Report for Project 4

1. **Realize Basic Functions of the Intelligent Cab**

Make the cab move in the environment, regardless of any problems about optimization in driving. Notice that your cab will provide the following information:

- Position of the next cross (according to the current position and forward direction);
- The state of the cross (red or green light, car existing or not);
- Current deadline (how much time left).

To complete this task, randomly choose one of the following movement for your cab. (None, forward, left, right) Regardless of the above information, set the deadline in the virtual environment, that is to set *enforce_deadline* as *False*, and watch the behavior of the intelligent cab.

Question 1: Watch and record the behavior of the intelligent cab when taking random actions. Does it finally achieve the destination? Are there interesting phenomena worth recording?

Answer 1: When I set action as random choice from None, "forward", "left" and "right", the cab will do as the choice, however, it takes long to reach the goal, because it just doesn't have order and waste lots of time looping around a same place. Also, I can see the reward is changing on the up left of the screen with its report of its deadline, inputs, action and reward in the command lines. Following is where I changed the code to make tests.

```
# TODO: Select action according to your policy
action = random.choice([None,"forward","right", "right"])
```

2. **Train the Intelligent Cab**

Now that your cab can move in the environment, your next task is to choose a suitable state to build models for the cab and its environment. State variables mainly come from the *inputs* of current road, but you do not need to show them all. You can explicitly define the states or input some implicit assembles. At every time node, you need to process input, using *self.state* to update its current state. Still, you should set *enforce_deadline* as *False* to observe how the cab report its state change.

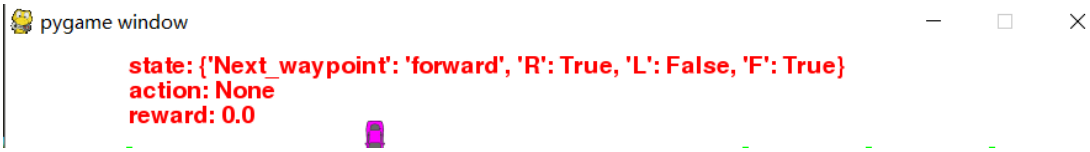
Question2: Which states do you think is suitable to build the model for cab and environment? Why do you think so?

Answer2: I think five states are important to evaluate the cab and environment: (1) the cab's location, (2) destination, (3)the cab's heading direction, (4, 5 and 6) the cab's actions for *forward, left* and *right* is safe or not. However, in order to reduce the number of states to train faster, I would more like to combine (1), (2) and (3) as the *next_waypoint*, which can very well represent the relationship between the three states and action choice. Thus, finally, I could choose the following states: (1) *next_waypoint*, (2) *forward_safe*, (3) *right_safe* and (4)

left_safe. If the cab's action makes it closer to the destination without accident, I think it will be positively rewarded, otherwise, if it does nothing or goes farther from the destiny it will be punished a little, but if it breaks the traffic rules or causes accident, it should be severely punished. Since cab's heading influences the choice of action, though it doesn't influence rewards directly, it should be considered. This reward mechanism is reasonable it real world and only considers these three states. Following is where I changed the code and the effect.

```
deadline = self.env.get_deadline(self)

inputs = self.env.sense(self)
self.next_waypoint = self.planner.next_waypoint()
light=inputs['light']
oncoming=inputs['oncoming']
left=inputs['left']
right=inputs['right']
forward_safe=light=='green'
right_safe=light=='green' or (light=='red' and not oncoming=='left' and not left=='forward')
left_safe=light=='green' and 'oncoming'==None
pro_inputs={'Next_waypoint':self.next_waypoint, 'F':forward_safe, 'R':right_safe, 'L':left_safe}
state=(self.next_waypoint, forward_safe, right_safe,left_safe)
self.state=pro_inputs
```



Optional Question: In this environment, how many states can the cab have? Is the number large enough for us to do Q-Learning for the cab? Tell the reason.

Answer: There are 3*2*2*2=24 states in this environment, where the 3 for 3 next way_points and the following three 2 for whether safe or not to go forward, right and left. I think it is large enough for us to do Q-Learning for the cab, since all the conditions are considered for the cab in those states.

3. **Realize Q-Learning Intelligent Cab**

When your intelligent cab is able to understand the input information, and has a map to state, your next task is to realize Q-Learning algorithm for your cab, to make it choose the best action according to current state and the Q values for the action. Every action of the cab will result in a reward based on the environment. Q-Learning cab should update the Q values in when need, considering those rewards. When finishing, set the environment variable *enforce_deadline* as *True*. Run the simulator and observe how the cab moves in every iteration.

Question 3: Compared with randomly choosing actions, what kind of changes do you find in your cab's actions? Why would those changes happen?

I think it definitely improved the rate of success, as I tested. And following is my test method.

1. Set *n_trials*=100 before running.

```
sim.run(n_trials=100)
```

2. Define a *success_time* in the environment and add 1 when succeed and print it after finish running 100 times.

```
class Environment(object):
    """Environment within which all agent
    valid_actions = [None, 'forward', 'le
    valid_inputs = {'light': TrafficLight
    valid_headings = [(1, 0), (0, -1), (
    hard_time_limit = -100  # even if en
    success_time=0
```

```
            self.done = True
            print "Environment.act(): Primary agent has reached destination!"  # [debug]
            self.success_time+=1
```

```
print e.success_time
```

3. Test random actions and result is 22:

```
action=random.choice(["forward","right","left",None])
```

```
 = forward,  reward = -0.5
Environment.step(): Primary agent ran out of time! Trial aborted.
22
```

4. According to trained Q-learning agent:

   4.1 Train

   ➢ Initialize state

```
deadline = self.env.get_deadline(self)

inputs = self.env.sense(self)
self.next_waypoint = self.planner.next_waypoint()
light=inputs['light']
oncoming=inputs['oncoming']
left=inputs['left']
right=inputs['right']
forward_safe=light=='green'
right_safe=light=='green' or (light=='red' and not oncoming=='left' and not left=='forward')
left_safe=light=='green' and 'oncoming'==None
pro_inputs={'Next_waypoint':self.next_waypoint, 'F':forward_safe, 'R':right_safe, 'L':left_safe}
state=(self.next_waypoint, forward_safe, right_safe,left_safe)
self.state=pro_inputs
current_state=state
```

   ➢ Get next state and reward

```
reward = self.env.act(self, action)

#Get next state
inputs = self.env.sense(self)
self.next_waypoint=self.planner.next_waypoint()
light=inputs['light']
oncoming=inputs['oncoming']
left=inputs['left']
right=inputs['right']
forward_safe=light=='green'
right_safe=light=='green' or (light=='red' and not oncoming=='le
left_safe=light=='green' and 'oncoming'==None
pro_inputs={'Next_waypoint':self.next_waypoint, 'F':forward_safe
state=(self.next_waypoint, forward_safe, right_safe,left_safe)
next_state=state
```

   ➢ Update Q value

```
self.Qlist.loc[state_dict[current_state],action]=(1-self.alpha)*self.Qlist.loc[state_dict[current_state],action]+self.alpha*(reward+self.gamma*self.Qlist.loc[state_dict[next_state]].max())
```

   Where in this formula, alpha=0.5, gamma=0.5

   4.2 Test

   ➢ Choose best action according to Q value

```
best_Q=self.Qlist.loc[state_dict[self.state]].max()
action_l=[]
for item in ["forward","left","right",None]:
    if self.Qlist.loc[state_dict[self.state],item]==best_Q:
        action_l.append(item)

action = random.choice(action_l)
```

   ➢ Result

```
 = forward,  reward = 12.0
99
```

The test above obviously confirmed my statement that choosing actions according to trained Q values is much better than randomly choosing actions. The reason is that I updated the Q values in the training process, which makes it much more reliable since it was based on experience. Moreover, the immediate reward and long term reward are both considered.

4. **Improve the Q-Learning Intelligent Cab**

In this project, your final task is to improve your intelligent cab, making sure that after enough training, it will achieve the destination in time and safely. Adjust alpha, gamma and epsilon to improve the success rate of your cab.

Question 4: Record the processing of adjusting perimeters, which is better? How good is it?
Answer4: Since my first try reached 99 success in 100 tries, I adjusted alpha and gamma a little, and set alpha as 0.6, gamma as 0.4, then in order to make the result more accurate, I tried 1000 times, but the result surprised me by presenting 1000 successful times, and most of them left more than 10 some even 20 in deadlines, so I think it is already good enough.

Question 5: Do you think you have find the best strategy? For example, reach the destination in shortest time? Or did not get punished? How do you define best strategy?
Answer4: Yes, I think it is already good enough, as I mentioned in question 4. Here, I define best as reach the destination fast but not regardless of the punishment.