Project 4 Train a Smartcab to Drive

Task 1: 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),*

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 the first step of the project I created an agent which chose a random action at each state, below is an excerpt of python code.

```
self.directions = [None, 'forward', 'left', ‘right']
…
def random_action(self):
    random_direction = random.choice(self.directions)
    return random_direction
```

After setting the deadline flag to false and watching the performance of the agent I noticed that the agent would occasionally reach the destination but often the episode would timeout. In order to gauge the actual performance I removed the upper time-step limit of 100 steps to see just how long would it take the agent to complete the episode, below is a log of 20 episodes for the random agent.

Random Agent Performance

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
252	21	438	112	27	11	82	4	1	51	100	186	46	10	116	80	66	138	196	28

Task 2: 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.

For the set of states to include I choose all the information necessary to make a choice at the current location. This includes the

- next_waypoint, which was obtained by calling the **`next_waypoint()`** method on the agent's super object.
- the traffic light from the **`environment.sense()`** method **`inputs['light']`**, this is important because the agent obtains a reward when going right on red but a penalty when going left or forward on red
- the oncoming traffic from the **`environment.sense()`** method **`inputs['oncoming']`** this is important because the agent is penalized when turning left with oncoming traffic as this can cause an accident.
- the left from the environment.sense method **`inputs['left']`** is required when making a right turn on the red light, so an accident can be avoided .

```
self.state = (self.next_waypoint, inputs['light'],
              inputs['oncoming'], inputs['left'])
```

The states I choose not to include are the right direction **`inputs['right']`** and the remaining steps until the **`deadline`** as part of my important learned states.

The agent is not penalized for making a right turn on red because it is a legal move. An oncoming car in the right direction has no impact on our learning agent and can be ignored.

I also choose not to include the deadline because it would make the Q learning table much larger, thus requiring more exploration and more trials to converge to an optimal policy. The agent should always try to make it to the end goal as soon as possible as directed by the planner. One potential downside of not including this states the agent could learn to intentionally not follow some of the driving rules and make it to goal right away if the agent is in

danger not reaching the end goal before the deadline depending on the anticipated reward / penalty tradeoff.

Task 3: 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.

After implementing the Q-Learning algorithm I noticed the agent preforms remarkably better as the number of episodes increase. During the first few episodes the agent makes mistakes similar to the random driving agent such as driving around in circles, but it quickly begins following the correct route guided by the **`next_waypoint()`** method. During the early episodes the agent does not follow the rules of the road, such as trying to go forward on a red light, or turning left into another car. As the agent progress through the episodes it will demonstrate a comprehension of the rules such as turning right on red and waiting at the stop lights when it needs to move forward.

Task 4 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.

We can see the performance of Q-learning agent has improved drastically as compared to the random action algorithm. Typically the agent will make some mistakes in the early trials but quickly learns from the mistakes; however the agent will still occasionally makes a mistake in the later trials. I believe these mistakes are due to randomly encountering an unexplored state. In the early trials the agent is also more likely to choose random actions, for exploration due to a decaying epsilon value rather than exploiting the best action from the Q table lookup.

This decaying works by creating a decreasing a value as the number of trials completed increases:

```
def get_epsilon_decay(self):
     '''decays epislon value over time. Used to promote exploration in
        earlier episodes.'''
     return float(self.epsilon) / (len(self.trial_rewards) +
                                        float(self.epsilon))
```

The agent is provided a seeded epsilon value which is initialized on the agent's instantiation. This value is a reciprocal inverse of the trial total number of trials we have encountered. Thus as the number of trials increase the value of the decayed epsilon will decrease.

At each step in the trial this decayed epsilon is then compared to a randomly generated value between 0 and 1.

```
def choose_action(self, state):
  '''decide if we should take a random action or a greedy action.'''
  epsilon = self.get_epsilon_decay()
  if random.random() < epsilon: #Roll the dice
      action = self.random_action() #Used for exploring the environment
  else:
     action = self.greedy_action(state)#Used for exploiting the environment
  return action
```

If the randomly generated number is greater than the decayed epsilon value, the agent will choose to explore the environment by selecting a random action. If the random number is less than decayed epsilon value the agent will choose to exploit the environment by taking a greedy action. As the number of trials increase the likelihood of the choosing a random number larger than the the decayed_epsilon value diminishes over time which encourages the agent to exploit the Q-Learning table rather that explore the environment.

The random state is as described above, it simply looks at our array of possible actions and selects one for exploration.

```
self.directions = [None, 'forward', 'left', ‘right']
…
def random_action(self):
    random_direction = random.choice(self.directions)
    return random_direction
```

The greedy action examines all the previous outcomes given a state and returns the direction which will yield the maximum reward. If there are multiple

actions with the same maximum value given the state, or if the agent has not yet encountered any of the actions (previous rewards are 0) the agent then randomly selects from the list of maximum values.

```
def greedy_action(self, state):
    '''returns the actions with the best known q value for a given state.'''
    q_vals = [self.get_Q_val(state, action) for action in self.directions]
    max_q_val = max(q_vals)
    if q_vals.count(max_q_val) > 1: #mult existing values
        best = [i for i in range(len(self.directions)) if q_vals[i] == max_q_val]
        direction_idx = random.choice(best)
    else:
        direction_idx = q_vals.index(max_q_val)
    return self.directions[direction_idx]
```

While I was developing the agent, I originally tried a non decaying epsilon value but found that the agent was more likely to choosing a random action in later stages when the agent has more of an idea of the best action to take.

To further tune the agent's performance after implementing the Q-Learning algorithm I tried adjusting the weighted parameters: Learning Rate, Discount factor, and Epsilon values to monitor changes in performance. These parameters are used to tune the rate at which new data overrides old data, the importance of future rewards, and exploration vs exploitation, respectively. Below is a table showing the performance of the agent with different combinations of Learning Rate and Discount.

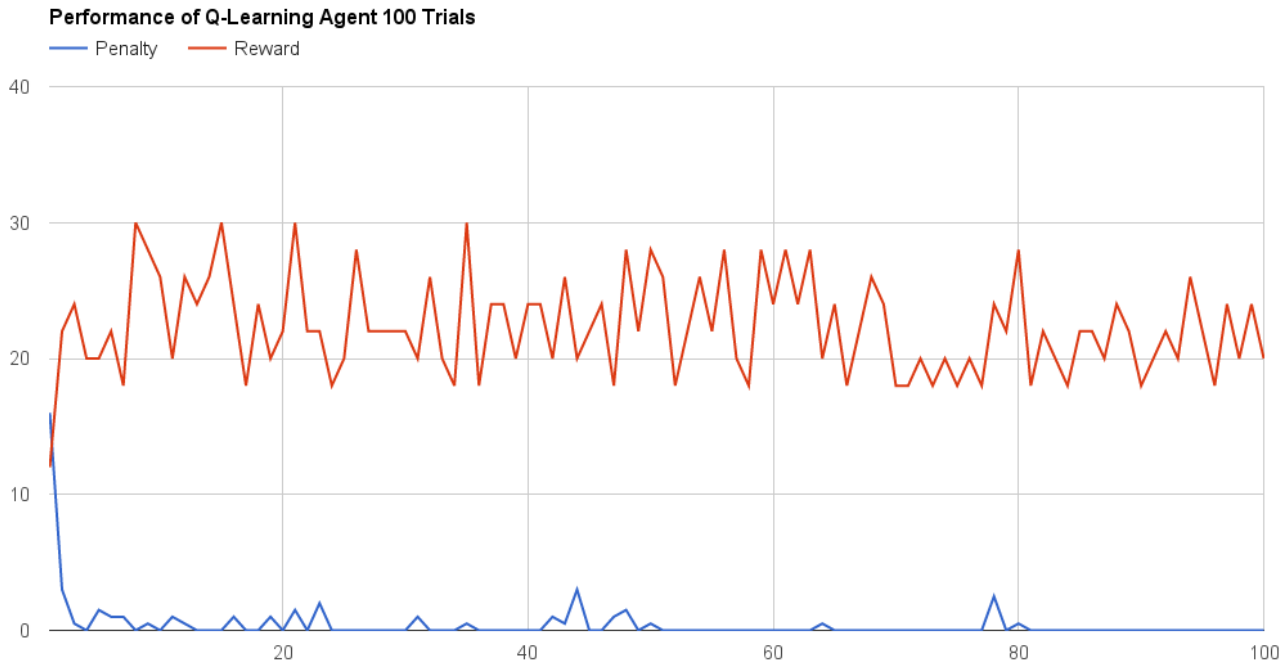
Q-Learning Agent Performance

Learning Rate	Discount Factor	Epsilon	Pass Rate	Explored States	Mean Reward	Failed Trial
0.1	0.9	0.5	92%	68	23.98	0,27,45,57,60,62,84,86
0.2	0.8	0.5	94%	69	25.855	0,27,33,35,47,87
0.3	0.7	0.5	97%	54	22.165	2,3,52
0.4	0.6	0.5	99%	46	23.07	0
0.5	0.5	0.5	99%	63	21.605	0
0.6	0.4	0.5	99%	56	21.89	0
0.6	0.4	0.001	98%	61	22.82	0,4
0.6	0.4	1.0	97%	68	22.465	0,9,49
0.7	0.3	0.5	98%	49	22.4	0,2
0.8	0.2	0.5	99%	53	22.95	0
0.9	0.1	0.5	98%	55	22.16	0,2

As the number of trials increase we can see the agent's policy converges toward an optimal policy; however the random movements of the other agents in the environment creates a stochastic environment in which every state is not explored during the 100 trials. In order to track the progress of my agent I created a 2-Dimensional array to analyze the Penalty**[0]** and Reward**[1]** values of each trial for 100 trials. Here is an example of the results using a Learning Rate of 0.6, Discount Rate of 0.4 and Epsilon of 0.5:

[[-16.0, 12.0], [-3.0, 22.0], [-0.5, 24.0], [0, 20.0], [-1.5, 20.0], [-1.0, 22.0], [-1.0, 18.0], [0, 30.0], [-0.5, 28.0], [0, 26.0], [-1.0, 20.0], [-0.5, 26.0], [0, 24.0], [0, 26.0], [0, 30.0], [-1.0, 24.0], [0, 18.0], [0, 24.0], [-1.0, 20.0], [0, 22.0], [-1.5, 30.0], [0, 22.0], [-2.0, 22.0], [0, 18.0], [0, 20.0], [0, 28.0], [0, 22.0], [0, 22.0], [0, 22.0], [0, 22.0], [-1.0, 20.0], [0, 26.0], [0, 20.0], [0, 18.0], [-0.5, 30.0], [0, 18.0], [0, 24.0], [0, 24.0], [0, 20.0], [0, 24.0], [0, 24.0], [-1.0, 20.0], [-0.5, 26.0], [-3.0, 20.0], [0, 22.0], [0, 24.0], [-1.0, 18.0], [-1.5, 28.0], [0, 22.0], [-0.5, 28.0], [0, 26.0], [0, 18.0], [0, 22.0], [0, 26.0], [0, 22.0], [0, 28.0], [0, 20.0], [0, 18.0], [0, 28.0], [0, 24.0], [0, 28.0], [0, 24.0], [0, 28.0], [-0.5, 20.0], [0, 24.0], [0, 18.0], [0, 22.0], [0, 26.0], [0, 24.0], [0, 18.0], [0, 18.0], [0, 20.0], [0, 18.0], [0, 20.0], [0, 18.0], [0, 20.0], [0, 18.0], [-2.5, 24.0], [0, 22.0], [-0.5, 28.0], [0, 18.0], [0, 22.0], [0,

```
20.0], [0, 18.0], [0, 22.0], [0, 22.0], [0, 20.0], [0, 24.0], [0, 22.0], [0,
18.0], [0, 20.0], [0, 22.0], [0, 20.0], [0, 26.0], [0, 22.0], [0, 18.0], [0,
24.0], [0, 20.0], [0, 24.0], [0, 20.0]]
```



This chart above shows the penalty vs reward the agent incurs over a number of trials. We can see as the agent progresses through more trials it makes fewer mistakes and gains more reward. When we are at the first trial we can see the reward value is quite low (12.0) and the penalty value is quite high (-16.0) when compared to the later trials. As the agent progresses through the trials the agent we see the errors quickly converge to 0 and the rewards converge to approximately 24.

In examining this agent I believe my agent is close to following an optimal policy which follows these three rules basic rules:

- The agent should obtain the largest possible cumulative reward for the trial.
- The agent should minimize the amount of penalty incurred through out the trail, follow the rules of the road and avoiding collisions with other agents.

- The agent would reach the goal state from the start state in as few time-steps as possible.

The agent I created is clearly is maximizing the total reward and incurring a minimal amount of penalties over time. While monitoring the agent I did notice some peculiar behaviors which is a deviation from the last bullet point. When stopped at a red light the agent should remain stopped but occasionally will loop around the block in order to maximize it rewards, this creates extra time steps and is a contradiction to the behavior of ideal policy as outlined in the third bullet point.