smartcab project

Q-learning algorithm summary

initialization

- initialize Q-value table with 0 for unknown combinations of states and actions
- select random action from all available action states

choose best action according to Q-state table or random

- set \(\epsilon\) (epsilon) as the probability of randomly selected action
- greedy strategy of action selection
    - choose best action \(\mathop{\arg\max}_{a} Q(s,a)\) at the probability of 1-\(\epsilon\)
    - choose random action from all available action states

act and update Q-state table (learning)

- act, get reward \(r\) of selected action in current environment and new state \(s'\)
- choose best action for new state according to the current Q-table \(a' = \mathop{\arg\max}_{a} Q(s',a)\)
- set \(\alpha\) parameter for Q-learning rate, and \(\gamma\) parameter for Q-learning discount rate
- update Q-state by $$ Q(s,a) = (1-\alpha) Q(s,a) + \alpha (r + \gamma Q(s', a')) $$

loop action selection and Q-state update process until

- Q-state becomes stable
- other conditions of ending the program (e.g. reach the destination / hard timelimit excelled in this project)

smartcab learning goals

Provided with the environment and planner classes, we want to implement a Q-learning agent that can find a optimal way towards the destination without severe penalties of not following traffic rules.

procedure

Implement a Basic Driving Agent

To begin, your only task is to get the **smartcab** to move around in the enviro nment. 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 oncomin g 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 fro m the set of possible actions (`None`, `'forward'`, `'left'`, `'right'`) at each intersection, disregarding the input information above. Set the simulation dead line enforcement, `enforce_deadline` to `False` and observe how it performs.

```
In [13]: import random
         from smartcab.environment import Agent, Environment
         from smartcab.planner import RoutePlanner
         from smartcab.simulator import Simulator
         from smartcab.agent import LearningAgent
         from collections import OrderedDict
         import numpy as np
```

```
In [14]: e = Environment()
         a = e.create_agent(LearningAgent)
         a.env.valid_actions

Out[14]: [None, 'forward', 'left', 'right']
```

```
In [20]: a.env.valid_actions[np.random.randint(3)]   # choose a random action

Out[20]: 'left'
```

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: With a random walk, in 100 trials, about half of the smartcab have reached the destination within the hard time limit(100), but not within the deadline. This observation may depend on our route planner which always point to the final destination, and simple grid network(8*6).

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.

In [26]:
```
a.env.agent_states[a]
```
Out[26]:
```
{'heading': (0, 1), 'location': (8, 3)}
```

In [ ]:
```
# define state as combination of inputs and next_waypoint planner
self.next_waypoint = self.planner.next_waypoint()  # from route planner, a
lso displayed by simulator
inputs = self.env.sense(self)
self.state = (inputs['light'], inputs['oncoming'], inputs['left'], inputs[
'right'], self.next_waypoint)
```

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 : I have defined the states from combinations of environment variables at the intersection(light, oncoming traffic, left traffic, right traffic) and the direction of our cab provided by a simple planner. Because the current environment and direction both affects the reward(penalty), I think these are the proper components for the problem.

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 : There are 2 (light) *4 (actions for oncoming cars)* 4 (actions for left cars) *4 (actions for right cars)* 4 (next_waypoint, actions for the cab to act according to simple plan) = 512 combinations. Given current goal of reaching the destinations in maximum(100, 5 *distance) steps, it's hot to learn all states. I ran 100 rounds of experiment in current problem setting, and get 64 states in total, because our system have default 3 dummy cars on the 8*6 traffic grids, in mostly case there are no oncoming, left, right cars at all.

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-L earning algorithm for your driving agent to choose the *best* action at each tim e step, based on the Q-values for the current state and action. Each action take n by the **smartcab** will produce a reward which depends on the state of the en vironment. The Q-Learning driving agent will need to consider these rewards when updating the Q-values. Once implemented, set the simulation deadline enforcemen t `enforce_deadline` to `True`. Run the simulation and observe how the **smartca b** moves about the environment in each trial.

The formulas for updating Q-values can be found in this video.

In [ ]:
```
    def __init__(self, env):
        super(LearningAgent, self).__init__(env)  # sets self.env = env, s
tate = None, next_waypoint = None, and a default color
        self.color = 'red'  # override color
        self.planner = RoutePlanner(self.env, self)  # simple route planner
to get next_waypoint
```

```
        # TODO: Initialize any additional variables here
        self.epsilon = 0.5    # probability of randomly select action
        self.alpha = 0.3   # Q-value learning rate
        self.discount = 0.9   # discount for future rewards
        self.Qvalue = OrderedDict()  # initialize Q-value table
        self.previous_state = None # initialize previous state
        self.rounds = 1  # initialize training rounds
```

In [ ]:

```
    def update(self, t):
        # Gather inputs
        self.next_waypoint = self.planner.next_waypoint()  # from route pla
nner, also displayed by simulator
        inputs = self.env.sense(self)
        deadline = self.env.get_deadline(self)

        previous_state = self.env.agent_states[self]
        previous_location = previous_state['location']
        previous_heading = previous_state['heading']

        # TODO: Update state with sensor information and guided directions
towards destination

        self.previous_state = (inputs['light'], inputs['oncoming'], inputs[
'left'], inputs['right'], self.next_waypoint)

        for a in self.env.valid_actions:  # initialize Q-value table for u
nknown combinations of states and actions
            if (self.previous_state, a) not in self.Qvalue.iterkeys():
                self.Qvalue[(self.previous_state, a)] = 0

        prob = np.array([self.Qvalue[(self.previous_state, None)] , self.Q
value[(self.previous_state, 'forward')], self.Qvalue[(self.previous_state,
'left')], self.Qvalue[(self.previous_state, 'right')]])

        # TODO: Select action according to your policy

        if random.random() < self.epsilon:
            action = np.random.choice(self.env.valid_actions)
        else:
            action = self.env.valid_actions[np.random.choice(np.where(prob
== prob.max())[0])]
            # prefer this rather than argmax, which can break the tie prefe
rence


        # Execute action and get reward
        oldValue = self.Qvalue[(self.previous_state, action)]
        reward = self.env.act(self, action)

        # TODO: Learn policy based on state, action, reward

        state = self.env.agent_states[self]
        location = state['location']
        heading = state['heading']

        nextInputs = self.env.sense(self)
        #print inputs, nextInputs
        self.state = (nextInputs['light'], nextInputs['oncoming'], nextInp
uts['left'], nextInputs['right'], self.planner.next_waypoint())

        for a in self.env.valid_actions: # initialize Q-value table for un
known combinations of current states and actions
            if (self.state, a) not in self.Qvalue.iterkeys():
                self.Qvalue[(self.state, a)] = 0

        futureValue = max([self.Qvalue[(self.state, a)] for a in self.env.
valid_actions])
        newValue = reward + self.discount * futureValue
        self.Qvalue[(self.previous_state, action)] =  (1-self.alpha) * old
Value + self.alpha * newValue
        # update Q-value table balanced by learning rate
```

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

Answer : according to my current setting of parameters, I can reach the destination in 45 rounds out of 100 trials within the deadline, that's a great learning process as we prefer the best action according to the up-to-date Q-table at 50% chance, which is a lot more reasonable than random choice.

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 t he allotted time safely and efficiently. Parameters in the Q-Learning algorithm, such as the learning rate (`alpha`), the discount factor (`gamma`) and the expl oration 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, particul arly 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 co mbination of parameters required for your driving agent to learn successfully.

- I decrease the epsilon as the training time increases, but not any further beyond a reasonable threshold

In [ ]:
```
    def reset(self, destination=None):
        self.planner.route_to(destination)
        # TODO: Prepare for a new trip; reset any variables here, if requir
ed
```

```
        self.rounds += 1 # increase count of training rounds
        if self.rounds > 100:
            self.rounds = 100
```

In [ ]:
```
    def update(self, t):
        self.epsilon = self.epsilon * (100-self.rounds)/100  # decrease ep
silon with increasing training rounds as confidence gains along
```

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

In [ ]:
```
        self.epsilon = 0.9   # probability of randomly select action
        self.alpha = 0.2  # Q-value learning rate
        self.discount = 0.9  # discount for future rewards
```

I finally choose the above parameters as the result shows the cab reach the destination in 99 rounds out of 100 trials most of the time.

QUESTION: *Does your agent get close to finding an optimal policy, i.e. re ach the destination in the minimum possible time, and not incur any penalties? H ow would you describe an optimal policy for this problem?*

After 100 rounds of training, some highest Q-values for each state are as follows, which correctly obey the traffic rules:

- when there is green light and no surrounding cars, you can move in your direction (('green', None, None, None, 'right'), 'right') 7.64943183796 (('green', None, None, None, 'forward'), 'forward') 9.21107577481 (('green', None, None, None, 'left'), 'left') 5.90768926833
- when there is red light and no surrounding cars, you can turn right if your direction is 'right' (('red', None, None, None, 'right'), 'right') 5.36415337755
- when there is green light, and right car turns left, your direction is right, you can turn right (('green', None, None, 'left', 'right'), 'right') 3.18866840061
- when there is green light, and right car turns right, your direction is forward, you can go forward (('green', None, None, 'right', 'forward'), 'forward') 3.84675308795

Observed with the final 5 test rounds, except only one little penalty for a single step, all the actions to the destination follow the optimal route, without idle time lost. I think given the not thorough Q-table, this is reasonable.