Train a Smartcab to Drive

Xiangwei Wang

1. Identify and update state

Firstly I change the code to be the follow:

```
self.next_waypoint = self.planner.next_waypoint()
inputs = self.env.sense(self)
deadline = self.env.get_deadline(self)
action = None
ran=random.randint(1,4)
if ran == 1:
   actino = None
elif ran==2:
   action = 'forward'
elif ran==3:
   action = 'left'
elif ran==4:
   action = 'right'
else:
   action = none
reward = self.env.act(self, action)
```

where **self.next_waypoint** is the next waypoint location, *inputs* in intersection state and deadline is current deadline value. And they are all the inputs to our agent. For the output or the action If the agent, it is just random from none, ‘forward’. ‘left’ or ‘right’.

When running in similar, we found that agent just drive blind and random, not approaching to the target, and can not arrive the target within a short time(I observe for about 5min, and it does not make it), the reward is varied about form -1 to 2.

2. Justify why you picked these set of states, and how they model the agent and its environment.

I only choose **light condition** and **next_waypoint** as our robot states.

There are six independent state to choose model our agent:

1. next_waypoints, it is direction form global router which can direct agent to his target with the reasonable and shortest path, so we must choose it as a state if we want to be able to reach the target.
2. Light condition, it is the traffic light condition which can be right and green, agent must obey the rule or it can not move. So it is another necessary state to model.
3. Left, right, oncoming, the three values is the intend of the car froming three different

direction, for example, the oncoming is what the car in oncoming direction want to do, 'go forward', 'turn left', 'turn right' or 'stay there'. For general purpose, they should states for model robot, however for this problem, they are necessary. Because, in the reward function, I find that the reward do not take it in account. Even though we add them as states, the agent cannot learn something right by Q method.

4. Deadline, it is the left time for robot to get to target. For this question it is not as necessary as usual. Because agent will not learning something relative to Deadline by Q method as well. Because each available move will get a reward and no extra penalty for each move.

As a result, I only choose **light condition** and **next_waypoint** as our robot states.

```
  AvailableInformation=[('Next Way Point:', self.next_waypoint),
        ('light',inputs['light']),
        ('On Comming:',inputs['oncoming']),
        ('Left:',inputs['left']),
        ('Right:',inputs['right']),
        ('Deadline',deadline)
  self.state=AvailableInformation[0:2];
```

And because I do not change the action of agent now, so the agent still move randomly.

3. What changes do you notice in the agent's behavior?

3.1 Implement Q-learning

I implement a Q-learning by the following three step.

1. Firstly, intial a Q value table by ordereddict and the states include light condition and next-waypoint. When initialize I set all value to zero.
2. Update the Q-value after every move by the following equation

   $$\hat{Q}=\left(1-\alpha\right)*\hat{Q}+\alpha*\left(r+\sigma*\max_{a}\hat{Q}\left(s^{'},a^{'}\right)\right)$$

   Where alpha is the learning rate and lamda is the time discount and r is the reward of the action according to state before the action, and s' is the state after the action.
3. Get the best action according the Q-value and current state. Just choose the action which can get biggest Q value. If there two or more action with the same Q value, I choose the action from those actions by random.

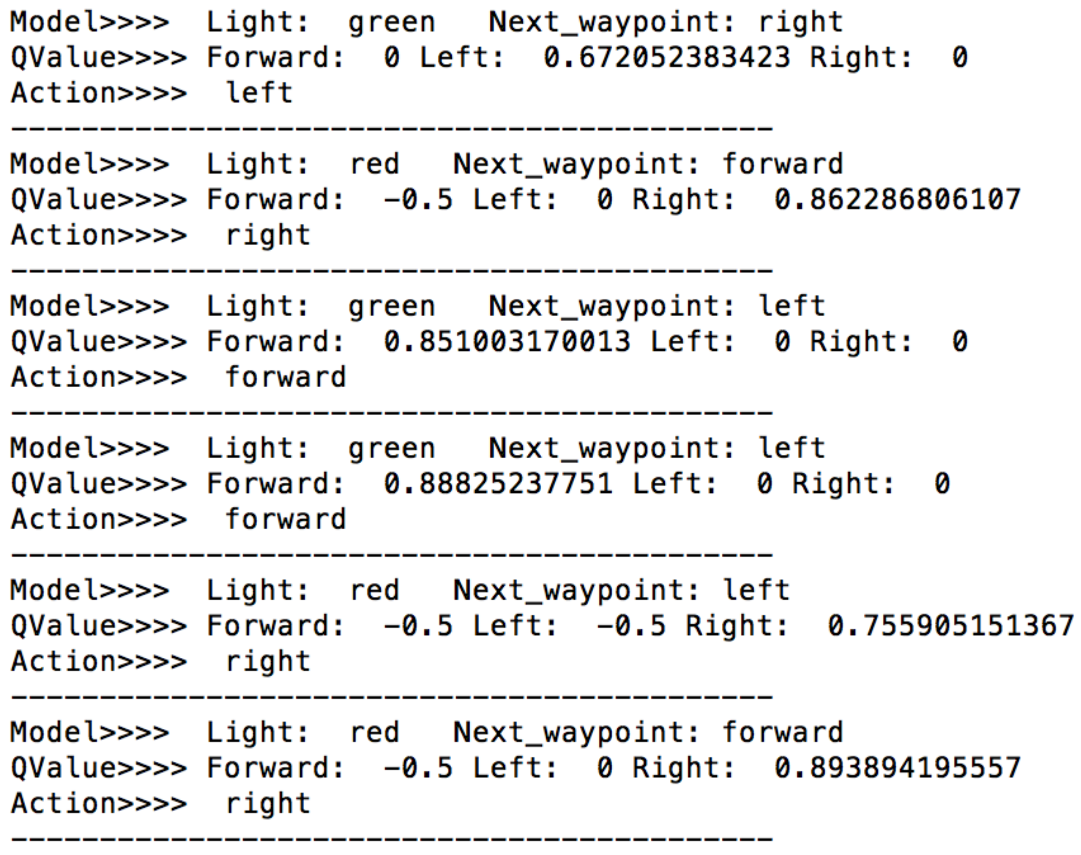
Note: for the first step, there is no Q value, so it is chosen by random.

3.2 Behavior Changes

After using Q learning, the agent no longer act by random, instead, he act by his current knowledge.

For example:

The following picture shows the agent's action. Model is the current state of the agent; QValue is the q value when agent choose different action; Action is the action agent choose. We can know that the action only depend on the QValue and current state. Agent learn how to move by Q learning.



```
Model>>>>  Light:  green   Next_waypoint: right
QValue>>>> Forward:  0 Left:  0.672052383423 Right:  0
Action>>>>  left
-------------------------------------------
Model>>>>  Light:  red   Next_waypoint: forward
QValue>>>> Forward:  -0.5 Left:  0 Right:  0.862286806107
Action>>>>  right
-------------------------------------------
Model>>>>  Light:  green   Next_waypoint: left
QValue>>>> Forward:  0.851003170013 Left:  0 Right:  0
Action>>>>  forward
-------------------------------------------
Model>>>>  Light:  green   Next_waypoint: left
QValue>>>> Forward:  0.88825237751 Left:  0 Right:  0
Action>>>>  forward
-------------------------------------------
Model>>>>  Light:  red   Next_waypoint: left
QValue>>>> Forward:  -0.5 Left:  -0.5 Right:  0.755905151367
Action>>>>  right
-------------------------------------------
Model>>>>  Light:  red   Next_waypoint: forward
QValue>>>> Forward:  -0.5 Left:  0 Right:  0.893894195557
Action>>>>  right
-------------------------------------------
```

However because agent the best action available from the current state based on Q-values, so he learn less and may stuck in local optimal. And it really does, because he always get reward if he turn right, so sometimes, he always turns right and move in a local circle.

4.Enhance the driving agent

4.1 improvement

We know that the bad performance before is the result of local optimal. So to enhance the driving agent, we choose to give him more chance to learn new things and we take two measures! Firstly, we make the agent 'only learn' and we just make it act randomly to meet more situation and learn more for the first 20 trial. Then after it, the agent randomly chooses act randomly or act following Q learning result by a possibility which can adjust.

4.2 Parameter adjustment

metrics: success rate, penalty times(after first 20 times)
performance with various random posibility

Parameter(possibility to random) lamda =0.5	Success rate	Penalty times
0	0.92	0
0.1	0.88	51

0.2	0.85	84
0.3	0.73	122

performance with various lamda

Parameter(lamda) P=0	Success rate	Penalty times
0.2	0.94	0
0.4	0.94	0
0.6	0.92	0
0.8	0.95	0

So we may choose the follow Q learning result after 20 times learning and the lamda matter little

4.3 Final Performance

Our agent after can reach an performance over an average 92% success rate(using over, because it can be better and better with more and more trials) and little penalty times.

So the best method is to learn first then use what you learn. For general purpose, we should also learn after the starting learn when use what we learn. Because they maybe some new or random thing in the real world and real problem. But for this problem, the model is simple, after the 20 times study, our agent learn enough knowledge for act in this world. Besides, no more new things will occurs for the problem. So the method with no random get best performance.

Reference

[1] OrderedDict not hashable http://stackoverflow.com/questions/15880765/python-unhashable-type-ordereddict
[2] MDP https://en.wikipedia.org/wiki/Markov_decision_process
[3] Udacity https://www.udacity.com/