Implement a basic driving agent

The agent chooses actions randomly with the same probability for each. The allowed actions are ‘turn right’, ‘turn left’, ‘go forward’ and ‘don’t move’. Random behaviour is observed on the plot as agent moves around without any strategy. In most trips agent does not reach target location, but sometimes does.

Identify and update state

The state is chosen to consist of five components with deadline not included. The environment returns deadline which tracks the number of steps the agent makes before it reaches the destination. Including this component into state will drastically increase the number of states as for each five components state there will be 10-30 more states with different deadlines. The huge number of states makes Q-learning overcomplicated.

State description:

state = (inputs['light'], inputs['oncoming'], inputs['left'], inputs['right'], self.next_waypoint)

- **inputs[‘light’]** captures traffic light signal, there are two possible values: ‘red’ and ‘green’.
- **inputs[‘oncoming’]** captures an action of traffic coming from ahead, there are four possible values: ‘left’(oncoming traffic turns left), ‘right’(oncoming traffic turns right), ‘forward’(oncoming traffic proceeds forward) and ‘None’(oncoming traffic do not move).
- **inputs['left']** captures an action of traffic coming from left, there are four possible values: ‘left’(traffic from left turns left), ‘right’(traffic from left turns right), ‘forward’(traffic from left proceeds forward) and ‘None’(traffic from left do not move).
- **inputs['right']** captures an action of traffic coming from right, there are four possible values: ‘left’(traffic from right turns left), ‘right’(traffic from right turns right), ‘forward’(traffic from right proceeds forward) and ‘None’(traffic from right do not move).
- **self.next_waypoint** represents intention of the agent, there are four possible values: ‘left’(agent plans to turn left), ‘right’(agent plans to turn right), ‘forward’(agent plans to proceed forward) and ‘None’(agent plans not to move).

First four components represent an environment the agent operates in. The last component represents actions an agent would make if no training is giving to it. All five components uniquely describe the state the agent is in.

Implement Q-Learning

The key part of Implementation of Q-learning is finding an optimal policy. Optimal policy tells the agent what action to take in order to reach the goal in the best way (get maximum reward). In Q-learning procedure the optimal policy is represented by Q-table. Q-table accommodates so-called q-values based on which next action is recommended. Q-value is assigned to (state, action) pair and keep learned knowledge about the state and action taken. Thus, Q-learning is the processes of updating Q-table which in its turn governs agent's behaviour.

Updating the Q-table is driven by this line of code:

self.Q_table[(state,action)] = (1-self.alpha)*q_value+self.alpha*(reward+self.gamma*next_q_value),

where

self.Q_table[(state,action)], q_value – q-value for current (sate, action) pair

self.alpha – the learning rate, set between 0 and 1. Setting it to 0 means that Q-values are never updated, hence nothing is learned. Setting a high values such as 0.9 means that learning can occur quickly

self.gamma – discount factor, set between 0 and 1. It models the fact that future rewards are worth less than immediate reward

Parameters alpha and gamma are chosen to be 0.5, setting the moderate learning pace and significance of future rewards.

reward – reward after performing action at current state

next_q_value – q-value at the next state, the state where agent occurs after performing an action advised by optimal policy at previous state

When agent starts learning it does not know anything about each state and what action is better to take. Hence, all q-values are zeros. In the code, *best_qvalue_action* function both initialise new (state, action) pairs, finds the best q-value for current state and identifies the best action for the current state.

When best action is found, the reward for performing this action is calculated. The line of code *reward = self.env.act(self, action)* implements this calculation.

The last bit needed for updating current q-value is finding the largest possible q-value for the next state. To accomplish this task the agent needs to sense the environment again. The following lines of code perform the task:

inputs = self.env.sense(self)

next_state= (inputs['light'], inputs['oncoming'], inputs['left'], inputs['right'], self.next_waypoint)

Then find q-value:

next_q_value, next_action = self.best_qvalue_action(next_state)

Now all parts of equation are available and Q-table can be updated accordingly.

Implementation of Q-learning changed the behaviour of agent significantly. Instead of performing chaotic movements, the agent moves towards the goal without making many unnecessary turns.

Results of agent's performance are discussed in Result discussion section.

Enhance the driving agent

Epsilon-greedy approach is implemented so that to enhance agent's performance. This approach allows the agent making actions at random rather than relying on learned experience. The methods ensures that if enough trials are done each actions will be tried an infinite number of times, thus ensuring an optimal action to be discovered. The action selected at random with a small probability *epsilon*, hence the name of the method.

When training evolves the agent makes more educated moves and hence random actions are less needed. An exponential function epsilon = exp(- number_of_trials) might be a good option to model smoothly decaying number of random actions.

The following changes have been added to basic implementation of Q-Learning:

The range of epsilon values is decreasing from 8.2% to 0.25% over 100 trials:

 # TODO: Initialize any additional variables here

self.epsilon = [np.exp(-val) for val in np.linspace(2.5,6,100)]

self.rand_count = 0

Skip to the next epsilon after each trial:

 # TODO: Prepare for a new trip; reset any variables here, if required

self.rand_count +=1

Choose action at random, happens with epsilon probability

if random.uniform(0,1) < self.epsilon[self.rand_count-1]:

action = random.choice(self.actions)

Results of agent's performance are discussed in Result discussion section.

Results discussion

Performance of last 10 trials only is considered for each simulation as a measure of how well the agent has learned driving as 90 trials are considered as a learning process.

Each simulation is done with constant gamma = 0.5 and alpha values to be 0.25, 0.5 and 0.75. Q-Learning with and without enhancement are compared in the table below.

Metrics explained:

- steps – an average number of steps the agent needs to reach the destination
- success – an average number of successful trips
- reward – an average total utility

	Q-Learning without enhancement			Q-Learning without enhancement		
(alpha, gamma)	steps	success	reward	steps	success	reward
(0.25, 0.5)	14.2	10	22.6	12.7	10	20.5
(0.50, 0.5)	16.2	10	23.4	13.9	10	23.2
(0.75, 0.5)	13.7	10	22.8	11	10	21.95

Although both types of learning result in agent always reaching the destination within last 10 trials, the number of steps and total rewards differ. On average, Q-learning with enhancement results in agent making less steps to reach the destination. It means that algorithm enhancement provides with better education as the agent goes the shortest path. The pair of parameters with alpha = 0.75 and gamma = 0.5 ensures the best performance with steps = 11.

Another characteristic of agent performance is how well it obeys the traffic rules. Let's introduce new metric RS = reward/step and compare quality of trainings based on it. The metric tells how much reward per step the agent receives. Small values indicate the agent disobey the traffic rules more often. For the models with best parameters found earlier, RS=22.8/13.7 = 1.66 and RS=21.95/11 = 1.99 for Q-learning without and with enhancement. Thus, the agent educated by the latter training receives less penalties than the one educated by former training.

As a final result, the best training is the one with decaying number of random actions and alpha = 0.75, gamma = 0.5.