Udacity Proj4 - Reinforcement Learning – Training A Smartcab

Implement a basic driving agent

I updated agent's action as:

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

1. what you see in the agent's behavior. Does it eventually make it to the target location?

When I assigned random action for the larning agent to take, the red car choose random action among none, left, right, and forward. It deviates from the reccomandation from planner most of the time. Though it drives randomly, it still obeys the traffic rule. In 100 simulator trials, there are few trails that can reach destination with positive deadline; there are roughly 50% of the trials drive to destination in tens of time steps and 50% could not reach destination with deadline > -100.

Identify and update state

I updated agent's self.state as traffic light and next waypoint suggested by the planner:

```
self.state = (inputs['light'], self.next_waypoint)
```

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

A smartcab should make decision in its particular environment which consists of intersection state (lights and presence of cars), navigation state (which is waypoint suggested by path planner) and time state (how much time budget remianed to reach target). The information collected by the environment should be adequate for learner in the agent to learn and make decisions.

For traffic condition, I take traffic light into account. Traffic light state can help the agent learn to obey traffic light rules. I didn't include traffic condition at intersection of oncoming, left and right direction. The reason is to reduce the state space so that Q-learning can converge faster. Also there are few cars in the grid map so reward deduction due to traffic rule violation due to other intersection cars will be small.

For navigation state, I take next waypoint into account. Next waypoint is suggested by the planner. It not only provides a suggested min-distance-path navigation direction, but also indirectly provides information of agent car relative position w.r.t destination. With next waypoint as state, the agent will get reward when taking the suggested next waypoint and learn to follow the planner's suggestion whenever it can.

For completely modeling the environment, time state should be considered. In this work I did not consider deadline as a state because it complicates Q-learning state space and also an agent following next waypoint suggested planner is adequate for it to reach destination within time limit.

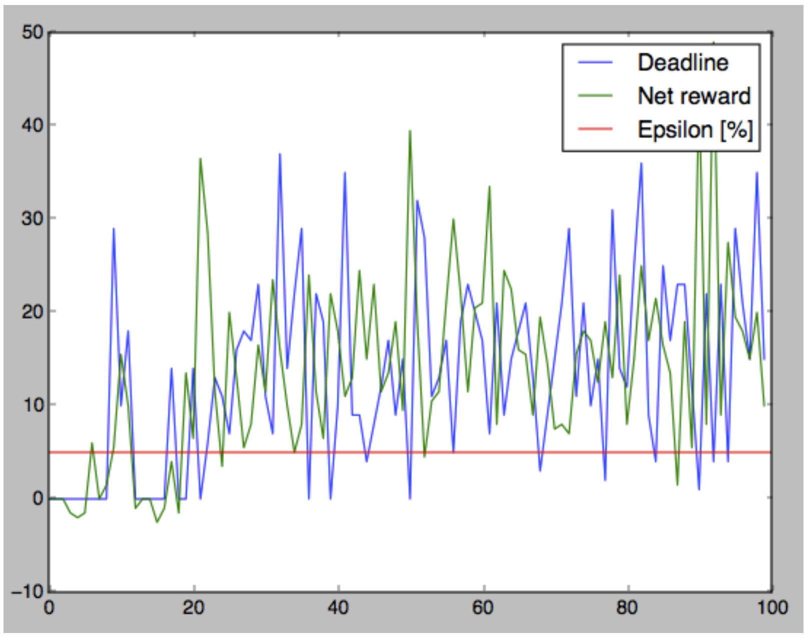
Implement Q-Learning

(a) The Q table is updated as
  i. The input argument is recorded in learning agents' lastState and state
    a) State contains intersection input, navigation state (waypoint suggested by path planner) and deadline.
    b) An additional argument in the 'lastState' records the action it took to come to current state.
  ii. Q table is initailized to be 0.
    a) The initialization is performed in fuction – self.queryQ()
  iii. Q(lastState with record of lastAction) ←Alfa← reward + Gamma * max{Q(currentState of AllActions)}
    a) Function carried out in self.learnDrive()
    b) y←Alfa←x means y = (1 – Alfa) * y + Alfa * x
    c) All actions in the current state (none, forward, left, right) are scanned to search for max Q
    d) Reward is recorded in self.reward
  iv. For each time step make decisions according to policy, update Q as rule iii.

(b) The decision is made by combination of exploration and exploitation.
  i. The function is performed in self.takeAction()
  ii. For each decision, make it explore with possibility of EPSON and take optimal action according to Q with (1-EPSON) possibility.
  iii. If explore, take random action among [none, forward, left, right]
  iv. If exploit, choose max Q of current state with all actions

<u>3. What changes do you notice in the agent's behavior?</u>

With learning rate ALFA = 0.8, Discount factor GAMMA = 0.8, Explore possibility EPSON = 0.05, initial Q values = 0, the following figure is achieved. The figure plots the deadline result (blue) and net reward (green) the learning agent achieved for each trial (X axis).



The change of agent's behavior is it has the trend to learn and improve it's own performance, net reward and deadline.

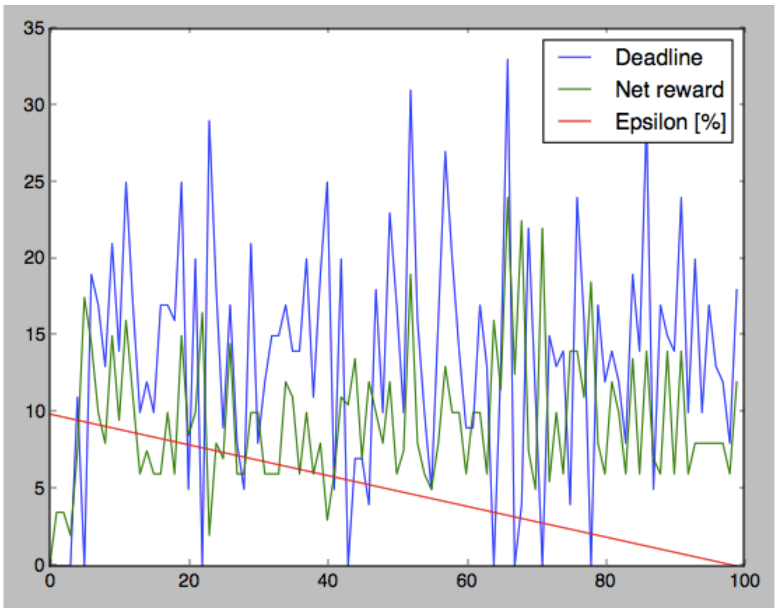
At the beginning, the agent does not know what is the right navigation direction and what is the right decision, its decision is almost random. It always went beyond deadline limit and got negtive reward. As trial number increases, the deadline number is increasing, indicating the smartcab is making more and more right decision to get destination faster. The net reward line also has a increase trend.
In the end, the agent makes much more right decisions then beginning and gets destination within time limit. However due to the explortion rate parameter EPSON = 0.05 and non-optimal parameter set, the agent sometimes makes a wrong random decision that leads to time limit violation.

Overall, the figures agent learning improvement of Q-learning implementation, while still not marginally meeting the requirement. The learning model needs to be improved and perhaps the rewarding rules need to be modified.

Enhance the driving agent

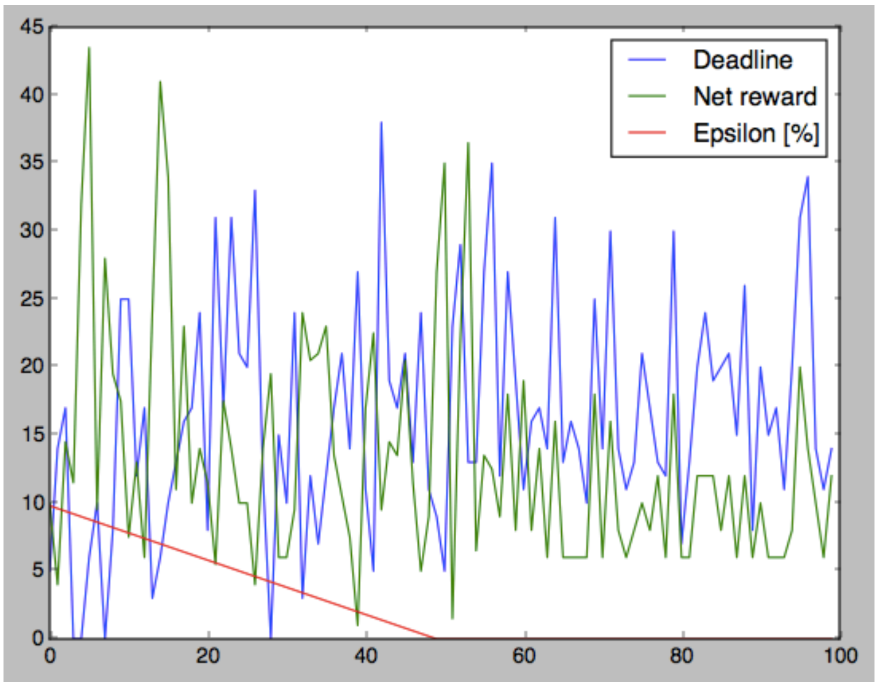
Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?

To get the best Q-learning performance, I applied epsilon decay and tried several parameter combinations:



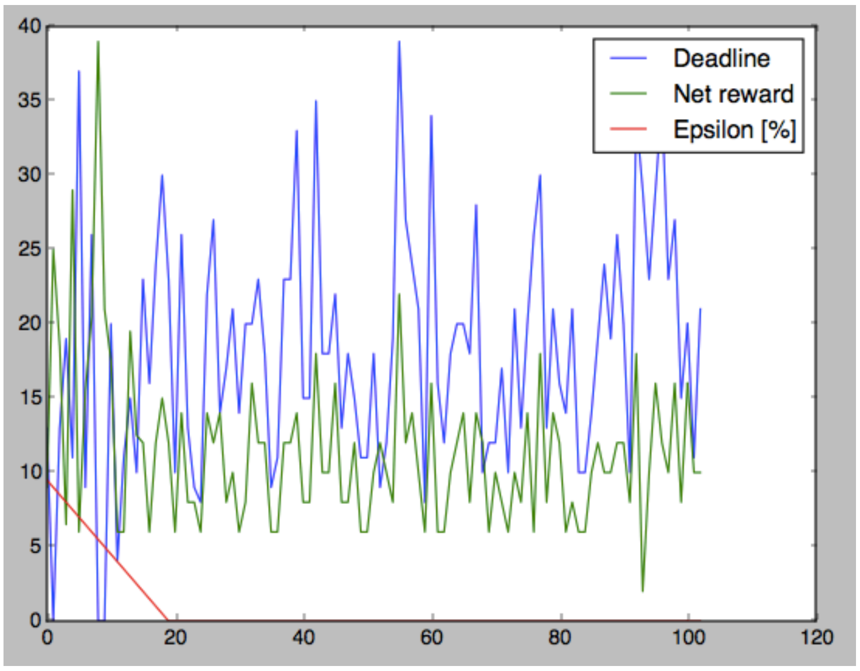
ALFA=0.3, GAMMA=0.8, EPSON=0.1, kEpson=-0.001, qInit=0

When Epsilon is large, high exploration rate let agent learn fast. When Epsilon becomes small in the end, the agent takes use of what it learned and accomplish job very well (within time limit and get positive reward).



ALFA=0.3, GAMMA=0.6, EPSON=0.1, kEpson=-0.002, qInit=2

The smaller GAMMA, faster Epsilon decay and larger qInit will help agent learn faster and achieve better performance.



ALFA=0.5, GAMMA=0.5, EPSON=0.1, kEpson=-0.01, qInit=2

Finally, the model with the searched optimal parameter set will learn fast and achieve 97% successful client delivery. As shown in the figure above, the red line is epsilon in percentage with respect to trial number:

- Before no. trial = 20, the epsilon drops from initial 0.1 at a rate of -1% per trial.
- After trial = 20, epsilon hits zero.

The code is implemented as:

```
epsilon = self.EPSON + self.kEpson * self.t
```

```
draw = random.random()
if draw < epsilon:
    #print "choose to explore"
    return random.choice(Environment.valid_actions)
else:
    #print "choose to use learned"
```

self.EPSON is the initial value of epsilon when trial number is zero. self.kEpson is the slope of epsilon with respect to trial number. I design this way because of the exploration and exploitation trade-off: If you only explore with random action, the agent learns by updating Q table with all the possible states, but the agent does not use what it learned and can not achieve good reward or performance. If you only take use of Q table you are not exploring all the possible state therefore Q table is not well updated --- the agent does not learn this way. So the best learning strategy is let the epsilon decrease as the agent is learning better and better.

At the beginning, the agent requires lots of exploration to learn and update Q table. Therefore I designed the agent to explore more to allow more learning. In the end, the agent uses a small epsilon to maximize performance according to what it has learned.

At the beginning of the game, the agent had not learn a effective policy and sometimes made random decision and did not meet time and intersection specifications. In the end (last 80 trials), the agent learned to obey the intersection rule --- it waits and stays when it desires to go along a direction but intersection rule does not allow it. And it also follows the direction that minimizes its path distance to destination.

4. Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties?

Yes it does find an policy that is close to optimal. Because it is aware of obeying intersection rules and going along with direction that minimizes its path distance to destination. The learning is motivated by the reward rule that drive the agent to learn this policy is in the code:

```
reward = 2.0 if action == agent.get_next_waypoint() else -0.5
```

as well as,

```
if not move_okey: reward = -1
```

so the model in the agent is motivated to follow the waypoint suggested by the planner as well as to obey the crosssection rule. Once it follows the planner, it is easy to get to destination within deadline.

The desired optimal policy shoud be more than that. When the agent gets an navigation direction suggestion but the intersection condition does not allow it to go, the agent waits and does not move. However, in this condition, the optimal policy is that the agent should check whether there's a secondary navigation direction that also minimizes path distance to destination while intersection rules allow it. To achieve this, the path planner should give two direction suggestions instead of one. Thus agent makes a second navigation choice when get deleyed by the intersection rule.