Train a Smartcab to Drive

A smartcab is a self-driving car from the not-so-distant future that ferries people from one arbitrary location to another. In this project, you will use reinforcement learning to train a smartcab how to drive.

Environment

The smartcab operates in an idealized grid-like city, with roads going North-South and East-West. Other vehicles may be present on the roads, but no pedestrians. There is a traffic light at each intersection that can be in one of two states: North-South open or East-West open. US right-of-way rules apply: On a green light, you can turn left only if there is no oncoming traffic at the intersection coming straight. On a red light, you can turn right if there is no oncoming traffic turning left or traffic from the left going straight.

Inputs

Assume that a higher-level planner assigns a route to the smartcab, splitting it into waypoints at each intersection. And time in this world is quantized. At any instant, the smartcab is at some intersection. Therefore, the next waypoint is always either one block straight ahead, one block left, one block right, one block back or exactly there (reached the destination). The smartcab only has an egocentric view of the intersection it is currently at (sorry, no accurate GPS, no global location). It is able to sense whether the traffic light is green for its direction of movement (heading), and whether there is a car at the intersection on each of the incoming roadways (and which direction they are trying to go). In addition to this, each trip has an associated timer that counts down every time step. If the timer is at 0 and the destination has not been reached, the trip is over, and a new one may start.

#1 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.

Question #1, in your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

Observation: Since the smartcab move randomly (None, 'forward', 'left', 'right') at each intersection regardless of the information it receives from the environment, and it doesn't follow the next waypoint provided by the route planner, basically the smartcab don't make it to the target location most of the time. The smartcab doesn't know its own exact location and the destination information. All it knows is the traffic information where it stays at each time step and it doesn't use the information it receives to decide its action.

The smartcab gets negative reward most of the time when its action doesn't follow the traffic rule. Coincidently, the smartcab reach the destination several times in 100 runs.

#2 Identify and update state

Identify a set of states that are appropriate for modeling the driving agent.

Question #2, justify why you picked these set of states, and how they model the agent and its environment.

Answer: I include {"Light", "Oncoming", "Left", "Right"} sensed by the car at each intersection from the environment. I also include the {"next_waypoint"} recommended by route planner into the states. {"Light", "Oncoming", "Left", "Right"} help the smart car learn the environment and traffic laws by observing the environment at each intersection. The reason to include "next_waypoint" is because the reward is calculated based off of the traffic light and next_waypoint.

Deadline and time step are not included into the state since they are not essential to the decision. And it would simplify the problem if we don't include Deadline or time step. The deadline and time step would greatly blow up the state space, thus taking a long time for the q-table to converge.

#3 Imlpement Q-Learning

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Pick the best action available from the current state based on Q-values. Each action generates a corresponding numeric reward or penalty. Agent take this into account when updating Q-values.

Question #3, what changes do you notice in the agent's behavior?

Answer: When the simulation starts, the agent still takes random actions because the Q-value table has just been initialized at this time and there is no Q-value been calculated and put into the Q-value table yet.

When a new state-action pair is detected and being added to the Q-value table, it will be initialized with 0. Then the Q-value of the state-action pair is updated based on the reward the agent gets and the maximum Q-value of the neighboring states.

After 10 to 15 runs, the agent starts to select the preferred actions according to Q-value table in certain states. This is useful to help the agent get to the destination. We can see that the agent is learning by continuously updating the Q-value table at each run.

However, sometimes the agent prefers to take "None" action or be in clockwise loops. This is because the agent only chooses the action with highest Q-value or chooses the best path it knows in some cases without fully exploring different paths which may get it better rewards. This is a local maximum problem. So for this kind of problem, we need to implement epsilon-greedy algorithm to optimize the system, which will be addressed in the following part.

#4 Enhance the driving agent

Apply the reinforcement learning techniques, and tweak the parameters (learning rate, discount factor, action selection method, etc.) to improve the performance of the agent.

Question #4, 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?

Answer: In order to eliminate the dilemma situation mentioned above in question #3, I implement the epsilon-greedy algorithm for selecting the optimal action. At every time step, generate a random number (between 0 and 1) and compare it with the epsilon which decays over the time (1/step t) during each run. At the beginning of each run, the agent has greater possibility to do random move because it is easy for the agent to get into local maximum situation at the early stage. When epsilon's value drops, there is small possibility for the agent to take random action and the action tends to select the optimal action according to the Q-value table since the table is improved and updated by reward and Q-value from neighboring states.

Initially, I set all the learning rate, decay factor and epsilon to 1/t where t is the time step during each run (1,2,3,4,5…). I run 5 simulations (100 runs each simulation), the average successful rate is 84.0 Then I try to change the value of Gamma to improve the result. However, if I set gamma to 0.9, the average successful rate is 69.0. If I set it to 0.1, the average successful rate is greatly improved to 87.8 With gamma fixed to 0.1. I try to change the alpha. When alpha equals to 0.1, success rate is 89.4. When alpha is set to 0.9, successful rate is 88.7.

I observe that the reason why the penalty rate is so high is because sometimes the epsilon is too high especially during the last several trials of the simulation. At the end of the simulation, since the agent has already learned a lot so we don't really need the epsilon to be set too high to allow the agent to act

randomly. So I change the epsilon from 1/step to 0.6/step while alpha = 0.9 and gamma = 0.1. I get 88.3 successful rate and 8.35% penalty rate, which is the best result I can get from 100 runs.

Epsilon	Alpha	Gamma	Ave. Success (10 simulations)	Ave. Penalty Rate (wrong actions/total moves)
1/step	**1/step**	**1/step**	**84.0**	**17.05%**
1/step	**1/step**	**0.9**	**69.0**	**31.37%**
1/step	**1/step**	**0.1**	**87.8**	**12.45%**
1/step	**0.1**	**0.1**	**89.4**	**12.48%**
1/step	**0.9**	**0.1**	**88.7**	**13.05%**
0.6/step	**0.9**	**0.1**	**88.3**	**8.35%**

Question #5, 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?

Answer: Based on the result above, the best successful rate I can get from 100 runs is 88.3 with penalty rate of 8.35%. It means that the agent can get to the destination 88.3 times out of 100 run. The penalty rate is low considering some of the wrong actions are generated due to the epsilon-greedy algorithm. The agent is getting close to find an optimal policy if it can have more runs on the map, which would further reduce the penalty rate.

I modify environment.py and simulator.py so that the agent n_trials can be passed to agent.reset(). If n_trail > 90, epsilon will be set to -1.0, which means that the agent will not take the random action if the number of runs is greater than 90.

From the result , we can see that, in the last 10 runs, the agent get to the destination. And the reward is either 2.0 or 0.0, which means that the agent is taking the right actions and following the rule of the road.

If we set the n_trails to 1000, the successful rate would increase to 95.5%.