Training a Smart Cab to drive automatically with Q-learning/Reinforcement Learning

Summary

In this project we applied reinforcement learning techniques for a self-driving agent in a simplified world to aid it in effectively reaching its destinations in the allotted time. We first investigated the environment the agent operates in by constructing a very basic driving implementation. Once your agent was successful at operating within the environment, we then identified each possible state the agent can be in when considering such things as traffic lights and oncoming traffic at each intersection. With states identified, we then implement a Q-Learning algorithm for the self-driving agent to guide the agent towards its destination within the allotted time. Finally, we improved upon the Q-Learning algorithm to find the best configuration of learning and exploration factors to ensure the self-driving agent is reaching its destinations with consistently positive results.

QUESTION: Observe what you see with the agent's behavior as it takes random actions. Does the **smartcab** eventually make it to the destination? Are there any other interesting observations to note?
Answer: We implemented the random actions using the following single line of code and then using the action to act on in the Environment.py for the project.

```
## Random actions
action = random.choice(self.env.valid_actions[0:])
reward = self.env.act(self, action)
```

The car will eventually reach the destination by making random choices but not within the allocated deadline (If this were not the case then the value of Q-learning wouldn't hold up)

QUESTION: What states have you identified that are appropriate for modeling the **smartcab** and environment? Why do you believe each of these states to be appropriate for this problem?
Answer: We identified the following state elements:

self.state = (self.next_waypoint, in_light, in_oncoming, in_left)

where they take the following values to create all the states:

self.next_waypoint in ['right', 'left', 'forward', None]
in_light in ['red', 'green']
in_oncoming in ['forward', 'right', 'left', None]
in_left in [0, 1] #1 means 'forward', 0 means 'left' or 'right'

We initialize these variables as:

```
for self.next_waypoint in ['right', 'left', 'forward', None]:
    for in_light in ['red', 'green']:
        for in_oncoming in ['forward', 'right', 'left', None]:
            for in_left in [0, 1]:
                state = (self.next_waypoint, in_light, in_oncoming, in_left)
                self.q[state] = dict()
                self.q[state]['forward'] = 0.04
                self.q[state]['left'] = 0.03
                self.q[state]['right'] = 0.02
                self.q[state][None] = 0.01
```

To understand why these states are necessary and sufficient :

Learning legal Actions (or why we included inputs['light'], inputs['oncoming'], inputs['left'] as a state element))

1) We looked @ the code in the DummyAgent and looked at the control conditions that define the legal conditions of driving. The code under DummyAgent lists all the control conditions needed to take legal actions and thus these conditions (and states that define these conditions are sufficient).
We see that the

a) State of traffic lights
b) The actions that the oncoming traffic and,
c) The actions that the traffic on our left will take

to be sufficient in our environment to determine all the legal actions that our agent can undertake. Thus these state elements and all the values that they can take determine all the states that our agent needs to be aware of to learn the legal actions in our environment. In some cases some states can be combined as the state elements changes between them are don't care to learn the legal actions. For example: The traffic on our left " #1 means 'forward', 0 means 'left' or 'right' " we have combined a few states here.

Learning efficient Actions (or why we included self.next_waypoint as a state element)

2) Learning legal actions is not sufficient. We also need to be able to efficiently reach our destination. In order for our agent to learn properly we use the actions provided by the RoutePlanner as a guide and hence a state element. Our agent will learn quickly that following the recommendations from

the RoutePlanner will generally lead to big rewards.

Why didn't we include deadline as a state element ?

There are multiple reasons why deadline was not included as state element. Here are a few:

1) The first and foremost reason deadline was not added as a state was because it does not add any value. The Q-values we learn do not depend on the deadline. For example we learn that it is not ok to red light and this is reinforced by a negative reward over and over when our agent runs a red light. It is not ok to run a red light when you are close to running out time and follow the laws when there is still lot of time left. In other words our environment does not change the rewards based on the deadline (what is an illegal action stays illegal and gets a negative reward whether you are far or close to running out of time.) Thus the Q-value we would learn for all states that have ONLY differing values for deadline will be the same. Thus adding deadline increases the state space without adding any value.

2) Deadline can be used to inform/change the learning rate or the discount factor. For example assume our experiment was a single trial with a large deadline and a large grid (i.e the agent and destination are very far away.) Thus the agent can learn/explore when it still has a lot of time. This would allow it to visit/explore all its state space when it still has a lot of time and allow its to converge to stable Q-values and then become more careful when its closer to end of time. This can be achieved by decreasing the alpha or the gamma as time progresses close to the deadline.
Thus deadline has a role in the Q-learning algorithm but not in the state space.

OPTIONAL: How many states in total exist for the **smartcab** in this environment? Does this number seem reasonable given that the goal of Q-Learning is to learn and make informed decisions about each state? Why or why not?
Answer: An infinite number of states exist for the smartcab but not all of them are necessary. In our program we have used 4 * 2 * 4 * 2 = 64 states and these seem to be sufficient given the results of our program, the smart cab seems to reach its detonation within time allotted ~ 90% of the time. Considering an average deadline of 20 (this is based on intuition and not based on actual measurement from the program) per trial and 100 trials we have 2000 steps to visit 64 states or an equal probability of 32 steps for each state (Assuming independence of each state, which is clearly not true.) Thus it is highly likely each state will be visited once (this is easy to check @ the end of the program by dumping out the q-values which we haven't done yet).

QUESTION: What changes do you notice in the agent's behavior when compared to the basic driving agent when random actions were always taken? Why is this behavior occurring?

Answer:

It is hard for us to answer this question because we coded the final algorithm all at once instead of piece-meal but here is pseudocode for the algorithm we have implemented and the problems/errors we faced

```
initialize the Q-values for each state    # problem 1
pick a random previous state PS
pick a random previous action PA
pick a random previous reward PR
find new State S from inputs from the environment:
    pick the action A to take from the Q-table. A is the action that produces the maximum value for S in Q-Table
    with a small probability P change A to a random choice from valid actions    # problem 2
    perform action and get reward R
    find action A' that produces the maximum value for S in Q-Table. Use this value as MAX_Value
    update the Q equation
    Q[PS][PA] = (1 - alpha) * Q[PS][PA] + alpha * (PR + gamma * MAX_Value)    # problem 3
    PA = A
    PR = R
    PS = S
    continue
```

problem1: Initialize to floats so you don't have precision issues and have at least a small value so the control function doesn't always evaluate False at the beginning (cold start problem)

```
if (self.q[self.state][action_t] > max_action):
```

problem2: Make sure the probability control is correctly coded. i.e with only a small probability we are making a random choice.

problem3: This is the most important and insidious problem in that it is hard to understand/catch but surprising how much the results are affected if the wrong states are updated. Be sure to understand which state/action pairs need to be updated for Q-values. In particular we were updating the previous state (PS) and the current action (A) and getting bad results. Once we figured out that we need to save it using the previous action (PA) our results improved from about 37% accuracy to close to 95% accuracy (accuracy defined as agent making it to the destination within allotted time.) It took us some time before we realized we had to save all three PS, PA, PR to update our states. Finally find out a good and valid decay function for alpha. The current one we are using decays too slowly but still produces very good results.

Comparison between random and learning agent:

An agent that picked its actions at random ended up reaching its destination with very low probability of less than 5% (deadline enforced = True). It also picked many illegal state/action pairs and racked up many negative rewards along the way. A learning agent was much better at reaching the destination and picking state/action pairs that were both efficient and legal. What is surprising is how quickly an agent that is learning learns the correct policy. For example with an accuracy of 95% the agent has to have come close to learning the optimal policy within the first 5 trials (though it learnt it even quicker than that because even some of the first 5 trials ended up as success)

The meat of the code:

```
self.state = (self.next_waypoint, in_light, in_oncoming, in_left)

# TODO: Select action according to your policy

## Find the action that has the max q-value in the current state
action = None
max_action = 0.0
for action_t in self.q[self.state]:
    if (self.q[self.state][action_t] > max_action):
        max_action = self.q[self.state][action_t]
        action = action_t


## with x% probability select a random action and 1-x% use the action that has max q value in the current state.
if ( random.randrange(0, 100, 1) > ((1 - self.epsilon) * 100) ):
            action = random.choice(self.env.valid_actions[0:])

# Execute action and get reward
reward = self.env.act(self, action)


# TODO: Learn policy based on state, action, reward
max_action = 0.0
for action_t in self.q[self.state]:
    if (self.q[self.state][action_t] > max_action):
        max_action = self.q[self.state][action_t]


self.q[self.prev_state][self.prev_action] = ((1 - self.alpha)* self.q[self.prev_state][self.prev_action]) + self.alpha * (self.prev_reward + self.gamma * max_action)
self.prev_reward = reward
self.prev_state = self.state
self.prev_action = action
```

QUESTION: Report the different values for the parameters tuned in your basic implementation of Q-Learning. For which set of parameters does the agent perform best? How well does the final driving agent perform?

Answer: We tuned the following values for our agent:

alpha:

We used a decay function for alpha. We needed to define a new method to be called that would decay the alpha.

```
def update_learn(self):
    self.alpha_called += 1
    self.alpha = 1.0/math.log(self.alpha_called + 16, 16)
```

This would be called every time the ENV was reset (which was at the beginning of every trial)

```
# Update learning rate in primary agent
self.primary_agent.update_learn()
```

	1/ln(t+3)	1/log(t+11, 10)	1/log(t+16, 16)
0	0.910239227	0.960252568	1
1	0.72134752	0.926628408	0.978602168
2	0.621334935	0.897711718	0.959249866
3	0.558110627	0.87250287	0.941635653
4	0.513898342	0.850274154	0.925512853
5	0.480898347	0.830482024	0.910680995
97	0.217147241	0.491781409	0.586494874
98	0.216679065	0.490815252	0.585403832
99	0.216217487	0.489861655	0.584326319
100	0.215762346	0.488920337	0.583262042

The table shows how some of the functions decay. We were getting good results for the 1/log(t+16, 16) but we need to explore the other ones too.

Gamma, Epsilon

We did a sweep through the gamma and epsilon values

```
for gamma in range(6,9,1):
    gamma *= 0.1
    for epsilon in range(5,15,5):
        epsilon *= 0.01
```

And the values we get for these are listed as:

gamma	epsilon	% reached
0.7	0.1	90
0.7	0.05	94
0.6	0.1	95
0.6	0.05	97
0.8	0.1	91
0.8	0.05	95

QUESTION: 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? How would you describe an optimal policy for this problem?
Answer: Yes our agent gets close to finding an optimal policy. Being greater than 90% correct by only learning in 100 trials is a very good result. Arguably this wouldn't work in real life where every trip would need to be completed, even if some take a little longer than optimal. It wouldn't be ok to say 90% of the trips completed correctly in real life. Thus our agent gets close to finding an optimal policy for this exercise (but not in any kind of realistic setting.)

The optimal policy would be a mapping from state to action that would:

1) NOT be illegal. i.e driving through red light etc.

2) would be efficient i.e the action brings it closest to its final destination.

In terms of our experiment we would say the agent worked with an optimal policy if its trip does not have any negative reward.

The last run of our agent shows the following:

Simulator.run(): Trial 99
Environment.reset(): Trial set up with start = (6, 1), destination = (3, 6), deadline = 40
RoutePlanner.route_to(): destination = (3, 6)
LearningAgent.update(): deadline = 40, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = right, reward = 2.0, alpha = 0.583262042114
LearningAgent.update(): deadline = 39, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = right, reward = 2.0, alpha = 0.583262042114
LearningAgent.update(): deadline = 38, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = right, reward = -0.5, alpha = 0.583262042114
LearningAgent.update(): deadline = 37, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None, reward = 0.0, alpha = 0.583262042114
LearningAgent.update(): deadline = 36, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None, reward = 0.0, alpha = 0.583262042114
LearningAgent.update(): deadline = 35, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None, reward = 0.0, alpha = 0.583262042114
LearningAgent.update(): deadline = 34, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = left, reward = 2.0, alpha = 0.583262042114
LearningAgent.update(): deadline = 33, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = right, reward = -0.5, alpha = 0.583262042114
LearningAgent.update(): deadline = 32, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None, reward = 0.0, alpha = 0.583262042114
LearningAgent.update(): deadline = 31, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None, reward = 0.0, alpha = 0.583262042114
LearningAgent.update(): deadline = 30, inputs = {'light': 'red', 'oncoming': 'right', 'right': None, 'left': None}, action = forward, reward = -1.0, alpha = 0.583262042114
LearningAgent.update(): deadline = 29, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None, reward = 0.0, alpha = 0.583262042114
Environment.act(): Primary agent has reached destination!
LearningAgent.update(): deadline = 28, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = left, reward = 12.0, alpha = 0.583262042114

We see that it made an inefficient choice as well as broke the law (highlighted in red above.) We think there are few reasons for this.

1) the agent hasn't visited the illegal state enough for the Q-value to converge to a large -ve number. A large -ve number would have trained the agent not to take that action in that state. This can be remedied by running for longer than 100 trials so that the agent has time to learn.
2) the probability of a random action is still 5% and hence the agent may still make an illegal choice based on random choice. In our algorithm the learning and actual trials are part of the same code. Typically the probability of random choice should drop to 0 after the q-values have converged.
3) finally the alpha has not decayed fully even in trial 100. Thus the agent is still exploring (and not just exploiting.) Thus it is possible that the agent will make an inefficient turn.

The largest and smallest 5 Q-values are:

State Waypoint	Light	Oncoming traffic	Left Traffic	Action	Values
('forward',	'green',	None,	1)	forward	9.952537298
('forward',	'green',	'right',	0)	forward	7.551328343
('right',	'green',	None,	0)	right	5.263050909
('forward',	'green',	'left',	0)	forward	4.676815079
('right',	'red',	None,	0)	right	4.601363421
('forward',	'red',	None,	1)	forward	-0.934597864
('left',	'red',	None,	0)	forward	-0.954259803
('forward',	'red',	'left',	0)	forward	-0.961637593
('forward',	'red',	'forward',	0)	forward	-0.967695219
('left',	'red',	None,	0)	left	-0.987408251