Implement a basic driving agent

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), follo

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.
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?

I run the simulation 5 times and it arrived surprisingly 3 times; one in t=52, another in t=77 and the last in one t=27. The behavior is obviously pretty random hitting a lot of negative rewards by having a lot of crashes, violating traffic rules or choosing bad directions.

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.

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.

The representation for the states I choosed is [light,car_oncoming_dir, car_right, car_left, waypoint_dir, deadline <5]

Most of the different components of the state are pretty obvious:

- light: Will show if the light is red or green.
- car_*: Will show if a car is in a particular direction. And the direction which is turning.
- waypoint_dir: Direction of the waypoint (probably the most useful).
- deadline <5: Will be true if the deadline is very near, this should incentivize the taxi to break the law when the deadline is near. Probably the reward will be greater than the penalty. I choose this approach to minimize the number of states.

The goal of this project is that the cab gets to its destination as fast as possible but with no crashes. I choose this implementation because is the more complete. It can cover all the cases where the cab could crash.

This are 768 different states let's hope the algorithm can handle them.

Implement Q-Learning

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?

This is Bellman's Equation for Q learning:

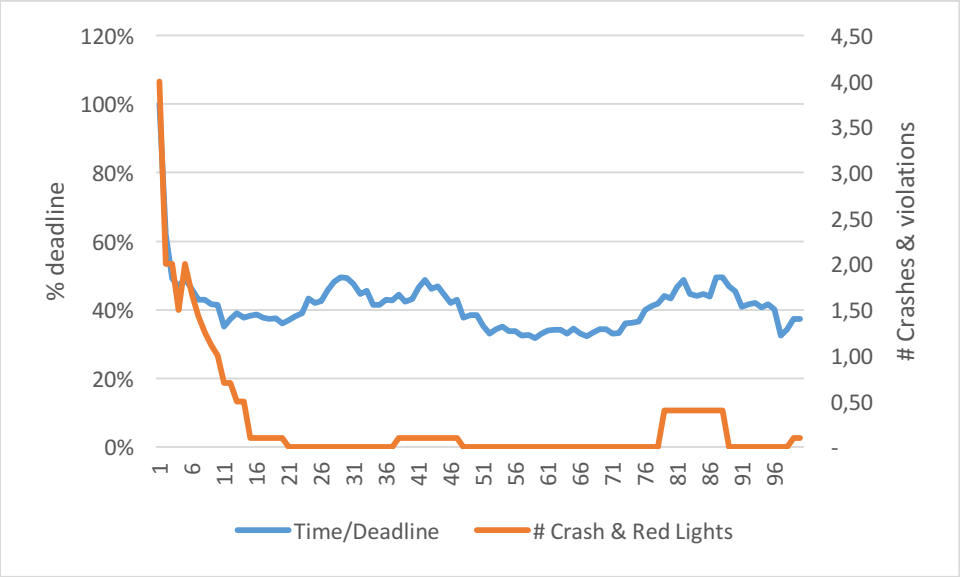
$$Q_{t+1}(s_t, a_t) \leftarrow \underbrace{Q_t(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t, a_t)}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q_t(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q_t(s_t, a_t)}_{\text{old value}} \right)$$

At first I did my implementation using a gamma value of 0.2 and an alpha value of 0.8.

The agent starts quite random, but starts to get smarter. It quickly starts to follow the direction of the waypoint and stop at the red lights, as this are the simplest and at the same time most rewarding actions.

At first it collides a lot with the other cars but as it starts to experience more and more crashes it learns from them and its crash rate starts to drop.

I will measure the model under two metrics: how fast it gets to waypoint (total time/deadline) and how many times it crashes or brakes the law. I choose this metrics because our two goals are that the car arrives quickly at its destination and that it arrives safe without breaking the law. This are the results for the model:



Both lines are a moving average of 10 iterations. Blue line is in the first axis and the orange one in the left axis.

As we can see our time to the destination decreases quickly, by the third iteration we are near the line of convergence of 40%. Also from the second iteration the cab manages to reach destination consistently without exception.

In terms of total reward, the model gets positive reward from iteration 2 and its average is 23, depending on how long is the trip and how was the traffic.

Crashes and traffic violations also go down very quickly but we still have some sporadic occurrences near the end. So let's see the crash log for those last events:

iteration	input	waypoint	action
79	{'light': 'red', 'oncoming': 'left', 'right': None, 'left': None}	forward	forward
79	{'light': 'red', 'oncoming': 'left', 'right': None, 'left': None}	forward	left
79	{'light': 'red', 'oncoming': 'left', 'right': None, 'left': None}	forward	right
99	{'light': 'red', 'oncoming': None, 'right': None, 'left': 'forward'}	left	right

In iteration 79 it came face to face with a car that was turning left while on a red light and in three different occasions tried different strategies. And in iteration 99 it tried to use its right of way but the other car was moving forward from the left.

This shows us that car must visit all the states, and tried all the action in order to learn the optimal policy. So let's see our Q function to see how many states we are visiting.

State, Action	Q
('green', None, None, None, 'left', False, 'forward')	-0,4
('red', 'right', None, None, 'forward', False, None)	0,054377
('red', None, None, None, 'right', False, 'left')	-0,8
('green', None, None, 'forward', 'forward', False, 'left')	-0,0577749
('green', None, None, None, 'forward', False, 'forward')	2,01681
('red', 'forward', None, None, 'forward', False, 'forward')	-0,628967
('red', 'right', None, None, 'forward', False, 'left')	-0,733959
('green', None, 'left', None, 'left', False, 'forward')	-0,0543839
('red', 'left', None, None, 'forward', False, 'forward')	-0,8
('green', None, None, None, 'right', False, 'left')	-0,144
('green', 'left', None, None, 'forward', False, 'left')	-0,0568797
('red', 'forward', None, None, 'forward', False, 'left')	-0,96
('red', None, None, None, 'forward', False, None)	0,407441
('green', None, None, 'left', 'right', False, 'right')	1,98432
('green', None, 'right', None, 'forward', False, 'left')	-0,0607566

('green', None, None, None, 'left', False, 'right')	-0,074368
('green', None, 'left', None, 'forward', False, 'right')	-0,0149183
('green', None, 'left', None, 'right', False, 'right')	2,29942
('red', 'left', None, None, 'forward', False, 'left')	-0,96
('red', None, None, None, 'forward', False, 'forward')	-0,441593
('red', None, None, None, 'forward', False, 'right')	-0,4
('red', None, None, 'right', 'left', False, None)	0,00142853
('green', None, 'right', None, 'forward', False, 'right')	-0,336966
('green', None, None, None, 'forward', True, 'forward')	1,6
('green', None, None, None, 'right', False, 'forward')	-0,4
('green', 'forward', None, None, 'forward', False, 'right')	-0,0156391
('green', None, None, None, 'forward', False, 'right')	-0,4
('green', None, None, 'forward', 'right', False, 'forward')	-0,0667184
('red', None, 'forward', None, 'left', False, 'right')	-0,743905
('red', 'left', None, None, 'forward', False, 'right')	-0,8
('green', None, None, None, 'right', True, 'right')	1,6
('green', None, None, 'left', 'forward', False, 'right')	-0,144
('red', 'forward', None, None, 'forward', False, None)	0,322578
('green', 'left', None, None, 'right', False, 'right')	1,65355
('green', None, None, None, 'forward', True, 'left')	-0,4
('green', None, None, 'forward', 'forward', False, 'right')	-0,0112955
('green', None, 'left', None, 'forward', False, 'left')	-0,077573
('green', None, None, None, 'forward', True, 'right')	-0,4
('red', None, None, None, 'left', False, 'right')	-0,00980365
('red', None, 'forward', None, 'forward', False, 'right')	-0,8
('green', None, 'forward', None, 'forward', False, 'forward')	2,39324
('green', None, None, 'right', 'forward', False, 'forward')	1,97467
('red', None, None, None, 'forward', False, 'left')	-0,8
('red', None, None, None, 'left', False, 'forward')	-0,8
('green', None, None, None, 'right', False, 'right')	2,06508
('red', None, 'right', None, 'forward', False, None)	0,0577679
('red', None, None, None, 'forward', True, None)	0,256
('green', None, None, None, 'left', False, 'left')	2,13733
('green', None, None, 'forward', 'forward', False, 'forward')	2,04163
('red', None, None, None, 'right', False, 'forward')	-0,478874
('red', None, None, None, 'right', False, 'right')	2,42312
('red', 'forward', None, None, 'forward', False, 'right')	-0,343379
('green', None, None, 'right', 'forward', False, 'left')	-0,0039759
('green', 'left', None, None, 'forward', False, 'right')	-0,346108
('red', None, None, None, 'left', False, 'left')	-0,8
('red', None, None, None, 'left', False, None)	0,379556
('green', None, None, None, 'forward', False, 'left')	-0,4

This are all the nonzero values which correspond to 57 different combinations of states and actions of the 3072 possible, this is no good because it means that too little of the different cases are visited.

Also is fun to watch the different values of Q for example the state/action has value ('green', None, None, None, 'forward', True, 'forward'), which means moving forward

when t there is green light, the waypoint is forward and there are no cars, has value 1,6 which is a very high number so it probably will always take that action given that state. In opposition state/action ('red', 'forward', None, None, 'forward', False, 'forward'), which means moving forward on a red light, has a very negative value of -0,62

Enhance the driving agent

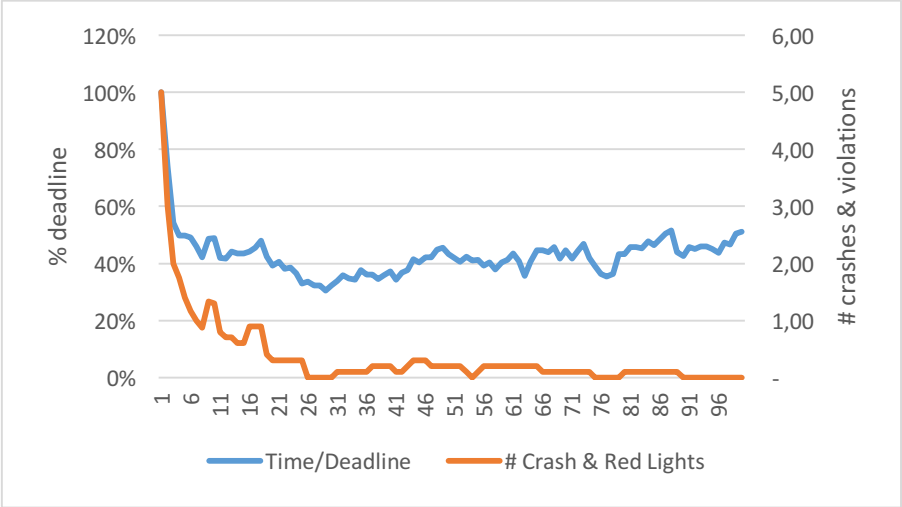
Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - 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 objective was achieved with the previous model but I still think is not enough I broke the law at iteration 97 so I want to improve my policy so that all the unfrequented cases are learn. My main goal is to not have crashes or break the law after the 80^{th} iteration.

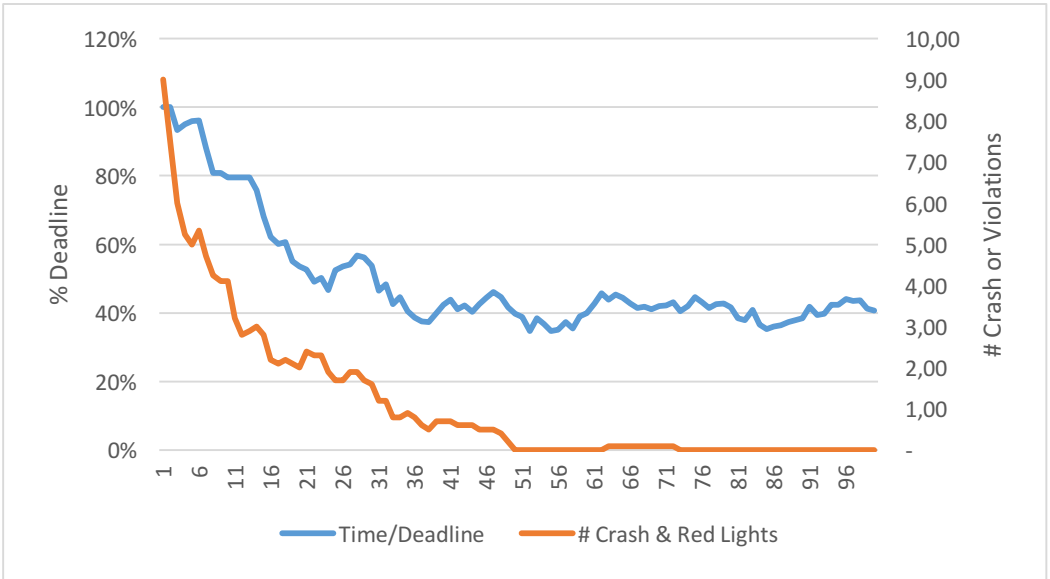
The first thing I want to do is to eliminate from the state “deadlines<5” because as we saw the taxi quickly learns to get to destination and is usually never near the deadline. This will reduce the states by half helping the model populate more states.

The results are the following:



We almost didn't we had a crash in the 80th iteration, but we are almost there. Actually now we have 66 nonzero Q values over 1536 which is an improvement.

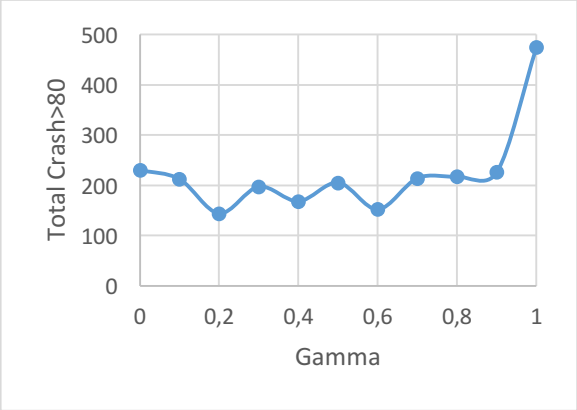
But my number of states visited is still low so we are going to implement e-greedy. We are going to start with an epsilon of 0.5 and then we are going to slowly decrease it so that by iteration number 50 epsilon would be 0. The results are the following:

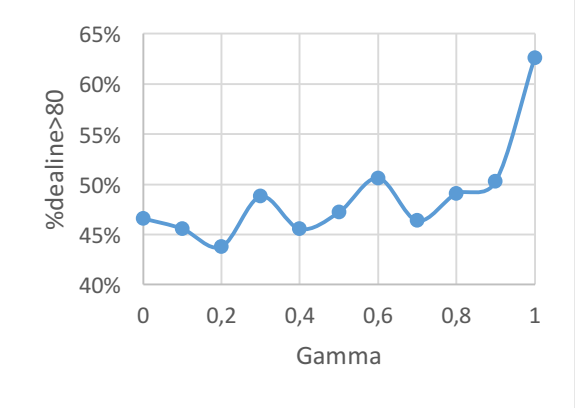


We did it! last crash/violation was at iteration 63 in which I go forward through a red light while having a car to the left.

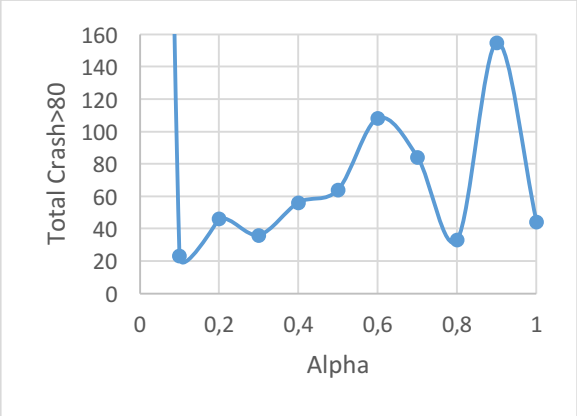
We have a little improvement also in the number of nonzero Q values, now they are 71.

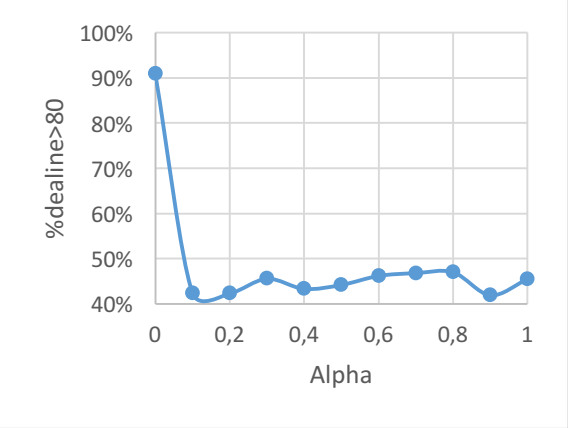
Finally I they a grid search over the values of gamma and alpha to get the optimal values. I decided to tracked performance in %deadline and # of crashes and violations. First let's see gamma





As we can see there is some variability in the scores. Clearly gamma=1 is a bad number. Here I am going to pick as candidates 0.2, 0.4 and 0.6 because they have the best scores.





For alpha I am going to pick values 0.2 and 0.8 as candidates. Here Alpha = 1 is clearly a very bad parameter.

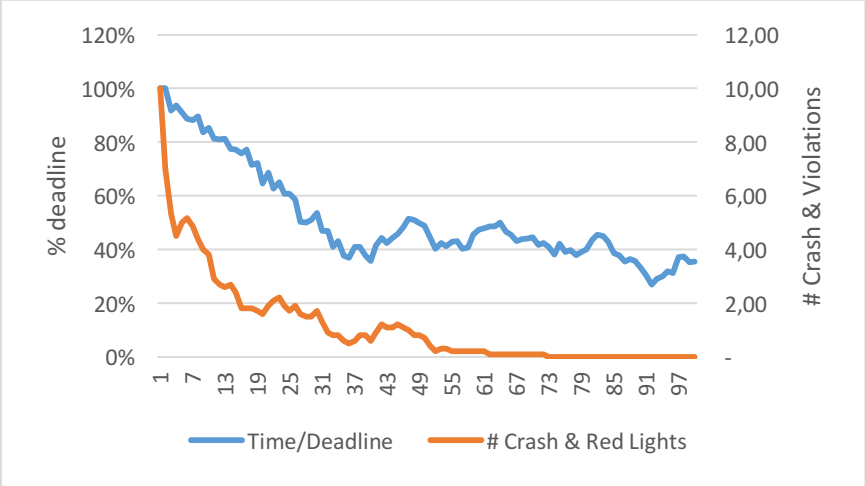
Let's see what happens when we combine the candidates:

Gamma	Alpha	Total Crash>80	Avarage %deadlien>80
0,2	0,1	2	45%
0,4	0,1	1	44%
0,6	0,1	0	48%
0,2	0,8	1	41%

0,4	0,8	0	38%
0,6	0,8	1	46%

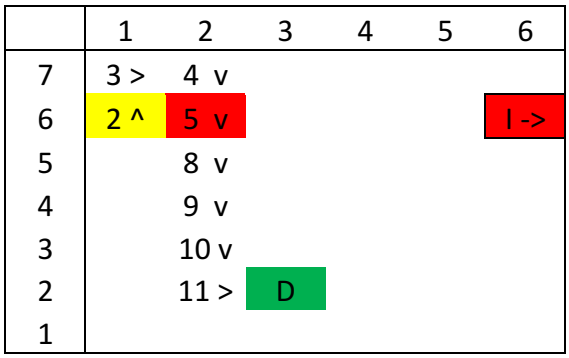
From this table I will pick gamma = 0.4 and alpha = 0.8 because they have the best scores.

With this optimal values let's see how we do.



In terms of crashes and traffic violations we did similar than the last time, with none of this error from iteration 67 and on. It terms of speed we had a little improvement at the end, we are below the 40% line.

Finally, I want to review if the route the taxi is taking is optimal. A very important component of the optimal policy is that the car finds the right path and that it doesn't run in circles. To avoid this last case, the car should always (when possible) follow the waypoint. And if it can't then it should take the fastest alternative route. If we check our error log there very few cases where this happens at the end and they are all when trying to avoid a crash with other car. This policy in most cases is ok because the agent chooses a -0.5 over a -1.0 reward. The problem is that maybe it detour to a suboptimal path. This is the case for the last iteration let see the last route taken:



Green is destination. Red are red lights that it encountered. Yellow is a Nav error. The number is the time and the arrow the direction taken.

As we can see the agent did a navigation error in position (6,1). The state at this point was {'light': 'green', 'oncoming': None, 'right': 'forward', 'left': None, 'waypoint': 'right'}, so the agent could not turn to the direction of the waypoint because there was a car there. This essentially is ok but the optimal action to take would have been to move forward and not to turn left. Because the car turned left it had to do a big detour and the path was not optimal.

The conclusion is that the policy is acting right most of the time, that is why we get good results, but it is still not optimal. It need to encounter more of the most uncommon cases to get to an optimal policy.

Ideas for Further improvements

The main problem with this model is top hard to see all the posible cases, so there are many states that just don't have a Q value. By looking through the the states that have 0 values in the Q table almost all of them correspond to situations where there are 2 or 3 cars present, so states very hard to come by. Thus we must learn this cases to have the optimal policy. And even more, as we saw with the route in the last example the values of Q for some cases are still not optimal because we need to visit those states more often to learn the optimal action to take.

One solution would be to associate some piece of the one state/action to other similar state/action, for example if I got a negative for moving foward in a red light I would propagate a piece of the negative reward to the same but maybe to moving right or to an state with a red light but with a car in the other side. The problem with this aproach is that the relation between the states can get a little bit messy and I think this approach is cheating because you are adding prior knowledge to the model when establishin relations between the states.

This is why I thought that maybe a good solution would be to learn this relations with another machine learning model. We could replace Q with a regression model, thus instead of calculating Q we would predict it. The inputs would be the different variables of

the state and the action, and the output would be the value of Q. We would train the model with the different values of <s,a,r,s'> and the predicted value would be Y = R(s)+gamma*Max(Q(s',a)).

With this approach not only the current state/action value of Q woud be updated, but also tehvalues of other Qs that the model seems to be related to it.

This looks like a great idea for my capstone project ☺