UDACITY

PROJECT

Train a Smartcab to Drive

A part of the Machine Learning Engineer Nanodegree Program

PROJECT REVIEW

CODE REVIEW

NOTES

SHARE YOUR ACCOMPLISHMENT!

Requires Changes

6 SPECIFICATIONS REQUIRE CHANGES

You are off to a good start here. Check out these sections for some insight in successfully completing this project. We look forward to seeing your next submission. And keep up the hard work!

Implement a Basic Driving Agent

Student summarizes observations about the basic driving agent and its behavior. Optionally, if a visualization is included, analysis is conducted on the results provided.

Question 1

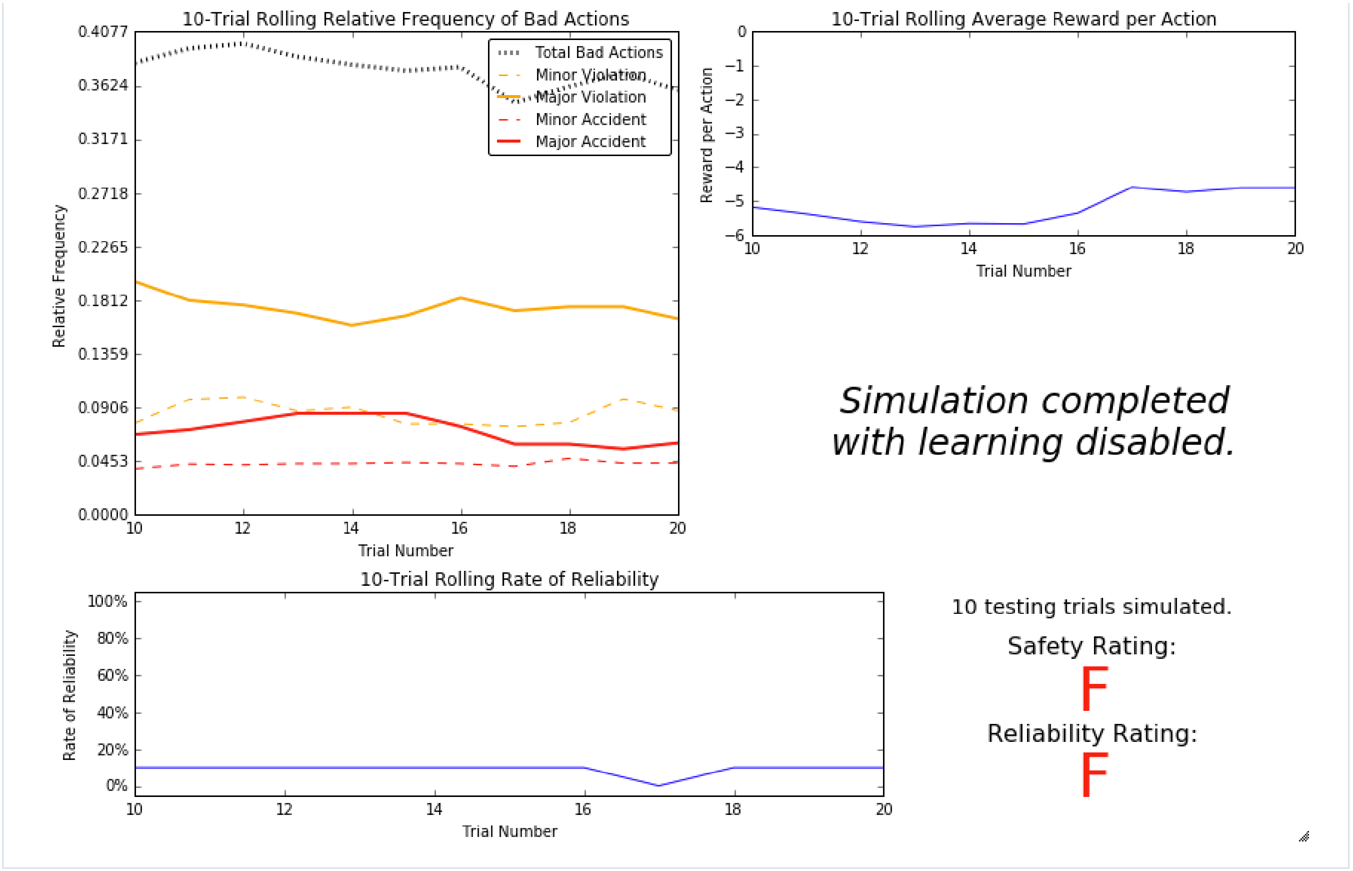
- **REQUIRED**: With your comment of "*While the light is red, it gets a positive reward for being (properly) idled at a red light. As soon as the light turns green, it starts receiving a negative reward for being (improperly) idled at a green light without oncoming traffic.*" Nice work with a red light(positive reward) and a negative reward when there is a green light without other traffic. However, there is actually one more instance that should also be discussed. What are the rewards when there is a green light **with** other traffic?

Question 2

- Good job checking out the environment. As there are many tuning knobs we can check out here and play around with! Would recommend not describing gamma, but something from the environment.

Question 3

- **REQUIRED**: Your visual isn't quite right here. As you will need to have `testing enabled` (so you can receive ratings) and run 20 training trials. As it seem that your simulation was cut short. Your visual should look more like this



Inform the Driving Agent

Student justifies a set of features that best model each state of the driving agent in the environment. Unnecessary features not included in the state (if applicable) are similarly justified.

You have modeled the environment very well.

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

As it is always an important thing to determine a good state before diving into the code, as this will pave the way to an easier implementation. And nice discussion for the need of all these features.

And good ideas regarding the deadline. As including the deadline could possibly influence the agent in making illegal moves when the deadline is near. Also that if we were to include the deadline into our current state, our state space would blow up, we would suffer from the curse of dimensionality and it would take a long time for the q-matrix to converge.

Question 5: The reason this is marked as *Requires Changes* is that your state space calculation is a bit high. Relook at your `waypoint` values of "*could be one of 4 values (forward, left, right, None)*". As it seems that you have included one extra action for the waypoint. Which one of these will the agent never see?

The driving agent successfully updates its state based on the state definition and input provided.

The driving agent successfully updates its state in the pygame window!

Implement a Q-Learning Driving Agent

The driving agent chooses the best available action from the set of Q-values for a given state. Additionally, the driving agent updates a mapping of Q-values for a given state correctly when considering the learning rate and the reward or penalty received.

Couple issues here. In your Q-Learning algorithm(as you essentially have)

```
self.Q[str((state, action))] = old_q * (1 - self.alpha) + self.alpha * (reward + max_state2_q)
```

As we should be setting gamma equal to zero!! As if we get the max q-value for the state `max_state2_q` we are essentially setting gamma equal to one. As for this project we should not be using gamma, since the future rewards don't matter. When setting gamma to 0(since 0 times anything is still 0), it should be

```
self.Q[self.state][action] = (1.0 - self.alpha) * self.Q[self.state][action] + self.alpha * (reward + 0 *
 maxNextQ)
```

or(when we remove gamma and the max for the next state, recommended)

```
self.Q[self.state][action] = (1.0 - self.alpha) * self.Q[self.state][action] + self.alpha * reward
```

In other words, next reward should not be included in this project.

Also in your `choose_action()` function and code of

```
possible_actions = {action: self.Q.get((self.state, action), 0) for action in self.valid_actions}
for an_action in self.valid_actions :
    if possible_actions[an_action] == max(possible_actions.values()):
        best_action = an_action
action = best_action
```

your agent should be choosing a random action from a choice of actions that have the highest Q-value. For example, since all actions are initialized with a reward of zero, it's possible that all four actions are considered "optimal". Not having the agent choose a random action from this would imply that it always chooses, perhaps, the first available option. This is incorrect behavior:

```
STATE X:
-- 'forward' : 0.00
-- 'left'    : 0.00
-- 'right'   : -1.023
-- 'None'    : 0.00
```

The agent should choose one of 'forward', 'left', or 'None' with equal probability, since they are all considered optimal with the current learned policy. Using a list would be a good idea.

Student summarizes observations about the initial/default Q-Learning driving agent and its behavior, and compares them to the observations made about the basic agent. If a visualization is included, analysis is conducted on the results provided.

"*The number of bad actions decreased (from 28% 5o 15% over the trial period) and the average reward got better (as in less negative) from -3 to -1 in the same period*"

Nice observations here. The agent is getting a bit better here, as it is actually starting to learn as the trials pass by. As this is definitely expected with a decaying epsilon. Therefore we can reduce the chances of random exploration over time, as we can get the best of both exploration vs exploitation.

Maybe with a slower decay rate and with more training trails, this default Q-Learning driving agent could actually improve and explore and learn more states. But for now we can use this agent as a good benchmark.

Improve the Q-Learning Driving Agent

Student summarizes observations about the optimized Q-Learning driving agent and its behavior, and further compares them to the observations made about the initial/default Q-Learning driving agent. If a visualization is included, analysis is conducted on the results provided.

N/A

The driving agent is able to safely and reliably guide the *Smartcab* to the destination before the deadline.

N/A

Student describes what an optimal policy for the driving agent would be for the given environment. The policy of the improved Q-Learning driving agent is compared to the stated optimal policy, and discussion is made as to whether the final driving agent commits to unusual or unexpected behavior based on the defined states.

N/A

RESUBMIT

DOWNLOAD PROJECT

Learn the best practices for revising and resubmitting your project.

Have a question about your review? Email us at review-support@udacity.com and include the link to this review.

RETURN TO PATH

Student FAQ