Train a Smartcab to Drive

Table of Contents

Project Description

A smartcab is a self-driving car from the not-so-distant future that ferries people from one arbitrary location to another.

Project Aim

The task is to use reinforcement learning to train a smartcab how to drive. The smartcab AI driving agent should receive the below-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.

The Rules of Engagements

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 it is assumed there is no pedestrian. 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.

Input Assumptions

- 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
- A higher-level planner assigns a route to the smartcab, splitting it into waypoints at each intersection.
- 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.

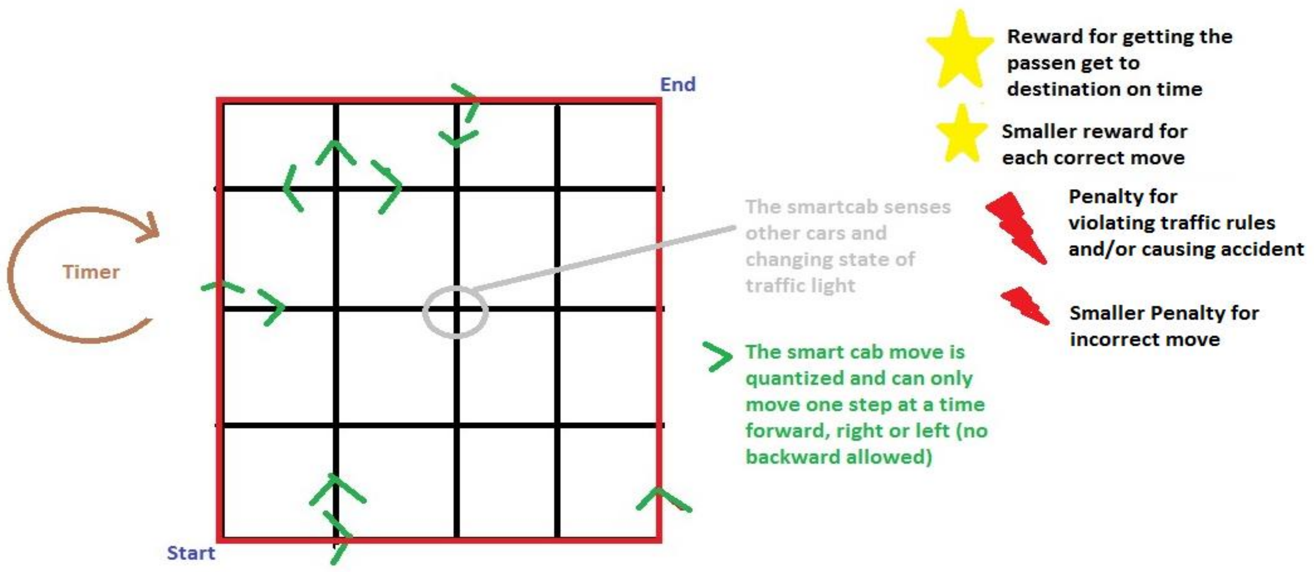
Output Assumptions

- 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 is allowed).

Reward

- 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.

High Level Representation


Reward for getting the passen get to destination on time
Smaller reward for each correct move
Penalty for violating traffic rules and/or causing accident
Smaller Penalty for incorrect move
End
Timer
The smartcab senses other cars and changing state of traffic light
The smart cab move is quantized and can only move one step at a time forward, right or left (no backward allowed)
Start

Software and Libraries

- Python 2.7
- Pygame 1.9.1
- iPython Notebook (with iPython 4.0)

Installation

1. Copy the content of the 'code' folder locally
2. Open a cmd line and run python SmartCab\agent.py

Implementation of a basic driving agent

The aim is to 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),
- Produce some random move/action (None, 'forward', 'left', 'right').
- The agent is given unlimited time to reach the destination

The implementation can be found in the 'Agent.py' file (see the Update() method). The user needs to explicitly set the agent to run as random (see a.set_agent_type('random') in the run() function). As the name suggests, the action is selected randomly for each agent update, from a vector containing four elements, namely [None, 'forward', 'left', 'right'] . The direct consequence is that the Agent does not seem to find the target, even after a large number of moves. It also generates la large proportion of negative rewards. The Agent was stopped by the user after a 2mins run, as it looked like it was running 'infinitely'. The 'RandomAgent.txt' shows that out of the 10 trials, only the first one was ever run. This is shown by the unique log entry relating to the number of trials, namely 'Simulator.run(): Trial 0'. If we gave it infinite time, it would probably hit the final destination. But this is not a workable model. The below table shows the reward broken down by category for this example. As this random, the data present very little value, as another run would produce different values. However, it provides the intuitive notion that a random behaviour is not the correct behaviour for AI optimised path finding.

Reward Value	**2**	**0.5**	**-1**	**1**	**10**
Rewad Description	**Correct Move**	**Incorrect Move**	**Traffic Violation**	**Wait**	**Successful Trip**
Count	33	73	101	0	0
Total	207				
Ratio	16%	35%	49%	0%	0%

In order to run this experiment, ensure the agent.py code is set as follows:

```
def run():
  (…)
 agent_type = 'random'
```

Identify and update state

In this next step, we will identify a set of states that are appropriate for modelling the driving agent. The main source of state variables currently live in the 'inputs' object. At each time step, we process the 'inputs' object and update the current state. We run it several times to observe how the reported state changes through the run.

The states that have been retained are:

- ***light***: ('red' or 'green'): it establishes whether the smart cab should stop or continue its route.
- ***oncoming***: it indicates whether the traffic comes from the 'left', 'right', 'forward' or None (i.e. no oncoming traffic) -> it establishes whether a move has a full or partial reward, depending on whether it is correct, incorrect or violating traffic rules.
- ***left***: it indicates whether the traffic coming from the 'left' is planning to turn 'left', 'right', 'forward' or None. This does matter, especially when a car coming from left wishes to go forward (as it has a green light), while the automated car faces a red light, but still wishes to turn right. The smart cab action should be none, else an accident will occur.
- ***waypoint***: it can take the following states: None, 'forward', 'left', 'right'. This indicates the next point the smartcab wishes to go to.
-

The states that were not retained are:

- ***right***: indicates whether the traffic coming from the 'right' is planning to turn 'left', 'right', 'forward' or None. When traffic light is green for traffic coming from the right, then there can be no collision with smartcab. Even when the smartcab faces a red traffic light but wishes to turn right.
  When the smart cab light is green, this could provoke collision with traffic coming from the right and turning right on a red light. But in this case, this would not be the smart cab fault. Therefore, this case is discarded.
- ***location***: this is used in conjunction with the heading to establish whether the destination is reached.
- ***heading***: this is used in conjunction with the location to establish whether the destination is reached.
- ***destination***: it establishes whether the destination is reached -> reaching destination is the result of change in state, not state per say.
- ***deadline***: it indicates whether the destination has been reached within the defined deadline -> being within a time deadline no related to the smart cab move, and therefore state change. The main reason deadline is not included is due to the number of states that the deadline variable can take. Consequently, the state space would increase exponentially. The size increase could not feasibly be explored by the agent in 100 trials of the game. There would be too many states for the agent to be able to visit. In order to learn the value of each action, the agent would need to visit every state many times over.

The ‘waypoint’ models the behaviour of the smartcab agent, by indicating where it wants to go on the grid. The ‘light’, ‘oncoming’ and ‘left’ states model the state of the environment. The ‘light’ tells the smartcab whether it should stop or continue its route. The ‘oncoming’ and ‘left’ indicate respectively what direction a dummy agent will take from the current intersection when it comes in front of the smartcab or from its left.

The code relating to the selected state changes is provided below (please see the agent.py file).

```
(…)
elif (self.__type == 'identify_and_update_state'):
        action_okay = True
        if self.next_waypoint == 'right':
           if inputs['light'] == 'red' and inputs['left'] == 'forward': action_okay = False
        elif self.next_waypoint == 'forward':
           if inputs['light'] == 'red': action_okay = False
        elif self.next_waypoint == 'left':
           if inputs['light'] == 'red' or (inputs['oncoming'] == 'forward' or  inputs['oncoming'] == 'right'): action_okay = False
        if action_okay == False: action = None
        self.state = ('next_waypoint:' + str(self.next_waypoint), 'light:' +
                 str(inputs['light']), 'oncoming:' +  str(inputs['oncoming']), 'left:' +
                 str(inputs['left']))
(…)
```

In order to run this experiment, ensure the agent.py code is set as follows:

```
def run():
  (…)
 agent_type = 'identfty_and_update_state'
```

Implement Q-Learning

In this section, we implement a free-model reinforcement learning technique, named Q-Learning. “Q-learning can be used to find an optimal action-selection policy for any given (finite) Markov decision process” (Wikipedia, 2016). Q-Learning is an “action-value function that ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter. A policy is a rule that the agent follows in selecting actions, given the state it is in. When such an action-value function is learned, the optimal policy can be constructed by simply selecting the action with the highest value in each state. (…) The goal of the agent is to maximize its total reward. It does this by learning which action is optimal for each state”. The Q-Learning algorithm used for this project is mathematically defined as follows (Wikipedia, 2016):

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

Parameters definition

$Q_{t+1}(s_t,a_t)$	The previous Q-table value given a state (s_t) and an action (a_t).
$\alpha_t(s_t,a_t)$	The learning rate. This is a number set by the user. It can take values between $0 < \alpha_t \leq 1$.
R_{t+1}	The reward table. It defines the reward based on a policy (i.e. the selection of a state) for each possible action. The reward table for the smartcab is defined below.
ϒ	The discount factor. It determines the importance of future rewards. A factor of 0 will make the agent "myopic" (or short-sighted) by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward.
max Q_t (s_{t+1},a_t)	An estimate of the optional future value. S_{t+1} is chosen randomly from universe of available states.

The logic

Step 1 – Initialisation Phase – $Q_t(s_t,a_t)$ is set to an arbitrary fixed value. Here,we have chosen a 128x4 matrix set to 0.
Step 2 – The Learning Phase – A learning iteration is started until the learning is stopped. (Burlap, 2016) describes the process as follows:

- ✓ Choose an action (here randomly) in the current world state (s_t) based on current Q-value estimates ($Q(s_t,-)$).
- ✓ Take the action (a_t) and observe the outcome state (s_t) and reward (R_{t+1})
- ✓ Update $Q_{t+1}(s_t,a_t)$ following the above formula.

The Reward table

Each row of the reward table corresponds to a potential state relating to i) the environment surrounding the smartcab (i.e. the ‘light’, ‘on_coming’,’left’ states) and ii) where the smartcab can go (‘next_way_point’). For each state, an action is possible and gets rewarded with a score based on the policy. In this case, the policy being followed corresponds to the logic defined by pre-built the environment.py code snippet:

```
def act(self, agent, action):
(…)
    if action is not None:
      if move_okay:
        (…)
        reward = 2 if action == agent.get_next_waypoint() else 0.5
      else:
        reward = -1
    else:
      reward = 1
 (…)
```

The Reward table is built following these rules:

- ✓ A possible action set to 'none' always returns 1
- ✓ A correct move, called 'move_okay' in the code above is set to 0.5 unless it corresponds to the 'next_way_point' then it provides a higher score of 2.
- ✓ An incorrect move returns a reward of -1
- ✓ There are cases, highlighted in yellow in the below table, where an action would provoke an accident. Therefore, the reward is set to -1.

	Descriptions (broken by state individual components)					Possible Actions				Comments
	Id	light	on_coming	left	next_way_point	none	left	forward	right	
States	0	green	none	none	none	1	0.5	0.5	0.5	
	1	green	none	none	right	1	0.5	0.5	2	
	2	green	none	none	left	1	2	0.5	0.5	
	3	green	none	none	forward	1	0.5	2	0.5	
	4	green	none	right	none	1	0.5	0.5	0.5	
	5	green	none	right	right	1	0.5	0.5	2	
	6	green	none	right	left	1	2	0.5	0.5	
	7	green	none	right	forward	1	0.5	2	0.5	
	8	green	none	left	none	1	0.5	0.5	0.5	
	9	green	none	left	right	1	0.5	0.5	2	
	10	green	none	left	left	1	2	0.5	0.5	
	11	green	none	left	forward	1	0.5	2	0.5	
	12	green	none	forward	none	1	0.5	0.5	0.5	
	13	green	none	forward	right	1	0.5	0.5	2	
	14	green	none	forward	left	1	2	0.5	0.5	
	15	green	none	forward	forward	1	0.5	2	0.5	
	16	green	left	none	none	1	0.5	0.5	0.5	
	17	green	left	none	right	1	0.5	0.5	2	
	18	green	left	none	left	1	2	0.5	0.5	
	19	green	left	none	forward	1	0.5	2	0.5	
	20	green	left	right	none	1	0.5	0.5	0.5	
	21	green	left	right	right	1	0.5	0.5	2	
	22	green	left	right	left	1	2	0.5	0.5	
	23	green	left	right	forward	1	0.5	2	0.5	
	24	green	left	left	none	1	0.5	0.5	0.5	
	25	green	left	left	right	1	0.5	0.5	2	
	26	green	left	left	left	1	2	0.5	0.5	
	27	green	left	left	forward	1	0.5	2	0.5	
	28	green	left	forward	none	1	0.5	0.5	0.5	
	29	green	left	forward	right	1	0.5	0.5	2	
	30	green	left	forward	left	1	2	0.5	0.5	
	31	green	left	forward	forward	1	0.5	2	0.5	
	32	green	right	none	none	1	0.5	0.5	0.5	
	33	green	right	none	right	1	0.5	0.5	2	
	34	green	right	none	left	1	2	0.5	0.5	
	35	green	right	none	forward	1	0.5	2	0.5	
	36	green	right	right	none	1	0.5	0.5	0.5	
	37	green	right	right	right	1	0.5	0.5	2	
	38	green	right	right	left	1	2	0.5	0.5	
	39	green	right	right	forward	1	0.5	2	0.5	
	40	green	right	left	none	1	0.5	0.5	0.5	
	41	green	right	left	right	1	0.5	0.5	2	
	42	green	right	left	left	1	2	0.5	0.5	
	43	green	right	left	forward	1	0.5	2	0.5	
	44	green	right	forward	none	1	0.5	0.5	0.5	

45	green	right	forward	right	1	0.5	0.5	2	
46	green	right	forward	left	1	2	0.5	0.5	
47	green	right	forward	forward	1	0.5	2	0.5	
48	green	forward	none	none	1	0.5	0.5	0.5	
49	green	forward	none	right	1	0.5	0.5	2	
50	green	forward	none	left	1	2	0.5	0.5	
51	green	forward	none	forward	1	0.5	2	0.5	
52	green	forward	right	none	1	0.5	0.5	0.5	
53	green	forward	right	right	1	0.5	0.5	2	
54	green	forward	right	left	1	2	0.5	0.5	
55	green	forward	right	forward	1	0.5	2	0.5	
56	green	forward	left	none	1	0.5	0.5	0.5	
57	green	forward	left	right	1	0.5	0.5	2	
58	green	forward	left	left	1	2	0.5	0.5	
59	green	forward	left	forward	1	0.5	2	0.5	
60	green	forward	forward	none	1	0.5	0.5	0.5	
61	green	forward	forward	right	1	0.5	0.5	2	
62	green	forward	forward	left	1	2	0.5	0.5	
63	green	forward	forward	forward	1	0.5	2	0.5	
64	red	none	none	none	1	-1	-1	0.5	
65	red	none	none	right	1	-1	-1	2	
66	red	none	none	left	1	-1	-1	0.5	
67	red	none	none	forward	1	-1	-1	0.5	
68	red	none	right	none	1	-1	-1	0.5	
69	red	none	right	right	1	-1	-1	2	
70	red	none	right	left	1	-1	-1	0.5	
71	red	none	right	forward	1	-1	-1	0.5	
72	red	none	left	none	1	-1	-1	0.5	
73	red	none	left	right	1	-1	-1	2	
74	red	none	left	left	1	-1	-1	0.5	
75	red	none	left	forward	1	-1	-1	0.5	
76	red	none	forward	none	1	-1	-1	0.5	
77	red	none	forward	right	1	-1	-1	-1	collision with left traffic
78	red	none	forward	left	1	-1	-1	-1	collision with left traffic
79	red	none	forward	forward	1	-1	-1	-1	collision with left traffic
80	red	left	none	none	1	-1	-1	0.5	
81	red	left	none	right	1	-1	-1	2	
82	red	left	none	left	1	-1	-1	0.5	
83	red	left	none	forward	1	-1	-1	0.5	
84	red	left	right	none	1	-1	-1	0.5	
85	red	left	right	right	1	-1	-1	2	
86	red	left	right	left	1	-1	-1	0.5	
87	red	left	right	forward	1	-1	-1	0.5	
88	red	left	left	none	1	-1	-1	0.5	
89	red	left	left	right	1	-1	-1	2	
90	red	left	left	left	1	-1	-1	0.5	
91	red	left	left	forward	1	-1	-1	0.5	
92	red	left	forward	none	1	-1	-1	0.5	
93	red	left	forward	right	1	-1	-1	-1	collision with left traffic
94	red	left	forward	left	1	-1	-1	-1	collision with left traffic
95	red	left	forward	forward	1	-1	-1	-1	collision with

									left traffic
96	red	right	none	none	1	-1	-1	-1	
97	red	right	none	right	1	-1	-1	-1	collision with on coming traffic
98	red	right	none	left	1	-1	-1	-1	
99	red	right	none	forward	1	-1	-1	-1	
100	red	right	right	none	1	-1	-1	-1	
101	red	right	right	right	1	-1	-1	-1	collision with on coming traffic
102	red	right	right	left	1	-1	-1	-1	
103	red	right	right	forward	1	-1	-1	-1	
104	red	right	left	none	1	-1	-1	-1	
105	red	right	left	right	1	-1	-1	-1	collision with on coming traffic
106	red	right	left	left	1	-1	-1	-1	
107	red	right	left	forward	1	-1	-1	-1	
108	red	right	forward	none	1	-1	-1	-1	
109	red	right	forward	right	1	-1	-1	-1	collision with on coming traffic
110	red	right	forward	left	1	-1	-1	-1	
111	red	right	forward	forward	1	-1	-1	-1	
112	red	forward	none	none	1	-1	-1	-1	
113	red	forward	none	right	1	-1	-1	-1	collision with on coming traffic
114	red	forward	none	left	1	-1	-1	-1	
115	red	forward	none	forward	1	-1	-1	-1	
116	red	forward	right	none	1	-1	-1	-1	
117	red	forward	right	right	1	-1	-1	-1	collision with on coming traffic
118	red	forward	right	left	1	-1	-1	-1	
119	red	forward	right	forward	1	-1	-1	-1	
120	red	forward	left	none	1	-1	-1	-1	
121	red	forward	left	right	1	-1	-1	-1	collision with on coming traffic
122	red	forward	left	left	1	-1	-1	-1	
123	red	forward	left	forward	1	-1	-1	-1	
124	red	forward	forward	none	1	-1	-1	-1	
125	red	forward	forward	right	1	-1	-1	-1	collision with on coming traffic
126	red	forward	forward	left	1	-1	-1	-1	
127	red	forward	forward	forward	1	-1	-1	-1	

Implementation

The code relating to the Q-learning implementation can be found in the following files.

File Name	Description
agent.py	Generates an action based on different policy, i.e. random or Q-learning
q_learner.py	Implements the Q-learning algorithm
q_table.py	Implements the Q-table helper functions
r_table.py	Implements the R-table helper functions
Utilities.py	Implements shared functions, such as the reward table

The calling code is as follows:

```
def update(self, t):
    (...)
    elif (self.__type == 'simple_q_learning'):
        #get the initial state
        self.state = {'next_waypoint': str(self.next_waypoint), 'light': inputs['light'], 'oncoming': inputs['oncoming'], 'left': inputs['left']}
        #find the corresponding state index
        state_idx = Utilities().parse_state(self.state)
        #learn... i.e compute q and update the q_table
        self.ql.run_step(state_idx)
        #update the action
        action = Utilities().parse_action(self.ql.get_qtable_best_action(state_idx))
    else:
        print "Type not supported: " + self.__type
    # TODO: Learn policy based on state, action, reward
    inputs = self.env.sense(self)
    # Execute action and get reward
    reward = self.env.act(self, action)
    #print "DummyAgent.update(): t = {}, inputs = {}, action = {}, reward = {}".format(t, inputs, action, reward)  # [debug]
    #print "DummyAgent.update(): next_waypoint = {}".format(self.next_waypoint)  # [debug]
    print "LearningAgent.update(): deadline = {}, inputs = {}, action = {}, reward = {}".format(deadline, inputs, action, reward)  # [debug]
```

The above code delegates into the ‘Utilities().parse_state(self.state)’ class to parse the string state into its corresponding index. The Q-learning algorithm learns at each step by calling ‘ql.run_step(state_idx)’ on the ‘QLearner(rt,0.5,0.5)’ object. The later can be found in the q_learner.py file. It implements the above mentioned mathematical formula in the ‘generate_entry_for_qtable’ function. The reward table(rt) is defined in the ‘Utilities().get_reward_matrix()’

In order to run this experiment, ensure the agent.py code is set as follows:

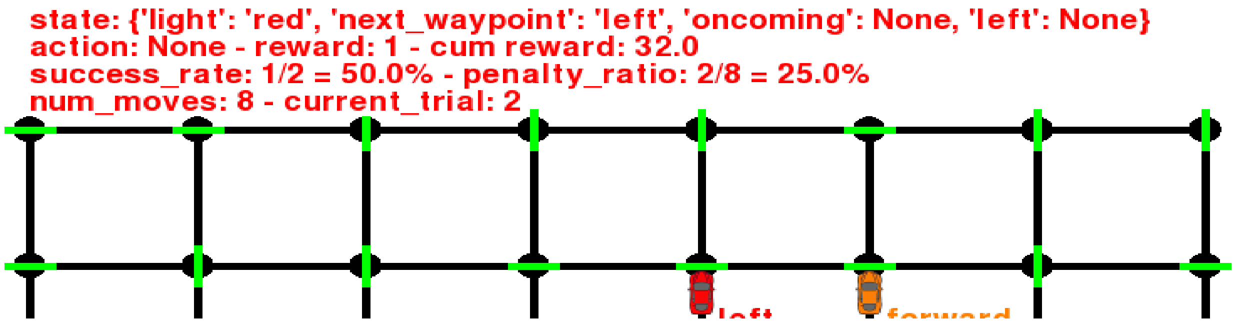
```
def run():
  (...)
 agent_type = 'q_learning'
```

Analysis

The ‘q_learning_log.txt’ log file was produced by running the agent.py file from a windows command line and routing the ‘print’ messages to a text file as follows:

>python SmartCab/agent.py >> " q_learning_log.txt".

The code base was modified to show basic statistics, as shown in figure 1 below.



```
state: {'light': 'red', 'next_waypoint': 'left', 'oncoming': None, 'left': None}
action: None - reward: 1 - cum reward: 32.0
success_rate: 1/2 = 50.0% - penalty_ratio: 2/8 = 25.0%
num_moves: 8 - current_trial: 2
```

Figure 1 – Display of basic statistics

Reward: the reward score for the current move
Cum Reward: the cumulative reward for trial run (whether destination is reached or not)
Success_Rate: the ratio of the trial run over competed run (i.e. when the destination is reached)
Penalty_ratio: the ratio of the number of penalties incurred for a given trial run over the total number of moves for the trial run
Num_moves: the number of moves for a given trial run
Current_trial: the number of trial run

The below table shows some basic statistics for a run set to 10 trials. There are a few points to note:

- ✓ The table shows a success rate of 80% for 10 trials, with a reward mean of 20.85. These numbers will change when the user starts a run from fresh. The success rate might be higher or lower.
- ✓ This experiment is run with a timer. It was notice that when the timer was set to ‘false’, some of the failed trial would succeed, but they required many more moves. Some of them would get stuck into an infinite loop, and the experiment had to be stopped by the user.
- ✓ The smartcab is following the traffic rules more carefully now. It reaches the destination faster and takes the shortest path as opposed to the initial configuration where the agent is just acting randomly. The previous behaviours, based on an action random selection, were more ‘hit and miss’.

Trial Id	Deadline	Learning Rate	Discount Factor	Cum. Reward	Penalty Ratio	# Moves	Success Trial
1	True	0.5	0.5	17.00	28.57%	14	True
2	True	0.5	0.5	13.00	0.00%	8	True
3	True	0.5	0.5	19. 00	52.38%	21	True
4	True	0.5	0.5	24.50	42.31%	26	True
5	True	0.5	0.5	22.00	28.5%	21	False
6	True	0.5	0.5	38.00	22.22%	36	False
7	True	0.5	0.5	12.50	33.33%	9	True
8	True	0.5	0.5	34.50	3.85%	26	True
9	True	0.5	0.5	31.50	23.08%	26	False
10	True	0.5	0.5	15.50	9.09%	11	True

Pass Rate: 80% - Reward Mean: 20.85 - Penalty Ratio Mean: 24.33%

Enhance the driving agent

The aim of this section is to performance tune the Q-Learning algorithm. A success trial is now defined as below

i) The agent consistently reaches destination within the allotted time, with a positive cumulated reward, within 100 trials.

ii) The agent should reach the destination at least 7 out of the last 10 trials.

One of the caveats to Q-Learning is that the agent has to explore enough of the grid world to eventually reduce the learning rate over time while using what the agent already knows to get a high reward. Eventually, the agent needs to stop exploring and start exploiting the values and information in the Q-Table. Here, we are the epsilon-greedy[1] algorithm, where epsilon is kept constant over time and trials. This is an improvement on the solution proposed previously.T he next action selection is now based on the epsilon-greedy algorithm, instead of being a pure random selection. The code can be found in the q_learner.py file. A snippet is provided below:

```
def run_step(self, current_state,use_epsilon_greedy=False,currrent_action_idx =-1, rdm_next_state_idx=-1):
(…)
    if (use_epsilon_greedy == False):
      #randomly select one possible action for the selected state
      if (currrent_action_idx == -1):
        currrent_action_idx = self.get_action_index()
    else:
      if np.random.random() < self.epsilon:
        #select a random action with epsilon probability
        currrent_action_idx = self.get_action_index()
      else:
        #select an action with 1-epsilon probability that gives maximum reward in given state
        currrent_action_idx = self.rTable.get_rtable_max_reward(currrent_state_idx)
str(currrent_state_idx)
(…)
```

The below table summarises the scenario run for the smartcab given 100 trials, a timer and different learning rate/discount factor.

Deadline	Learning Rate	Discount Factor	Epsilon	Mean Run Cum. Reward	Pass Rate	# of successful last 10 trials	Success Trial	Failed Trial	Log File Name
True	0.1	0.9	0.5	21.83	81%	8	True	15,21,31,70,72,82,90,91,94	enhanced_q_learning_lr01_df09.txt
True	0.2	0.8	0.5	20.66	95%	10	True	47,57,63,77,79	enhanced_q_learning_lr02_df08.txt
True	0.3	0.7	0.5	24.36	79%	8	True	9,16,23,34,42,53,72,83,88,91,100	enhanced_q_learning_lr03_df07.txt
True	0.4	0.6	0.5	21.41	95%	10	True	16,27,29,53,69,84	enhanced_q_learning_lr04_df06.txt
True	0.5	0.5	0.5	22.91	96%	9	True	44,67,76,96	enhanced_q_learning_lr05_df05.txt
True	0.6	0.4	0.5	23.88	92%	9	True	21,22,46,67,76,80,85,100	enhanced_q_learning_lr04_df06.txt
True	0.7	0.3	0.5	24.08	93%	9	True	2,22,78,85,87,88,95	enhanced_q_learning_lr03_df07.txt
True	0.8	0.2	0.5	21.57	94%	10	True	8,12,23,37,58,77	enhanced_q_learning_lr02_df08.txt
True	0.9	0.1	0.5	22.87	93%	10	True	11,22,32,41,49,61,90	enhanced_q_learning_lr01_df09.txt

1 In short, the epsilon-greedy is a policy where a random action with epsilon probability is selected. When the action is contained within a 1-epsilon probability, then the action corresponds to the maximum action reward for a given state. Epsilon can also decay over time (this is not implemented here). We can start epsilon off relatively high (close to 1.0 to force the agent to explore instead of exploit). As epsilon decreases with respect to elapsed time (or trial number), the agent's frequency of exploiting the values and information in the Q-table increases.

In the light of the data provided in the above tabe, the optimal solution is the one highlighted in yellow. Indeed, the performance tuned version of the initial Q-Learning algorithm implementation shows an increase of 16% (96%-80%) of accuracy, with a learning rate of 0.5 and a discount factor of 0.5. The agent does not go into circle and find the most optimal route. The cumulative mean of rewards is not 200 (max reward = 2 * 100 iterations) . This means the smartcab takes some incorrect traffic laws action during its travel.

References

Wikipedia, Q-learning, https://en.wikipedia.org/wiki/Q-learning, [Accessed 04 July 2016]
Burlap, http://burlap.cs.brown.edu/tutorials/cpl/p3.html#qltest, [Accessed 04 July 2016]