Train a Smart Cab to Drive

A smartcab is a self-driving car from the not-so-distant future that ferries people from one arbitrary location to another. In this project, you will use reinforcement learning to train a smartcab how to drive.

Environment

The smartcab operates in an ideal, grid-like city (similar to New York City), with roads going in the North-South and East-West directions. Other vehicles will certainly be present on the road, but there will be no pedestrians to be concerned with. At each intersection there is a traffic light that either allows traffic in the North-South direction or the East-West direction. U.S. Right-of-Way rules apply:

- On a green light, a left turn is permitted if there is no oncoming traffic making a right turn or coming straight through the intersection.
- On a red light, a right turn is permitted if no oncoming traffic is approaching from your left through the intersection. To understand how to correctly yield to oncoming traffic when turning left, you may refer to this official drivers' education video, or this passionate exposition.

Inputs and Outputs

Assume that the **smartcab** is assigned a route plan based on the passengers' starting location and destination. The route is split at each intersection into waypoints, and you may assume that the **smartcab**, at any instant, is at some intersection in the world. Therefore, the next waypoint to the destination, assuming the destination has not already been reached, is one intersection away in one direction (North, South, East, or West). The **smartcab** has only an egocentric view of the intersection it is at: It can determine the state of the traffic light for its direction of movement, and whether there is a vehicle at the intersection for each of the oncoming directions. For each action, the **smartcab** may either idle at the intersection, or drive to the next intersection to the left, right, or ahead of it. Finally, each trip has a time to reach the destination which decreases for each action taken (the passengers want to get there quickly). If the allotted time becomes zero before reaching the destination, the trip has failed.

Rewards and Goal

The **smartcab** receives a reward for each successfully completed trip, and also receives a smaller reward for each action it executes successfully that obeys traffic rules. The **smartcab** receives a small penalty for any incorrect action, and a larger penalty for any action that violates traffic rules or causes an accident with another vehicle. Based on the rewards and penalties the **smartcab** receives, the self-driving agent implementation should learn an optimal policy for driving on the city roads while obeying traffic rules, avoiding accidents, and reaching passengers' destinations in the allotted time.

Implement a Basic Driving Agent

To begin, your only task is to get the **smartcab** to move around in the environment. At this point, you will not be concerned with any sort of optimal driving policy. Note that the driving agent is given the following information at each intersection:

- The next waypoint location relative to its current location and heading.
- The state of the traffic light at the intersection and the presence of oncoming vehicles from other directions.
- The current time left from the allotted deadline.

To complete this task, simply have your driving agent choose a random action from the set of possible actions (None, 'forward', 'left', 'right') at each intersection, disregarding the input information above. Set the simulation deadline enforcement, enforce_deadline to False and observe how it performs.

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: The implementation of a basic driving agent is in agent.py. The random agent eventually reaches the target location, but the actions are randomly chosen, which means the agent does not learn from the previous behaviour and improve the action. So as I observed, sometimes the agent's action is not optimal, for instance, the agent keep going forward when there is a red light, or remains in the same position even there is no other oncoming cars or red light. So the reward is terrible. The Interesting observation that I noted is the result eventually reaches the destination through trial and error but not for considerable time as it moves in random directions.

Inform the Driving Agent

Now that your driving agent is capable of moving around in the environment, your next task is to identify a set of states that are appropriate for modeling the **smartcab** and environment. The main source of state variables are the current inputs at the intersection, but not all may require representation. You may choose to explicitly define states, or use some combination of inputs as an implicit state. At each time step, process the inputs and update the agent's current state using the self.state variable. Continue with the simulation deadline enforcement enforce_deadline being set to False, and observe how your driving agent now reports the change in state as the simulation progresses.

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: Intersection State and Next waypoint location are the states that are appropriate for modelling the smartcab and environment.

Intersection State: I chose intersection state (traffic and presence of cars) as one of the state because we want to train the cab to perform legal moves and follow the US right-of-way rules.

Next Waypoint Location: We can't choose destination and location as variables since destination changes all the time and location will cause an issue of having large amount of states and taking a long time to converge, so we choose next waypoint location as variable.

Question:

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: The task was to identify the possible states, the possible options are Intersection state (traffic light and presence of cars), Next waypoint location and Deadline, This number seems not resonable because if there are too many state to visit, it will make q value a log time to converge and q-table will be very large in memory thus I do not include Deadline.

Implement a Q-Learning Driving Agent

With your driving agent being capable of interpreting the input information and having a mapping of environmental states, your next task is to implement the Q-Learning algorithm for your driving agent to choose the *best* action at each time step, based on the Q-values for the current state and action. Each action taken by the **smartcab** will produce a reward which depends on the state of the environment. The Q-Learning driving agent will need to consider these rewards when updating the Q-values. Once implemented, set the simulation deadline enforcement enforce_deadline to True. Run the simulation and observe how the **smartcab** moves about the environment in each trial.

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:

Here are the steps for

Implementing Q-Learning Algorithm:

1. Initialize the Q-values: I initialized q-values as 5.0.

2. Observe the current state: I considered the following state variables:

    - Next waypoint

    - Intersection state (traffic light, (oncoming, left, right))

3. For the current state, choose an action based on the selection policy(epsilon-greedy), which is available in get_action function. The epsilon-greedy approach chooses an action with the highest Q value with a probability of (1-epsilon) and explores with a probability epsilon, given the state, at every time step and then deciding to make either a random action(exploration) or follow the optimal policy given the inequality.

4. Take the action, and observe the reward and as the new state.

5. Update the Q-value for the state using the observed reward the the maximum reward possible for the next state:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

6. Set the state to the new state, and repeat steps until smartcab arrives at destination.

This version of the program did run without error, however the learnig agent frequently got stuck in a policy thah was clearly not optimal. Compared to breaking traffic rules or going the wrong direction, the action of not moving(selecting "None" for action) is preferable. Thus, in the first instance where a state was encountered, if the first random action explored was to remain stationary, a local Q optimum would be created. In subsequent of the state, the agent would interpret the 'best' action as remaining still. While this policy avoid traffic violations but also avoids exploring the state for the most rewarding potential action (which obviously would be to follow traffic rules allowed). Nevertheless, this initial implementation of Q-Learning did train the agent to follow rules as negative rewards were quickly relegated to penalized action.

For controlling the trade-off between exploration and exploitation is to add an epsilon clause within the program. The epsilon clause is a way to introduce exploration of the state-action space, even if a 'best' action does exits for the specific state. A randome float is generated, and compared to epsilon value. If the float was less than epsilon, the best action would be taken according to the policy. If the rendom float eclipsed epsilon, then a random action would be taken. This feature allows the agent to explore more of the state and avoid being trapped in a sub-optimal policy thah had found a local minimum Q.

Performance:
Alpha value(learning rate) was set to 1, Gamma value(discount factor) was set to 0.05, Epsilon was set to 0, I set these values by trying many possible values and compare the performances, the details will be shown later.

Implemented the Qlearning after determining these variables. The agent has reached the destination before deadline 95 times out of 100 runs.

Improve the Q-Learning Driving Agent

Your final task for this project is to enhance your driving agent so that, after sufficient training, the **smartcab** is able to reach the destination within the allotted time safely and efficiently. Parameters in the Q-Learning algorithm, such as the learning rate (alpha), the discount factor (gamma) and the exploration rate (epsilon) all contribute to the driving agent's ability to learn the best action for each state. To improve on the success of your **smartcab**:

- Set the number of trials, n_trials, in the simulation to 100.
- Run the simulation with the deadline enforcement enforce_deadline set to True (you will need to reduce the update delay update_delay and set the display to False).
- Observe the driving agent's learning and **smartcab's** success rate, particularly during the later trials.
- Adjust one or several of the above parameters and iterate this process.

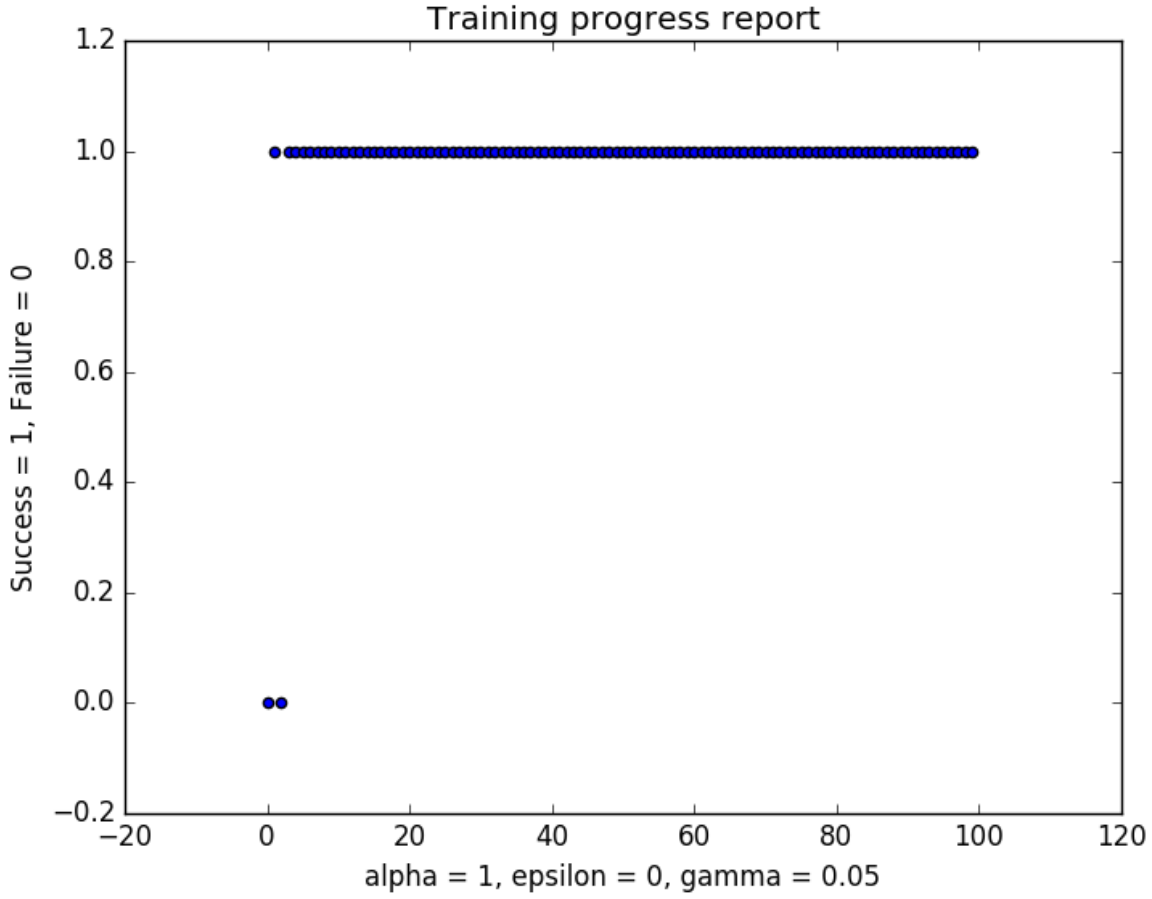
This task is complete once you have arrived at what you determine is the best combination of parameters required for your driving agent to learn successfully.

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:

alpha	gamma	Epsilon	success	moves(n =100)	reward(n=100)
0.5	0.05	0.9	26/100	23	10.5
0.5	0.05	0.5	55/100	12	13
0.9	0.4	0.5	61/100	14	12
0.6	0.2	0.4	76/100	20	12
0.9	0.2	0.05	95/100	19	20.5
0.9	0.2	0.1	96/100	18	23
1	0.05	0	98/100	12	18.5



Performance:
Alpha value (learning rate) was set to 1, Gamma value (discount factor) was set to 0.05, Epsilon was set to 0, I set these values by trying many possible values and compare the performances.

Implemented the Qlearning after determining these variables. The agent has reached the destination before deadline 95 times out of 100 runs.

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:

```
Simulator.run(): Trial 97
Environment.reset(): Trial set up with start = (6, 2), destination = (5, 6), deadline = 25
RoutePlanner.route_to(): destination = (5, 6)
LearningAgent.update(): deadline = 25, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = forward, reward = 2.0
LearningAgent.update(): deadline = 24, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint':
'left', 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 23, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint':
'left', 'left': None}, action = left, reward = 2.0
LearningAgent.update(): deadline = 22, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = forward, reward = 2.0
LearningAgent.update(): deadline = 21, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = forward, reward = -1.0
LearningAgent.update(): deadline = 20, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 19, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = forward, reward = 2.0
LearningAgent.update(): deadline = 18, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 17, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = forward, reward = -1.0
LearningAgent.update(): deadline = 16, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = None, reward = 0.0
Environment.act(): Primary agent has reached destination!
LearningAgent.update(): deadline = 15, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = forward, reward = 12.0
Simulator.run(): Trial 98
Environment.reset(): Trial set up with start = (3, 3), destination = (1, 5), deadline = 20
RoutePlanner.route_to(): destination = (1, 5)
LearningAgent.update(): deadline = 20, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint':
'right', 'left': None}, action = right, reward = 2.0
LearningAgent.update(): deadline = 19, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = forward, reward = -1.0
LearningAgent.update(): deadline = 18, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 17, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 16, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = forward, reward = 2.0
LearningAgent.update(): deadline = 15, inputs = {'light': 'red', 'oncoming': 'right', 'right': None, 'next_waypoint': 'left',
'left': None}, action = left, reward = -1.0
LearningAgent.update(): deadline = 14, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint':
'left', 'left': None}, action = left, reward = 2.0
LearningAgent.update(): deadline = 13, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = forward, reward = -1.0
LearningAgent.update(): deadline = 12, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 11, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint':
'forward', 'left': None}, action = None, reward = 0.0
```

```
Environment.act(): Primary agent has reached destination!
LearningAgent.update(): deadline = 10, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = forward, reward = 12.0
Simulator.run(): Trial 99
Environment.reset(): Trial set up with start = (1, 4), destination = (7, 3), deadline = 35
RoutePlanner.route_to(): destination = (7, 3)
LearningAgent.update(): deadline = 35, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'right', 'left': None}, action = right, reward = 2.0
LearningAgent.update(): deadline = 34, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint': 'right', 'left': None}, action = right, reward = 2.0
LearningAgent.update(): deadline = 33, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 32, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 31, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = forward, reward = 2.0
LearningAgent.update(): deadline = 30, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = right, reward = -0.5
LearningAgent.update(): deadline = 29, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'left', 'left': None}, action = right, reward = -0.5
LearningAgent.update(): deadline = 28, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'right', 'left': None}, action = forward, reward = -1.0
LearningAgent.update(): deadline = 27, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'right', 'left': None}, action = right, reward = 2.0
LearningAgent.update(): deadline = 26, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint': 'right', 'left': None}, action = right, reward = 2.0
LearningAgent.update(): deadline = 25, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 24, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 23, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 22, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 21, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 20, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = forward, reward = 2.0
LearningAgent.update(): deadline = 19, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = forward, reward = 2.0
LearningAgent.update(): deadline = 18, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = forward, reward = 2.0
Environment.act(): Primary agent has reached destination!
LearningAgent.update(): deadline = 17, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': None}, action = forward, reward = 12.0
```

So the best combination is Alpha = 1, Gamma = 0.05 and Epsilon = 0. Now we compared the Q-Learning agent with random agent, the success rate is around 95% which is much better than the basic random agent (19%).
The optimal policy would be.

1. Obey traffic rules and
2. Move in the direction of the way point provided. If traffic laws not obeyed the way point direction do not move.

We noticed that after learning optimal policy, the agent get to the destination quickly with less number of moves in the end, which means the agent not only learned how to get to the destination but the best route to get to the destination instead of going in circles.

Also, the cab always reaches the destination with large positive cumulative reward, the cumulative is smaller in the end compared to the beginning because of the smaller number of moves. So basically the cab are learning to obey traffic rules and make reasonable action.
Given above are the result of last three trials we can see whenever wrong action is taken which is not optimal policy it gets negative rewards. Whenever the action taken is correct with respect to optimal policy it gets positive rewards.
So I think the cab is acting optimally not only learning an optimal policy by taking the right moves but also choosing the smaller route.