Project 4: Reinforcement Learning

Implement a Basic Driving Agent

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?

```
action = random.choice((None, 'forward', 'left', 'right'))
```

Commit: https://github.com/lucasdupin/machine-learning/commit/5026353bcc5421a8726b8fc9b223a8ca3232e694

Yes, it will reach the destination but its path is far from optimal.
It will get there by chance, not because we calculated the best strategy.
It also doesn't use any quality function or calculate the utility of any move chosen.

Inform the Driving Agent

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?

```
# Update state
self.state = {
    #"deadline": deadline,
    "light": inputs["light"],
    "oncoming": inputs["oncoming"],
    "left": inputs["left"],
    "right": inputs["right"],
    "next_waypoint": self.next_waypoint
}
```

Commit: https://github.com/lucasdupin/machine-learning/commit/c45ae8b1d1f730bfc2dc2f0c00529a3cae2820bf

I've identified the following components for a state:

Explanation under each bullet

- light: Semaphore
  Affects if a move is valid or not, you can go forward if it’s red.
- oncoming: traffic coming towards you
  You won’t be able to turn left if there is oncoming traffic, which makes this key relevant
- left: traffic coming from the left
  affects whether you can turn right or not
- right: traffic coming from the right
  affects whether you can turn left or not
- next_waypoint: the direction that minimizes manhatan distance to the target position
  important since the target point affects the reward
- deadline: How long do the smartcab still have to reach the destination
  In this particular case I decided to **ignore** the deadline because of the curse of dimensionality.
  The great number of possible values for this state would make training much longer than necessary

Implement a Q-Learning Driving Agent

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?

on init:

```
self.alpha = 0.5
self.possible_actions = ('forward', 'left', 'right', None);
```

on update:

```
# Unique hash to represent this state when learning
state_hash = hash(frozenset(self.state.items()))
# Create dictionary to hold q values
if not state_hash in self.q_table:
    self.q_table[state_hash] = dict((a, 0) for a in self.possible_actions)

# Select action according to your policy
action = random.choice(self.possible_actions)
if (random.random() > 0.1): # Avoid getting locked in local optima
    q_values = [(action, self.q_table[state_hash][action]) for action in
self.possible_actions]
    action = q_values[np.argmax(q_values, axis=0)[1]][0]

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

# Learn policy based on state, action, reward
self.q_table[state_hash][action] = (1.0 - self.alpha) * self.q_table[state_hash]
```

```
[action] + self.alpha * reward
```

Commit: https://github.com/lucasdupin/machine-learning/commit/2a925b7b6f608c4efddb1f342674399686c214d7

It starts moving in an erratic and random manner, like in the first example, but then starts to learn what gives it better rewards, and avoid moves that return negative rewards.

This is accomplished by keeping track of rewards given at each state - in a weighted manner - and applying a learning rate.

I also implemented a feature to avoid getting stuck in local optima, by randomly picking a direction that may not necessarily look optimal at first, but might end up modifying the Q table.

Improve the Q-Learning Driving Agent

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?

The parameters used for tuning were: “alpha”, “action_cost”, “number of trials” and “random direction rate”.
“Action Cost”, in this case, represents a cost for an action.
After training the algorithm 100 times, I turned off “random choices” to be able to compute the results without stochasticity.
Please consider the following table:

Random rate	Alpha	Action Cost	Mean score	number of penalties in 100 iterations	out of time
0.0	0.10	-0.10	19.99	4	1
0.0	0.10	-1.00	7.25	35	0
0.0	0.50	-0.10	19.45	4	0
0.0	0.50	-1.00	11.10	3	0
0.1	0.10	-0.10	19.70	1	0
0.1	0.10	-1.00	9.65	1	0
0.1	0.50	-0.10	20.27	1	0
0.1	0.50	-1.00	10.50	1	0
0.5	0.10	-0.10	22.80	0	0
0.5	0.10	-1.00	7.50	0	0
0.5	0.50	-0.10	23.71	1	0
0.5	0.50	-1.00	8.00	1	0

I also tried to incorporate *deadline* into the state but it takes too long to train without any perceived improvement.

Random rates bigger than 0 keep Q from getting stuck. It's clear that the worst results had `random_choice = 0`
Alpha only makes a difference when `random_choice` is set to 0, this happens because it's easier to make the algorithm change it's mind with higher alphas.
Action cost closer to 0 result in less penalties, since there it's ok to take longer to reach the destination, and it ended up not affecting the number of times we ran out of time.

I consider the model well trained since the final reward is always positive, it usually takes only 1 penalty after 100 iterations and and the number of steps is met.

Optimal set:

```
random rate: 0.1
alpha: 0.5
action_cost: -0.1
with final score: 23.71
```

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?

This problem requires penalties to be avoided, cars aren't supposed to crash, they must protect their passengers. That's the most essential rule.
This is what I had in mind while tweaking the values, keeping the reward not only positive, but keeping the whole smartcab experience safe.

And yes, it gets close to the optimal policy, having usually 1 penalty after 100 runs.
After training the algorithm for 100 times, I turned off all random actions and ran it again 100 times, using the resulting Q table.
Here is a list of penalties taken, with the trained Q table:

```
penalty for action forward at state: {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': 'left'}
penalty for action right at state: {'light': 'red', 'oncoming': 'forward', 'right': None, 'next_waypoint': 'forward', 'left': 'forward'}
penalty for action left at state: {'light': 'green', 'oncoming': 'forward', 'right': None, 'next_waypoint': 'right', 'left': None}
penalty for action left at state: {'light': 'red', 'oncoming': None, 'right': 'forward', 'next_waypoint': 'forward', 'left': None}
penalty for action right at state: {'light': 'red', 'oncoming': None, 'right': None, 'next_waypoint': 'forward', 'left': 'right'}
penalty for action left at state: {'light': 'red', 'oncoming': None, 'right': 'left', 'next_waypoint': 'forward', 'left': None}
penalty for action right at state: {'light': 'red', 'oncoming': None, 'right': 'left', 'next_waypoint': 'left', 'left': None}
penalty for action left at state: {'light': 'red', 'oncoming': None, 'right': 'forward', 'next_waypoint': 'left', 'left': None}
```

Analyzing the penalties, we notice that the most common situation is when the traffic lights are red, and the cab tries to turn left.

Obiviously, while training, it starts behaving randomly and learns over time. Starting with high bias, and moving to a point where the current model saturates.
During the first 10 trials I was seing a sum of 18 penalties, with the optimal parameters, moving 0 on the last ten. Even though 0 is a great number it's important to know that running that optimal, saturated model, for 100 times again, I still get 1 penalty sometimes.