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).

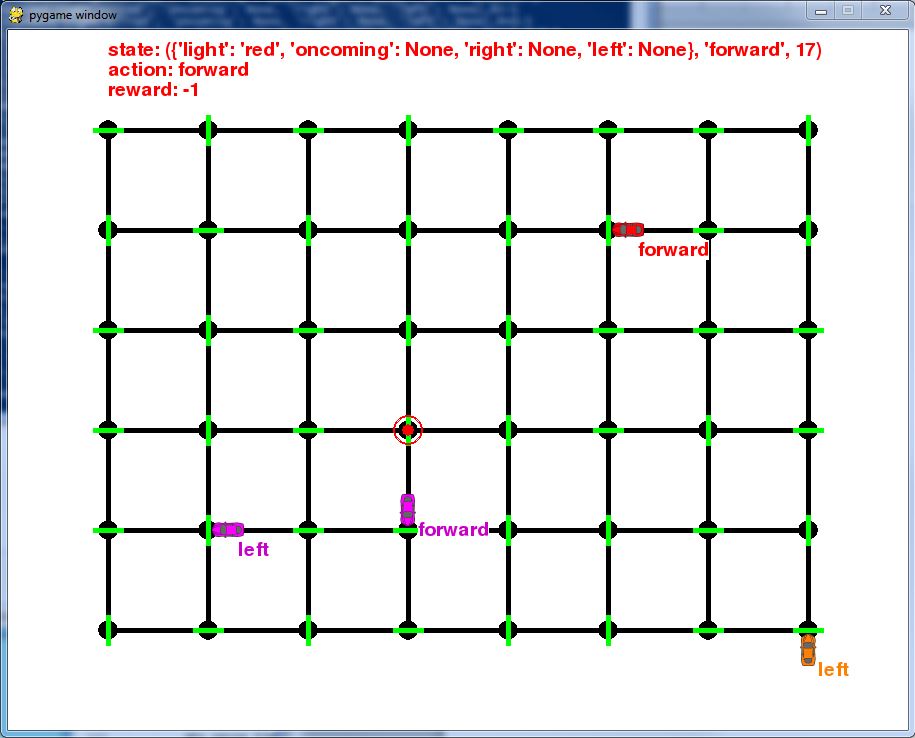


Figure 1 Basic driving agent

# 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 mentioned in a [Udacity forum post](https://discussions.udacity.com/t/q-learning-is-this-the-right-algorithm/170487?u=sbaek) the Q-learning tends be subject to its initial random choice and to stick with ‘action=NONE’ as NONE gives the reward of one, whereas other actions only gives 0.5. Reward of two points would be given when the light is green and a randomly chosen action happens to be same as ‘next\_waypoint,’ but the probability of that instance happening is low. Therefore, the agent would continuously choose its initial selection for a state and it will be augmented even further as the trial goes forward.

This is because the agent only conducts ‘exploitation,’ and not ‘exploration’ except at the very first time. Figure XX shows the success rate from the basic Q-learning with 100 trials.

Q\_new = (1-alpha)\*Q\_old + alpha[R + max(Q’)]

Where

alpha: learning rate

R: immediate reward

Q’: Utility (future reward)