Train a Smartcab to Drive

A smartcab is a self-driving car from the not-so-distant future that ferries people from one arbitrary location to another. In this project, you will use reinforcement learning to train a smartcab how to drive.

Environment

Your smartcab operates in an idealized grid-like city, with roads going North-South and East-West. Other vehicles may be present on the roads, but no pedestrians. There is a traffic light at each intersection that can be in one of two states: North-South open or East-West open.

US right-of-way rules apply: On a green light, you can turn left only if there is no oncoming traffic at the intersection coming straight. On a red light, you can turn right if there is no oncoming traffic turning left or traffic from the left going straight.
To understand how to correctly yield to oncoming traffic when turning left, you may refer to this **official drivers' education video**, or this **passionate exposition**.

Inputs

Assume that a higher-level planner assigns a route to the smartcab, splitting it into waypoints at each intersection. And time in this world is quantized. At any instant, the smartcab is at some intersection. Therefore, the next waypoint is always either one block straight ahead, one block left, one block right, one block back or exactly there (reached the destination).

The smartcab only has an egocentric view of the intersection it is currently at (sorry, no accurate GPS, no global location). It is able to sense whether the traffic light is green for its direction of movement (heading), and whether there is a car at the intersection on each of the incoming roadways (and which direction they are trying to go).

In addition to this, each trip has an associated timer that counts down every time step. If the timer is at 0 and the destination has not been reached, the trip is over, and a new one may start.

Outputs

At any instant, the smartcab can either stay put at the current intersection, move one block forward, one block left, or one block right (no backward movement).

Rewards

The smartcab gets a reward for each successfully completed trip. A trip is considered "successfully completed" if the passenger is dropped off at the desired destination (some intersection) within a pre-specified time bound (computed with a route plan).

It also gets a smaller reward for each correct move executed at an intersection. It gets a small penalty for an incorrect move, and a larger penalty for violating traffic rules and/or causing an accident.

Goal

Design the AI driving agent for the smartcab. It should receive the above-mentioned inputs at each time step t, and generate an output move. Based on the rewards and penalties it gets, the agent should learn an optimal policy for driving on city roads, obeying traffic rules correctly, and trying to reach the destination within a goal time.

Tasks

Implement a basic driving agent

Implement the basic driving agent, which processes the following inputs at each time step:

- Next waypoint location, relative to its current location and heading,
- Intersection state (traffic light and presence of cars), and,
- Current deadline value (time steps remaining),

And produces some random move/action (None, 'forward', 'left', 'right'). Don't try to implement the correct strategy! That's exactly what your agent is supposed to learn.
Run this agent within the simulation environment with enforce_deadline set to False (see run function in agent.py), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.

In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

The random agent is eventually reaching the destination with variables at play (other vehicles, oncoming traffic, traffic lights, etc) but the approach to reach the goal is clearly random and not optimal. In fact the agent is not optimizing the rewards gain. The agent does not prefer actions that it has tried in the past and found to be effective in producing reward. In fact the agent is constantly in "discovery" mode where he try new action without any knowledge of the past rewarding (or not) actions.
The agent does not exploit what it has already discover in order to obtain reward. For example the agent does not understand, run after run, that running a red-light produce bad reward.
We noticed that several times, the agent, being close to the goal choose a different direction than the preferred to reach the destination.
We can also remark the fact that the agent sometimes remain in position (action = `None`) even when there is no oncoming traffic or red light.

Identify and update state

Identify a set of states that you think are appropriate for modeling the driving agent. The main source of state variables are current inputs, but not all of them may be worth representing. Also, you can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state.

At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

Justify why you picked these set of states, and how they model the agent and its environment.

The goal of the agent is to reach the destination before a limited amount of time runs out while maximizing its collection of rewards. Based on the reward structure, we can easily find out 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 exponential 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 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.

We have many options base the state update. For example we can use the `next_waypoint` from the planner or traffic light or even the traffic data of potential cars coming on the left or right. The deadline is also another factor that can be used to update the state.

For this model we choose to use the data available at each intersection. This include the traffic lights (red / green), the oncoming traffic (left, right or None) from 2 other cars (input[1] = oncoming, input[3] = left). Due to the circulation rules, the traffic coming from the right does not matter and will therefore be ignore from the states.

We also decided to use the result of `next_waypoint` to defined the target to reach the destination. The agent can use this information to act properly to reach the destination in an efficient manner.

Basic Q-Learning Implementation

In this section we explored what a very basic Qlearning algorithm does for our agent. In the basic approach, I implemented the following Qlearning 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 simple 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 Qvalue, 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 netrewards 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 Qtable will then update states based on those parameters, both of which are directly.

Enhanced Q Learning

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

Q(State, Action) = (1Alpha) * 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.

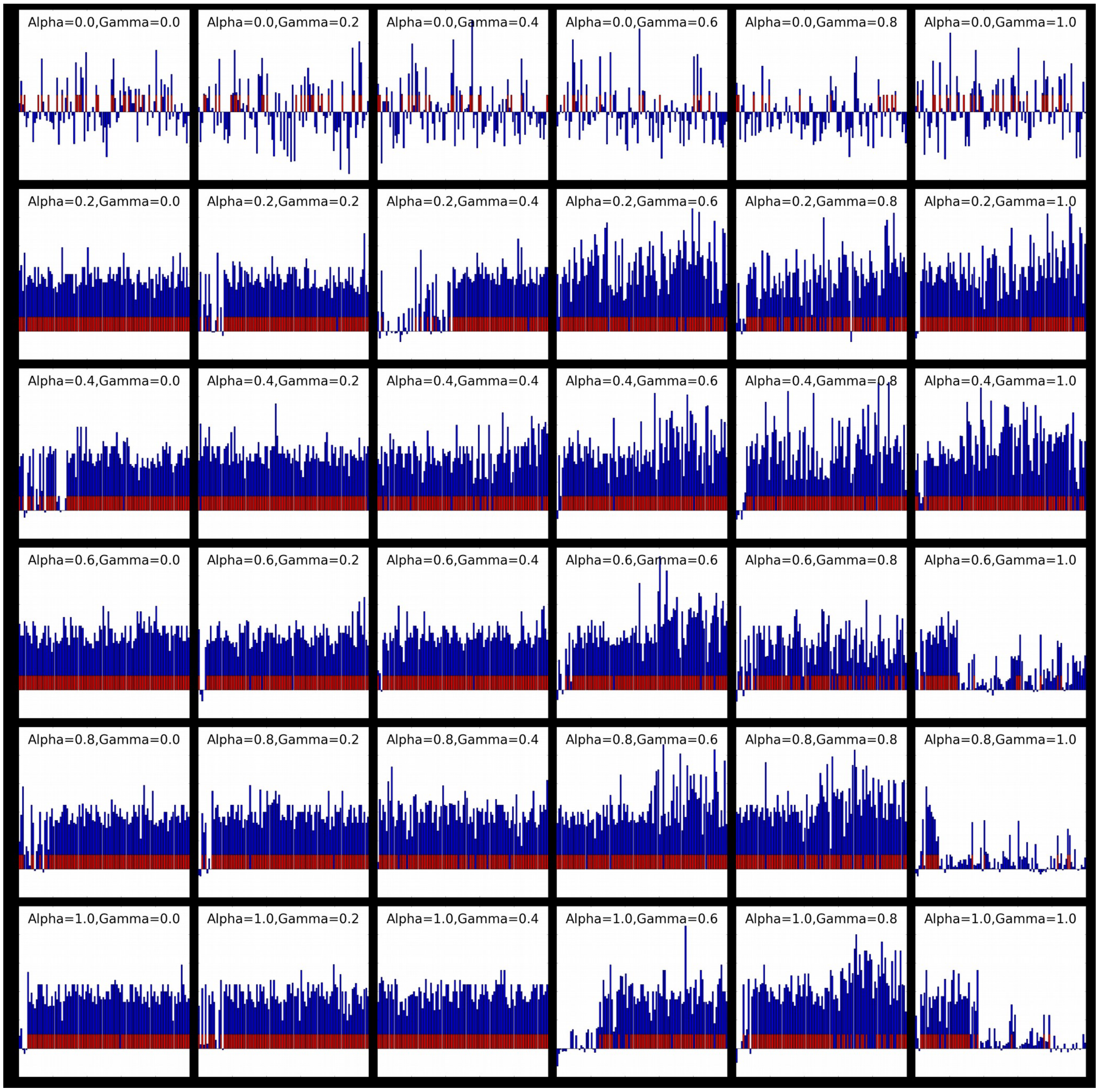
Results of Enhanced Q Learning.
With the state of (light, next_waypoint) and epsilon set to 0.01, I ran the a gridsearch over the following Alpha and Gamma settings.
Alpha = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]
Gamma = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]

In order to evaluate which settings were the most optimal, I looked at two different barcharts. The first chart is the netreward 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 barchart 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. The red ticks represent the agent reaching the goal, the blue represents a normalized net reward.

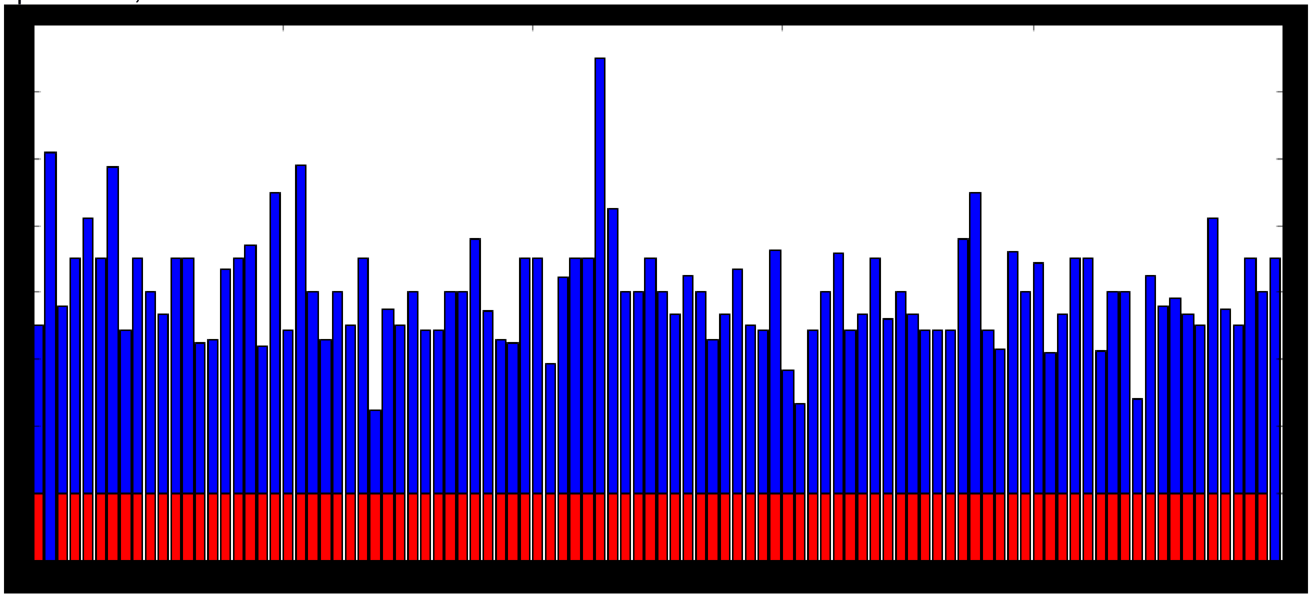
A few things to note about grid. The Yaxis is actually not the same per row, most of them are at a range of 8, although the first row max is 6 and second and third rows have a max of 10. All xaxes are the same, ranging from the first trial 0, to the last 100.


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

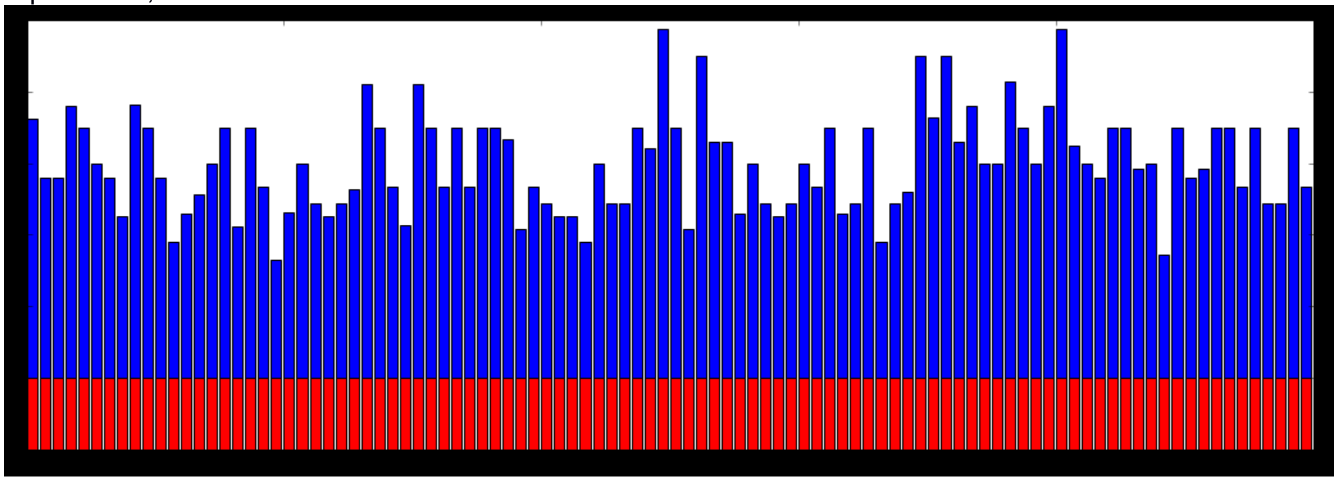
In order to evaluate the gridsearch, consider that the most important variable for consideration is “reached”, thus any graph with a full bar of red, or mostly full ,did exceedingly well. In addition graphs that have full red towards to the 100, are of interest as they indicate that the agent is consistently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

Based on the graph above, I’d say Alpha = 0.4 with Gamma = 0.2 and Alpha = 0.6 with Gamma = 0.0 seem quite promising. We see very high normalized netrewards and near perfect reach of destination.

Alpha = 0.4, Gamma = 0.2



Alpha = 0.6, Gamma = 0.0



It’s worth noting that the agent also performs very well in situations such as alpha = 0.2 and gamma = 0.0. There are other situation also where agent did very well. 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 reran 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 = 0.6 and gamma=0.0 because it didn't miss a single destination. 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 routeplanner and I believe that is beyond the scope of this project.