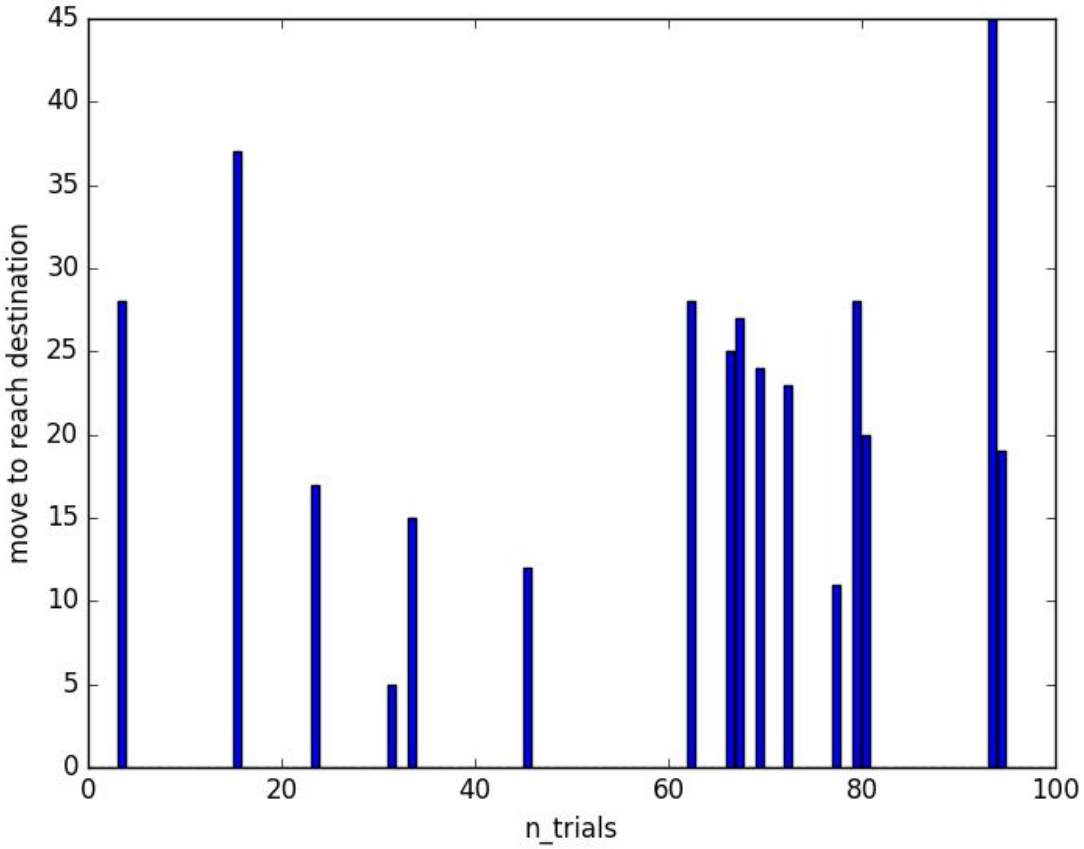
Implement a basic driving agent

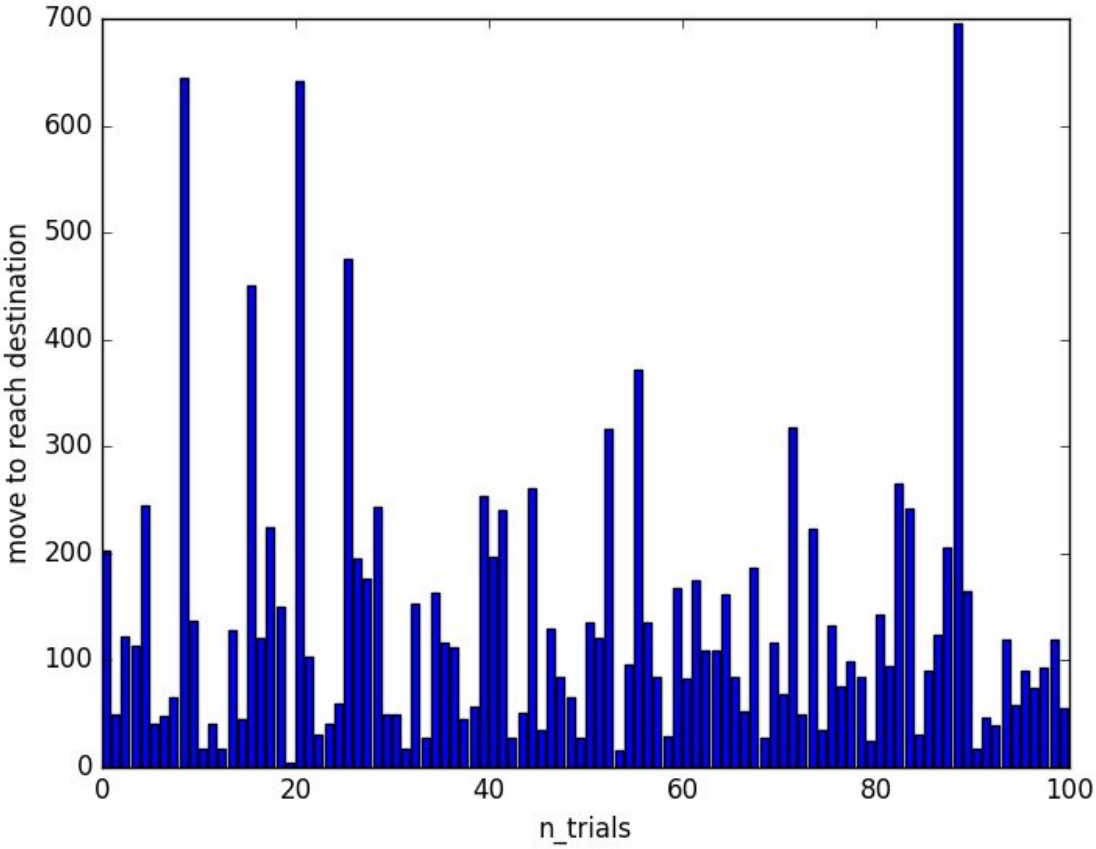
There are four possible actions (None, Forward, Left, Right) available for the agent to take. In the first stage, we have made the agent to choose action randomly. Out of four possible actions, the agent is taking any one of those randomly at a time without considering the consequence. As a result, it 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

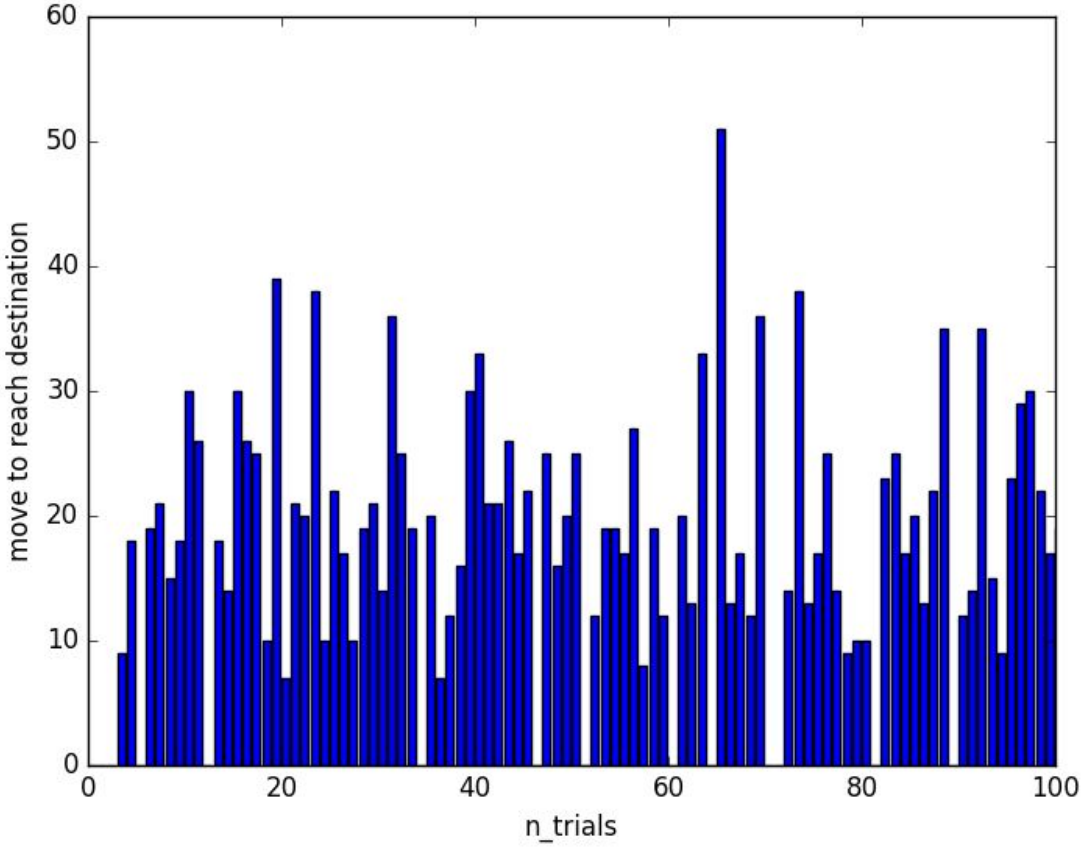
At any given time, we can describe the next move of agent using some combination of parameters. These are *light* and *oncoming*. *Light* and *oncoming* together define the current scenario of intersection. *Light* has two values (red, green) and *oncoming* has four (none, forward, left, right). Using 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.

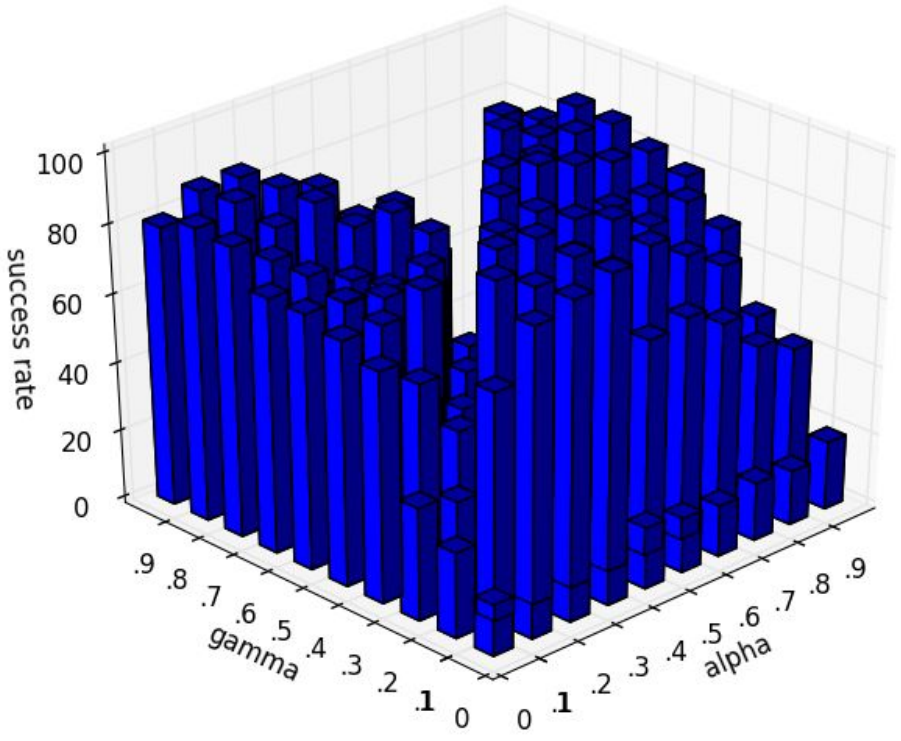
There are three other parameters which we can consider to create a state. These are *right*(traffic from right), *left*(traffic from left) and *deadline*(allotted time to reach destination). We have added the *left* as an item in state to properly simulate the agent behavior after watching this intersection rule video (Traffic coming from left is necessary to take waypoint decision). Incorporating this parameter in state has increased the performance by *4%-6%* on average. Based on the video described earlier, in US, if the *light* is green, traffic coming from right doesn't matter. That's why we have decided to not include this in state space, because the *light* parameter can properly infer this scenario. Deadline parameter is not important for our purpose, because reaching target in time doesn't gives any extra reward. As the deadline is described as *deadline = self.compute_dist(start, destination) * 5*, including this in state will create *2080(=32*13*5)* (*13* is the maximum possible distance in the grid) states in Q table which will make the learning slower. That's why we have decided to not include this in state space.

Implement Q Learnings

We 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*. We have written a code (*runner.py*) which runs agent.py using various values of *alpha* and *gamma* ranging from *0* to *1*. For each pair of *alpha* and *gamma* we have calculated the success rate of the agent. The resultant graph is given below.



After implementing Q Learnings the agent shows tendency to reach target location within specified timeframe. In general, now the agent can move towards the target having knowledge about the surrounding environment. This knowledge helps it to make right choice during various state (*light*, *oncoming*, *left* and *waypoint*). By using the Q Learning, the agent now can decide the best move to reach the target which in turns helps it to avoid collision and reduce random move within the grid.

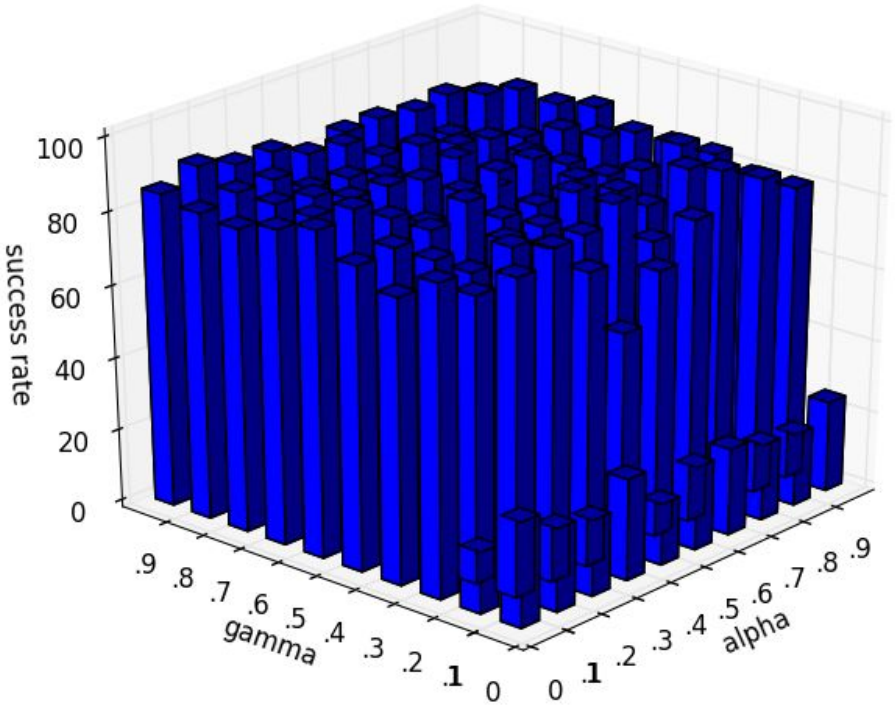
Enhance the driving agent

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. we 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.

Optimal policy

The optimal policy of the agent is to reach the target destination using shortest distance/move without breaking traffic law (without colliding with other cars, without making wrong move). So in optimal policy an agent should reach the target destination in time without breaking traffic laws.



By checking the above grid search graph, we can say that the agent's performance is going towards optimality. Now the performance is *95% (alpha=0.7, gamma=0.9)* accurate. However, this is the result of adding *left* parameter as a state. The agent now takes less time to reach the target

destination, that's why it's performance has been increased dramatically. As the *epsilon* is set to *0.4,* agent's move is not accurate *100%* of times. To achieve true optimality, we can introduce different reward structure to reward optimal shortest distance. In current scenario, the agent may move to and fro to collect some extra reward and reach the target in time.