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 in `agent.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?

The implemented agent chooses randomly to perform a move/action not considering waypoint location, intersection state or traffic. Although the move/action is 100% random, the agent eventually gets to destination but it's very rare that it arrives on deadline. In the first 5 trial the agent arrived to destination in 200+ time steps but on the 6th it arrived on time (note that this results are totally random).

Another thing that caught my eye was that this random agent tends to get stuck in intersections because it is not considering the environment to take an action. So for example it may choose to move right on North-South traffic or just choose the 'None' action when there could be a reasonable action to do. In this sense, I made the Agent choose a random action based the possible action it can take given the environment (considering both the traffic lights and the oncoming traffic). For example, it can not choose to go forward on red lights. Although this new agent gets to destination at a much lower number of time steps, it still doesn't get to destination on deadline always and is quite random in the number of time steps it takes.

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.

I picked the state based on all the environmental inputs that have and impact in the reward gained by performing a certain action. In this case the next waypoint, intersection lights and traffic determine what reward is gained by taking an action. The general explanation is that the agent gets a big reward for following the waypoint but incurs in a penalty if it doesn't follow the traffic light directions or if it crashes with another car. So in this case, my state is determined by the following statement:

```
self.state =
tuple([('waypoint',self.next_waypoint),('light',inputs['light']),('oncoming',inpu
ts['oncoming']),('right',inputs['right']),('left',inputs['left'])])
```

I considered not including the 'oncoming', 'right' and 'left' inputs since they were almost always set to 'None'. This is because of the low amount of cars as traffic, but I wanted to implement an agent that could learn a feasible policy independent of the number of cars in traffic so I kept this inputs. After solving the problem with 3 cars in traffic I decided to test the results using 10 cars in traffic and results showed a feasible policy after 100 trials.

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?

I used the Q-Learning algorithm explain in the Reinforcement Learning lessons using the following formula:

$$\widehat{Q}(s,a) \overset{\alpha}{\leftarrow} r + \gamma\, Max_{a'}\, \widehat{Q}(s',a')$$

where:

$$V \overset{\alpha}{\leftarrow} X$$

$$V \leftarrow (1-\propto) \cdot V + \propto \cdot X$$

So:

$$\widehat{Q}(s,a) \leftarrow (1-\alpha) \cdot \widehat{Q}(s,a) + \alpha \cdot \left(r + \gamma\, Max_{a'}\, \widehat{Q}(s',a')\right)$$

I had a hard time implementing the max Q value for the next action, so I decided to implement this policy using the previous values. So I ended up implementing the following formula:

$$\widehat{Q}(s_{t-1},a_{t-1}) \leftarrow (1-\alpha) \cdot \widehat{Q}(s_{t-1},a_{t-1}) + \alpha \cdot \left(r + \gamma\, Max_{a'}\, \widehat{Q}(s_t,a_t)\right)$$

I set the initial value for all Q values at 0.

After implementing this I decided to run the agent with it only choosing actions based on the policy. This lead to an scenario where the agent always performed the same action, 'None'. This in fact makes sense because the agent didn't learn anything from random actions so it choose to make the same action always.

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?

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?

After watching the agent perform the same action always I implemented the Exploration – Exploitation dilemma with a little twist. I wanted to implement an agent that performed random action to 'Explore' (or learn) and to choose the best action to 'Exploit' this learning. I realized that in the first trials there was very little to exploit so it may not be worth it to perform actions based on the policy. Since we wish to have a feasible policy at 100 trials, I implemented an agent who starts doing only random actions and with each new trial the probability of choosing an action based on the policy increases. At 100 trials the agent should only choose actions based on the policy.

In the first tests with this implementation I used a random value for the learning rate (α=0.9) and discount factor (γ=0.8). This lead to a strange behavior where the agent some times decided to take the same action most of the times (for example always take right) which lead to a loop that made the agent get to destination way past the deadline.

My first impression was that I had implemented the Q-Learning formula the wrong way so I spent quite a lot of time searching for the problem. After some time I realized the parameters had a huge impact on the results of the agent. After a couple if test with different parameters I got to the following values: α=0.8 and γ=0.3 that got me to a feasible policy after 100 trials.

After 1000 trials (past the 100 trial mark) this agent gets to destination on time with a probability of 98.7%, incurring in 0 penalties (it never chooses a action that would get a negative reward) and getting to destination after 13 time steps on average.