Implement a basic driving agent

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

The agent picks a random action from one of the 4 possible values of 'forward', 'left', 'right' and None. The car sometimes reaches its destination.

Identify and update state

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

Below states were picked up as valid states for the agent to be in at any moment of time.

	VALID STATES					
	Car to my Left		Car Oncoming			
Sr #	Yes or No	Direction	Yes or No	Direction	Traffic Light	Next Waypoint
1	No		No		Red	F
2	No		No		Red	L
3	No		No		Red	R
4	No		No		Green	F
5	No		No		Green	L
6	No		No		Green	R
13	**Yes**	**F**	**No**		**Red**	**R**
22	**Yes**	**F**	**No**		**Green**	**R**
37	No		Yes	F	Green	L
39	No		Yes	R	Green	L

At any given point of time (or state), the agent may see cars to its left, cars to its right, and oncoming traffic. It will also have a sense of which direction each of the earlier mentioned cars are headed. Also, the agent will have a notion of the direction to which it is headed from the next_waypoint input. However, the agent should not be concerned with the car on the right as the agent if at all it has to turn right will be following that car. In a way, the agent can ignore if there is a car ahead of it. Hence, while considering total states car to the right is not considered.

Total number of states that are possible:
num_of_light_options = 2 (light can have 2 choices: 'red', 'green')
num_of_next_waypoint_options = 3 (next waypoint could be one of: 'forward', 'left', 'right')
num_of_directions = 3 (F, L, R)

No car oncoming, No car left: 2*3=6 (num_of_light_options * num_of_next_waypoint_options)
No car oncoming, Car left: 2*3*3=18 (num_of_light_options * num_of_next_waypoint_options * num_of_directions)

Car oncoming, No Car left: 2*3*3=18 (num_of_light_options * num_of_next_waypoint_options * num_of_directions)
Car oncoming, Car left: 2*3*3*3=54 (num_of_light_options * num_of_next_waypoint_options * num_of_directions * num_of_directions)

So, total number of states possible: 96
However, many states are more or less the same. For example, a state in which there is a car on agent's left and is turning right is same as no car on left. This list was reduced to 36 states. Following logic was used.

```
valid_left = ['forward', 'otherwise']
valid_oncoming=['forward','right','otherwise']
valid_lights = ['red', 'green']
valid_next_waypoint = ['forward', 'left','right']
```

So, total number of states possible: 2*3*2*3 = 96
However, many states could not be reached by the agent. So, the states were further reduced to only a set of 10. For detail list of states, please see attached xls file.

The following information from the inputs was not chosen for a valid state:
1.) deadline: This input was not chosen as this is a decreasing number. This is more like a timer. Realistically speaking, if a person was learning how to drive and was given a deadline within which reach a target, he or she would end up in an accident or jumping red lights. Jumping red lights is something that we do not intend our agent to learn. Hence this input was not used.

2.) car to agent's right information: The reason is because we are following US traffic rules. Consider a driver in the US trying to make a right turn, he or she is always going to be behind the car on the right. If the driver was making a left turn, he or she would look for oncoming traffic as traffic from the left would be at a red stop. Hence, the information that a car is at the driver's right does not help someone drive in the US. Hence, this information was not included.

Implement Q-Learning

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

Q-learning was implemented in the agency.py file. Following is the Q-learning equation.
Q_hat(current state, next action) <--alpha-- (reward for reaching future state) + gamma*Q_hat(future state, action that would give max value for that future state))

Some action was taken in the past that brought agent in the present state. Replacing above equation with one time step before, we get
Q_hat(past state, past action) <--alpha-- past reward + gamma*Q_hat(present state, action that maximizes Q val for present state)

The action that maximizes the qval was chosen. Below logic was used.

qidx = np.argmax(self.Qdf.loc[self.state, :].values)
action = action_arr[qidx]

Following Q matrix was reached after 100 trails.

Qdf:

	forward	left	right	None
0	-0.900000	-0.900000	-0.450000	1.823457
1	-0.900000	-0.900000	-0.450000	2.430977
2	-0.900000	-0.900000	3.062123	0.000000
3	3.840011	0.000000	0.000000	0.000000
4	-0.450000	3.582976	0.000000	0.000000
5	-0.450000	-0.450000	3.376375	0.000000
6	0.000000	0.000000	0.000000	0.000000
7	0.000000	0.000000	0.000000	0.000000
8	0.000000	0.000000	0.000000	0.000000
9	0.000000	0.000000	0.000000	0.000000

	Car to my Left		Car Oncoming			
Sr #	Yes or No	Direction	Yes or No	Direction	Traffic Light	Next Waypoint
1	No		No		Red	F
2	No		No		Red	L
3	No		No		Red	R
4	No		No		Green	F
5	No		No		Green	L
6	No		No		Green	R
13	**Yes**	**F**	**No**		**Red**	**R**
22	**Yes**	**F**	**No**		**Green**	**R**
37	No		Yes	F	Green	L
39	No		Yes	R	Green	L

First 3 rows are for 'forward', 'left' and 'right' on a 'red' light. The agent learns that the best action for the above 3 states are None, None and Right, which are correct. The agent also learns the right action in case of green light and no traffic elsewhere. However, the agent is not able to learn the last 4 states. One reason is that these states are very rare. Hence in the current 100 trails, these states were never hit and hence, the agent could not learn the best action.

When the number of trials were increased to 10000, the agent could learn the following Q matrix.

Qdf:

	forward	left	right	None
0	-0.900000	-0.900000	-0.450000	1.333562
1	-0.900000	-0.900000	-0.281761	6.197208
2	-0.900000	-0.900000	3.808616	0.000000
3	2.673026	0.000000	0.000000	0.000000
4	-0.450000	4.835311	0.000000	0.000000
5	-0.252000	-0.253771	7.914226	0.000000
6	0.925246	0.000000	0.000000	0.000000
7	3.782319	0.000000	0.000000	0.000000
8	-0.327166	0.899741	0.000000	0.000000
9	0.936767	0.000000	0.000000	0.000000

	Car to my Left		Car Oncoming			
Sr #	Yes or No	Direction	Yes or No	Direction	Traffic Light	Next Waypoint
1	No		No		Red	F
2	No		No		Red	L
3	No		No		Red	R
4	No		No		Green	F
5	No		No		Green	L
6	No		No		Green	R
13	**Yes**	**F**	**No**		**Red**	**R**
22	**Yes**	**F**	**No**		**Green**	**R**
37	No		Yes	F	Green	L
39	No		Yes	R	Green	L

Here, we have q values that are learnt are okay. But the true action that was to be learnt in the last 4 rows should that be of None. But the agent has learnt one value that is completely wrong. The value for state 6. The agent has learn that it should jump the red light and go forward in case if it reaches state 6. However, values for state 7 and state 9 that are learnt are to go forward which is okay, since the lights are green. The action learnt for state 8 is that of making a left even though there is oncoming traffic in the forward direction. One reason for this learning could be that number of times this state is hit would be small.

The negative values indicate that there is penalty in taking an action in that state. For example, a penalty on all actions except None for row 1 indicates that the best action in case the agent hits a red light and the next waypoint is forward is to wait for light to turn green.

Enhance the driving agent

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 values chosen for alpha is 0.9 and that of gamma is 0.5. The agent reaches destination but whether it reaches in the minimum possible time is not known.

One strategy to include exploration into the learning process is to have the agent take random actions even if it has learnt what the best action for that state is. This approach is similar to simulated annealing approach and is called greedy epsilon approach. In this approach, epsilon is initialized to a value smaller than 1, and action determined by policy is chosen based on this number. The exact math is that agent follows the action determined by the policy with probability 1-epsilon and choses random action with probability epsilon. In most scenarios, an epsilon value of 0.1 would suffice, meaning 1 out of 10 times, random action is taken and 9 times the action determined by the policy is chosen. However, there is a problem in this simple approach. The problem is that the learning agent would continue making a random choice 1 out of 10 times even if it had spent a lot of time learning the policy. This would mean that there is good probability of agent getting a negative reward even after knowing what the right action for that state is. So, this issue can be removed with having epsilon decay over time. This way, as the agent visits a particular state over and over, it is able to guess the right action. And with time, as epsilon is decreasing, its probability of choosing a random action also decreases. Intuitively speaking, the agent becomes wiser with time just like how a normal human would learn. Well this all sounds good, but is not applicable in this situation. The reason is that in order to Q-learning to converge, the learning agent has to visit all the states infinite number of times. And this is a condition which is not satisfied here. Also, if epsilon is decreasing with time there is a possibility that by the time learning agent visits a state that is less often visited, epsilon has decreased so much that there is no possibility of randomness left. Concretely, say, number of trails = 100. Epsilon starts decreasing with each new state. So epsilon would decrease continuously. Since most of the states that are visited are 0 to 5, randomness benefit is only applicable to these states. When learning agent visits state 9, epsilon has already approximating zero and hence there is no randomness or exploration in states that are less often visited.

So, in order to have learning agent not receive negative rewards in trails 90-100 and explore states that it has not visited yet, my learning agent uses this logic.

```
if self.state_visit_hist[self.state]>=50:
    action = action
else:
    action = random.sample(action_arr,1)[0]
```

If a particular state is visited 50 times, I assume that learning agent is able to figure out the right action and it stops exploring. However, if it has not visited a state 50 times, I assume there is still scope of exploration and hence the random sampling for action.

Below figure shows the histogram of number of times a state is visited (state_visit_hist).

```
Qdf:
    forward      left     right      None
0  0.796504 -0.434550  0.010305  0.226592
1 -0.598784 -0.624192 -0.270316  0.756058
2  0.566632  0.408967  3.804952  1.954755
3  3.392050  1.241875 -0.079861  1.843329
4  1.241552  2.679069  0.930116  1.471629
5  1.352422  1.118992  3.910622  1.844966
6  2.651134  0.000000  0.000000  0.000000
7  0.966505  1.420527  0.000000  0.000000
8  0.000000  0.000000  0.000000  0.000000
9  0.644363  0.000000  0.000000  0.000000

state_visit_hist:
[ 568.  183.  305.  494.  186.  298.    1.    3.    0.    1.]
```

Below screenshots correspond to a policy when epsilon=0.25 and there is no decay

```
Qdf:
    forward      left     right      None
0 -0.190928  0.265553  1.445686  0.117506
1  0.184015  0.506077  0.730488  0.162072
2  0.776494  1.090442  3.675180  1.914646
3  2.651791  1.222010  0.892045  1.184799
4  0.800819  3.408767  1.264090  1.864482
5  1.379271  2.091116  3.322188  1.908410
6  0.373698  0.268796  3.348544  0.000000
7  2.555203  1.328679  3.935635  1.752033
8  1.225404  2.983883  0.984951  2.958966
9 -0.359551 -0.060053  0.000000  0.000000

state_visit_hist:
[10971  3704  7069  9763  3549  6818    19    74    11     7]
```

```
state_visit_hist:
[10970  3704  7067  9762  3549  6817    19    74    11     7]
state: 0
LearningAgent.update(): deadline = 17, inputs = {'light': 'red', 'oncoming': Non
e, 'right': None, 'left': None}, action = right, reward = -0.5

Qdf:
    forward      left     right      None
0 -0.190928  0.265553  1.348565  0.117506
1  0.184015  0.506077  0.730488  0.162072
2  0.776494  1.090442  3.912953  1.914646
3  2.651791  1.222010  0.892045  1.184799
4  0.800819  3.408767  1.264090  1.864482
5  1.379271  2.091116  3.288817  1.908410
6  0.373698  0.268796  3.348544  0.000000
7  2.555203  1.328679  3.935635  1.752033
8  1.225404  2.983883  0.984951  2.958966
9 -0.359551 -0.060053  0.000000  0.000000

state_visit_hist:
[10971  3704  7067  9762  3549  6817    19    74    11     7]
```

The screen on the right receives a negative reward even in the last iteration run. The logic of having a constant epsilon did well for states that were visited rarely (states 6 to 9). We can see that as the values are filled for states 6 to 9. However, this strategy is not acceptable because it would result in a random action (and most likely a negative reward) for states that are visited with high frequency.

Below is the matrix that was achieved after 10000 trails. The intention was to reach states that get less often visited.

```
Qdf:
    forward      left       right      None
0 -0.824167 -0.702273   0.045055  1.617300
1 -0.655610 -0.512573  -0.326563  1.584695
2  0.752282  0.359611  12.058924  1.164059
3  3.166077  1.266666   1.744243  1.833850
4  0.393070  5.218138   1.410136  2.141897
5 -0.048626  1.378056   5.578498  1.885503
6  1.031251  1.810084   2.772142  5.130196
7  2.518124  2.139174   5.487581  2.602984
8  0.893740  1.456651   3.397022  1.778077
9  2.076352  0.454359   1.226139  2.641404

state_visit_hist:
[52204 10198  6078 42579  7120  6075    17    76    20    15]
```

	Car to my Left		Car Oncoming			
Sr #	Yes or No	Direction	Yes or No	Direction	Traffic Light	Next Waypoint
1	No		No		Red	F
2	No		No		Red	L
3	No		No		Red	R
4	No		No		Green	F
5	No		No		Green	L
6	No		No		Green	R
13	**Yes**	**F**	**No**		**Red**	**R**
22	**Yes**	**F**	**No**		**Green**	**R**
37	No		Yes	F	Green	L
39	No		Yes	R	Green	L

The learning agent I able to successfully learn after 10000 trails. The agent learns that in states 0 and 1, the best action is None. For state2, the best action learnt is to make a right. This is correct driving as per US rules when traffic signal is red. For states, 3 to 6, when the traffic light is green, it appropriately learns to take forward, left and right when there is no oncoming traffic from any side. The learning is pretty impressive for states 6 to 9. In state 6, learning agent correctly learns that it has to wait at red stop when it is attempting a right turn and there is oncoming traffic from the left. In state 7, it learns that it can take a right even though there is oncoming traffic from left as there is green light at the traffic signal. In state 8, the agent is trying to learn the best action when it is at a green light and attempting a left turn while, there is oncoming traffic. Although, the action that is learnt is to make a right is a valid option but ideal response would be None. In state 9, the agent correctly learns to perform a None action when it is attempting a left turn on a green light with oncoming traffic attempting a right turn.

The agent does not incur and penalties and the policy learnt is very close to optimal policy.