

PROJECT

Train a Smartcab to Drive

A part of the Machine Learning Engineer Nanodegree Program

PROJECT REVIEW

CODE REVIEW 2

NOTES

Meets Specifications

SHARE YOUR ACCOMPLISHMENT



Getting Started

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

You can see how the software handles reward when the agent is idle by looking at the code [here](#).

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

Good discussion of the flags.

Nice job identifying the `act` function.

Good discussion of the rendering functions.

You correctly identified that East-West is considered first.

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.

You have correctly run your first trial with `Learning = False`. Great job using `self.valid_actions`. Not using this could have the agent trying to make invalid moves.

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

The key here is that it is not learning. Nice work.

Nice work here. You could think about this phase as the part where you make sure everything works in your system before you begin working on your agent. Great job!

As you reflect on this project, you can think about working on it three phases: MVP, initial implementation, and tuning. This is a common pattern in development.

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.

deadline

You are correct in noting that `deadline` is not necessary, and in fact, causes a drastic increase in the dimension of the state space. It is also of note that including `deadline` could influence the agent to make illegal moves to meet the `deadline`.'

`waypoint`

Obviously critical as this determines where the cab intends to go next. Note that your code is not required to determine this step, only if it is safe.

`light`

Obviously critical for safety.

`oncoming`

Critical for left turns. There is currently a glitch in the environment file that handles this case for you. It is in the spirit of the project, however, to treat the environment as a complete unknown.

`left`

`left` is critical for handling the edge case of turning right on a red light.

`right`

US traffic laws make `right` unnecessary. This said, while excluding right makes sense for a human, including `right` makes the problem closer to one an actual AI might face. Actual traffic laws are far too complex to explain in a series of if-then statements, not to mention that many human drivers treat traffic laws as "flexible". The spirit of the project is for the agent to learn that `right` is not necessary. The agent should be able to in less than 400 trials, depending upon your learning rate and exploration decay rate.

I love that you included all the inputs in your definition of state space. Had you eliminated features beyond `deadline`, your project would have been verging on being a knowledge-based approach to AI [reference]. This is a reinforcement **learning** project which means you must teach the AI to learn, not program its behavior. You are technically *allowed* to restrict its input to some extent, but it should be choosing actions based upon its own learning. Note that this may take hundreds of trials to be successful.

In terms of learning the theory behind the project, here you only have six inputs. If you had 6000 inputs, would you know which features to eliminate? In my opinion, the best solution to this project is one in which has many features as possible are included and the agent is allowed to learn for itself which features are important. If you were to include all four inputs and the waypoint feature, it wouldn't take you more than 400 or 500 trails to train your agent. In this reinforcement learning task, its fine to run hundreds or thousands of trials. Tesla collected hundreds of millions of miles of data before it activated its self-driving car software!

Have a look at the Monte Carlo simulation below. Could you use this to try to optimize the number of trials you run?

Rate this review

**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.**

Your calculation of dimension is correct. 384 states is a reasonable number of states to learn in a with a good number of training trials and a good epsilon decay rate.

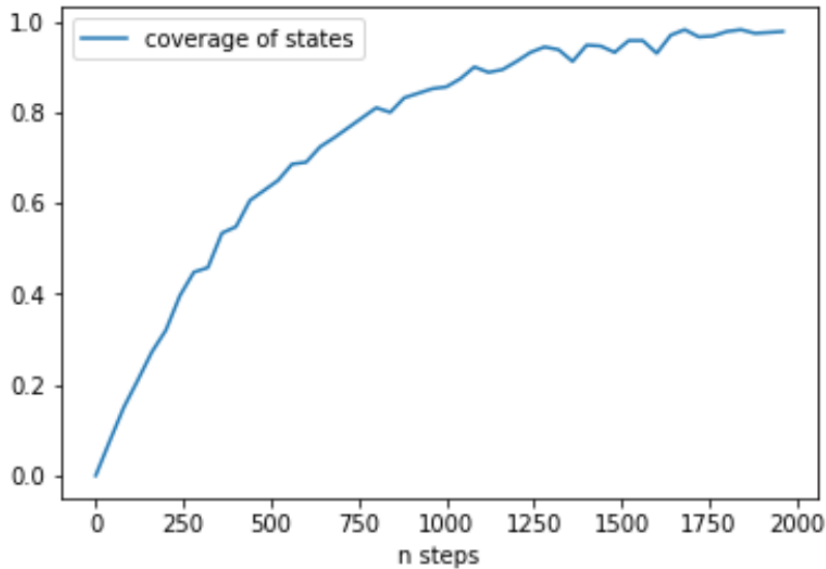
You might think about a Monte Carlo simulation to verify this. From there you might try different numbers of steps to see if every state could be visited via random exploration. Given a number of steps, how many trials are needed to reach that number?

```
import numpy as np
import random

def percent_visited(steps, states):
    visited = np.zeros(states, dtype=bool)
    for _ in range(steps):
        current_state = random.randint(0, states-1)
        visited[current_state] = True
    return sum(visited)/float(states)
```

```
1 import matplotlib.pyplot as plt
2 %matplotlib inline
3
4 states = 500
5
6 n_steps = [s*40 for s in range(50)]
7 coverage = [percent_visited(steps, states) for steps in n_steps]
8 plt.plot(n_steps, coverage, label='coverage of states')
9 plt.xlabel('n steps')
10 plt.legend()
```

<matplotlib.legend.Legend at 0x112bebbd0>



Each step here is a decision made by the agent. This is for a state space of dimension 500. Given that 2000 steps are approximately needed to visit every state, how many trials would be needed?

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

Your code matched your report. Great job!

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

Excellent implementation of your `choose_action` function.

Your `choose_action` function handles the edge case when more than one action have the same max Q.

Your `createQ` function is valid.

Your `get_maxQ` function is valid.

Your `get_maxQ` function is also pythonic.

Your `learn` function correctly executes only when `self.learning` is `True`.

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.

Good analysis. If you have a look at your log file `sim_default-learning.txt` you can see that there are many states that your agent has not yet learned. This would show these states have not been visited by the agent.

Your state space has a dimension of approximately 400 (Note that it will never actually need to consider when `waypoint == None`). At this point, you have done very little exploration. Your agent is learning, but it hasn't done enough exploration.

Improve the Q-Learning Driving Agent

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

In `choose_action`, You have correctly implemented your explore versus exploit logic.

Nice work. It looks like your agent has done enough exploration to become reliable.

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.

Considering you ran over 1750 trials you were able to include `right` in your input, and have the agent learn that it is not necessary. There is an interesting discussion to be had about reducing the size of the state space by eliminating inputs versus doing more exploration and letting the agent learn. In terms of approaches to AI, not including features is a knowledge-based or heuristic approach. It was *the* approach to AI until very recent advances in computational power have made learning-based approaches to AI feasible. The RL project is intended to be a learning-based approach to AI. Sure you could eliminate features that are redundant or even irrelevant or develop sets of heuristics (`if-then` statements about what the agent should do), but if you do *this is not reinforcement learning*. Reinforcement learning is where *your agent* learns what rules to follow *without you telling it*.

Interestingly, in this light, not using `deadline` is appropriate *not* because it blows up the dimension of the state space, but because it actually might shift the goals of the agent so that breaking safety rules might be “ok” if the deadline has almost expired!!

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

Nice job! The agent has to be extremely reliable. In a real-life situation, this is a zero-fault tolerance application. Even one mistake can result in deaths!

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.

You missed one thing in your policy. As this was your only flaw in the project, I have passed you:

1. Best set of instructions.

A. If light is red, take action 'None'.  
B. If light is green.

a. If waypoint is 'forward'.

i. Take action 'forward'.

b. If waypoint is 'left'.

i. If oncoming is 'forward' or 'right'.

1. Take action 'None'.

ii. Else.

1. Take action 'left'.


c. If waypoint is 'right'.

i. Take action 'right'.

d. If waypoint is 'None'.

i. Take action 'None'.



Agent can turn right on a red light

Nice job discussing optimal and sub-optimal policies. You can read more about learning rates for Q-learning here:

- <http://www.jmlr.org/papers/volume5/evendar03a/evendar03a.pdf>
  - <http://karpathy.github.io/2016/05/31/rl/>
  - <https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0>
  - <http://mnemstudio.org/path-finding-q-learning-tutorial.htm>
  - <https://www-s.acm.illinois.edu/sigart/docs/QLearning.pdf>
  - <http://www.umiacs.umd.edu/~hal/courses/ai/out/cs421-day10-qlearning.pdf>

https://review.udacity.com/#!/reviews/897970

4/5

 [DOWNLOAD PROJECT](#)

2 [CODE REVIEW COMMENTS](#) 

[RETURN TO PATH](#)

[Student FAQ](#)