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

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

   Where alpha is the learning rate and sigma 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.

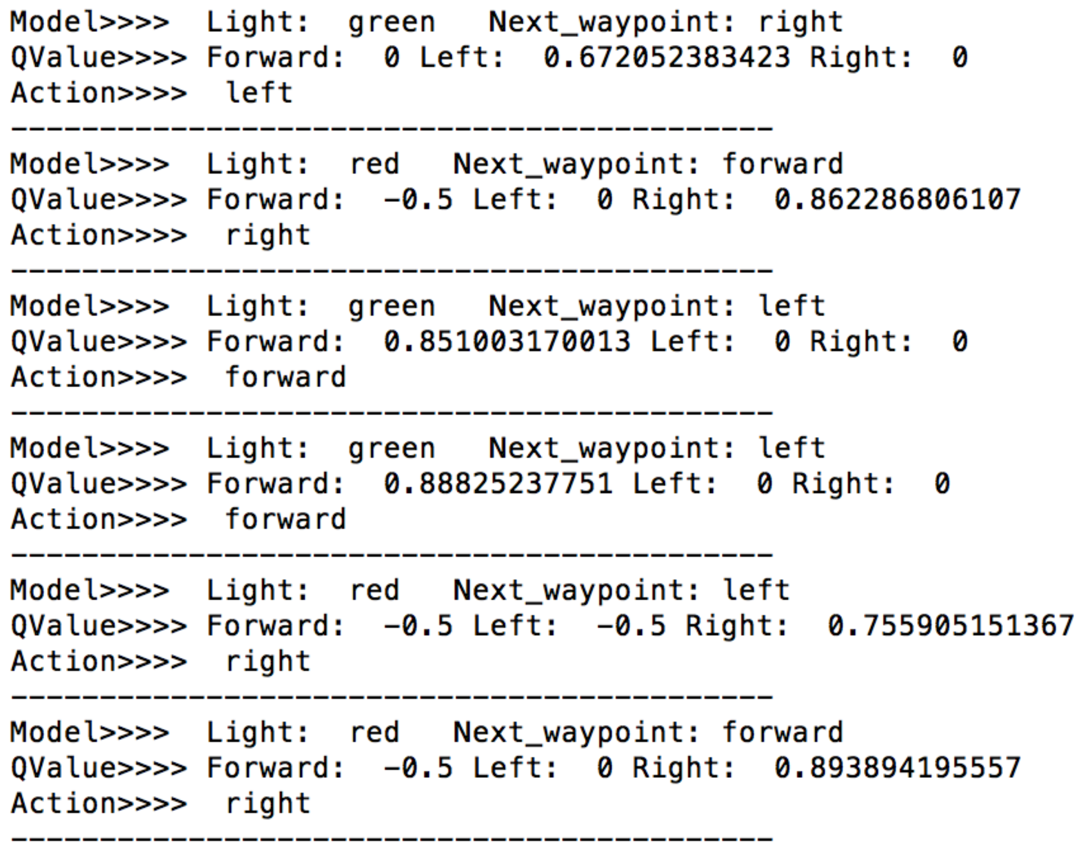
Note: for the first step, there is no Q value, so it is chosen by random.

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(need adjust) 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

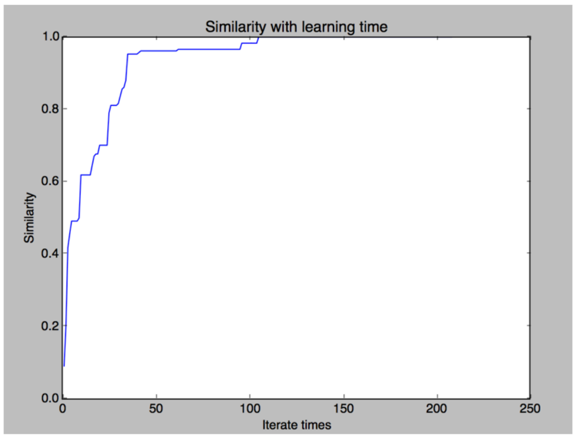
1. adjust learn times

We set learning rate as 1/t(a decay learning rate), let agent random move change reward discount sigma in [0 0.2 0.4 0.6], and identify the required learning time by he following method. Firstly, I defined an expected Q value matrix(may not the best Q value matrix) by
if act == None:

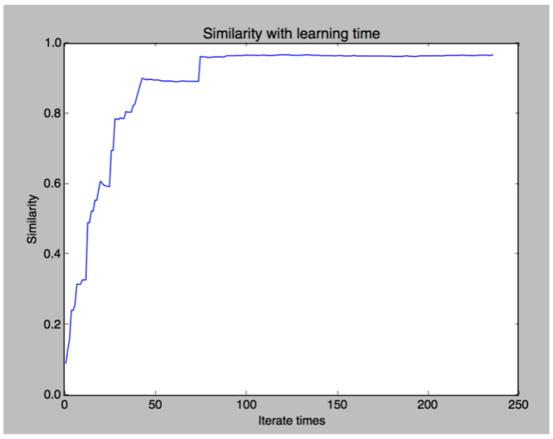
```
    self.q_value_ground_truth[(light_cond,next_waypoint_cond,act)]=1;
else:
    if light_cond == 'red' and act != 'right':
        self.q_value_ground_truth[(light_cond,next_waypoint_cond,act)]=-1;
    else:
        self.q_value_ground_truth[(light_cond,next_waypoint_cond,act)]= 2 if act == next_waypoint_cond else 0.5
```

and if using this Q matrix, agent will obey traffic rules and try to get to target as soon as possible.

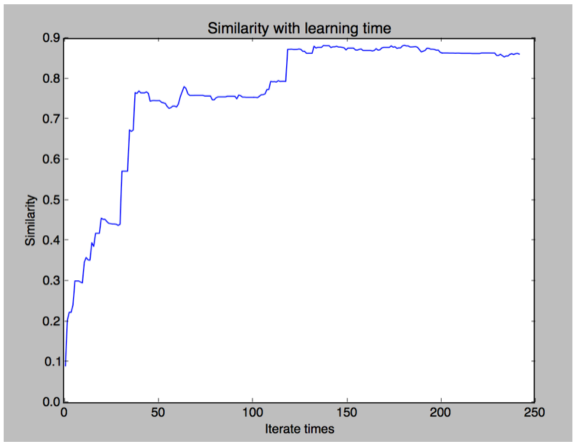
then I calculate the cos distance of each Q value when learning matrix with the expected Q value matrix by time. And got the following image.



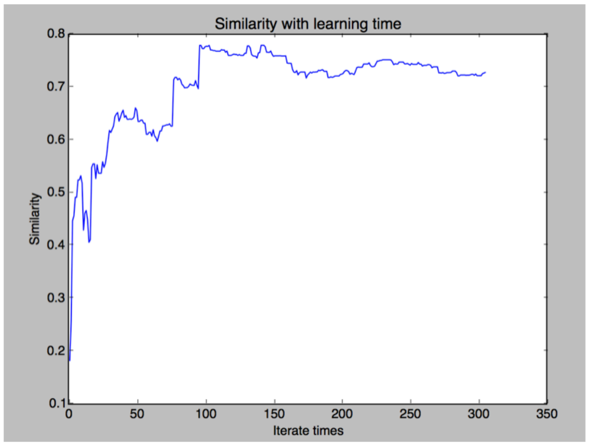
sigma=0



sigma=0.2



sigma=0.4



sigma=0.6

From the image, we can know that as the beginning, with the iterate time increase, performance increase as well, after iterated about 100~150 times, the performance still stable or decrease.

We know that for each trial, the learning time is as least 25, so we can choose the first 6 trials as learning trail. To be stability, we add 2 times and choose 8 trials as learning trail

metrics: success rate, penalty times(after first 8 times)
performance with various random posibility

Parameter(possibility to random) sigma =0.5	Success rate	Penalty times
0	1	0
0.1	0.98	41
0.2	0.91	87
0.3	0.88	128

performance with various sigma, possibility of choose random action is zero after the first learning trial

Parameter(sigma) P=0	Success rate	Penalty times
0.0	1	0
0.2	1	0
0.4	1	0
0.6	1	0
0.8	0.99	0

As a result, we may choose the follow Q learning result after 8 times learning and the sigma matter little. For any sigma in form 0 to 0.8 can make agent get a very good performance.

4.3 Final Performance
Our agent after can reach an performance of about 100% success rate and with almost zero penalty times.
As I mentioned above, a ideal policy should be

Light	Next_way_point	Action
red	right	right
red	else	None
green	x	x

In this way agent will never obey traffic rules and can reach the target with a relative short time.

When sigma is 0.5, our Q value matrix and policy is

Light	Next_way_point	Action
red	forward	**None 1.969** Forward -0.012 Left -0.031 Right 1.625
red	left	None 2.219 Forward 0.219 Left 0.219 **Right 2.438**
red	right	None 2.742 Forward 0.871 Left 0.871 **Right 3.742**
green	forward	None 2.468 **Forward 2.984** Left 2.242 Right 1.718
green	left	None 2.875 Forward 2.375 **Left 3.750** Right 2.242
green	right	None 2.969 Forward 2.436 Left 2.375 **Right 2.984**

From it we can know for most situation, the agent move correctly. The only different is when the traffic is red and the next way point is left the agent may go right, It may choose a longer path to target(extra 2 steps) but may also arrive on time of course and if it is fortunate with traffic light it can be faster then wait there. It will not stuck into right turn for many times.

When sigma is equal to zero

Light	Next_way_point	Action
red	forward	**None 1** Forward -1 Left -1 Right 0.5
red	left	**None 1** Forward -1 Left -1 Right 0.5
red	right	None 1 Forward -1 Left -1 **Right 2**
green	forward	None 1 **Forward 2** Left -1 Right 0.5
green	left	None 1 Forward 0.5 **Left 2** Right 0.5
green	right	None 1 Forward 0.5 Left 0.5 **Right 2**

It is the same with the expected policy, and the only weekness is that if the red never turn green, the robot will never move however it is not common.

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 8 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/