Training a smart cab to drive

From Udacity Machine Learning Nanodegree

By Tasmiah Tahsin Mayeesha

Definitions

Environment

The smartcab operates in an ideal, grid-like city (similar to New York City), with roads going in the North-South and East-West directions. Other vehicles will certainly be present on the road, but there will be no pedestrians to be concerned with. At each intersection there is a traffic light that either allows traffic in the North-South direction or the East-West direction. U.S. Right-of-Way rules apply:

- On a green light, a left turn is permitted if there is no oncoming traffic making a right turn or coming straight through the intersection.
- On a red light, a right turn is permitted if no oncoming traffic is approaching from your left through the intersection.

Inputs and Outputs

Assume that the smartcab is assigned a route plan based on the passengers' starting location and destination. The route is split at each intersection into waypoints, and you may assume that thesmartcab, at any instant, is at some intersection in the world. Therefore, the next waypoint to the destination, assuming the destination has not already been reached, is one intersection away in one direction (North, South, East, or West). The smartcab has only an egocentric view of the intersection it is at: It can determine the state of the traffic light for its direction of movement, and whether there is a vehicle at the intersection for each of the oncoming directions. For each action, the smartcab may either idle at the intersection, or drive to the next intersection to the left, right, or ahead of it. Finally, each trip has a time to reach the destination which decreases for each action taken (the passengers want to get there quickly). If the allotted time becomes zero before reaching the destination, the trip has failed.

Rewards and Goal

The smartcab receives a reward for each successfully completed trip, and also receives a smaller reward for each action it executes successfully that obeys traffic rules. The smartcab receives a small penalty for any incorrect action, and a larger penalty for any action that violates traffic rules or causes an accident with another vehicle. Based on the rewards and penalties the smartcab receives, the self-driving agent implementation should learn an optimal policy for driving on the city roads while obeying traffic rules, avoiding accidents, and reaching passengers' destinations in the allotted time.

Implement a basic driving agent :

QUESTION: Observe what you see with the agent's behavior as it takes random actions. Does the smartcab eventually make it to the destination? Are there any other interesting observations to note?

The task for this part of the project is to get the smartcab to move around in the environment without using any optimal driving policy.

The driving agent is given the following information at each intersection:

- The next waypoint location relative to its current location and heading.
- The state of the traffic light at the intersection and the presence of oncoming vehicles from other directions.
- The current time left from the allotted deadline.

If I run the simulation without any modification the chosen agent(red car) does not move at all regardless of the input or how the other agents are interacting in the environment.

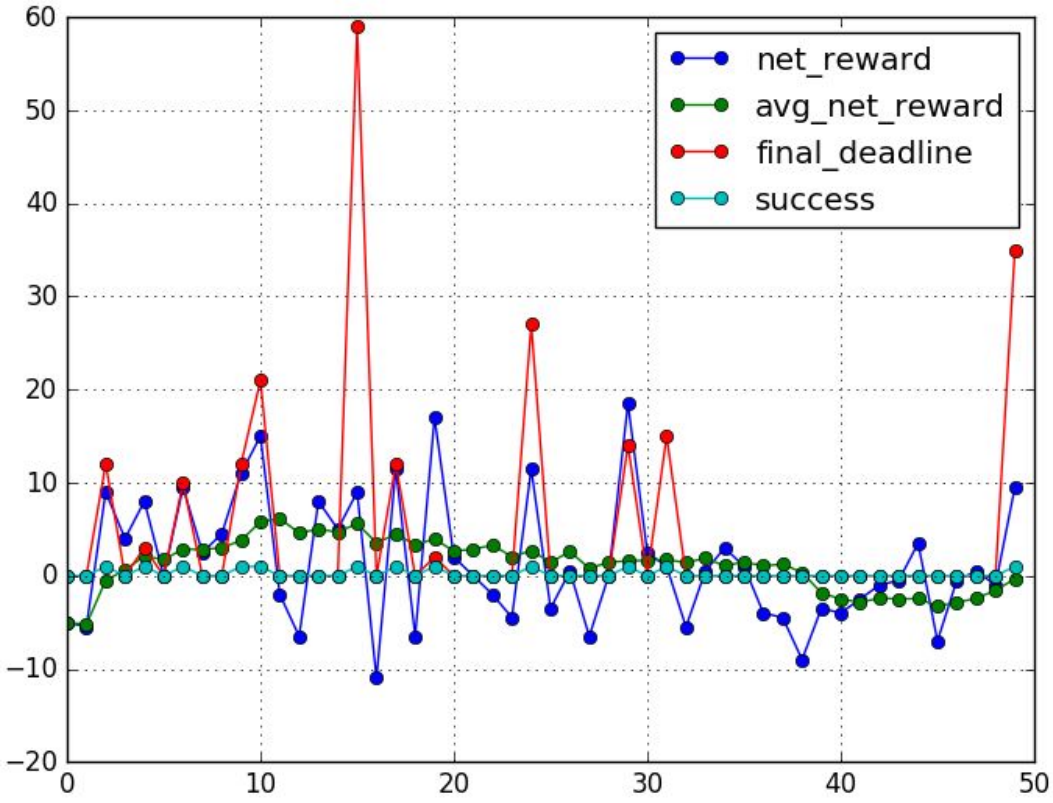
After modifying the action policy to choose a random action from the set of available actions (None,"forward","left","right") the agent started moving in the environment by choosing to go left, right, forward or choosing to not move at all.

But randomly choosing policies to move in some direction rarely took the agent to it's destination. Instead of that, it managed to collide with the other agents(vehicles) and attempted to violate traffic rules, thus incurring large penalties/negative rewards.

Mostly the agent simply moved around in the grid world even if the deadline was not enforced, often choosing to move in the opposite direction even if the destination was just one waypoint ahead of it given it was not given any goal to go to the destination.

To see it visually, here's a graph that shows the net reward, avg_net_reward, deadline and where a trial was successful or not over 50 trials. (100 trials take quite a long time in my machine but the result is likely to be similar). As we can see, choosing actions randomly rarely, if ever is successful. Out of 50 trials, only 12 trials were successful with almost all of the trials in the end as failures. Here the total net reward was only 1.44 after 50 trials with an average net

reward of 1.23



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

OPTIONAL: How many states in total exist for the smartcab in this environment? Does this number seem reasonable given that the goal of Q-Learning is to learn and make informed decisions about each state? Why or why not?

In the Q-learning implementation , following parameters were initially used for representing a state :

- `Light : Red/Green`
- `Oncoming : None/Left/Right/Forward`
- `Waypoint : None/Left/Right/Forward`

State variable “deadline”(deadline = 5*(destinations index - sources index) was removed because it’s variable and depends on the destination and source index per trial. Including deadline would have made the state space significantly larger as even with equal values for the other variables, the agent would consider a state as different for different values of deadline. Under only 100 trials it’s unlikely the agent would encounter equal values of state parameters along with equal deadline multiple times. With a huge state space the Q-learning algorithm would fail to converge to optimal policy. It was important to include traffic light so that the agent does not violate traffic rules. “Waypoint” was included as it signals the agent where to move next and “Oncoming” parameter was included to reduce the chances of crashing into another car.

With these state parameters, we have 2 * 4 * 4 = 32 states. A state's form can be similar to this :

```
(('light', 'red'), ('oncoming', None), ('waypoint', 'left'))
```

We have four actions for each state in the action space which are 'right','left', 'forward' and None. So the Q-matrix size would be [32 x 4] where the states are the rows and the actions are the columns.

However, after observing the agents behavior again(with reviewer feedback) I noticed even if the agent frequently reaches the destination with positive reward and good timing, it has the tendency to collide with other agents in the environment when the information about those agents(how the left and the right car is going to behave) is not being incorporated into the algorithm. So I’ve decided to switch the state to following variables :

- `Light : Red/Green`
- `Oncoming : None/Left/Right/Forward`
- `Waypoint : None/Left/Right/Forward`
- `Left : None/Left/Right/Forward`
- `Right : None/Left/Right/Forward`

So we have 2*4*4*4*4 = 512 states and the Q-matrix would be [512 x 4] with four actions(Left, Right, Forward and None). A state's form can be similar to this :

```
(('light', 'red'), ('oncoming', None), ('waypoint',
'left'),('left','None'),('right','None'))
```

If I don't incorporate "Left" and "Right" and only keep "Oncoming", then the agent understands that it should not move when another car is coming straight and it should not move when the light is red, however, it'd not stop from colliding with other cars. Adding left and right related information improved the agents behavior as it mostly stopped colliding with other cars after a few collisions in the first few trials. However, in the first few trials when we take random actions as Q-learning implementation with epsilon greedy technique , there's a chance that the agent will collide with other cars, but that can't be avoided if we want to focus on exploration.

Implement Q-Learning

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

In the first implementation of Q-learning algorithm, my Q-table is a nested dictionary dictionary where the states are the keys(implemented as a tuple) and the actions are the values where action is another dictionary with action as key and Q-value as value. We start with an empty Q-table and iteratively fill it as the agent encounters each state.
(See : http://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html for pseudocode)

Q table with only one state-action pairing and initialized with random values is of this form(in order to visualize)

```
Q = {(('light', 'red'), ('oncoming', None), ('waypoint', 'left'))
: {None: 24, 'forward': 26, 'left': 22, 'right': 21}}
```

We initialize the Q-table with a default value(0 or greater). In each step, we observe the state and for that state we generally choose the action that maximizes the Q-value(which is the action that maximizes the long term reward). However, in order to ensure that we explore the state space as much as exploit the already learnt Q-values, every once in a while, with a small probability, say-epsilon (ε) we choose a random action with uniform probability. We update the Q-table with following equation :

$$Q(state,action)=(1 - \alpha)*Q(state,action) + \alpha*Q(reward+\gamma*max(Q(state'))$$

Parameters/conditions that influence the behavior of the algorithm :

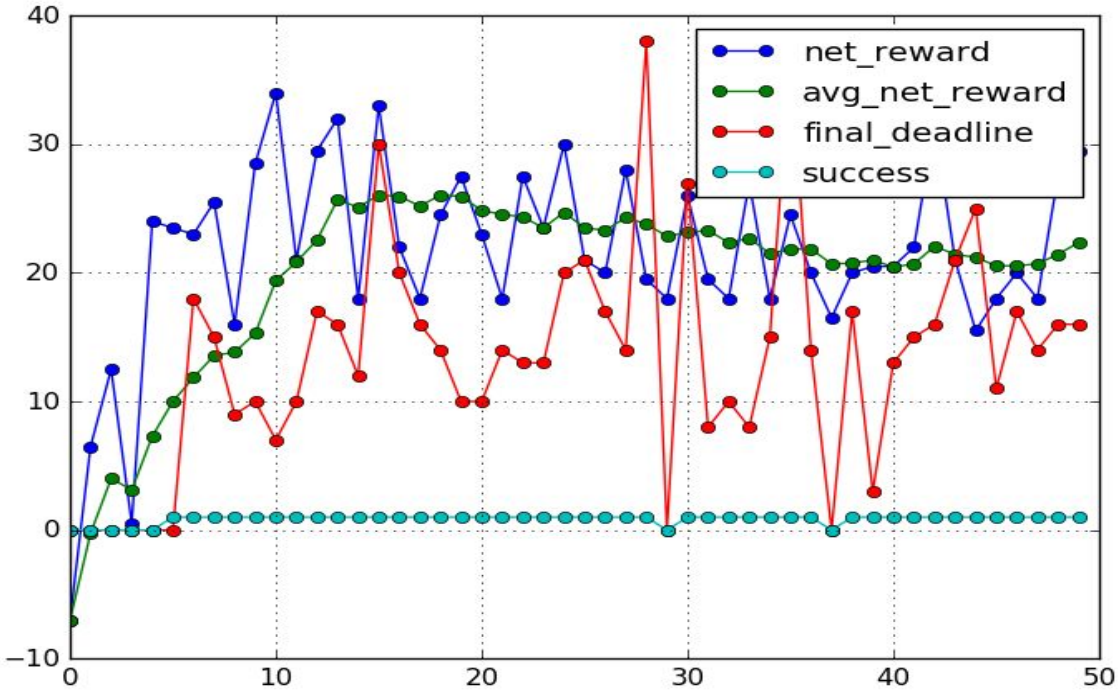
1. Learning Rate(α) : α(Alpha) determines how much we update Q-values each step and it varies from 0 to 1. A learning rate of 0 (α = 0) means we don't learn anything and keep the initial values while a learning rate of 1 (α = 1) means the agent will consider only the most recent information. If α = 0.5 it means we don't bias for either way against the new and the old Q-value. I used α = 0.5 for the initial implementation of the algorithm.
2. Discount factor(γ): Discount factor γ(gamma) determines the importance of the future rewards and varies from 0 to 1. A discount factor of 0(γ = 0) means we only care about the immediate reward and the agent makes myopic(short sighted) decisions while the discount factor of 1 (γ = 1) means we don't differentiate between getting reward now vs getting rewards later. As we have a deadline here it's better to discount future long term rewards as we want to reach the destination as fast as possible . We choose γ = 0.3 for initial settings.

3. Initial conditions : Initial(default) Q-value that we start with for each state also influences the behavior of the algorithm.High initial values(known as 'optimistic initial condition') can encourage exploration. No matter what action is selected, the update rule will lower the Q-values and eventually converge to correct values. The value of epsilon(ε), parameter that controls exploration vs exploitation can also influence the behavior of the agent as it determines how much we 'randomly' choose an action to explore the state space. Initially we choose a value of 15. Note all of these trials are done with the first state space where I had not incorporated "left" and "right" state variables.

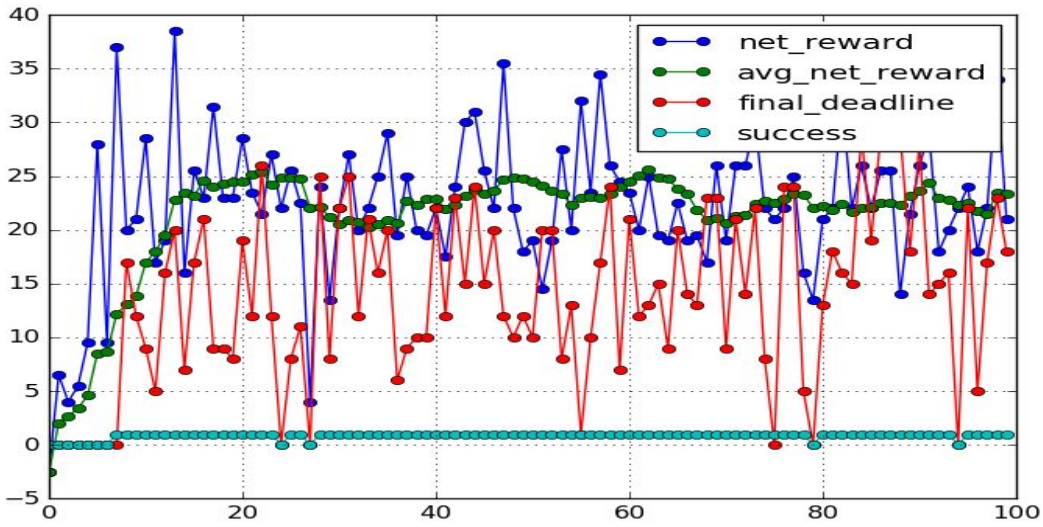
Implementing Q-learning vastly improves on random choice of the actions as the success rate goes from 24%(over 50 trials) to around 90%(measured over 100 trials), however we include a episode with only 50 trials for comparison too. Parameters chosen along with their values and initial conditions are explained above.

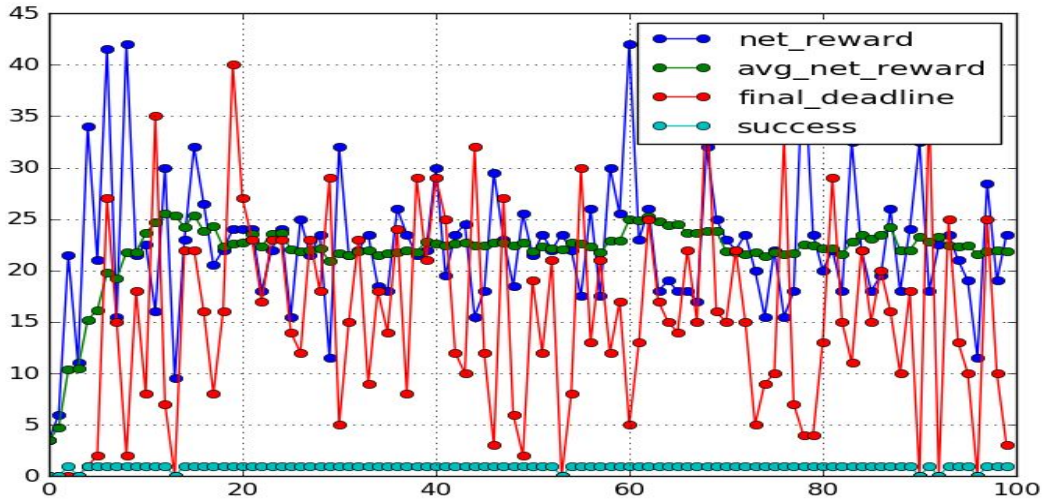
Episode	Number of trials	Average Net Reward	Average Time Remaining	Number of success	Success Rate(Percentage)
1	50	21	15.16	44	88%
2	100	22.8	13.8	89	89%
3	100	22.3	15.9	92	92%

Here's the graph with 50 trials. We can see the agent takes a few trials to succeed but after that it is mostly successful.


net_reward
avg_net_reward
final_deadline
success
40
30
20
10
0
−10
0
10
20
30
40
50

The 2 episodes with 100 trials also show good success rates(89% and 92%) respectively with similar average net reward and they also take similar time to finish the trials as the mean deadline remaining is quite close.





Enhance the driving agent

a) 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? (Here you should give a detailed list of parameter tested and report the results obtained).

Here are the trials that I've tried with different parameter values.

- Trials with similar initial conditions(same learning rate and discount factor) but different initial values(ranging from 0 to 20)

Alpha	Gamma	Initial value	Number of trials	Average net reward	Average time remaining	Number of success	Success Rate(Percentage)
0.5	0.33	0	100	22.74	17.28	98	98%
0.5	0.33	5	100	21.86	14.81	92	92%
0.5	0.33	10	100	22.015	16.59	96	96%
0.5	0.33	15	100	21.59	16.11	94	94%
0.5	0.33	20	100	22.7	14.95	91	91%

So far starting with an initial value of 0 seemed to yield the best score with 98% accuracy and reasonable average reward.

- Trials with different values of alpha(ranging from 0 to 1), initial value at 0 (but I'd experiment with a few others to see the results) and similar discount factor.

Alpha	Gamma	Initial value	Number of trials	Average net reward	Average time remaining	Number of success	Success Rate(Percentage)
1	0.33	0	100	22.93	17.12	95	95%
0.8	0.33	0	100	22.7	16.8	96	96%

0.6	0.33	0	100	22	15.41	95	95%
0.5	0.33	0	100	22.74	17.28	98	98%
0.4	0.33	0	100	21.64	15.00	95	95%
0.3	0.33	0	100	22	17.06	98	98%

- Trials with different discounting factors, Alpha = 0.5/0.3 given until now they performed best at 98% score and initial value of 0.

Alpha	Gamma	Initial value	Number of trials	Average net reward	Average time remaining	Number of success	Success Rate(Percentage)
0.5	1	0	100	22.47	9.77	67	67%
0.5	0.8	0	100	24.60	13.64	86	86%
0.5	0.6	0	100	21.93	16.36	94	94%
0.5	0.4	0	100	22.405	16.19	94	94%
0.5	0.3	0	100	22.82	15.33	98	98%

It appears from the table that a high discounting factor essentially makes the success rate decrease while a discounting factor of 0.33 along with a learning rate of 0.5 and initial value of 0 has consistently yielded good results(around 97-98%). So I'll choose these parameters as the final Q-learning implementation.

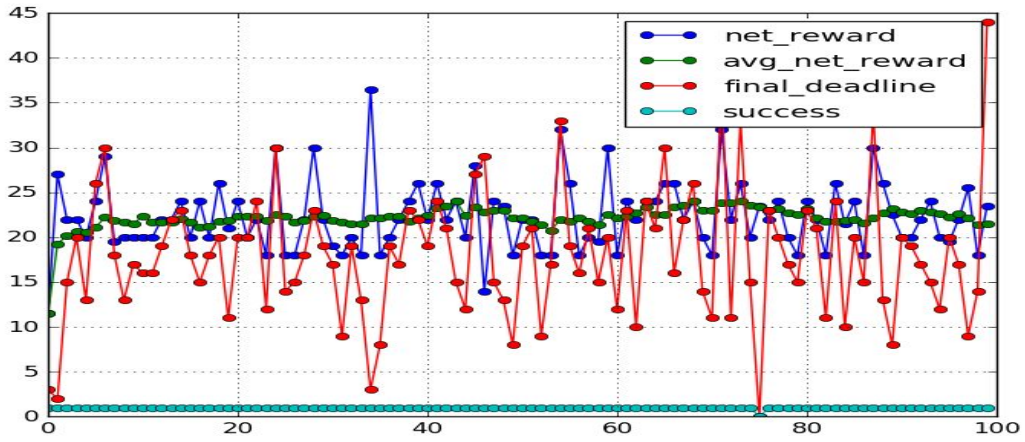
b) Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties? (Here you should observe and analysis the behavior of the agent, does it always go in an optimal path and follow traffic laws and if not under what specific circumstances does it not follow the optimal policies).

The agent generally is successful with the final chosen values, and the success rate is around 98%, however, there are some random failures, initially I felt it's because of the epsilon parameter making the agent go for exploration.

I've tried to decay the epsilon parameter with respect to trial number (ϵ/trial number) because if the trial numbers go up, we don't need to explore much and we can just exploit what we have learnt and this yielded a 99% success rate.

Alpha	Gamma	Initial value	Number of trials	Average net reward	Average time remaining	Number of success	Success Rate(Percentage)
0.5	0.1	0.3	100	22.14	17.99	99	99%

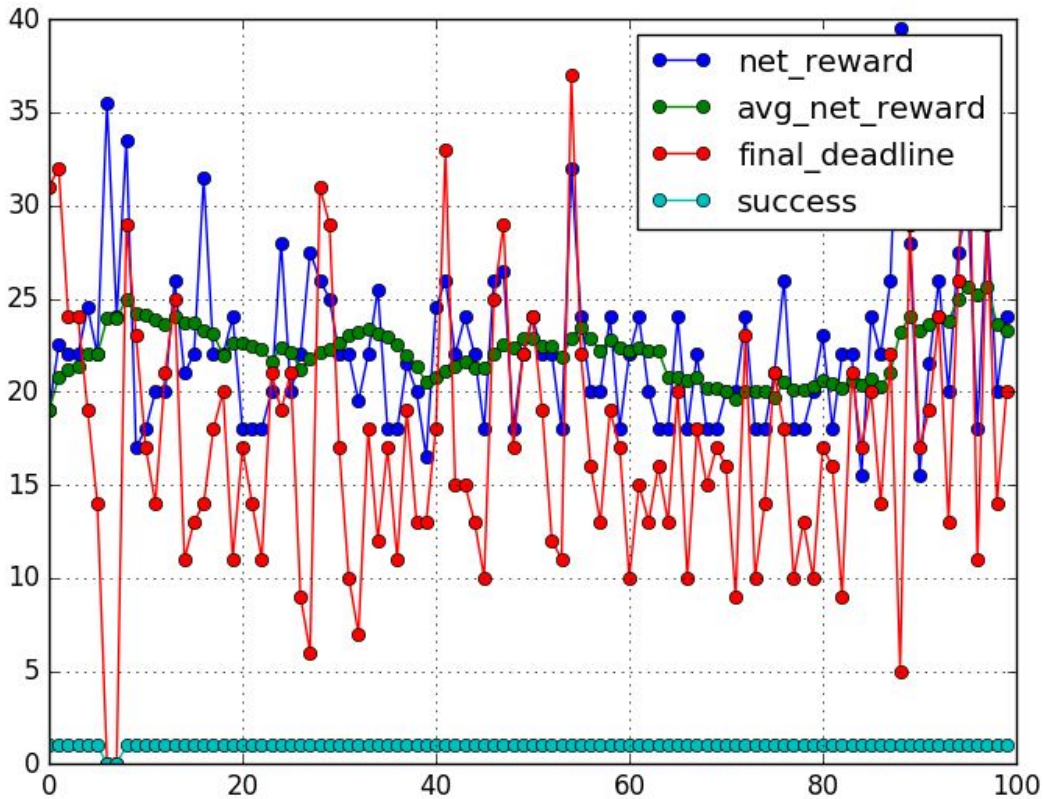
Here's the visualization for the final version for first submission that had not used the new state variables :



After implementing Q-learning with the new state variables to decrease collisions, I've again tried to capture the new performance with the following table. First I've tried the optimal parameters I had found earlier and then I've tried some other combinations :

Alpha	Gamma	Initial value	Number of trials	Average net reward	Average time remaining	Number of success	Success Rate(Percentage)
0.5	0.3	0	100	22.265	17.32	98	98%

As we see from the visualization for the new state space, even if there's a few mistakes in the beginning, this Q-learning manages to finish most of the trials successfully in a row.



After visual inspection I've found that in general the mistakes I considered random in the beginning happened because the agent failed to learn optimal policy with the previous state variables. However even with the new state variables often the agent wanted to avoid collisions and ended up going in opposite directions to avoid them. Also in the beginning sometimes the agent does take random actions and ends up far from the destination. If both combines, random actions and forced waiting to avoid collisions, the agent generally runs out of time and fails to reach the destination, however, in a real world setting that would be much preferable compared to hitting some other cars.

For inspection, I've tried some other combinations of the parameters and initial values :

Alpha	Gamma	Initial value	Number of trials	Average net reward	Average time remaining	Number of success	Success Rate(Percentage)
0.5	0.3	15	100	22.085	16.19	95	95%
0.5	1	0	100	18.29	13.47	97	97%
0.8	0.5	0	100	22.41	16.27	98	98%
0.5	0.5	0	100	22.8	17.3	97	97%

Despite the fact that gamma = 0.5 and alpha = 0.8 has similar results as alpha = 0.5 and gamma = 0.3, the last failure in the visualization above was near 10th trial while the alpha = 0.5 and gamma = 0.5 combination has the last failure around 50th trial. Setting a high initial value tend to lower successful trials, possibly because the agent was exploring even after learning the policy. As long as the parameters are mostly reasonable(gamma around 0.3-0.5, alpha around 0.5-0.6) I felt the results were pretty much robust and the agent was able to be successful after around 50-60th trials, however for the final implementation I choose to go with gamma = 0.3 and alpha = 0.5 along with the initial value of 0 and epsilon decayed by trial and as we can see from the visualization that it results in optimum policy where we get a long streak of successes only after

the 10/12th trial. I thought including both the right and left values for state would result in a much bigger Q-matrix(512 states instead of initial 32) however in practice the program didn't take long to run so I've improved upon after learning about my mistakes.