Task1

Implement a Basic Driving Agent

Simply have the driving agent choose a random action from the set of possible actions (`None`, `'forward'`, `'left'`, `'right'`) at each intersection, disregarding the input information above. Set the simulation deadline enforcement, `enforce_deadline` to `False` and observe how it performs.

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?*

In agent.py I set the following

```
action = random.choice(Environment.valid_actions)
```

This makes the agent take a random action from one of the 4 available actions – None, forward, left, right.

I added a variable called success_rate that measures how many times the agent reaches the destination. This is seen to be around 65%. However, it doesn't follow the traffic rules like in this case, it made a left turn on a red light.

The results for 100 trials are in task1.txt

```
LearningAgent.update(): deadline = 3, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'right', 'left': None}, action = left, reward = -1.0
```

Also some other times, it doesn't take any action although the light is green and there is no oncoming traffic. I also noticed that the oncoming traffic is in general not toggling mostly in the simulation.

```
LearningAgent.update(): deadline = 11, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint': 'left', 'left': None}, action = None, reward = 0.0
```

There is also no learning feedback based on its actions and the agent has no incentive or reward to take the right action, hence not an optimal solution.

When I enforce the deadline, the success rate falls to 24% as expected (refer task1a.txt) and we can use this as a benchmark for the remaining tasks.

Task 2

Inform the Driving Agent

I decided to use inputs light and oncoming traffic along with the next_waypoint as the set of states for the learning agent. Next_waypoint is the next suggested action by the routeplanner. However the learning agent can use this information along with some of its own decision making to maximize its reward, learn faster. The inputs light and oncoming,right,left traffic is essential to ensure the smartcab takes the right actions so it doesn't meet with accidents and also follows the US right of way rules. I noticed that in this simulation oncoming, right and left rarely changes, so it would have been ok to leave this out from the state, but I kept it anyway. I didn't consider deadline as there are no rewards for arriving early and wasn't worried about changing the course of action because of deadline approaching. I would rather have the agent make the right action irrespective of the deadline than take a wrong action pressurized by the deadline although we could use deadline to maximize rewards.

```
self.next_state = inputs
        self.next_state['next_waypoint'] = self.next_waypoint
        self.next_state = tuple(sorted(self.next_state.items()))
```

Task3

Implement a Q-Learning Driving Agent

The basic gist of Q-learning is that you have a representation of the environmental states s, and possible actions in those states a, and you learn the value of each of those actions in each of those states. Intuitively, this value, Q, is referred to as the state-action value. So in Q-learning you start by setting all your state-action values to 0 (this isn't always the case, but in this simple implementation it will be), and you go around and explore the state-action space. After you try an action in a state, you evaluate the state that it has led to. If it has led to an undesirable outcome you reduce the Q value (or weight) of that action from that state so that other actions will have a greater value and be chosen instead the next time you're in that state. Similarly, if you're rewarded for taking a particular action, the weight of that action for that state is increased, so you're more likely to choose it again the next time you're in that state. Importantly, when you update Q, you're updating it for the previous state-action combination. You can only update Q after you've seen what results.

Q learning equation

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

Q-learning implementation can be found in Qagent.py and has the following steps

1. The current state considers next_waypoint and the inputs light, oncoming traffic, left and right
2. The qlearning agent take an action calling the next_action function
3. The next_action function uses an epsilon greedy approach. I set epsilon to 0.05. Using this policy either we can select random action with epsilon probability or we can select an action with 1-epsilon probability that gives maximum reward in given state. This is one of the suggested ways to trade off exploration and exploitation.
   To determine the maxQ value, we look at the Qtable that is implemented as a dictionary

   ```
   def getQvalue(self, state, action):
           key = (state, action)
           return self.qTable.get(key, 12.0)
   ```

4. We calculate the reward for this action
   next_reward = self.env.act(self, action)
   We now update the QTable for this current_state, action, next_state, reward combination based on the reward calculated and the discounted estimated future value. I initialized the Qtable values to 12 as this was the maximum reward I noticed in the random experiment. (The most extreme optimism in the face of uncertainty would be to initialize all of your initial Q values to a value *higher* than *any other Q values you might encounter).* No matter what, your Q value for encountered state action pair will be lower than your Q values for unencountered state action pairs

5. Make next_state as current_state and repeat this until current_state becomes the end state

With this setup, I got a success rate of 97,99,98% over 3 runs of 100 trials each stored in task3.txt, task3a.txt,task3b.txt for alpha=0.9, gamma=0.2,epsilon=0.05

The smartcab execute with an optimal policy and almost gets to the destination most number of times way ahead of the last few trials, with enough deadline to spare.

Task4

Improve the Q-Learning Driving Agent

Learning Rate – Aplha
0<apha<=1 value of 0 indicates the agent doesn't learn and 1 indicates it learns the most from the newly acquired information. Like expected, for higher values of alpha closer to 1, the success rate increases.

Discount rate - Gamma
I read an interesting discussion on the forums as to how in this project, discount rate need not be required as the current reward and not the future rewards play an important role. However we cannot disregard the fact that future rewards information has little correlation on the actions and hence gamma value needs to be low. Future rewards in regards to traffic lights and conditions. It is true that the environment is non-deterministic and thus makes it harder for the agent to properly learn the rules of the road. I also observed that for higher values of gamma the reward is higher.

Epsilon Value
Epsilon value needs to be lower as we want the qlearning agent to take more policy actions than random actions which is the whole point of learning. However we still need the option of random to avoid circling of the smartcab.

Based on the experiments below, we can clearly see for the combination of alpha=0.9, gamma=0.2, epsilon=0.05, the smartcab performed very well. I also tried increasing gamma for this combination to 0.6, but although that increased the rewards, dropped the success rate to 83%, indicating we cannot ignore gamma. I also tried iterating the Qtable initial values from 5,12,25, but 12 seemed the best choice.

alpha	gamma	epsilon	success rate
0.1	0.6	1	19
0.2	0.6	0.8	32
0.3	0.5	0.7	30
0.4	0.4	0.5	51
0.5	0.3	0.3	76
0.7	0.2	0.1	93
0.9	0.2	0.05	97
0.9	0.1	0.05	96
1	0.1	0.05	97
1	0	0.05	94

Final output – task3e.txt

This shows which actions were random vs greedy, total time taken to reach the destination, success rate, total rewards.

Incorporating review feedback

I printed out the formatted Qtable whenever the qagent makes it to the destination. Please find the results in task3f.txt

I noticed some mistakes that the qagent is still making. The qagent gets a negative reward of -1 for taking a forward action on a red light.

```
inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'left',
'left': None}, action = forward, reward = -1.0
```

However I see that in the initial trials, the qagent makes a lot of these mistakes, as it learns towards the last few trials, it significantly reduces taking this action, however this isn’t eliminated completely.

In the final Q table I see that the Q value for this combination is -0.934 which indicates as per the algorithm, the agent will not pick this action as it’s a high –ve value.

```
((('left', None), ('light', 'red'), ('next_waypoint', 'left'), ('oncoming', None),
('right', None)), 'forward'): -0.9342220309530338,
```

This could possibly be improved by modifying the reward/penalty scheme to be more severe for taking a forward action on a red light. The qagent could do better considering the US right of way traffic rules.

Overall, I observed high positive values for legal actions that seem to match the suggested next_waypoint.

```
((('left', None), ('light', 'green'), ('next_waypoint', 'left'), ('oncoming', None),
('right', None)), 'left'): 2.4843328199199446,
```

We can also see how for the right actions, the Q value for that state,action pair increases by small amounts from the first iteration till the last and how for the wrong actions, the Qvalues keep decreasing and then take a –ve value.

References

https://discussions.udacity.com/t/resources-for-coding-mdp/394550

https://discussions.udacity.com/t/states-is-this-on-the-right-track/442737

https://discussions.udacity.com/t/next-state-action-pair/44902/1050

http://artint.info/html/ArtInt_265.html