**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](https://www.youtube.com/watch?v=TW0Eq2Q-9Ac), or this [passionate exposition](https://www.youtube.com/watch?v=0EdkxI6NeuA).

**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**

**Setup**

You need Python 2.7 and [**pygame**](http://pygame.org/) for this project: <https://www.pygame.org/wiki/GettingStarted>  
For help with installation, it is best to reach out to the pygame community [[help page](http://www.pygame.org/wiki/info), [Google group](https://groups.google.com/forum/#!forum/pygame-mirror-on-google-groups), [reddit](https://www.reddit.com/r/pygame/)].

**Download**

Download [smartcab.zip](https://s3.amazonaws.com/content.udacity-data.com/courses/nd009/projects/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 inagent.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?*

Since the action was randomly chosen, the agent wasn’t optimizing the future action from the past rewarding actions. If we set the ‘enforce\_dealine’ to be false, it will reach the target somehow, but with too many unnecessary steps. If we set the ‘enforce\_dealine’ to be true. The agent always failed running out of times.

**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 three places where we could find the state update. The first is next waypoint output from planner, as one action is complete, and then the next waypoint begins. The second is often in the process of intersection. A cab needs to handle traffic lights and oncoming traffic so as to choose the right path and avoid accident. The third is counting down the deadline.

In this agent2.py, we are going to update the second one, the handling of traffic rules in the intersection.

### 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 anice 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?