UDACITY

PROJECT

Train a Smartcab to Drive

A part of the Machine Learning Engineer Nanodegree Program

PROJECT REVIEW

CODE REVIEW 2

NOTES

SHARE YOUR ACCOMPLISHMENT!

Meets Specifications

You have a good agent here and very impressed with your understanding of these reinforcement learning techniques. Hopefully you can learn a bit more from this review. Wish you the best of luck in your future!

Getting Started

Student provides a thorough discussion of the driving agent as it interacts with the environment.

Nice job examining the rewards for the agent! One way to force the agent to move is through simulating annealing.

Student correctly addresses the questions posed about the code as addressed in Question 2 of the notebook.

Good job checking out the environment! As there are many tuning knobs we can check out here and play around with.

Implement a Basic Driving Agent

Driving agent produces a valid action when an action is required. Rewards and penalties are received in the simulation by the driving agent in accordance with the action taken.

Agent produces a valid action!

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

> "As the number of trials increase, the outcome of results do not appear to change significantly for this no learning scenario."

Good. We do see some random noise here, but this is really not a trend, as this smartcab really isn't that smart yet. This lack of success and a bit of randomness make sense as the agent is only exploring and not learning, collecting many negative rewards. As this behavior is similiar to a random walk.

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. Students argument in notebook (Q4) must match state in agent.py code.

You have modeled the environment very well

```
state = (waypoint, inputs['light'], inputs['left'], inputs['right'], inputs['oncoming'])
```

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.

> "We opt not need to include Deadline as an input as this would significantly increase the number of states we need to train on."

Good! As 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. Also note that including the deadline could possibly influence the agent in making illegal moves when the deadline is near.

The total number of possible states is correctly reported. The student discusses whether the driving agent could learn a feasible policy within a reasonable number of trials for the given state space.

> "The number of states (384) seems to be large but I think it is not too large for safe and reliable trips to the destinations by the driving agent."

Would agree. As a total of 384 total states isn't too many to learn with a feasible number of training trials and a good epsilon decay rate. Maybe another idea to confirm this would be to run a Monte Carlo simulation(considering that all these state are uniformly randomly seen). Would 750 be enough? How many training trials could represent 750 steps? What about every state, action pair? Try changing the step

```
from sets import Set
from random import choice

def chance_of_visiting_all_states(iterations, k, n=24):
    r = range(n)
    total = 0
    for i in range(iterations):
        s = Set()
        for j in range(k):
            s.add(choice(r))
            if len(s) == n:
                total +=1
                break
    return float(total)/iterations

steps = 750
print "Chance of visiting all states in {st} steps: {ch}".format(st = steps, ch = chance_of_visiting_all_states(2000, steps, 384))
```

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: (1) Chooses best available action from the set of Q-values for a given state. (2) Implements a 'tie-breaker' between best actions correctly (3)Updates a mapping of Q-values for a given state correctly while considering the learning rate and the reward or penalty received. (4) Implements exploration with epsilon probability (5) implements are required 'learning' flags correctly

Code looks great! Check out the code review.

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.

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.

Thus with a slower decay rate and with more training trails(as I see that you have actually done), 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

The driving agent performs Q-Learning with alternative parameters or schemes beyond the initial/default implementation.

Great parameter tuning here. As more training trials with increased exploration is exactly what the agent needs, since the main thing in the training phase is to explore, learn and fill up the Q-Values! As it is crazy how the number of bad actions / accidents / violations seem to be directly correlated to your epsilon value in these graphs. This might be an interesting one to check out in terms of a sigmoid function

```
self.trial_count = self.trial_count + 1
self.epsilon = 1 - (1/(1+math.exp(-k*self.alpha*(self.trial_count-t0))))
```

- Where k determines how fast the agent performs the transition between random learning and choosing the max q-value. k also determines how fast the sigmoid function converges to 0.
- The t0 value can also be chosen empirically; by using the sigmoid function, we can make sure that the agent would have sufficient time to explore the environment completely randomly, in order also to fill the Q-value matrix with the correct values.

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.

Would recommend mention what your learning rates actually represent here. As these are typically not used in the actual epsilon function, but you should create a new variable `a` for the epsilon decay rate.

Remember that the learning rate is used in the actual Q-Learning algorithm

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

What does a low learning rate represent? How does this determine how the Q-table is updated?
(https://en.wikipedia.org/wiki/Q-learning#Learning_rate)
(http://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html)
(http://www.jmlr.org/papers/volume5/evendar03a/evendar03a.pdf)

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

Congrats on your ratings! One thing to look into is to determine how reliable your agent actually is. Try changing the number of testing trials to something like 100 or even 200. How does your agent perform then? Is it ready for the real world?

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. Student presents entries from the learned Q-table that demonstrate the optimal policy and sub- optimal policies. If either are missing, discussion is made as to why.

Awesome analysis here and nice pick up on your sub-optimal states. These are probably from the low learning rates you used, since the Q-table isn't updated too much at each iteration.

Another idea to think about in terms of what an optimal policy might be

- when the agent is blocked by a red light or traffic, the best course of action is always to wait until they can follow the waypoint. However, it is often the case that they could be taking alternate routes that decrease the distance to the goal (proceeding along the step-wise diagonal path to the goal). A vector waypoint system could greatly aid navigation by rewarding paths that decrease the 2D distance to the goal instead of just rewarding following a single waypoint. In general, it may also be the case that patterns in traffic signals can be learned, but not in this environment.

Student correctly identifies the two characteristics about the project that invalidate the use of future rewards in the Q-Learning implementation.

Awesome ideas in terms of the two characteristics about the project that invalidate the use of future rewards in the Q-Learning algorithm

- agent -- This is due to egocentric nature of the agent, as well as the random aspects of state that cannot be predicted ahead of time. Were the simulation changed to an allocentric view, where the agent could sense where it was in relation to the destination, etc, then gamma term would be a more important parameter for optimal performance and policy generation.
- environment -- That the destination itself moves after every trial. If we attempted to propagate reward away from the goal, we would eventually propagate reward away from every intersection.

DOWNLOAD PROJECT

2 CODE REVIEW COMMENTS ›

RETURN TO PATH

Student FAQ