Train a Smartcab to Drive

Summary

This is my response document answering a Udacity's project Train a Smartcab to Drive, where we let a reinforcement learning algorithm driving around a city in a cab, while learning the fastest way to reach its destination without violating (too much) traffic rules.

Answers to Rubric Questions

Implement a basic driving agent

In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

Initially, an action is taken randomly from these valid actions: None, 'forward', 'left', 'right'. As expected the agent (red car) moves randomly across the map. It eventually makes it to the target location but rarely, if at all, under the specified time bound.

Although it does not perform well, we can still use it as a proof of concept that Q-learning works. I adjusted the code to show some means of measurement of its performance: I added code to record number of successes and total runs. Successes are recorded when our learning agent reaches the goal within deadline time.

I then made the code runs on 100 trials. That way we can see how large the success rate is before and after Q learning is applied.

I added net reward statistics but did not really use it in this project, since I think with how the reward system is structured, it is more profitable for the agent to run in circles or even wait in the intersection before the goal, then go to goal intersection right before the deadline ends. I don't think that is a good measure, so instead I used ratio of how many times agent successfully reached destination / number of trials. I then ran the script a few times / episodes and averaged this score.

With basic driving agent, testing **5 episodes** with **100 trials** gave an average score of **19/100** success rate. Later we will see how Q-learning improves this.

Identify and update state

Justify why you picked these set of states, and how they model the agent and its environment.

In the final Q-learning algorithm, I used the following state variables:

- **light**: 'red' and 'green' (2 variants)
- **oncoming**: None, 'forward', 'left', 'right' (4 variants)
- **left**: None, 'forward', 'left', 'right' (4 variants)
- **right**: None, 'forward', 'left', 'right' (4 variants)
- **waypoint**: None, 'forward', 'left', 'right' (4 variants)

*Later on I tried using only two states, **light** and **waypoint**, that does improve the performance greatly. More details in later section.*

The states here is a combination of all of the state variables. So one state could be, for example: `{light: 'red', oncoming: 'forward', left: 'right', right: None, position: [1,1], waypoint: 'left'}`. In total, there should be **2 x 4 x 4 x 4 x 4 = 512 states**.

There are four actions: None, 'forward', 'left', 'right', so our Q matrix should be of size **[512 x 4]**. Instead of creating the entire matrix I decided to fill the values iteratively as the agent roams around this world.

I initially added **deadline** into the list of states, but it greatly lowered the performance, so I removed it. The low performance was caused most likely by the fact that the Q matrix got too sparse, that the prediction step often cannot find the required value in Q matrix. Another way to say this is that the Q values did not converge (it eventually will, with much more trials).

Implement Q-Learnings

What changes do you notice in the agent's behavior?

I implemented a standard Q-learning algorithm which basically selects the agent's action by Q values, that are learned by going one step ahead of a trial to get its `reward + discounted expected max Q value of next state`. Here are the initial settings:

- All Q values are given a default value of 0.
- Discount factor (γ in Udacity courses, or β in official Bellman's equation formula) of 0.33.
- Learning rate (α) of 0.9.
- ε (Epsilon) for simulated annealing approach, set to 0.1 The larger it is, the more likely it is for agent to try out new paths.

Again, running the script 5 times gave me an average score of **84/100** with mode of **85/100**.

Enhance the driving agent

Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?

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?

Interesting finding 1: What if the agent doesn't care about other cars?

When the agent doesn't care about other cars, i.e. use only **light** and **waypoint** as states, performance improved to **86/100** in 5 episodes of 100 trials, and the mode was **93/100**.

My intuition was that it was caused by significantly less size of Q matrix (8x4 as opposed to 512x4 previously). That means **with more trials we could probably get better performance**. I

later proved this to be correct by running a few more test with all **5 state variables** which gave me **87/100** score as opposed to **84/100** earlier)

I wonder how it affected the penalty rate (total number of penalties / total number of moves), since intuitively one would think that if we don't care about other cars then more penalties would happen.

One possible reason to this is that the starter code does not correctly punish the agent for violating traffic rules. This is probably why your error rate is so low.

Interestingly, they don't differ much, as they are both at **0.03** penalty rate. Maybe it was since there weren't too many training data, so I added the **number of cars to 30** to see if the difference would show:

- **30 cars with 5 state variables gave 0.04 penalty rate on average** (74 penalties / 2031 moves). Total moves are averaging at above 2000.
- **30 cars with 2 state variables gave 0.03 penalty rate on average** (54 penalties / 1591 moves). Total moves are averaging at between 1500 to around 2000.

Well, this is counterintuitive. Looks like 2 state variables, **light** and **waypoint**, is better in all fronts compared to including all 5 state variables, so let's use this instead for the final Q-learning algorithm.

Interesting finding 2: What if the default reward is not 0

In many Q-learning algorithms tutorials I found that the default Q value was set to **0**. I tried setting it to **30** and found that it greatly improved performance to an average of **96/100** but with penalty rate of **0.04** with average total moves of 1500-under 2000 (and it was setup with **2 state variables**).

What this means is that with given default Q value of 30, the agent is somehow more likely to reach its destination, but with slightly higher probability of violating traffic rules. I am not quite sure what's the intuition behind this, but a default Q value of **1** gives a performance of **94/100** with penalty rate of **0.03**. More investigation is required to really understand this behavior but that is probably way outside the scope of this project.

One can justify that by setting the default Q value to high number, the agent will basically always try out actions it hasn't tried before a couple of times, ensuring that it quickly explores its state space and then has a good picture of the environment.

To answer the last question, yes, the agent seemed able to find the optimal policy by being able to find the quickest path with very small error rate once Q values are converged.