Training a SmartCab

Implementing a Basic Driving Agent

This is my solution:

```
action = random.choice([None, 'forward', 'left', 'right'])
```

Unfortunately `pygame` wasn't working for me. I ran the simulation multiple times. The smart cab reaches its destination eventually. In some cases it did even so within the deadline. It did rack up a lot of negative rewards though, mainly due to traffic violations

Informing the Driving Agent

At each intersection the smart cab receives the receives the following inputs:

- Next Waypoint (North, South, East or West from current location)
- Traffic Light State in direction of Travel
- Vehicles from North
- Vehicles from South
- Vehicles from East
- Vehicles from West
- Deadline

The policy of the smart cab will have to maximize rewards, while reaching the goal in time. The smart cab can receive the following rewards:

- **++** — For each successful completed Trip
- **+** — For each action taken, that obeys traffic rules
- **-** — For each action taken, that violates traffic rules
- **- -** — For each action taken, that causes an accident

The smart cab drives in a homogenous environment. Thus, the smart cab's actions are not influenced by the current location. That's why I decided to exclude the global location from my states. This allows me to derive a reasonable policy within 100 iterations, like the rubric demands.

Still, mapping all the other inputs directly to states would result in 4480 possible states. Since the last three rewards are directly influenced by the action the smart cab takes, I choose to summarize the inputs in three risk levels:

- **drive_safe:** Moving to next waypoint won't violate any traffic rules

- drive_safe_left: green light, no oncoming traffic
- drive_safe_forward: green light
- drive_safe_right: green light / red light, no traffic from left

- **drive_reckless:** Moving to the next waypoint will violate traffic rules, but won't cause an accident
    - drive_reckless_left: red light, no oncoming traffic
    - drive_reckless_forward: red light, no traffic from left, no traffic from right
- **drive_accident:** Moving to the next waypoint will violate traffic rules and will cause an accident
    - drive_accident_left: green light, oncoming traffic / red light, oncoming traffic
    - drive_accident_forward: red light, traffic from left or right
    - drive_accident_right: red light, traffic from left

This essentially relieves the algorithm from figuring out the traffic rules and focuses the algorithm on choosing the appropriate risk tolerance based on the remaining time. I think this is an appropriate step to take, since in real life I would want an autonomous cars to have a clear understanding of the traffic rules.

The deadline is a bit problematic, because including the deadline directly in the possible states significantly increases the number of possible states. That's why I decided to use binning for the deadline:

- 100% - 75% remaining → 3
- 75% - 50% remaining → 2
- 50% - 25% remaining → 1
- 25% - 0% remaining → 0

I'll have to test this to ensure that these four bins are a good summarization of the remaining deadline. I may go back and adjust these bins.

The remaining time, next waypoint and the traffic situation together describe the state of my smart cab. In total there are 48 different combinations of these attributes, which is low enough to be learned fairly quickly and still large enough to allow for a robust policy. I'll represent these states in a dictionary like this:

```
state = {"waypoint": "forward", "deadline": 2, "risk_level": "drive_safe"}
```

I tuned the states in the last section, because some assumptions turned out to have an adverse effect on driving agent performance.

Implementing a Q-Learning Driving Agent

I went back to the videos with Michael and Charles and went over the section on Q-Learning on more time.
For the brain of the agent (the Q Matrix) I decided to use pandas, because DataFrames are fast and easy

to use, allow for robust indexing and are easy to export for analysis. The structure of the Q Matrix is as follows:

	stay	left	forward	right
state 1	0	0	0	0
state 2	...	...	...	...

With this DataFrame set up, I implemented the q-learning update formula:

```
self.q_values.loc[self.state, action] = (1 - self.alpha) * self.get_qvalue
(self.state, action) + self.alpha * (reward + self.gamma * self.max_q(new_
state))
```

The next problem that I addressed was the exploration-exploitation dilemma. I decided to perform random restarts with the degrading probability `self.epsilon`. In order to address the problem of local optima I implemented a function that given a state will prefer actions that have not been taken over actions with a high q value.

Actions taken are more targeted and in most cases the agent reaches the destination considerably faster than the agent that's taking random actions. This is because the agent is interested in getting the final reward. Another interesting thing is that the agent often drives in circles when enough time is remaining in order to maximize rewards - I wouldn't want a taxis driver like that ;). To solve this problem small negative rewards should be given during each iteration.

Improving the Q-Learning Driving Agent

This section probably took me the longest to complete, because I challenged a lot of the assumptions that I made earlier on. Below is a brief summary of what I considered:

States:

- I found that summarizing the inputs into risk levels was hurting performance and only resulted in about 85% trip completion. I decided to remove the risk level and include the traffic inputs in my states.
- After that adjust I still found that a lot of columns of the q-matrix were empty, which is sub-optimal. That's why I decided to reduce the number of states further by reducing the number of deadline options to two.

Final States:

- *Deadline:*
  - 1, if more than 40% of time is remaining

    - 0, if equal or less than 40% of time remaining
- *Next Waypoint*
- *Traffic Light*
- *Traffic from Left*
- *Oncoming Traffic*
- *Traffic from Right*

Paramters:
I tested various different combinations. These are the final numbers:

- Alpha: 0.9 — at the end of every trial: `self.alpha -= self.alpha * 0.05`
- Gamma: 0.5
- Epsilon: 0.9 — at the end of every trial: `self.epsilon -= self.epsilon * 0.6`

The final driving agent reaches the destination in about 97/98 out of 100 cases. My agent uses only about 45% of the available trip time. It uses about 0.7 more moves on average than are necessary to get to the destination and has a 22% chance of causing an accident in any given trial.
The optimal policy would be zero accidents with the minimal number of additional steps towards the destination. The best way to dress this would be to increase the negative reward the agent receives, when causing an accident with another driver.