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.

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 most 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 I chose 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 when making a right turn on red.

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

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 still occasionally makes a mistake in the later trials. I believe these mistakes are due to randomly encountering unexplored states. In the early trials the agent is more likely to choose random actions, for exploration due to the decaying epsilon value rather than exploiting the best action from the Q table lookup.

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 Rate	Epsilon	Pass Rate	Explored States	Mean Reward	Failed Trial
0.1	0.9	0.5	99.0%	48	22.54	20
0.2	0.8	0.5	98.0%	56	21.665	1,6
0.3	0.7	0.5	98.0%	58	22.19	1,2
0.4	0.6	0.5	99.0%	55	22.33	3
0.5	0.5	0.5	99.0%	53	23.005	0
0.6	0.4	0.5	99.0%	46	21.58	0
0.6	0.4	0.001	98.0%	52	21.76	0,86
0.6	0.4	1.0	95.0%	55	22.45	0,1,2,30,32
0.7	0.3	0.5	98.0%	48	21.605	0,81
0.8	0.2	0.5	96.0%	66	21.85	0,52,63,74
0.9	0.1	0.5	99.0%	56	22.395	0
1.0	1.0	0.5	98.0%	47	21.115	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:

[[-9.0, 6.0], [-3.5, 22.0], [-2.5, 22.0], [-0.5, 24.0], [0, 20.0], [0, 18.0], [0, 24.0], [-1.0, 22.0], [-1.0, 20.0], [-0.5, 20.0], [0, 20.0], [0, 20.0], [0, 18.0], [-2.0, 26.0], [0, 20.0], [0, 26.0], [-0.5, 24.0], [0, 22.0], [-1.5, 34.0], [0, 18.0], [0, 24.0], [0, 26.0], [0, 22.0], [0, 20.0], [-0.5, 32.0], [-2.5, 26.0], [0, 18.0], [0, 22.0], [0, 26.0], [0, 22.0], [-1.0, 22.0], [0, 20.0], [0, 28.0], [0, 22.0], [-0.5, 26.0], [0, 20.0], [0, 22.0], [-1.0, 24.0], [0, 30.0], [0, 18.0], [0, 18.0], [0, 18.0], [0, 24.0], [0, 20.0], [-2.0, 24.0], [-1.0, 28.0], [-1.5, 26.0], [-1.0, 22.0], [0, 26.0], [0,

```
24.0], [-0.5, 20.0], [0, 18.0], [-2.5, 26.0], [0, 18.0], [0, 20.0], [0,
26.0], [0, 22.0], [-1.0, 24.0], [0, 24.0], [0, 20.0], [0, 22.0], [0, 28.0],
[0, 20.0], [0, 28.0], [0, 28.0], [0, 18.0], [0, 22.0], [0, 22.0], [0, 26.0],
[0, 20.0], [0, 18.0], [0, 18.0], [-1.5, 22.0], [0, 22.0], [-0.5, 20.0], [0,
22.0], [0, 26.0], [0, 24.0], [0, 26.0], [0, 20.0], [0, 20.0], [0, 26.0], [0,
26.0], [0, 28.0], [0, 22.0], [0, 18.0], [0, 18.0], [0, 20.0], [0, 26.0], [0,
18.0], [0, 24.0], [0, 20.0], [0, 20.0], [0, 24.0], [0, 24.0], [0, 18.0], [0,
22.0], [0, 18.0], [0, 20.0], [0, 30.0]]
```

This data shows that as the agent progresses through the trials it makes fewer mistakes **`[first value]`** and gains more reward **`[second value]`**.