## Train a Smartcab How 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.

## Tasks

Download smartcab.zip, unzip and open the template Python file `agent.py` (do not modify any other file). Perform the following tasks to build your agent, referring to instructions mentioned in `README.md` as well as inline comments in `agent.py`.

Also create a project report (e.g. Word or Google doc), and start addressing the questions indicated in *italics* below. When you have finished the project, save/download the report as a PDF and turn it in with your code.

### 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 the basic agent is available in `agent.py`
>
> The random agent is eventually reaching the destination with variables at play (other vehicles, oncoming traffic, traffic lights, etc) but the approach to reach the goal is clearly random and not optimal. In fact the agent is not optimising the rewards gain. The agent does not prefer actions that it has tried in the past and found to be effective in producing reward. In fact the agent is constantly in "discovery" mode where he try new action without any knowledge of the past rewarding (or not) actions. The agent does not exploit what it has already discover in order to obtain reward. For example the agent does not understand, run after run, that running a red-light produce bad reward. We noticed that several times, the agent, being close to the goal choose a different direction than the prefered to reach the destination. We can also remark the fact that the agent sometimes remain in position (action = `None`) even when there is no oncoming traffic or red light.
>
> In order to compare the performance of the agent, we force `enfore_deadline` to `True` and report the results of 100 trials made with an `update_delay` of 0.2. Results for the random agent can be seen in the `output_random1.txt`, `output_random2.txt` and `output_random3.txt` files available the report directory.
>
> The success rate of the random agent are respectively 16%, 19% 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.*

We have many options base the state update. For example we can use the `next_waypoint` from the planner or traffic light or even the traffic data of potential cars coming on the left or right. The deadline is also another factor that can be used to update the state.

For this model we choose to use the data available at each interesection. This include the traffic lights (red / green), the oncoming traffic (left, right or None) from 2 other cars (input[1] = oncoming, input[3] = left). Due to the circulation rules in US, the traffic coming from the right does not matter and will therefore be ignore from the states.

We also decided to use the result of `next_waypoint` to defined the target to reach the desitnation. The agent can use this information to act properly to reach the destination in an efficient manner.

**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 is implemented in it's own class `QLearn.py`.
We can notice that QLearning give to the agent a better appreciation of the world. He is learning the traffic lights rules and is trying the reach the destination.

For example, we can notice that the agent doesn't choose to stay in position anymore in case of no traffic of green light. This is already a nice improvment. We can also notice that the agent is not choosing the red light direction once he know the bad impact of this 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?*

In this implementation we introduce exploration instead of the normal tactic of choosing the max Q value. An `epsion` value has been added so that we generate a random number. If that value is less than `epsion`, a random action is choosen, otherwise, the max q value is selected from the Q table.

In order to avoid the problem of picking a random move even if we know for sure the best option, a special logic has been implemented to cover this case.

If the randomly generated value is less than `epsilon`, then randomly add values to the q values for this state, scaled by the maximum q value of this state. In this way, exploration is added, but we're still using the learned q values as a basis for choosing the action, rather than just randomly choosing an action completely at random.

*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 is reaching destination after some trial steps. Several value of alpha, gamma and epsilon has been tried out in order to have the best learning experience for our agent. The first line bellow is an implementation where the exploration is not activated.

| alpha | gamma | epsion | results |
|---|---|---|---|
| 0.2 | 0.5 | 1 | 13 wins |
| 0.2 | 0.5 | 0.1 | 67 wins |
| 0.2 | 0.8 | 0.1 | 39 wins |
| 0.1 | 0.8 | 0.1 | 74 wins |
| 0.1 | 0.9 | 0.1 | 76 wins |
| 0.5 | 0.9 | 0.1 | 56 wins |
| 0.05 | 0.9 | 0.05 | 60 wins |
| 0.1 | 0.9 | 0.05 | 83 wins |

The results of these simulation are available in the report directory.

By trial and error we can isolate the ideal parameter combination.

The last line in the above table is the best results we can reached with the Q-Learning implementation. Better results may be possible by constanlty reducing the random factor once the agent is getting more and more skills over time.

## reference

Studywolf - REINFORCEMENT LEARNING