Task -1

<u>Implement a basic driving agent</u>

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.

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

Ans:- I choose state in the format of (('light',inputs['light']),('oncoming',inputs['oncoming']),('right',inputs['right']),('left',inputs['left']),self.next_waypoint)

In this agent choose any random action out of these four, so sometimes randomly it reaches to the target location

<u>Identify and update state</u>

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.

Ans:-First choose state in the format of (('light',inputs['light']),('oncoming',inputs['oncoming']),('right',inputs['right']),('left',inputs['left']),self.next_waypoint,deadline)

So, initially I was taking deadline as one of the parameter in my state, which is increasing my no of states to many folds as it can take any big value

Then I removed deadline and started modelling based upon new states with only inputs and next waypoint

Self.state=

(('light',inputs['light']),('oncoming',inputs['oncoming']),('right',inputs['right']),('left',inputs['left']),self.next_waypoint)

Task-2

<u>**Implement Q-Learning**</u>

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that.

Each action generates a corresponding numeric reward or penalty (which may be zero). Your agent should take this into account when updating Q-values. Run it again, and observe the behavior.

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

Ans:- I decided to choose a dictionary for my table which holds the value of (state,action) as key.

So initially as all value of my Q-table are zero it chooses action randomly from a list of all possible value and as my Q-Table fills it chooses the best possible action.

Sometimes it gets good accuracy in trials but other times it tends to stuck at local minima

<u>Enhance the driving agent</u>

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?

Ans:-

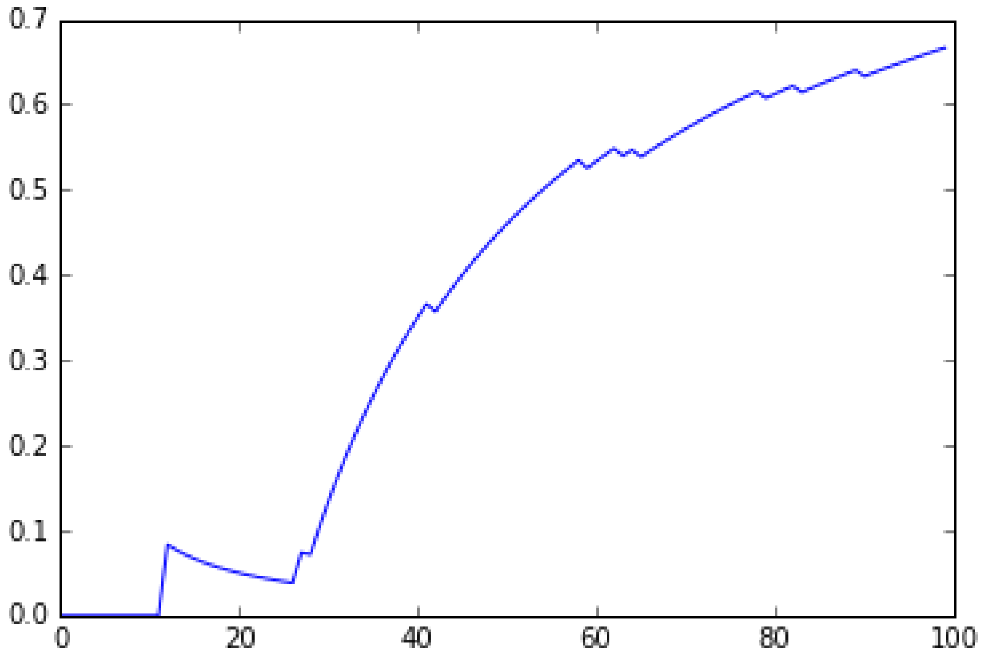
In order to optimize I used E-Greedy exploration and decaying learning rate (1/t)

in E-greedy exploration I choose a random action by epsilon probability and best action with

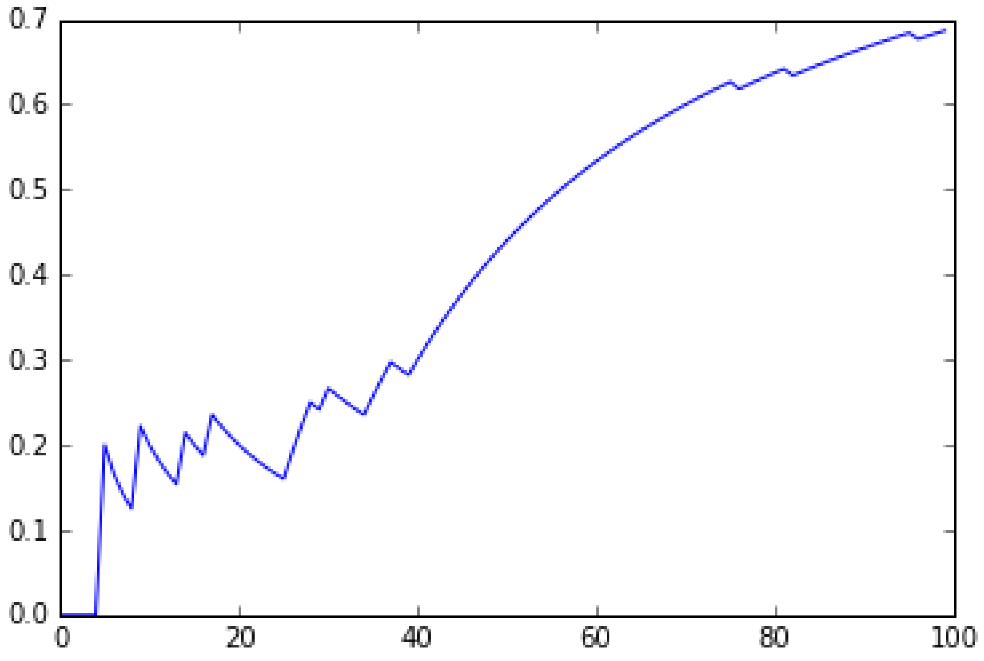
(1-epsilon) probability. And after some learning it starts to reach destination almost every time

I choose learning rate 1/t so, that It always converges.

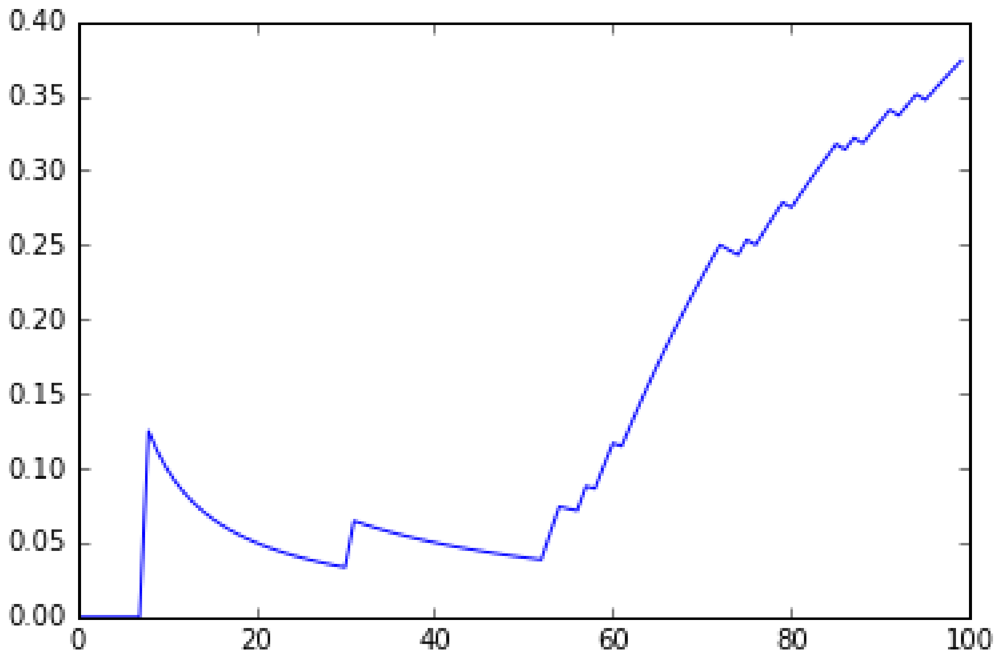
Some Success Average Graph for different value of gamma and alpha:



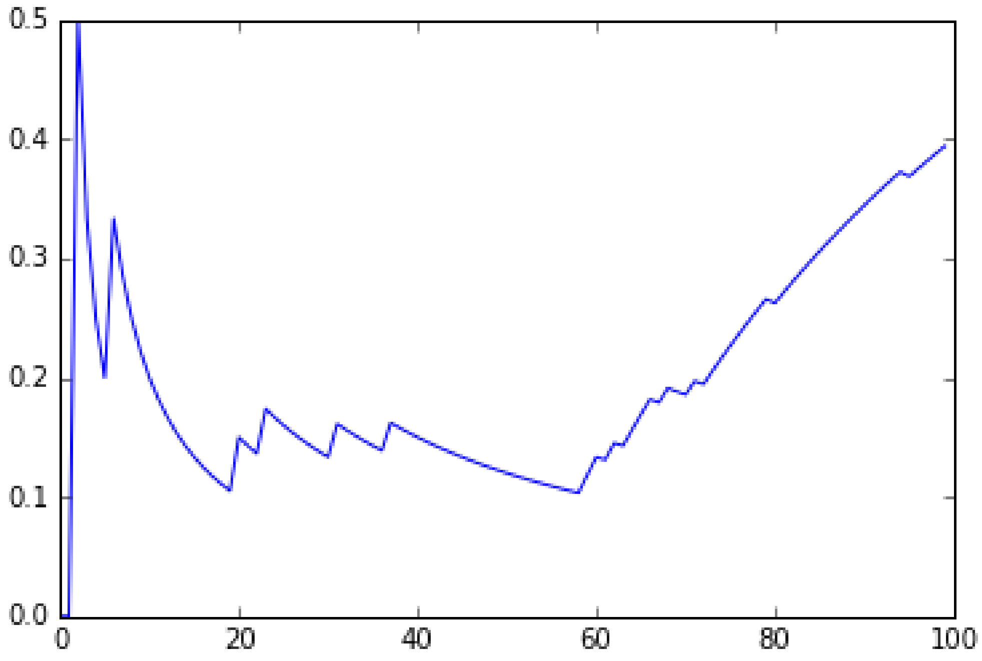
Gamma =0.4 , Alpha=1/t



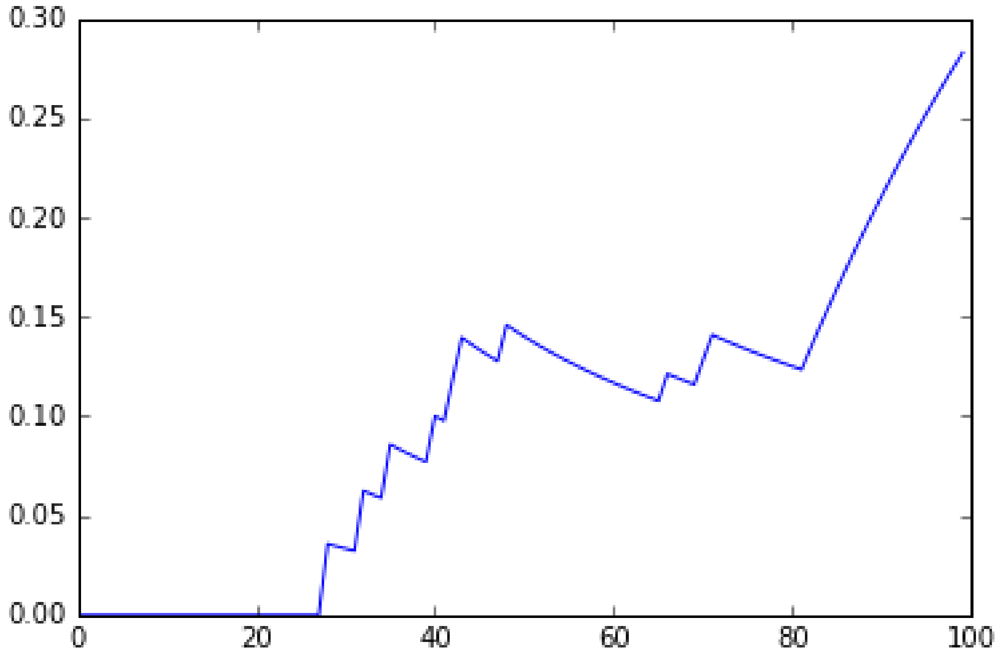
Gamma=0.9 , Alpha =1/t



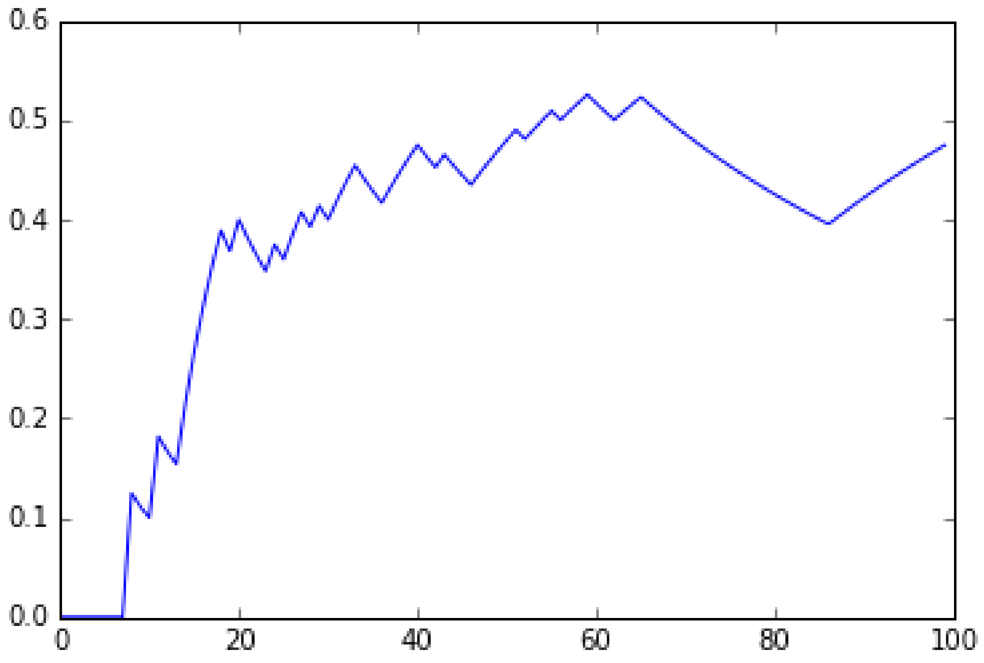
Gamma =0.4



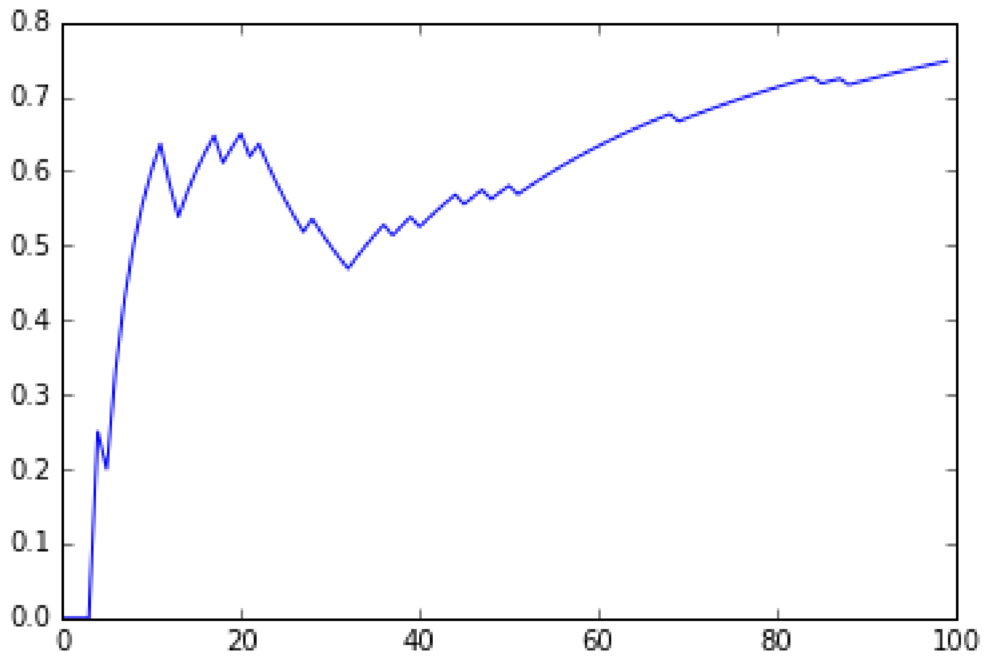
Gamma =0.9t , Alpha= 1/2t



Gamma =0.4t, alpha =1/2t



Gamma =0.9, alpha = 1/ (1+t)



Gamma =0.4, alpha =1/(1+t)

Using alpha as 1/2t from the graph above we can see learning starts late as learning rate is made half, while using alpha as 1/ (1+t) there is not much difference while using alpha 1/t.

Optimal policy:-

In my testing I was able to reach destination before deadline almost everytime in the last 10 trials of total 100 trials,

On 90th trial I saw one traffic light skip.

So I think my policy is really close to the optimal policy of reaching destination before deadline without breaking any traffic rules.