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
Your 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.

To understand how to correctly yield to oncoming traffic when turning left, you may refer to this official drivers' education video, or this passionate exposition.

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.

Outputs
At any instant, the smartcab can either stay put at the current intersection, move one block forward, one block left, or one block right (no backward movement).

Rewards
The smartcab gets a reward for each successfully completed trip. A trip is considered “successfully completed” if the passenger is dropped off at the desired destination (some intersection) within a pre-specified time bound (computed with a route plan).

It also gets a smaller reward for each correct move executed at an intersection. It gets a small penalty for an incorrect move, and a larger penalty for violating traffic rules and/or causing an accident.

Goal
Design the AI driving agent for the smartcab. It should receive the above-mentioned inputs at each time step t, and generate an output move. Based on the rewards and penalties it gets, the agent should learn an optimal policy for driving on city roads, obeying traffic rules correctly, and trying to reach the destination within a goal time.

Run
Make sure you are in the top-level project directory `smartcab/` (that contains this README). Then run:

python smartcab/agent.py

OR:

python -m smartcab.agent

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.

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?

The implementation of a basic driving agent is in agent.py. The random agent eventually reaches the target location, but the actions are randomly chosen, which means the agent does not learn from the previous behavior and improve the action. So as I observed, sometimes the agent's action is not optimal, for instance, the agent keep going forward when there is a red light, or remains in the same position even there is no other oncoming cars or red light. So the reward is terrible. To check the success rate of reaching the destination during the deadline, I set update_delay=0.01, enforce_deadline = True and n_trails = 100, then implemented the basic agent 5 times, the average success rate is 19%.

2. 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.

Before implementing Q-learning algorithm, the first task was to identify the possible states, here are some possible options:

- Intersection state (traffic light and presence of cars): I chose intersection state as one of state variables because we want to train the cab to perform legal moves and obey the US right-of-way rules.

- Next waypoint location: I chose this to model the destination relatives to the current location, we cannot directly choose destination and location as variables since destination changes all the time and location will cause an issue of having large amount of states and taking a long time for q values to converge, so we choose next waypoint location instead.
- Deadline: I do not include this variable as state variable here because if there are too many state to visit, it will make q values a long time to converge and the q-tables will be very large in memory

So finally I considered Intersection state(traffic light and presence of cars) and Next waypoint location as two state variables.

3. 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?

Q-Learning Implementation is in Qlearning.py. Here are the steps for implementing Q-Learning Algorithm:

1. Initialize the Q-values: I initialized q-values as 5.0
2. Observe the current state: I considered the following state variables:

- next waypoint
- Intersection state (traffic light, (oncoming, left, right))

3. For the current state, choose an action based on the selection policy(epsilon-greedy), which is available in get_action function. The epsilon-greedy approach chooses an action with the highest Q value with a probability of (1-epsilon) and explores with a probability epsilon, given the state, which is like flipping a coin at every time step and then deciding to make either a random action(exploration) or follow the optimal policy given the inequality.
4. Take the action, and observe the reward and as the new state
5. Update the Q-value for the state using the observed reward the the maximum reward possible for the next state:

$$Q_{t+1}(s_t, a_t) \leftarrow \underbrace{Q_t(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t, a_t)}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q_t(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q_t(s_t, a_t)}_{\text{old value}} \right)$$

6. Set the state to the new state, and repeat steps until smartcab arrives at destination

Performance:
Alpha value(learning rate) was set to 0.9, Gamma value(discount factor) was set to 0.2, Epsilon was set to 0.05, I set these values by trying many possible values and compare the performances, the details will be shown later.

Implemented the Qlearning.py after determining these variables. The agent has reached the destination before deadline 95 times out of 100 runs.

Enhance the driving agent
Apply the reinforcement learning techniques you have learnt, 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?

I tuned parameters(alpha, gamma and epsilon) by trying different combinations, here are the performances:

alpha	gamma	epsilon	success	# moves(n = 100)	cum_reward(n = 100)
0.5	0.05	0.9	26/100	23	10.5
0.5	0.05	0.5	55/100	10	13.0
0.5	0.2	0.5	62/100	15	14.0
0.7	0.2	0.5	69/100	16	11.0
0.9	0.6	0.05	89/100	20	27.5
0.7	0.2	0.05	91/100	16	23
0.5	0.2	0.05	93/100	28	34.5
0.9	0.05	0.05	93/100	15	20.5
0.9	0.4	0.05	94/100	20	23
0.9	0.2	0.05	98/100	12	18.5

So the best combination is Alpha = 0.9, Gamma = 0.2 and Epsilon = 0.05. Now we compared the q-learning agent with random agent, the success rate is around 95% which is much better than the basic random agent(19%).

We noticed that after learning optimal policy, the agent get to the destination quickly with less number of moves in the end, which means the agent not only learned how to get to the destination but the best route to get to the destination instead of going in circles.

Also, the cab always reaches the destination with large positive cumulative reward, the cumulative is smaller in the end compared to the beginning because of the smaller number of moves. So basically the cab are learning to obey traffic rules and make reasonable action.

So I think the cab is acting optimally not only learning an optimal policy by taking the right moves but also choosing the smaller route.