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?

When I ran the agent with these settings, the agent has no concern about the route suggested by the planner and or where it needs to go. It just wanders around randomly. So we assume that all the grid points are likely to be visited by the agent at some point or the other, then eventually the agent will reach the destination. Infact the chance that agent will not reach destination in the limit is zero. So in other words the agent will always find the goal if it is allowed to run for long enough time.

The informal mathematical proof can go along the lines….

Suppose the grid has total of "n" grid points. Also since the action is random, the probability of agent being at any grid at some time tick say "t" after a long run is equally likely and it is equal to 1/n

And this is true for all the ticks there on.

So if a game is allowed to run enough number of times, the probability of finding the agent at the destination will be

1/n + 1/n + …. So on

Which in the limit will become 1 – i.e. the agent will surely reach home

A better proof could be modeled using "random walk in 2 d" which would state that after t ticks, the agent is likely to be found sqrt(t) distance away from starting point. And as (t -> infinity), the possible locations where the agent could go would cover the entire grid.

To test this, I ran the simulation with "enforce deadline = False". I also modified the code to observe the number of steps/ticks the agent takes to reach destination. I ran the simulation for 100 trials. The observations are as under:

1) The agent reached its destination in all the 100 cases
2) The number of steps it took varied widely but it seem to hover around 100-120 steps. A more detailed modelling would be required to find out the relationship between the original starting distance between start and destination and the average number of ticks required to cover this journey.

Q2 - 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.

In my work on this project, this was the trickiest part. I spent maximum time in trying various combinations of how to define a state.

First I started with storing all the values collected i.e. next-waypoint, all inputs (lights and traffic condition) as well as deadline. My original idea was to store as much information as possible in "state" so that learner is able to learn all the possible combinations. However, soon I realized that adding deadline will lead to no learning. Deadline decreases each time step , so no state will be repeated in a single run of simulation.

Second attempt was to then model only using inputs and next_waypoint. Next_waypoint coupled with actual action taken by agent would help in learning that following next_waypoint is a good thing to do. Using inputs in state will help in learning that following traffic rules is right behavior. From there on I played around with these two basic concepts to define the states. In its raw form, state space size was

3(waypoint)*2(light)*4^3 (for oncoming, left and right traffic) = 384 which when coupled with 4 possible actions make the (s,a) space to be 1536 values

NOTE: Reviewer 1 commented "There is a miscalculation in your computation of the number of state spaces--on page 2 you calculate using 4 possible values for the inputs 'oncoming', 'left', and 'right'; however these are binary variables so there are only two possible values for them. That makes 48 states rather than 384. The rest of the analysis still holds, but it does make changes beyond this less urgent."

MY RESPONSE: However, if you see the code "sense" in "environment.py", the inputs 'oncoming', 'left', and 'right' can have 4 values based on the direction the vehicle from that side is headed towards. To check this, I also ran the code by increasing the number of cars to 20 (self.num_dummies = 20 in environment.py) and got following values

inputs = {'light': 'green', 'oncoming': 'left', 'right': None, 'left': 'left'}
inputs = {'light': 'red', 'oncoming': 'forward', 'right': None, 'left': None}
inputs = {'light': 'red', 'oncoming': None, 'right': 'forward', 'left': None}
inputs = {'light': 'green', 'oncoming': None, 'right': 'left', 'left': None}
inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': 'forward'}
inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': 'right'}

'Oncoming' and 'left' does have 4 values; 'right' also have 3 values - possibly 4 if I explore more.

I am trying to understand how does 'Reviewer 1" see each of these three variables as 'binary'? Could you please help me understand that.

While trying various combinations of state definition, I was also playing around with various values of gamma (discount), alpha (learning rate) and epsilon(exploration). So eventually the definition of state representation emerged after lot of trial and error and it was not something that was fixed apriori before starting the modelling exercise.

Soon I realized that I need to bring down number of states (or (s,a) possible values) to a more manageable level there by ensuring a more general and accelerated learning. I realized that inputs['right'] has no role to play since all the rules about "right of way" do not deal with this input. So next step was to remove this from state space which took the size from 384 to 96.

In my first submission as I had tried to create couple of state variables to represent the traffic rule. Based on the feedback I dropped them.

After dropping input['right'], with some more experimentation, I also dropped input['left'] and input['oncoming']. The idea was to make the agent learn traffic rules based on the learning environment presented to it – be it a left-side traffic or right-side traffic environment or any other types of "right of way" rules. The learning agent should be able to learn effectively from different types of environments.

So eventually, my state consisted of only two variables:

a. The suggestion from planner as to where should I be headed (next_wayPoint) – 3 possible values
a. Inputs['light'] – just the traffic light with two possible values.

So my state space reduced to 6 options and when clubbed with actions (4 possible values), the total (s,a) space had 24 values.

Q 3- Implement Agent behavior with each action being the best action based on currently learnt Q-value

Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that.

What changes do you notice in the agent's behavior?

Right from beginning, to test the agent's learning, I also coded a copy of LearningAgent into another python class called "FinalAgent". This was to test the agent with no learning and no exploration, in order to check the performance – kind of a "TEST SET"

When I made epsilon to zero alongwith original qvalue of unexplored states also equal to 10 i.e. the maximum reward of reaching destination, the number of (state, action) pairs (i.e. Q-value pairs) explored stayed close to 24 which was the full space. It made me realise that another way to force exploration is to have Qvalues of unexplored states equal to the maximum reward. To try this further, while keeping epsilon zero, I made the Qvalue of unexplored states equal to zero. Under this scenario, the explored (Q,a) space dropped to 50% of the total space.

Compared to the "random action", the agent reached the destination within deadline many a times and many a times it got stuck in local maxima.

This exercise should me the need to do exploration early and slowly reduce it and shift to exploitation strategy more as the learning progressed. It also showed me the ability to force exploration by setting initial Qvalues to some high value, higher than the maximum possible reward. With such a setup, the best action initially will be forced to choose from unexplored states.

Q4 - Enhance the driving agent

Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive.

Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?

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?

The changes I made to the basic Q-learning was following:

a) Added a discount factor: I played around a lot with discount factor. Making it close to 1 would mean rewards now vs rewards later have no difference. Conceptually this looked ok to me as the goal of agent is to reach destination within deadline. Soon I realized that a discount factor will encourage the agent to reach destination earlier to get the big reward of reaching there faster. After many iterations, I settled for a value of 0.30. This discount forced the agent to hurry back home

b) Learning Rate: For learning rate, I have used a decaying rate. Initially I tried 1/t where t is a running number starting from time zero of first simulation run and does not get reset to zero for subsequent simulation runs. I also tried different combinations and then finally settled for x/(x+t) where x is some number. With some experimentation, I chose the following strategy to adjust learning rate

$\alpha = \frac{100}{100+t}$ where t is the tick from start of simulation and does not get reset to zero between two consecutive runs.

I used the above value of 100 to decay the learning at a rate slower then 1/t

c) Epsilon: I used the similar strategy as above to decay epsilon. However, for a given run I kept it constant. This was to have a fixed epsilon for an individual run and to decay it even slower than alpha. The exploration probability drops from 50% for first run to about 1% by 100th run. I achieved this by using the following formula

$\varepsilon = \frac{1}{1+run\_no}$

As noted above, exploration can also be encouraged by setting initial Q-values to higher than then maximum reward.

The agent trained with this, when run, does reach the goal majority of times. The state explored during training varies from 90-100% of total (state, action) space.