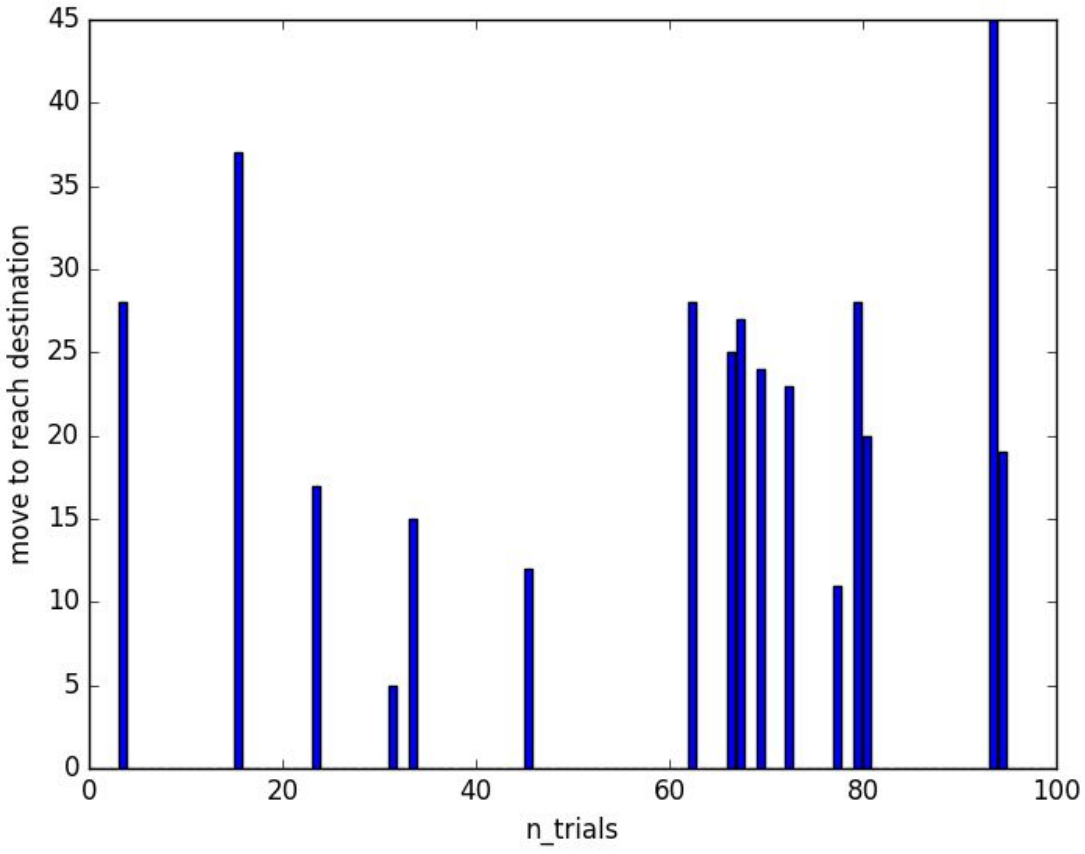
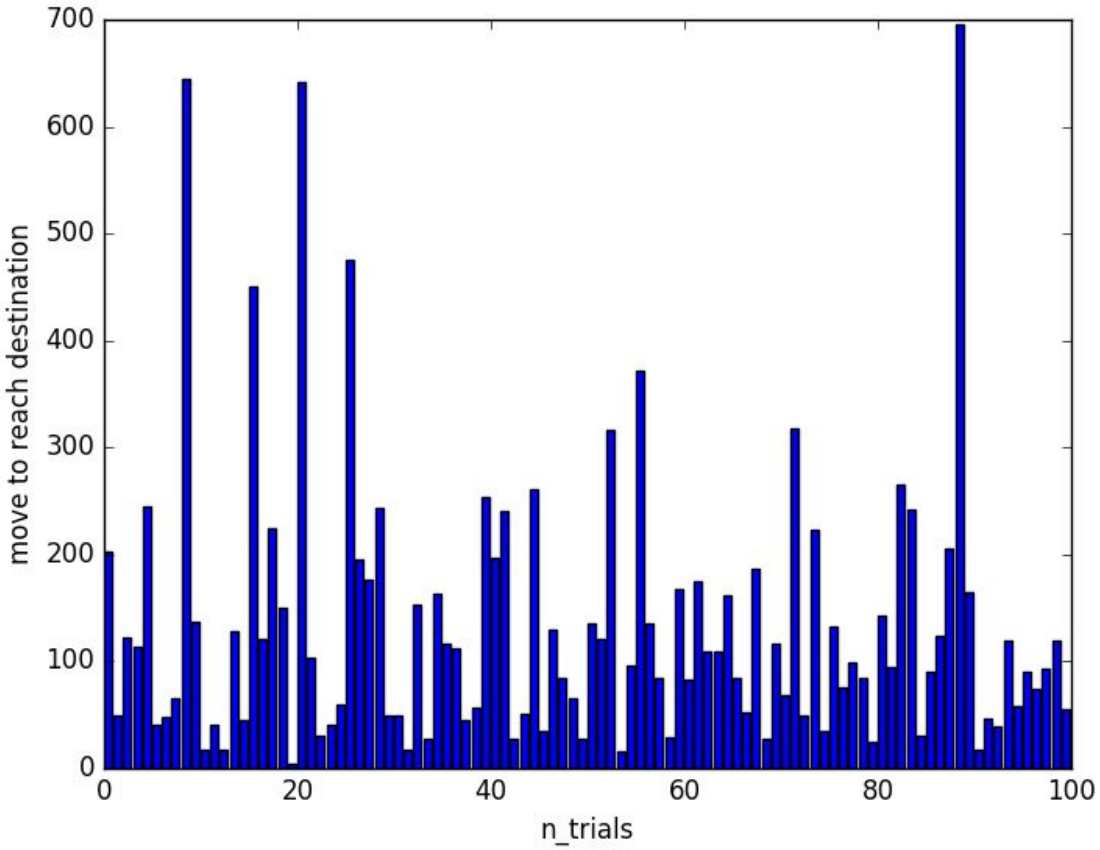
Implement a basic driving agent

There are four possible actions (None, Forward, Left, Right) available for the agent to take. In the first stage, we have made the agent to choose action randomly. As a result, it moves arbitrarily in the grid without having any knowledge about how to reach the target location. For this reason, the agent fails to reach the target destination most of the time. Out of *100* trials, setting the enforce_deadline parameter true, it has reached the destination *16* times. As we can say, the success rate is *16%*. We will use this rate for benchmarking in next sections. A figure related to this situation is given below. Here the height of a bar denotes the number of moves needed to reach the destination. A zero move means that the agent is unable to reach the target within given deadline.



On the contrary, if the enforce_deadline is set to False, the agent can reach all target location, though it takes a lot of time. A figure depicting this scenario is given below.



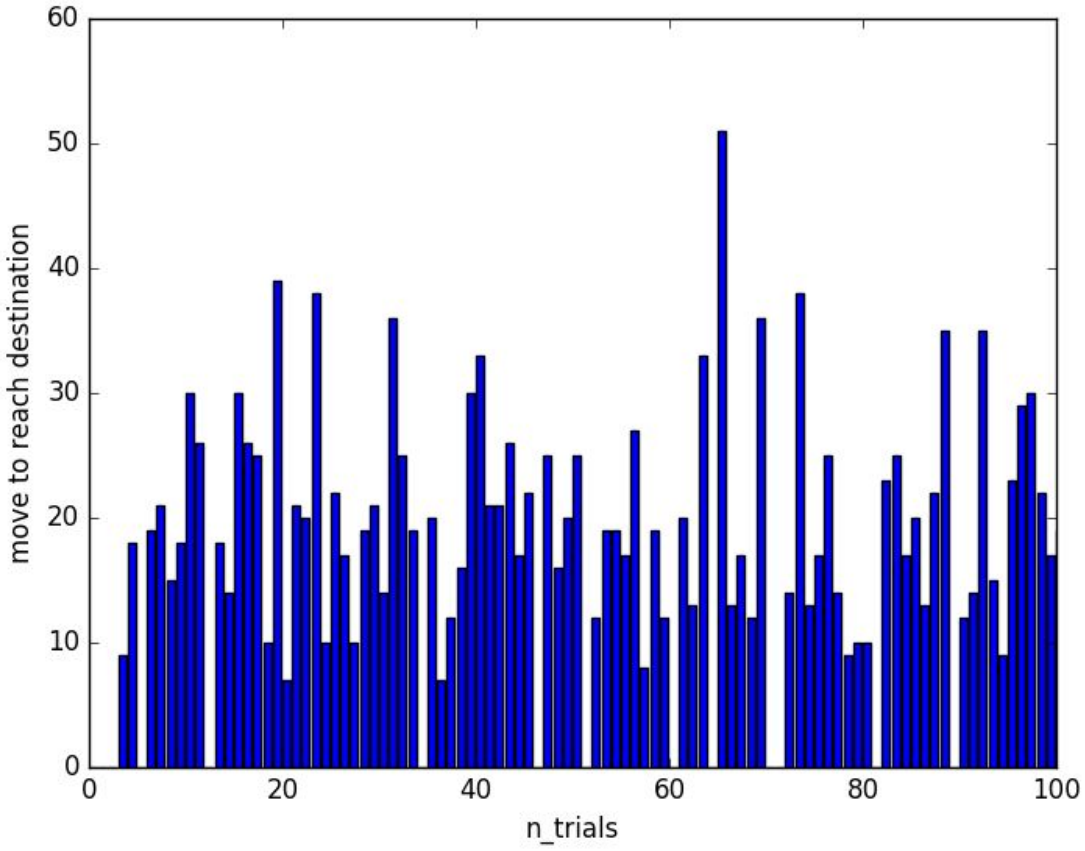
Identify and update states

At any given time, we can describe the next move of agent using some combination of parameters. These are *light* and *oncoming*. *Light* and *oncoming* together define the current scenario of intersection. *Light* has two values (red, green) and *oncoming* has four (none, forward, left, right). Using the combination of various values of *light* and *oncoming*, agent can decide whether to stall or go left, right or forward. For this reason, an agent state is defined using *light* and *oncoming*. *Waypoint* is added in the state to use it in Q table. Because the reward is based on the *waypoint* of

the agent which is chosen based on *light* and *oncoming* value. *Waypoint* has four different values (none, forward, left and right). As a result, we need to store 32(=2x4x4) states in Q table.

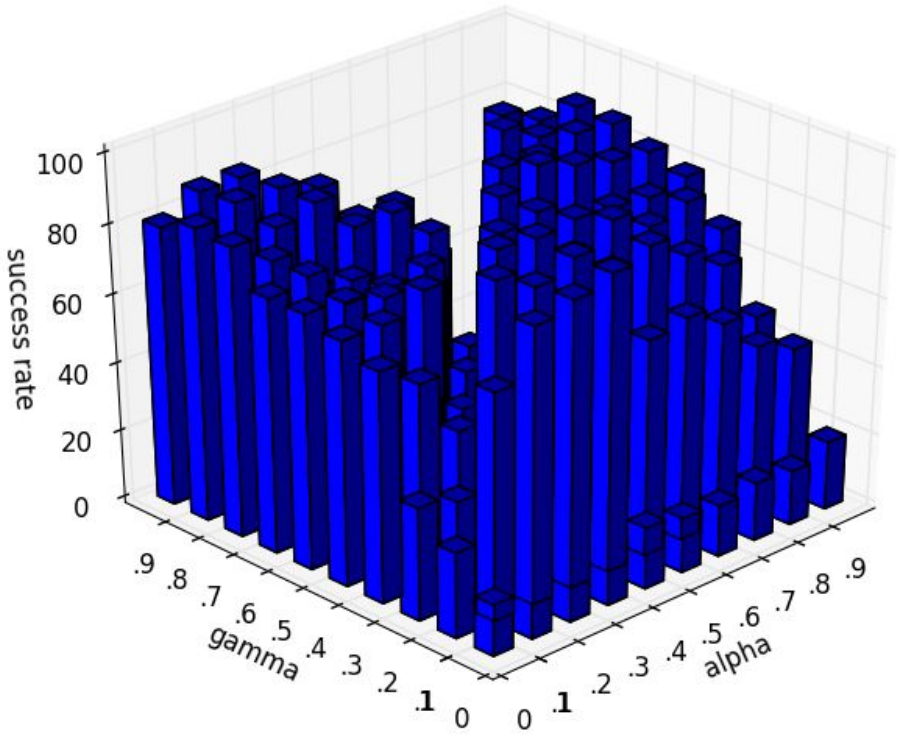
Implement Q Learnings

I have used python *Counter* class as Q table. It is used like a map. By default, every non inserted value is mapped a zero. When a new state is inserted in the table, a corresponding value has been assigned to that. The initial value of alpha and gamma is set to *0.5* and *0.2* respectively. For this setting, the agent reaches the target *88* times out of *100* trials. That means, *88%* success rate. Please check the following figure.



Lets check the agent's behaviour using different values of *alpha* and *gamma*. I have written a code which runs agent.py using various values of alpha and gamma ranging from *0* to *1*. For each pair of

alpha and gamma I have calculated the success rate of the agent. The resultant graph is given below.



Enhance the driving agent

After many trial and error calculation and using the above figure we have found that *alpha = 0.9* and *gamma = 0.4* can give us success rate around *90%* on average. I have only tweaked these two parameters to find the best success rate.

By looking at the above figures, we can say that the agent can reach the target using fewer number of trials and number of trails reduces over time. Besides the cumulative rewards get larger with trials. That means, the agent starts making better decision over time.