Implement a basic driving agent

The most basic driving agent, is one which has an absolutely random policy and path. At every step the agent randomly chooses from the following options “left”, “right”, “forward” or None. In this way, at each step the agent will either go left, go right, go forward, or do nothing. Of course this is a less than optimal taxi cab! However it’s worth noting, that with no deadline enforced, in the limited grid, the agent does indeed always accomplish the task and reach the goal. It just sometimes takes a really really long time.

Implement states for the driving agent.

The goal of the agent is to reach the destination before a limited amount of time runs out while maximizing its collection of rewards. If we examine the reward structure, it becomes easy to see which inputs should be used in the agent state. The agent receives, 10 points for completing the trip within the given amount of time, thus we should include the time variable, however the exact implementation needs to be thought about carefully. If we include all steps of T, then the state space will explode and the agent will need a very long time to learn the correct Q values (much longer than 100). For this reason, I omitted the time from the state. For future work, I may want to implement a time variable which is >2 or <2, in this way the agent would have two additional parameters for state, and would not need to explore so much in order to incorporate this state variable into the Q table efficiently.

The agent receives 2 points for getting to the next waypoint, thus we should include the next waypoint. The agent receives -1 points if it breaks a traffic law, thus we need to include the variable for this : the traffic light. In total we get the following three variables for state: Time_to_destination, Light, and next_wapoint. From these three variables for state, our agent should be able to learn the rules of traffic and be aware of when it’s worth breaking a rule in order to make it on time. However due the complexity that Time_To_Destination introduces, I went ahead with only two variables for the state.

Basic Q-Learning Implementation.

In this section we explore what a very basic Q-learning algorithm does for our agent. In the basic approach, I implemented the following Q-learning equation.

Q(State, Action) = (1-Alpha) * Q(State,Action) + Alpha* (Reward + Max_Q(Action_Prime, State_Prime)

Where _Prime, denotes the result of the current action. In laymens terms, this comes down to, “combine the value of the current action with the value of the next action, based on what we know so far.” Overtime by exploration, the agent will learn the Q-value, or utility for any given (state, action) pair.

When I implemented this model, I chose alpha = .5. By choosing .5, we are evenly combining the value of the current state, with the future state and not biasing towards either. The results were

somewhat promising. The agent's performance in this environment was a bit erratic. On a few runs of 100 trials, it did as well as 100% accuracy on one end of the spectrum, down to 41 percent accuracy on the other.

After documenting the net-rewards per trial, I can see that the agent is either learning the correct policy early on, or failing to. Whatever happens in the beginning is sort of "stuck" and the agent either succeeds regularly, or not much at all. Although the agent do always succeeds at least 40% of the time. The agent does learn both traffic rules (stops for red lights) and it successfully follows waypoints. However, this only happens in the trial runs where the agent learns the correct steps early on, else it will constantly fail.

The changes in the behavior are the agent slowly begins to learn the traffic rules and follows them. Additionally it will learn to follow the next_waypoint in order to get the rewards. This makes sense because the state contains both the traffic light and the next_waypoint. The Q-table will then update states based on those parameters, both of which are directly

Enhanced Q Learning

The next step is to enhance the Q-learning algorithm and tune the parameters. The main enhancement to make to the Q-learning procedure is to include a gamma value for the future rewards. The update equation looks like this:

Q(State, Action) = (1-Alpha) * Q(State,Action) + Alpha* (Reward + *gamma* *Max_Q(Action_Prime, State_Prime).

This gamma variable, is a discount factor on the future rewards. It discounts future rewards that we believe we will receive. This has the effect of "add the the current value and *some* of the future value when updating the current (state,action) pair".

I also implemented an random(action) function, which chooses a random action with probability Epsilon. I set this to 1%, so that for each action, 99% of the time the ARG_maxQ is returned as the action, but 1% of the time it will totally random. This encourages exploration occasionally.

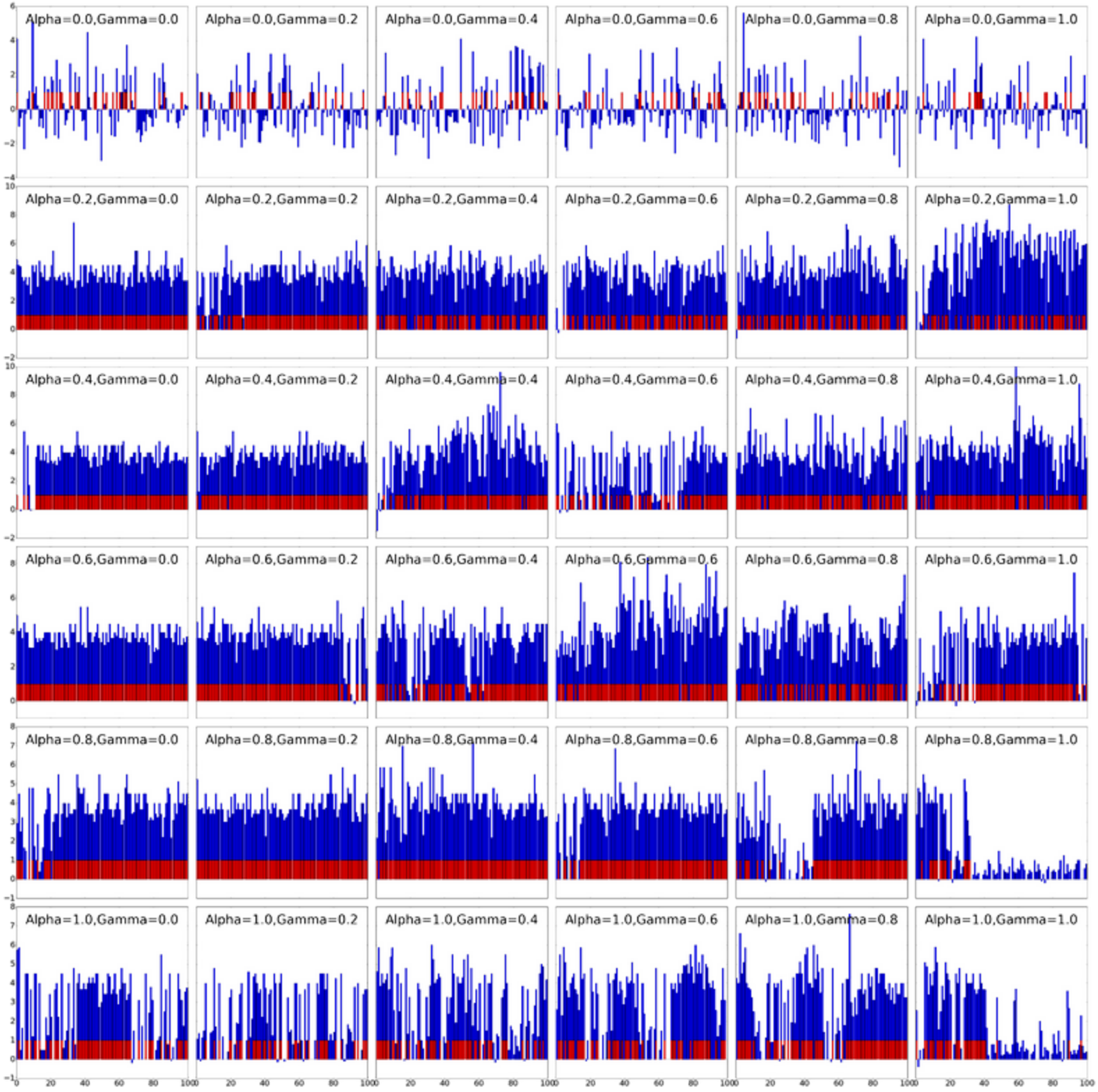
Results of Enhanced Q Learning.
With the state of (light, next_waypoint) and epsilon set to 0.01, I ran the a grid-search over the following Alpha and Gamma settings.
Alpha = [0. , 0.2, 0.4, 0.6, 0.8, 1.]
Gamma = [0. , 0.2, 0.4, 0.6, 0.8, 1.]

In order to evaluate which settings were the most optimal, I looked at two different bar-charts. The first chart is the net-reward normalized by the distance to the goal from the starting position. That is to say, for any given trial, I totaled the net reward earned by the agent and divided it by the L1 distance from the starting position to the destination. I used the environment.compute_dist() method to calculate this. In this way, we can compare net rewards across different trials.

The second bar-chart is much more simple, it is a binary chart of “reached”, either the agent reached the goal or did not. Reached is denoted as 1, so the red bars indicated reaching the goal, while the white space is a failure the reach the goal.

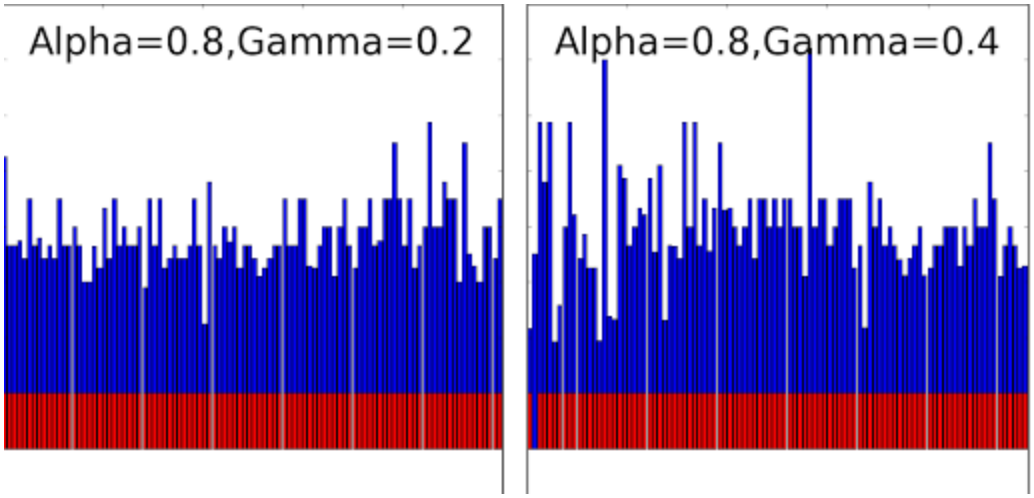
The results for grid search is included below It is also included a jpg, which can be opened separately and viewed while reading this report. The red ticks represent the agent reaching the goal, the blue represents a normalized net reward.

A few things to note about grid. The Y-axis is actually not the same per row, most of them are at a range of 8, although the first row max is 6 and second and third rows have a max of 10. All x-axes are the same, ranging from the first trial 0, to the last 100. Please scroll down to the next page to see the figure.


Alpha=0.0,Gamma=0.0
Alpha=0.0,Gamma=0.2
Alpha=0.0,Gamma=0.4
Alpha=0.0,Gamma=0.6
Alpha=0.0,Gamma=0.8
Alpha=0.0,Gamma=1.0
Alpha=0.2,Gamma=0.0
Alpha=0.2,Gamma=0.2
Alpha=0.2,Gamma=0.4
Alpha=0.2,Gamma=0.6
Alpha=0.2,Gamma=0.8
Alpha=0.2,Gamma=1.0
Alpha=0.4,Gamma=0.0
Alpha=0.4,Gamma=0.2
Alpha=0.4,Gamma=0.4
Alpha=0.4,Gamma=0.6
Alpha=0.4,Gamma=0.8
Alpha=0.4,Gamma=1.0
Alpha=0.6,Gamma=0.0
Alpha=0.6,Gamma=0.2
Alpha=0.6,Gamma=0.4
Alpha=0.6,Gamma=0.6
Alpha=0.6,Gamma=0.8
Alpha=0.6,Gamma=1.0
Alpha=0.8,Gamma=0.0
Alpha=0.8,Gamma=0.2
Alpha=0.8,Gamma=0.4
Alpha=0.8,Gamma=0.6
Alpha=0.8,Gamma=0.8
Alpha=0.8,Gamma=1.0
Alpha=1.0,Gamma=0.0
Alpha=1.0,Gamma=0.2
Alpha=1.0,Gamma=0.4
Alpha=1.0,Gamma=0.6
Alpha=1.0,Gamma=0.8
Alpha=1.0,Gamma=1.0

In order to evaluate the grid-search, consider that the most important variable for consideration is “reached”, thus any graph with a full bar of red, or mostly full ,did exceedingly well. In addition graphs that have full red towards to the 100, are of interest as they indicate that the agent is consistently reaching the destination.

The region of that achieves is alpha [.2,.4,.6]. Within those three rows, I’d say the intersection of Alpha = 0.8 with Gamma = 0.2, 0.4 seem quite promising. We see very high normalized net-rewards and near perfect reach of destination.



It’s worth noting that the agent also performs very well in situations such as alpha = 0.2 and gamma = 0.0. However, I think that this is more likely due to the random nature of the learning, the agent happens into the right steps early on (following the next waypoint) and then the future values really don’t matter at all. I believe if we re-ran the experiment multiple times and averaged, it would not always perform so well.

The optimal policy.

In theory I believe the optimal policy to be one that guides the agent to the destination within the stipulated time, this ensures that the passenger reaches their destination in a timely manner. Additionally the agent should follow all traffic laws in order to ensure the highest level of safety. So the optimal policy is one where the agent reaches the destination without breaking laws. I believe my agent(s) have found this policy, with the above values of alpha = 0.8 and gamma=0.2 / 0.4. I suppose if I’m required to choose one optimal setting I’d have to choose gamma=0.2, since it didn’t miss a single destination (although luck probably played a role).

Finally, I believe a slightly superior policy would be one that trains the agent to take strategic right turns in order to avoid traffic delays etc, however this would probably require optimization of the route-planner and is probably beyond the scope of this project.