was the lowest.

In Figure 1 the 10 parameter combinations with the highest number of successes are presented.

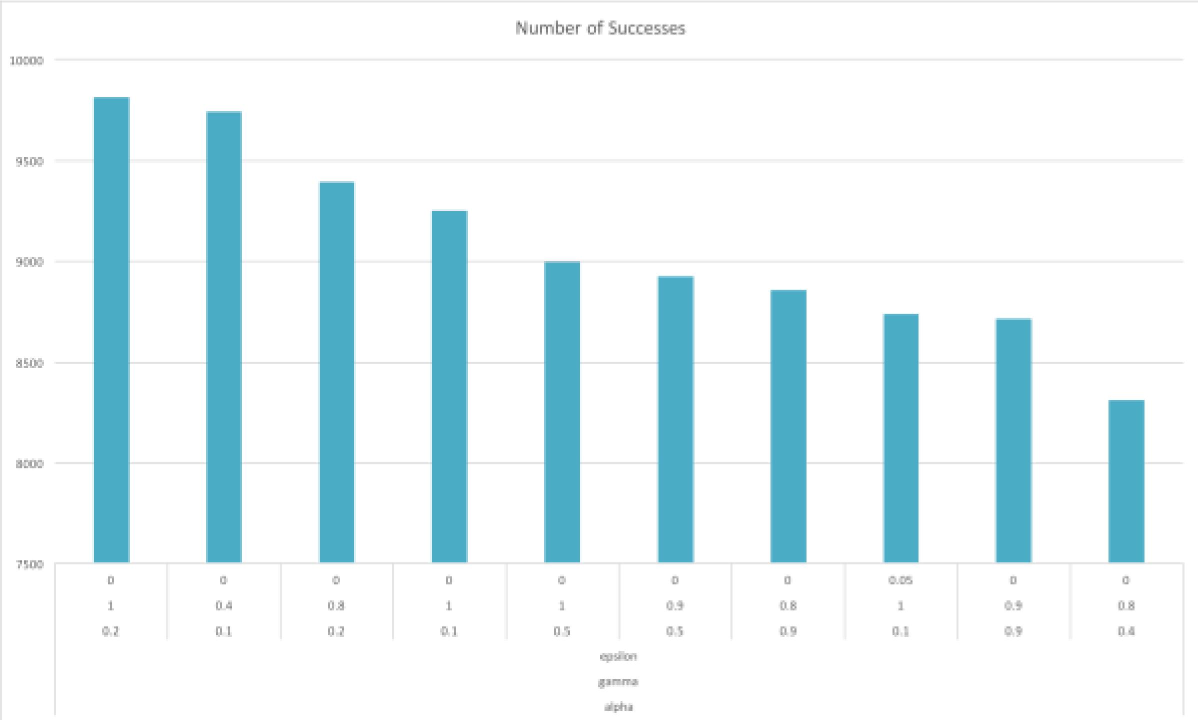

Figure 1 Parameter combinations with the most successes

In Figure 2 we can see the results for the same parameter combinations with regards to average rewards per step.

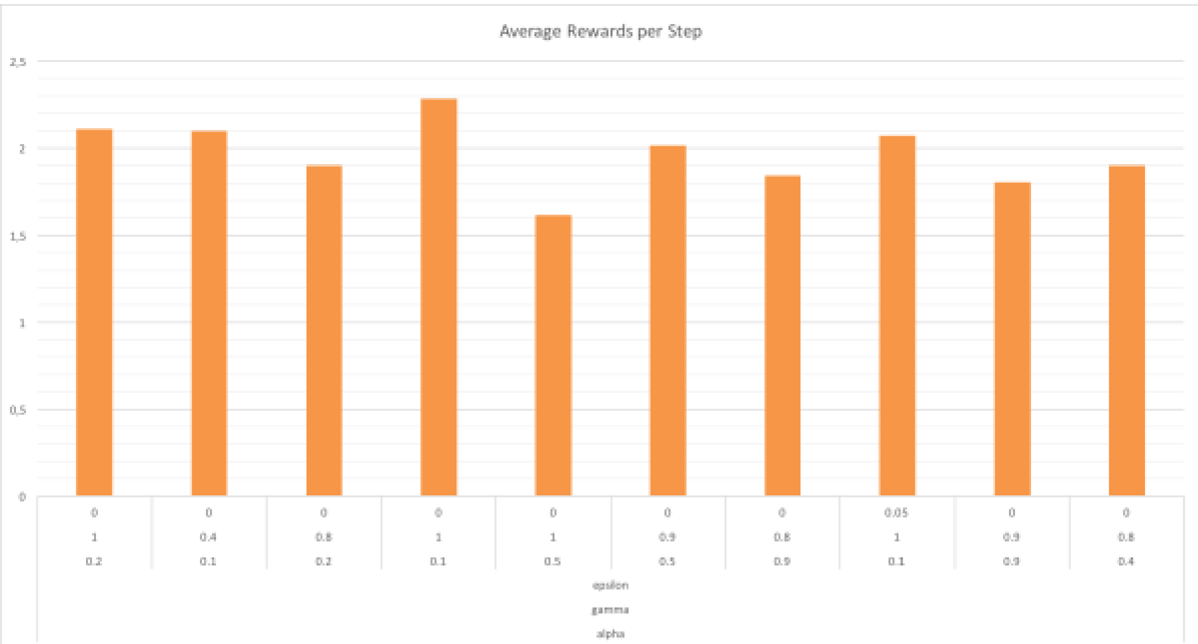

Figure 2 Average rewards per step

As can be seen, the combination with the highest average rewards is the one with alpha = 0.1, gamma = 1, and epsilon = 0. At the same time, the parameter combination chosen as optimal, had the second best performance with regards to this metric.

Finally, in Figure 3 is presented the number of moves required by each of the 10 parameter combinations, in order to reach the target. As can be seen, the combination chosen as optimal has the lowest average number of moves in order to get to the goal, compared to all the rest.

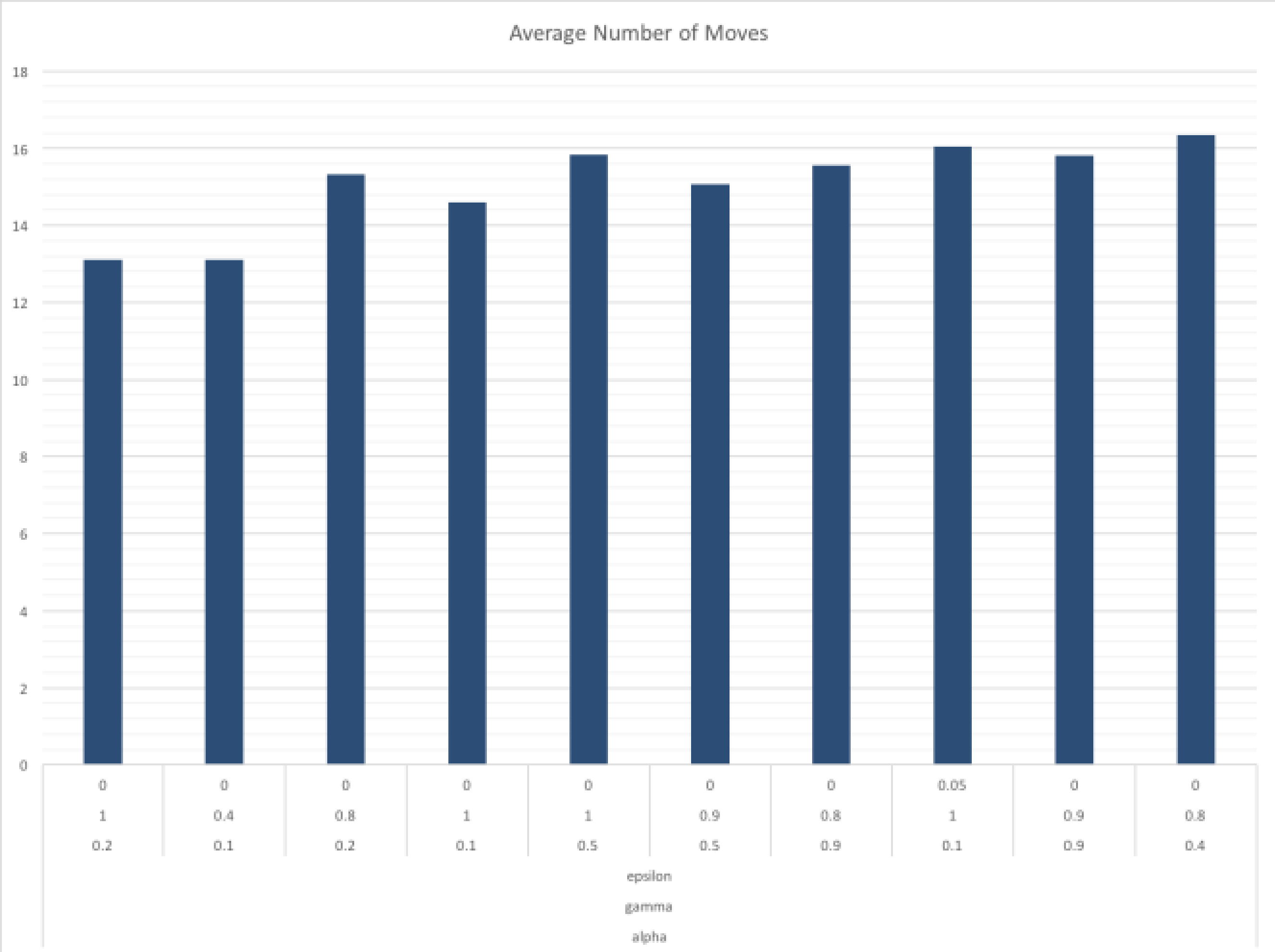

Figure 3 Average number of steps required to reach the goal

To summarize, the success percentage (car finding the goal) was in many combination trials 100%. The success rate when using the "optimal" parameter combination was 98.17% (9,817 times reaching the goal, out of 10,000). In the previous submission that percentage was estimated near 1.5%, and that was for 2 reasons:

- *The Q table was reset during every run. In this updated version Q is global, and does not get reset during every run.*

- *The 1.5% was not the percentage of the successful runs, but the percentage of the successful moves. The estimation was wrong, since the number of successes or failures was updated during every call of the update() method, hence every move. Thus, the results appeared to be significantly worse than what they actually were.*

QUESTION: Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties? How would you describe an optimal policy for this problem?

ANSWER: I believe that my agent has found (or at least is close to finding) an optimal policy. The agent should be learning during the beginning, and during the last attempts should mostly find the goal. In Figure 4 we can see the times (out of a 100 altogether) that the 10 "best" parameter combinations reached the goal during the last 10 attempts. As can be seen, for the "optimal" combination (alpha = 0.2, gamma = 1, and epsilon = 0) this happened in 93 out of the 100 cases. (With the second best combination at 92/100)

Additionally, the agent had the lowest average number of moves in order to get to the goal. This shows that the agent not only had the highest success rate, but this was also done in the least *moves possible.*

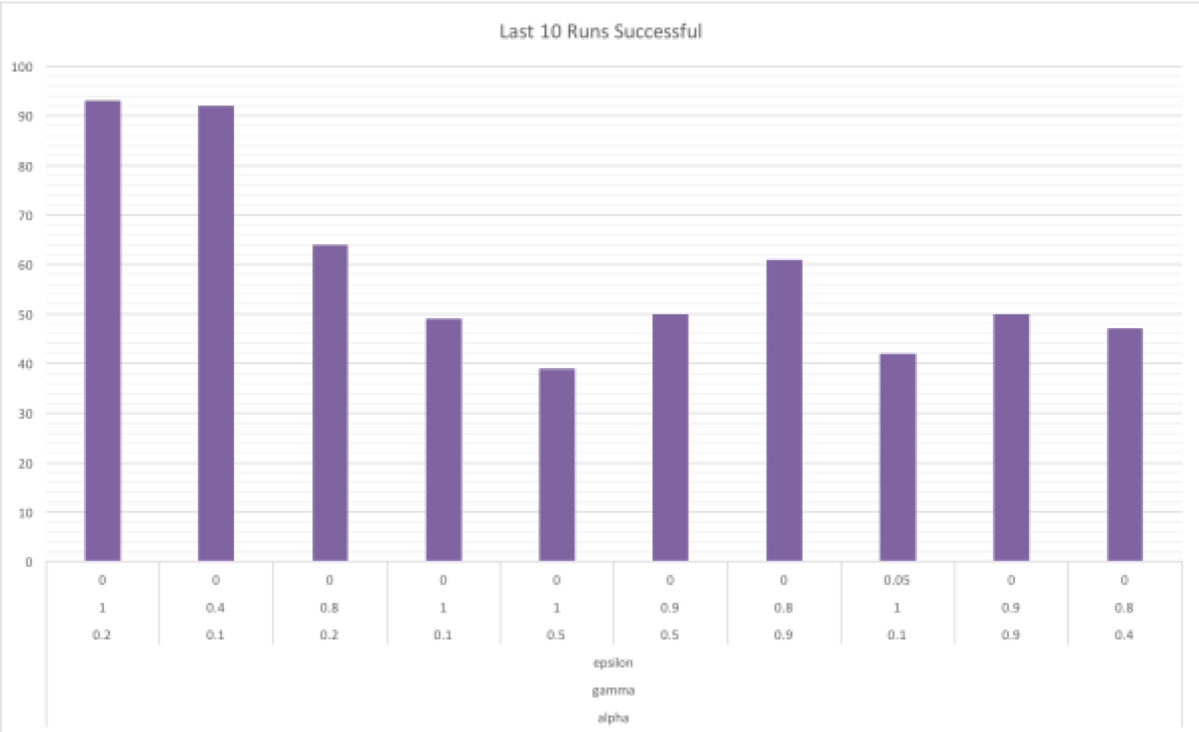

Figure 4 Number of times the last 10 attempts to find the goal where successful