Here are the types of the inputs and outputs of the smart cap from the Udacity Project Description.

Inputs:

1. The next waypoint
2. Traffic light for its direction of movement
3. Other cars' location and their next movement directions when they are near the agent
4. deadline

Outputs:

1. Agent's action – stay put, move one block forward, one block left or one block right

Implement a basic driving agent

`<agent.py>`

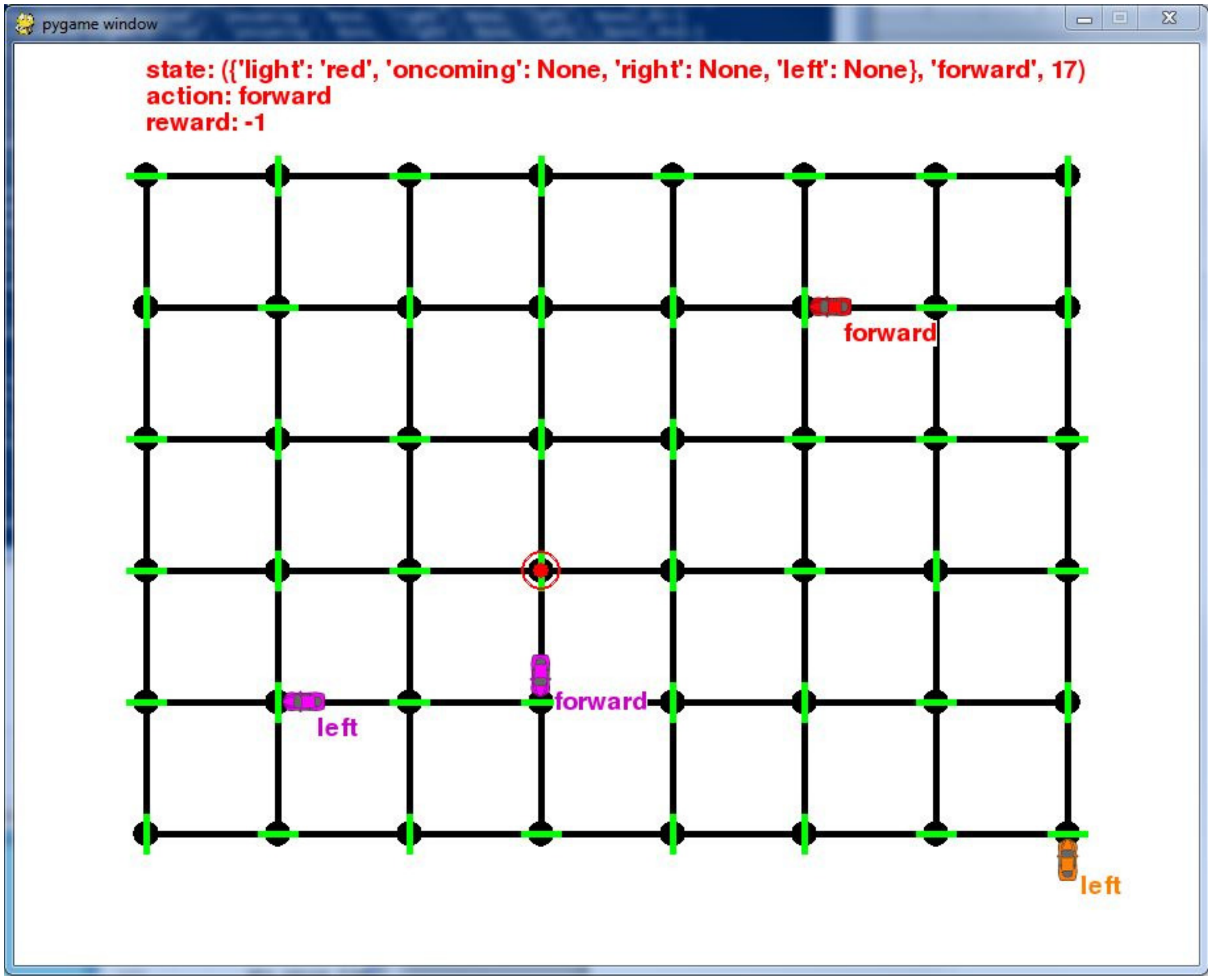
```
# Gather inputs
self.next_waypoint = self.planner.next_waypoint()  # from route planner,
also displayed by simulator
inputs = self.env.sense(self)
deadline = self.env.get_deadline(self)

# TODO: Update state

# TODO: Select action according to your policy
action = random.choice([None, 'forward','left','right'])
```

A basic driving agent with random action regardless of its surrounding conditions was implemented. The reward system was verified based on the next waypoints, traffic lights, neighboring cars and the agent's (random) action. As shown the Figure 1, the agent received the reward of -1 when it moved forward when the light was red.

The agent very rarely does arrive at the destination, but since the movements are completely random, it should not be called 'smart' cab. It did not take into account any of the Inputs; the traffic light, the other cars, the next waypoint information or the time steps (deadline).



state: ({'light': 'red', 'oncoming': None, 'right': None, 'left': None}, 'forward', 17)
action: forward
reward: -1

Figure 1 Basic driving agent

Identify and update state

The state of the agent was initiated and updated with the pre-specified variables in the script (self.next_waypoint, inputs, deadline). These inputs were chosen for defining state because;

- next_waypoint is used to calculate the reward en route to the final destination.
- inputs consists of traffic information that will be used to train the agent to find the optimal action. Only traffic light among the input items was selected as other car's location and heading have little impact on the agent's next move. (right-of-way violation has yet to be implemented in the code per Udacity coaches)
- deadline is also important to find the optimal route when available time is limited.

```
# Gather inputs
self.next_waypoint = self.planner.next_waypoint()  # from route planner,
also displayed by simulator
inputs = self.env.sense(self)
deadline = self.env.get_deadline(self)

# TODO: Update state
self.state = (inputs['light'], self.next_waypoint, deadline)

# TODO: Select action according to your policy
action = random.choice([None, 'forward','left','right'])
```

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.

Q-learning algorithm was implemented successfully, but it finds local minimum frequently based on its initial stage of learning processes. As I posted in the Udacity forum post the Q-learning tends be subject to its initial random choice and sticks with ‘action=NONE’ as NONE gives the reward of 1, whereas other actions gives less value of 0.5. Reward of two points would be given when the light is green and the random action happens to be same as ‘next_waypoint,’ but the probability of that instance occurring is lower than staying with ‘NONE.’ Therefore, the agent would continuously choose its initial selection for a state and it will be augmented even further (value in q-table gets even larger) as the trial goes forward.

```
Q_new = (1-alpha)*Q_old + alpha*[R + gamma*max(Q')]

Where
alpha: learning rate
R: immediate reward
Q': Utility (future reward)
Gamma: discount factor
```

This is because the agent only conducts ‘exploitation,’ and not ‘exploration’ except at the very first time. Table 1 below shows the success rate from the basic Q-learning of 10 simulations, each with 100 trials. The success rate is staggering. It is hard to define which combinations of alpha and gamma values would achieve the most successful learning.

Table 1 Success rate (in percent) from Basic Q-learning

	Alpha = 0.8	Alpha = 0.8	Alpha = 0.8
simulations	Gamma = 0.0	Gamma = 1.0	Gamma = 0.5
1	8	1	59
2	28	29	0
3	1	2	0
4	1	0	52
5	72	0	0
6	1	0	26
7	18	0	44
8	39	1	12
9	0	1	7
10	99	36	0

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.

In order to improve the success rate of Q-learning algorithm, a simulated annealing scheme was introduced with parameter epsilon, which starts with 1 and decays over time as the trial numbers increases. The epsilon defines the probability of random choice of actions that the agent takes in the earlier stage of simulation. During the course of trials, a random number between 0 and 1 is generated and compared with the epsilon. If the epsilon is larger than the random number, than the agent takes a random action regardless of the Q-table values. It promotes 'exploration' and prevents the agent from falling in local minima as it did in the basic Q-learning algorithm.

```
Q_new = (1-alpha)*Q_old + alpha*[R + gamma*max(Q')]

Where
alpha: learning rate
gamma: discount factor
R: immediate reward
Q': Utility (future reward)

** epsilon <- epsilon / (epsilon + trial_number)
```

Table 2 shows simulation (100 trials each) results for various combinations of alpha and gamma parameters. There could be many ways to find the optimal policy, but I chose to maintain the alpha and gamma values fixed and apply the simulated annealing to encourage more exploration in the earlier stage of each simulation.

The results shows that (alpha = 0.8 and gamma = 0.2) have generated the best success rate.

It's worth noting that (alpha = 0.4 and gamma = 0.2) would generate comparably good success rate, but with same alpha values, the success rates are much more dependent on the gamma value.

NOTE :
With given variables and the structures of python scripts associated with agent.py script, the behaviors of dummy cars don't seem to affect the reward points given to the agent. The agent only receives rewards based on the fact whether it follows the next waypoint or it stays put based on the traffic lights. The optimal sets of parameters will likely be changed once all other pieces of traffic information are to be used to define the state of the agent.

Table 2 Success rate (in percent) from Enhanced Q-learning with simulated annealing

simulations	Alpha = 0.8 Gamma = 1.0	Alpha = 0.8 Gamma = 0.5	Alpha = 0.8 Gamma = 0.2	Alpha = 0.4 Gamma = 1.0	Alpha = 0.4 Gamma = 0.5	Alpha = 0.4 Gamma = 0.2
1	7	89	100	1	95	95
2	2	97	97	15	86	83
3	3	85	96	47	51	95
4	5	99	94	69	67	93
5	0	87	95	93	86	84
6	2	88	99	38	95	97
7	9	97	97	17	91	90
8	2	88	97	34	92	68
9	9	89	63	28	73	99
10	6	89	88	1	91	67