Train a smart cab to drive

Implement a basic driving agent

Implement the basic driving agent, which processes the following inputs at each time step:

Next waypoint location, relative to its current location and heading, Intersection state (traffic light and presence of cars), and, Current deadline value (time steps remaining), And produces some random move/action (None, 'forward', 'left', 'right'). Don’t try to implement the correct strategy! That’s exactly what your agent is supposed to learn.

Run this agent within the simulation environment with enforce_deadline set to False (see run function in agent.py), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.

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

It is reasonable to observe that car just move randomly. I watched the car running for 10 min, it didn't reach the destination. I believe the car can eventually reach the destination with enough time. But this method is really low efficient.

Identify and update state

Identify a set of states that you think are appropriate for modeling the driving agent. The main source of state variables are current inputs, but not all of them may be worth representing. Also, you can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state.

At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

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

The object of this agent is guide the cab to get the destination as soon as possible. So the most important information is the current location and the next direction it will go. As a result, the waypoint is a critical factor of the state. Additionally, in the *environment.py* file, there is a penalty (negative reward) if the cab violates the red light police, so the state also should include the current traffic light condition.

Implement Q-Learning

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that.

Each action generates a corresponding numeric reward or penalty (which may be zero). Your agent should take this into account when updating Q-values. Run it again, and observe the behavior.

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

Since if the cab moves towards the destination, it can get more rewards. So when the cab starts moving, the probability of cab go towards to destination increases gradually until it reached the destination, on the other hand, it is less and less possible that cab moves away from the destination point. After a short time (compare to the random agent), the cab can always reach the destination.

Enhance the driving agent

Apply the reinforcement learning techniques you have learned, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive.

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?

The first the parameter I tweak is the initial Q value assigned to each state, I noticed that if the initial Q value of every states is 0 or 1, the cab will frequently stuck at some places since the Q value of each action is same, so the cab cannot make further action and doesn't move forever. So I need increase the initial value to 5. In this situation, the cab agent can perform learning process, and finally approach destination. In fact, I also check other initial Q values greater than 5, the performance is same.

Next important parameters is the learning rate and discount factor. The learning rate determines to what extent the newly acquired information will override the old information. A factor of 0 will make the agent not learn anything, while a factor of 1 would make the agent consider only the most recent information. However, the discount factor determines the importance of future rewards. A factor of 0 will make the agent only consider current rewards, while a factor approaching 1 will make it strive for a long-term high reward.

At beginning, I started with parameters pair (0.5, 0.5) (the first is learning rate, and the second is discount factor), and results turns out very good. In order to analyze the performance of the learning agent, I wrote a python function to compute the success rate and running time in all trails. The agent is implemented 5 times, the its performance is recorded in the table below.

	Time	**Success rate**
1	19.69	87
2	23.47	76
3	24.40	73
4	26.93	56
5	21.73	82
Average	23.24	74.8

But I found out that the cab learning slowly for early trails. And I silently adjust the parameters as (0.6, 0.4), which make better results. Also, its performance is shown in the table below.

	Time	**Success rate**
1	21.16	96
2	19.63	89
3	20.76	93
4	21.99	90
5	20.89	87
Average	20.89	91.0

From the comparison of above two tables, we can find out that the parameter set of (0.6, 0.4) performed better than (0.5, 0,5) with faster learning speed and higher success rate.