Smartcab Project Report

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.

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 runfunction 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 the basic agent is represented in file agent_basic.py.

The basic agent is randomly reaching the destination with a combination of variables, such as traffic light, oncoming traffic, and other vehicles. This basic agent did not take into consideration of rewards gain and therefore it does not taking actions that will maximize rewards gain.

This basic agent neither remember the past correct choices and reinforce the correct choices in the future, nor remember the past bad choices and avoid them in the future. It is constantly trying new actions without considering the past history. For example, even it

was punished and gain negative rewards many times by passing a red light, it still pass the red light next time at the same chance. In addition, the basic agent does not choose optimal actions. For example, it sometimes took no action while there is no traffic or no red light.

For comparing the performance of different agents, we quantified the results for three trails. In the simulation, the basic agent enforce_deadline = Trure, trials = 100, update_delay = 0.01. The success rate of the basic agent are respectively 29%, 27% and 22%.

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.
There are many variables that could be used as the inputs of the states, including but not only the traffic lights, oncoming traffic, left vehicles, right vehicles, next_waypoint, deadline, etc. However, only the useful ones should be chosen. For example, according to US traffic rule, the right vehicle states should not affect the agent's decision, and therefore it is not necessary to be included in the state in the model.

Here four variables were chosen, traffic light (input[0]), oncoming traffic(input[1]), left vehicles (input[3]) and next_waypoint. The first three are obvious to choose. The agent needs to stop when the light is red and move forward when the light is green. It also need to avoid clashing with oncoming traffic and left vehicles. We chose next_waypoint, because it made the agent moving towards the destination in an efficient manner, instead of random direction. The deadline should not be included, because otherwise the size of the stat space will be larger.

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.

The initial value of Q has been set to 0. When gathering the inputs for the next_state, I have used the updated version of self.next_waypoint.

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

Q-Learning is implemented in file agent.py

If we think the previous basic agent as a person that only take actions by judging the current environment, the agent implemented with Q-Learning is like a person installed with a strong brain, having a good memory of the past experience and learned from that.

The Q-learning agent has a better understanding of the world. It learned the traffic rule and heading to the destination. If it gained bad rewards by passing red lights in the past, it knows to avoid passing red light in the future. It also won't stay still and take no action, if moving is a better choice.

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?

For the basic Q-Learning, the max Q value is chosen. Here an exploration factor epsilon was added to the current version of Q-Learning. In this setting, a random value was generated. If the value is less than epsilon, the agent will take a random action, otherwise the agent will take action towards max q. The smaller epsilon is, the lower the chance random action will be taken.

In the case random action is taken, max q is still taken for the next_state to update the q for current state q to avoid completely random choice at this case. In conclusion, we introduce exploration factor to this learning by introducing epsilon.

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 agent reached the destination. Several combinations of alpha, gamma, and epsilon have been tried to gain different learning experience for the agent. The results for various trails are as follows:

epsilon	alpha	gamma	results
1	0.3	0.6	23 success
0.5	0.5	0.5	51 success
0.3	0.2	0.8	65 success
0.1	0.3	0.9	79 success
0.2	0.3	0.6	85 success
0.1	0.2	0.8	53 success
0.1	0.1	0.8	79 success
0.05	0.1	0.9	57 success

By trying, error and improve, we can get optimized parameter combinations. It seems random factor decrease and the success increases to some extent. These are only several trials. Better results could be obtained by continuing training the agent. For 100 trails, the best parameter are (epsilon = 0.1, alpha = 0.1, gamma = 0.8) and (epsilon = 0.05, alpha = 0.1, gamma = 0.9). Both reached an accuracy of 87 successes out of 100 trails.

The agent is optimally trained. Fro example, while epsilon = 0.1, alpha = 0.1, gamma = 0.8 , in the first 10 trails, 3 out of 10 failed, in the last 10 trails, only 1 out of 10 failed. The accuracy increased after training.

In the last 10 trials, in 162 actions, there are 2 wrong actions. Both wrong actions are turing left while (Input light = red, oncoming =left, right = None, left = None).

In the last 10 trains, the agent did not always follow the planner. A few times the agent act None while it should not.