Implement a Basic Driving Agent

When the agent's actions are chosen purely by random from a set of possible actions `[None, 'forward', 'left', 'right']`, it does eventually reach it given unlimited time. However, this is just a matter of chance, and it can take a really long time for the agent to reach its destination.

Inform the Driving Agent

Our `LearningAgent` is given the following information at each intersection:

- `self.next_waypoint`: The next waypoint location relative to its current location and heading.
- `inputs`: The state of the traffic light at the intersection and the presence of oncoming vehicles from other directions.
- `deadline`: The current time left from the allotted deadline.

However, we should not use all of them to build our agent state because that will make the state space too large thus our smartcab would need a very long time to learn. We must discard some of the above informations. Out of all the informations provided, I choose to ignore two of them.

- Deadline: Since the variable deadline has so many differetn values, including it to build agent states would make the state space enormous and make the learning really slow. Even though it holds some value(we want the agent to reach destination as soon as possible), considering we need to let the agent learn sufficently within 100 trials, deadline is ignored in my model.
- Traffic to the right: This doesn't affect how the smartcab excutes its action in all three different scenarios. (1)If traffic to the right is empty, clearly there is nothing to worry about. (2)If traffic to the right is proceeding straight or turning left, its traffic light must be green and therefore the light for the learning agent must be red. That means the smartcab can only turn right, then they won't interfere each other. (3)If traffic to the right is turning right, again the traffic doesn't affect our cab's action.

Then we build our model using following variables.

- The state of the traffic light: `['green', 'red']`
- Oncoming traffic: `[None, 'forward', 'left', 'right']`
- Traffic to the left: `[None, 'forward', 'left', 'right']`
- Direction of next waypoint: `['forward', 'left', 'right']`

So there are 96 states in total for our smartcab to learn in this environment. The deadlines in different trials could be from 20 to 50. Since we plan to run 100 trials, even though the smartcab reaches the destination before the deadline, the smartcab can at least learn over 1000 times. The

state space is reasonable and the smartcab should be able to learn and make informed decisions about each state.

Implement a Q-Learning Driving Agent

Q learning implementation

The way I build my Q table is that I include a state into the table and set the all Q values of that state to be 1 each time the agent senses a new state(this or next state), like shown below.

```
if not(self.state in self.Q.keys()):
    for action in self.env.valid_actions:
        self.Q[self.state][action] = 1.
```

The learning agent will try to take the optiaml action, which has the largest Q value out of all the actions in a state. However, if there exists more than one optimal actions, my agent would randomly choose one of them.

```
def get_optimal_action(self, state):

    actions = self.Q[state]
    optimal_actions = []
    maxQ = max(actions.values())

    # find all the actions that have the max Q value
    for key, value in actions.iteritems():
        if value == maxQ:
            optimal_actions.append(key)

    # if the optimal action is not unique, randomly choose one
    if len(optimal_actions) > 1:
        return random.choice(optimal_actions)
    else:
        return optimal_actions[0]
```

With all of these, I can use the value iteration formula to update my Q value table.

```
self.Q[self.state][action] = (1.-self.LR)*old + self.LR*(reward+self.DF*next_Q)
```

Where `self.LR` is the learning rate, `self.Df` the discount factor, `old` the old Q value for this state/action pair, `next_Q = self.Q[next_state][next_optimal_action]` is the Q value for the next state taking the optiaml action.

LearningAgent's behavior

The smartcab now consistently learn how to take actions to reach its destination. For the early trials, about 30, the cab sometimes run out of time and can't reach the destination before deadline. However, this trend changes in the later 50 trials. After updating the Q table for some time, the agent can take very smart actions and almost always reach the destination before the deadline.

Previously, the basic agent was choosing random actions at each step. It is now using the reward received after taking an action to update the Q value of that state/action pair. After enough trials, the Q values should converge to reveal a reasonable policy. This is consistent with the behavior observed above.

Improve the Q-Learning Driving Agent

In order to optimize the learning agent, I run the simulation with different combinations of the three parameters, learning rate, discount factor and exploration rate. The performance of each case was measured by an average success rate of 5 simulations. Since we mainly care about the later trials, the success rates were calculated in the last 25 trials.

The results are shown in the table below.

```
           |-------------------------------------------------------------|
           | 5% ExplorationRate | 10%ExplorationRate | 15%ExplorationRate |
           |--------------------|--------------------|--------------------|
           |LR=0.7|LR=0.8|LR=0.9|LR=0.7|LR=0.8|LR=0.9|LR=0.7|LR=0.8|LR=0.9|
|----------|------|------|------|------|------|------|------|------|------|
| DF = 0.4 | 96.8 | 96.0 | 98.4 | 93.6 | 95.2 | 94.4 | 88.0 | 92.0 | 87.2 |
| DF = 0.5 | 88.0 | 95.2 | 95.2 | 85.6 | 95.6 | 88.0 | 75.2 | 84.8 | 80.0 |
| DF = 0.6 | 89.6 | 91.2 | 93.6 | 82.4 | 80.8 | 84.0 | 78.4 | 75.2 | 75.2 |
|-------------------------------------------------------------------------|
```

Based on the results, I believe that the best combination of the parameters is 5% exploration rate, 0.9 learning rate and 0.4 discount factor. In this case, 98.4% success rate means that our smartcab almost always reaches its destination before the deadline.

The final Q value table of the optimal parameter combination after 100 trials is shown below. The states are built using a tuple of waypoint, light status, oncoming traffic status, left traffic status.

```
(forward, red  , None   , None   ):forward:-1.00,right: 0.73,None: 0.00,left:-1.00
(forward, red  , None   , forward):forward:-0.44,right: 0.51,None: 0.46,left:-0.44
(forward, green, None   , left   ):forward: 1.90,right: 0.05,None: 0.46,left: 0.75
(forward, red  , left   , None   ):forward:-0.44,right:-0.19,None: 0.46,left:-0.44
(right  , red  , None   , None   ):forward: 0.08,right: 3.14,None: 1.26,left: 0.14
(forward, green, forward, None   ):forward: 1.90,right: 1.00,None: 1.00,left:-0.44
```

```
(right  , green, left   , None   ):forward: 1.00,right: 1.00,None: 1.00,left: 0.67
(right  , green, forward, None   ):forward: 1.00,right: 3.02,None: 1.00,left: 1.00
(forward, green, right  , None   ):forward: 1.00,right:-0.35,None: 1.00,left: 1.00
(left   , green, None   , left   ):forward: 0.41,right: 0.74,None: 1.00,left: 1.00
(left   , green, None   , None   ):forward: 0.30,right: 0.64,None: 1.26,left: 2.76
(left   , green, None   , right  ):forward: 1.00,right: 1.00,None: 1.00,left: 1.00
(forward, green, None   , forward):forward: 3.17,right: 1.00,None: 1.00,left: 0.67
(right  , green, None   , forward):forward: 1.00,right: 3.02,None: 1.00,left: 1.00
(forward, red  , None   , left   ):forward: 1.00,right: 0.78,None: 1.00,left: 1.00
(right  , red  , forward, None   ):forward: 1.00,right: 1.90,None: 1.00,left: 1.00
(right  , red  , left   , None   ):forward: 1.00,right: 3.01,None: 0.46,left: 1.00
(forward, green, left   , None   ):forward: 2.62,right: 1.00,None: 0.46,left: 1.00
(left   , red  , forward, None   ):forward:-0.44,right: 0.41,None: 0.21,left:-0.44
(left   , red  , None   , None   ):forward:-0.94,right:-0.49,None: 0.00,left:-0.99
(forward, green, None   , None   ):forward: 3.14,right:-0.40,None: 1.27,left: 0.36
(left   , red  , left   , None   ):forward:-0.44,right:-0.35,None: 0.46,left: 1.00
(right  , green, None   , left   ):forward: 0.55,right: 1.00,None: 1.00,left: 1.00
(right  , green, None   , None   ):forward: 0.33,right: 2.35,None: 0.80,left: 0.76
(forward, red  , forward, None   ):forward: 1.00,right: 0.81,None: 1.00,left:-0.44
(forward, green, None   , right  ):forward: 1.00,right: 1.00,None: 1.00,left: 1.00
(left   , green, None   , forward):forward: 1.00,right: 0.55,None: 0.46,left: 1.00
```

These Q values haven't converged. Some of the Q values are still the initial value 1.0 and the cab haven't visited all the state/action pairs. So the policy given by these Q values should not be considered as an optimal one. In addition, as shown in the below table, our smartcab still incures many penalties. The table shows those moments when the smartcab get a negative reward during the last 25 trails. We can notice that even though the smartcab have very good performance(high success rate), it still makes a lot of mistakes in the latter stage of the 100 trails. That is to say, our smartcab still have potential for improvement.

```
iteration                        input                                  waypoint  action
-------------------------------------------------------------------------------------------
    77    {light:red  , oncoming:left,  right:None, left: None }, forward, right
    78    {light:green, oncoming:None,  right:None, left: None }, forward, right
    79    {light:green, oncoming:None,  right:left, left: left }, left,    right
    80    {light:green, oncoming:None,  right:None, left: None }, forward, right
    80    {light:red  , oncoming:None,  right:None, left: left }, forward, left
    81    {light:green, oncoming:None,  right:None, left: None }, forward, right
    84    {light:green, oncoming:None,  right:None, left: right}, forward, left
    86    {light:red  , oncoming:None,  right:None, left: None }, left,    right
    88    {light:red  , oncoming:None,  right:None, left: None }, left,    forward
    88    {light:red  , oncoming:None,  right:None, left: None }, left,    right
    88    {light:red  , oncoming:None,  right:None, left: None }, forward, forward
    89    {light:red  , oncoming:None,  right:None, left: None }, forward, left
    90    {light:red  , oncoming:None,  right:None, left: None }, forward, left
    91    {light:red  , oncoming:None,  right:None, left: None }, left,    right
    92    {light:red  , oncoming:None,  right:None, left: None }, left,    right
    92    {light:green, oncoming:None,  right:None, left: None }, forward, right
    92    {light:red  , oncoming:left,  right:None, left: None }, forward, left
    93    {light:red  , oncoming:None,  right:None, left: right}, forward, left
    95    {light:red  , oncoming:None,  right:None, left: None }, forward, forward
    96    {light:red  , oncoming:None,  right:None, left: None }, left,    right
    96    {light:red  , oncoming:None,  right:None, left: None }, forward, right
```

```
 96    {light:red  , oncoming:None,  right:None, left: None }, forward, right
 96    {light:red  , oncoming:None,  right:None, left: None }, forward, right
 97    {light:green, oncoming:None,  right:None, left: None }, forward, right
 97    {light:red  , oncoming:None,  right:None, left: None }, forward, right
 97    {light:red  , oncoming:None,  right:None, left: None }, forward, right
 97    {light:red  , oncoming:None,  right:None, left: None }, forward, right
 97    {light:green, oncoming:right, right:None, left: None }, forward, right
 99    {light:green, oncoming:None,  right:None, left: None }, forward, right
 99    {light:red  , oncoming:None,  right:None, left: None }, left,    right
```