P4: Train a smart cab to drive

Implement a basic driving agent

A 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)
- Current deadline value (time steps remaining)

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 agent eventually makes it to the target location quite a few times when the actions are chosen at random. However, it also misses it's target location and hits a hard stop a few times. It is purely chance that the agent reaches it's destination and often does so way past the deadline. It's equally likely that it doesn't reach the deadline either.

Further, the following was noticed here with respect to the agent, environment and rewards:

1. The agent had no regard for traffic rules, lights, right-of-way, waypoint and sense of direction.

2. It seems to be penalized for the following actions as seen from the results:

   a. `-0.5` when, the action taken doesn't lead to the next waypoint.
   Ex: `{'tripId': 1, 'right': None, 'light': 'green', 'oncoming': None, 'deadline': 23, 'waypoint': 'forward', 'action': 'right', 'reward': -0.5, 'left': None}`

   OR

   when, the light is red and an action to turn right is executed provided there is no traffic at the waypoint.
   Ex: `{'right': None, 'tripId': 2, 'light': 'red', 'oncoming': None, 'deadline': 12, 'waypoint': 'forward', 'action': 'right', 'reward': -0.5, 'left': None}`

   b. `-1` When, the action taken doesn't obey the traffic signal or when it takes an action against the U.S. right-of-way.
   Ex: `{'tripId': 100, 'right': None, 'light': 'red', 'oncoming': None, 'deadline': -64, 'waypoint': 'right', 'action': 'forward', 'reward': -1.0, 'left': None}`

3. No reward is given when no action is taken by the agent while it could have instead.
   Ex: `{'action': nan, 'deadline': -50, 'left': None, 'light': 'green', 'oncoming': None, 'reward': 0.0, 'right': None, 'tripId': 100, 'waypoint': 'right'}`

4. A positive reward is given when the agent executes actions that takes it to the `next_waypoint` without disrupting traffic.
   Ex: `{'action': 'right', 'deadline': -83, 'left': None, 'light': 'green', 'oncoming': None, 'reward': 2.0, 'right': None, 'tripId': 7, 'waypoint': 'right'}`

However, this is arguably not the desired behavior expected from a smartcab. It must be able to plan and move in an optimal path from the source to the destination. Choosing a random action suggests that there is no learning and hence nothing smart about the cab.

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

A state is representative of the condition that the agent is in with respect to the environment at any particular given point in time. It seems appropriate to represent it as a combination of sensory inputs and the responses derived from the environment. Hence, we may choose to represent the state with the following inputs:

1. `Light: ['Red','Green']`
2. `Waypoint: ['None','Left','Right','Forward']`
3. `Left: ['None','Left','Right','Forward']`
4. `Oncoming: ['None','Left','Right','Forward']`
5. `Right: ['None','Left','Right','Forward']`
6. `deadline: Number of points in the Grid`

`Light`, `action` and `oncoming` captures information about traffic rules in the environment and the `waypoint`guides the agent to the next waypoint that will lead it to the destination.

Adding the `deadline` increases the state space and we wish to keep the state space as small as possible. Further, it's felt that since the `waypoint` is already guiding the agent toward the deadline already `deadline` seems redundant and unnecessary.

The final state space is now composed of `light`, `waypoint`, `left`, `oncoming` and `right` resulting in `2*4*4*4*4 = 512` possible states.

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 agentâ€™s behavior?

The agent reaches the destination more often and is significantly faster after Q-Learning is implemented. It's however, important to note that the agent begins to learn only as the number of trials increase. In the beginning *(first 10 trips)* although the agent reaches the destination on time it takes longer and collects more negative rewards to reach the destination while towards the end *(last 10 trips)* it begins to accumulate more positive rewards and reaches the destination much faster.

This shows that the agent learns from its mistakes progressively and learns the behavior that the environment rewards as the trips progress. Further, it was noticed that the agent explores the environment in the beginning and begins to exploit as the trip count increases.

At this point the `learning rate` (α) and the `discount rate` (γ) have been arbitarily chosen to be `0.5` and `0.5` respectively. Further, no greedy-exploration of the environment is implemented yet.

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?

The action selection method is already quite optimal and the vannila Q-learning implementation proved to be quite effective. Toward the end of 100 trips it can be seen that the agent seldom collects negative feedback.

However, we could tinker with the `learning rate` (α) and the `discount rate` (γ) to optimize this further. We can also implement `greedy limit infinite exploration` so that the agent explores the environment to ensure that it reaches the destination in the minimum possible time.

Further, the state space could be reduced. The car in the `right` although present adds no value with the current reward system in place. Further, in the U.S. right-of-way rules we are concerned with the car `oncoming` and the car on the `left`. Hence, it makes sense to ignore `right` while reducing the state space.

Since, we wish to change the hyperparameters of learning i.e. α , γ and ε we must have a metric or measure to judge how well the agent is doing and must be able to compare the results using this metric/measure. While there could be multiple measures/metrics we could use, it was decided that we shall go with `MeanNetReward/Action`. Mathematically, this is: $$\frac{NetRewards}{Actionstaken}$$ Here,
`NetReward = Sum of all rewards in 100 trials`
`Actionstaken = Total number of actions taken in 100 trials`

Further, it also makes sense to track the number of times the agent reaches the destination during the 100 trials

These have been recorded in the tables below for all the three attempts:

1. Random Actions picked
2. Vanilla implementation of Q-Learning.
3. Enhanced Q-Learning.

Random Actions

Trial	MeanReward/Action	Reached	Not Reached
1	0.03326023392	23	77
2	0.01367488444	19	81
3	0.05433130699	24	76
4	0.01201550388	22	78
5	0.01397168405	23	77
6	0.08922226609	30	70
7	0.008302583026	19	81
8	-0.004234297812	19	81

9	0.03193960511	22	78
10	0.03458049887	23	77
Mean	0.02870642686	22.4	77.6

Vanilla Q-Learning (learning_rate = 0.5, discount_factor=0.5)

Trial	MeanReward/Action	Reached	Not Reached
1	1.930419269	100	0
2	1.741129032	97	3
3	1.656502572	97	3
4	1.799159664	98	2
5	1.740930599	100	0
6	1.664661654	96	4
7	1.624905944	96	4
8	1.603606789	98	2
9	1.701638066	96	4
10	1.707354056	96	4
Mean	1.717030764	97.4	2.6

Enhanced Q-Learning (learning_rate = 1, discount_factor=0.1)

Trial	MeanReward/Action	Reached	Not Reached
1	1.789624183	99	1
2	2.005597015	99	1
3	1.837073982	98	2
4	1.900690846	99	1
5	1.926165803	100	0
6	1.802474062	99	1
7	1.848229342	100	0
8	1.893435635	100	0
9	1.753949447	99	1
10	1.938958707	100	0
Mean	1.869619902	99.2	0.8

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?

The enhanced agent is quite close to finding an optimal policy. It reaches the destination in very little time when compared to the vanilla Q-learning implementation. This is visible from the fact that on average of 10 sets of 100 trials the agent with an α`= 1` and γ`= 0.1` has a better MeanRewards per Action performed and reaches the target on a average 99.2 times.

Furthermore, it was quite counter-intuitive to notice that implementing the Greedy limit - Infinite exploration actually reduced the performance of the Agent although ε was `0.1` and hence this was reverted back