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

4 SPECIFICATIONS REQUIRE CHANGES

Off to a great start! The Q-learning algorithm is correctly implemented, but there are a few more areas in your report in which you'll need to further expand upon. I've provided some feedback/suggestions below to help out. Great job so far, keep it up!

Implement a basic driving agent

✓

Student is able to implement the desired interface to the agent that accepts specified inputs.

✓

The driving agent produces a valid output (one of None, 'forward', 'left', 'right') in response to the inputs.

Really like the results you've shown here. This provides a solid benchmark when comparing the agent's performance now (i.e. random agent) and to the case where Q-learning is implemented. But for now, what we see is that instead of taking actions based on intelligence using the Q-Learning methods, the randomized series of decisions here result in the smart cab taking longer to reach the destination and majority of the time it does not, few cases it does (i.e. enforcing the deadline).

Trials	I Success	I Ran Out of time	I Hard Deadline
1. 1000	682	475	318
2. 1000	666	460	334
3. 1000	655	474	345
4. 1000	678	476	322
5. 1000	652	426	348

The driving agent runs in the simulator without errors. Rewards and penalties do not matter - it's okay for the agent to make mistakes.

Note the following errors occurs when running your code:

```
File "agent.py", line 116, in chooseAction
 bestAction = max(self.Q[self.state].iteritems(), key=operator.itemgetter(1)
[0]
File "agent.py", line 43, in getDefaultQvalues
 for action in self.env.getActions():
AttributeError: 'Environment' object has no attribute 'getActions'
```

```
def getDefaultQvalues(self):
     Q = {}
     for action in self.env.getActions():
         Q[action] = 1
     return Q
```

Unless you made changes to the `environment.py` file then you will need change `self.env.getActions()` with `Environment.valid_actions`. You did not include this file in your submission so I cannot verify whether any changes were made. You should notate this in your next submission.

```
class Environment(object):
    """Environment within which all agents operate."""
    valid_actions = [None, 'forward', 'left', 'right']
```

Identify and update state

Student has identified states that model the driving agent and environment, along with a sound justification.

"I decided to include following states: light (green, red), oncoming (None, left, right, forward), left (None, left, right, forward) and a next waypoint." -- Nicely done identifying the sensory inputs that were included in the state space in addition to the next waypoint. Good discussion as to why these variables are necessary so that the agent has enough critical information about the environment to make decisions and thus eventually

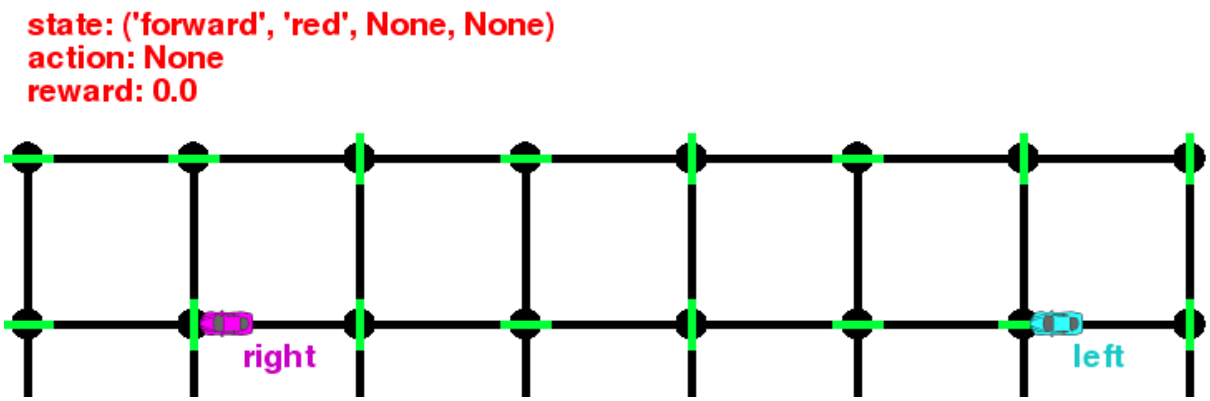
learning the optimal rules of the road. One piece that you'll need to discuss is whether or not deadline should be included in the state space. For example, given the number of states that the deadline variable can take, will it blow up our state space into a size that cannot be feasibly explored by the agent in 100 trials if deadline is included?

✓

The driving agent updates its state when running, based on current input. The exact state does not matter, and need not be correlated with inputs, but it should change during a run.

States correctly update as shown in the PyGame window:

```
# Update state
self.state = (self.next_waypoint, inputs['light'], inputs['oncoming'], inp
uts['left'])
```



Implement Q-Learning

✓

The driving agent updates a table/mapping of Q-values correctly, implementing the Q-Learning algorithm.

Q-learning is correctly implemented, nice work!

```
newQ = reward + self.gamma * newAction[1]
currentQ = self.Q[self.state][action]
self.Q[self.state][action] = (1 - learningRate) * currentQ + learningRate
 * newQ
```

✓

Given the current set of Q-values for a state, it picks the best available action.

Great job selecting the best available action at each time step in addition to implementing epsilon-greedy to address the trade-off between exploiting the information learned thus far in the Q-table and exploring the

environment by taking a random action and learning the value of that action taken.

```
bestAction = max(self.Q[self.state].iteritems(),
key=operator.itemgetter(1))[0]
if random.random() < self.epsilon:
   random_actions = [action for action in self.env.valid_actions if action
 != bestAction]
   return random.choice(random_actions)
```

Similarly to how you decayed the learning rate, I would also suggest adding a decay factor for epsilon as well. This works relatively well and the reason is that eventually at some point, the agent needs to stop exploring and start exploiting the values and information in the Q-Table.

Student has reported the changes in behavior observed, and provided a reasonable explanation for them.

"Smartcab started reaching destination each time while not hitting a hard-deadline. It improves with each run and consistently achieves 80%+ success rate. It does so, because it uses a Q learning algorithm instead of taking random actions." -- Nice observations and good work discussing how the performance significantly increases one Q-learning has been implemented. Can you discuss why that is? Additionally, can you elaborate more on the agent's behavior and whether the agent obeys traffic laws better, takes the most direct route, etc. given the implementation of Q-learning?

Enhance the driving agent

The driving agent is able to consistently reach the destination within allotted time, with net reward remaining positive.

Success rate for the last 10 trials (num trials = 100) is 100%. Great! The agent is consistently reaching the destination within the allotted time.

Specific improvements made by the student beyond the basic Q-Learning implementation have been reported, including at least one parameter that was tuned along with the values tested. The corresponding results for each value are also reported.

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?

No answer under this question. In your code, you have the right idea though; to systematically check multiple values of alpha, gamma, and epsilon to determine which values produce the optimal performance:

```
alphas = [0.8]
gammas = [0.1]
epsilons = [0.005]
# A lot of work :)
for alpha in alphas:
     for gamma in gammas:
         for epsilon in epsilons:
             runWithParams(alpha, gamma, epsilon)
```

In your report you only show the range of values you investigated:

"I've tried following sets of parameters:
alphas = [1, 0.8, 0.6, 0.3, 0.1]
gammas = [0.5, 0.3, 0.1]
epsilons = [0.2, 0.1, 0.05, 0.01, 0.005]"

I assume the values in your code are the final tuned hyperparameters? How will your audience know this? You also decayed the learning rate (good idea), but you should mention this here as well.

```
learningRate = 1.0 / (t + 1) ** self.alpha
```

The idea here is that you should output some numerical results for the other values as well in order to illustrate to your reader how well/poor the agent performs under these conditions. This way it helps confirm that your final tuned values were not just "randomly" selected. One approach here is to summarize the simulation results, given a range of values for the hyperparameters, in a table format. This will help give your audience an easy way to interpret your work clearly.

A description is provided of what an ideal or optimal policy would be. The performance of the final driving agent is discussed and compared to how close it is to learning the stated optimal policy.

For this specification, you should explicitly discuss whether you believe the agent is following the optimal policy or not. So for instance, does the final learnt agent just go in circles or does it find the most direct route? Another example would be does the agent always obey traffic laws or is there certain times it does not (i.e. taking right turn on red with oncoming traffic)? If the agent does not behave optimally, discuss what specific circumstances (state, actions) this occurs. One example is that you could highlight instances where the agent still does not follow traffic laws towards the end of the simulation in a table/log format. Here's an idea:

(('light', 'red'), ('oncoming', None), ('next_waypoint', 'forward'), ('left', None)) forward
(('light', 'red'), ('oncoming', None), ('next_waypoint', 'left'), ('left', None)) forward
(('light', 'red'), ('oncoming', None), ('next_waypoint', 'left'), ('left', None)) left
(('light', 'red'), ('oncoming', None), ('next_waypoint', 'forward'), ('left', None)) left

RESUBMIT

DOWNLOAD PROJECT

Best practices for your project resubmission

Ben shares 5 helpful tips to get you through revising and resubmitting your project.

Watch Video (3:01)

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