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:

self.state = (inputs, self.next_waypoint, deadline)

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. The information in collected by the environment should be adequate for learner in the agent to learn and make decisions.

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()

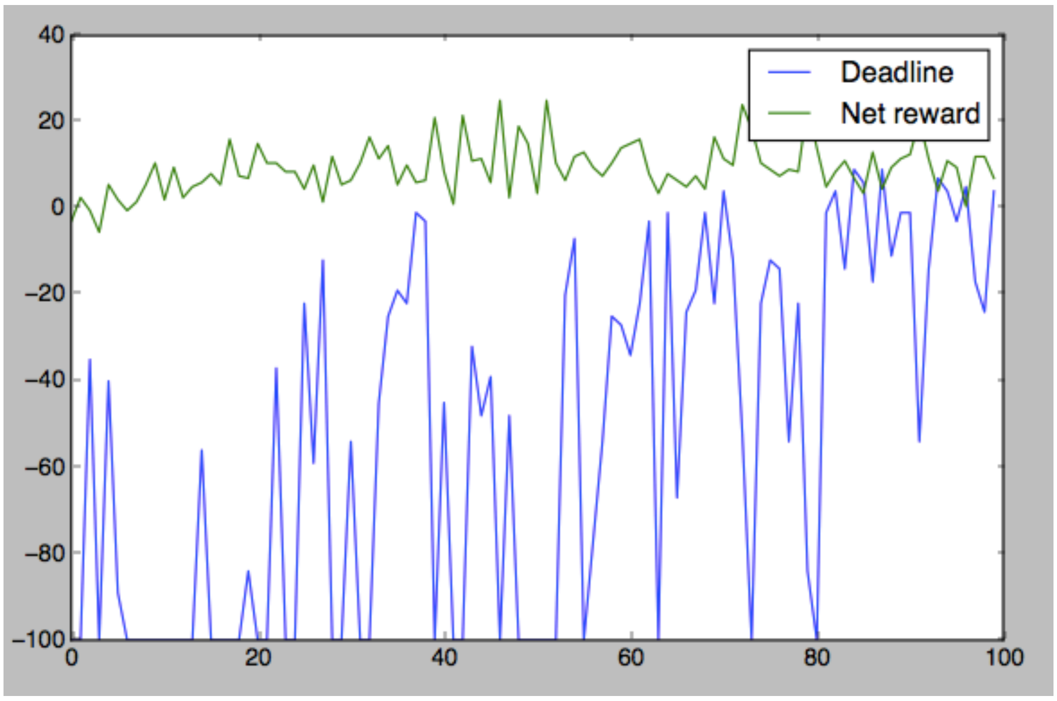
ii. For each decision, make it explore with possibility of EPSON and take optimal action according to Q with (1-EPSON) possibility.

iii. If explore, take random action among [none, forward, left, right]

iv. If exploit, choose max Q of current state with all actions

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

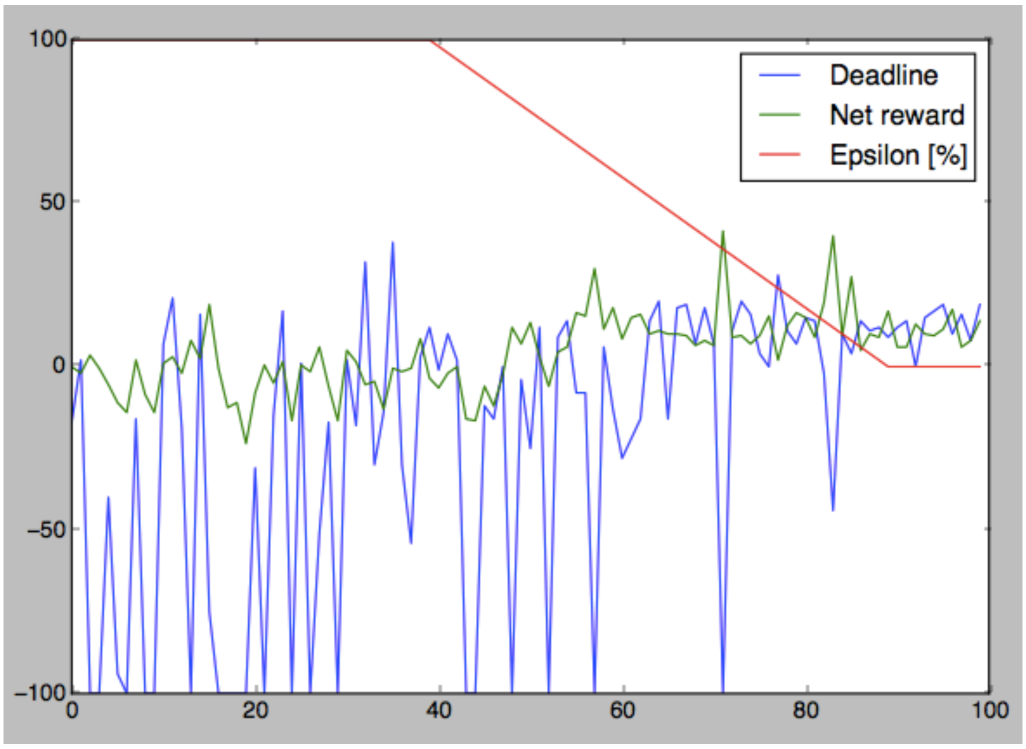
With learning rate ALFA = 0.06, Decaying rate GAMMA = 0.98, Explore possibility EPSON = 0.1, the following figure is achieved. The figure plots the deadline result (blue) and net reward (green) the learning agent achieved for each trial (X axis).



The change of agent's behavior is it has the trend to learn and improve it's own performance, net reward and deadline. The deadline number is increasing according to trial number, indicating the smartcab is making more and more right decision to get destivation faster. The net reward line has only a slight increase trend. Overall, the figures shows the effectiveness 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?



With learning rate ALFA = 0.15 and decaying rate GAMMA = 0.95, I have modified the exploration probability epsilon in each decision step. As shown in the figure above, the red line is epsilon in percentage with respect to trial number. Before no. trial < 40, it remains one. After no. trial > 40, the epsilon drops at a rate of -2% per trial. In the end, for the last 10 trials, epsilon hits zero. 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. In the end, the agent uses a small epsilon to maximize performance according to what it has learned.

As can be seen from the figure above. The deadline (blue line) is expriencing a nice improvement as more and more trials are performed. In the end (last 10 trials), it meets the requirement with deadline > 0 and net reward positive. The net reward (greed line) is also presenting a incresing trend and remain positive after trial no. > 60.

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?

The agent finds a nice policy but not an optimal policy. How well the agents learns is essentially dependent on reward.
The agent learns a nice policy in the sense that it accomplished the task within deadline after learning. It knows a way to go to destination. 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
```

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

But the action it takes ignores the intersection rules (because the planner is not aware of intersection rules). It should happen that once the agent obeys the intersection rule and made decision ‘none’ for stopping, it gets reward.

```
move_okay = True
if action == 'forward':
    if light != 'green':
        move_okay = False
elif action == 'left':
    if light == 'green' and (inputs['oncoming'] == None or inputs['oncoming'] == 'left'):
        heading = (heading[1], -heading[0])
    else:
        move_okay = False
elif action == 'right':
    if light == 'green' or (inputs['oncoming'] != 'left' and inputs['left'] != 'forward'):
        heading = (-heading[1], heading[0])
    else:
        move_okay = False
if not move_okay:
    reward = -1
```

But the rewarding rule in class Environment does not give such a reward or avoid penalty. Therefore the agent does not learn to obey the intersection rules.

The agents however does not find a optimal policy. This is mainly due to the planner, the path planner should give two direction suggestions instead of one. Therefore when agent make a second navigation choice when get deleyed by the intersection rule. The reward rule for intersection should be also modified so that the agent can learn to be aware of intersection condition and choose to obey the rule or choose a second navigation direction.