Machine Learning Engineer Nanodegree

Project 4: Train a Smart Cab to Drive

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.

Q: In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

A: By taking random actions in each step, the basic driving agent will basically perform a random walk on the grid until it finally reaches the destination in each trial. However, as expected, the agent did not reach the destination in a reasonable time or did not obey traffic rules most of the time.

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.

Q: Justify why you picked these set of states, and how they model the agent and its environment.

A: In my model, I have included the following parameters as part of the state:

1. self.next_waypoint: the next waypoint provided by the route planner, as an important signal of what the next optimal action should be. It helps guide the agent towards the destination
2. inputs['light']: the state of the traffic light, as an important signal for learning the rules of the road

3. inputs['left']: the state of the car on the left, as an important signal for learning right of way at the junctions
4. inputs['oncoming']: the state of the oncoming car, as an important signal for learning right of way at the junctions
5. self.distance: manhattan distance to the destination, to gauge how war the agent is from the destination

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.

Q: What changes do you notice in the agent's behavior?

A: After implementing the Q learning and taking optimal actions in each step using the resulting Q table, it is observed that:

1. the agent takes proper actions at each traffic light and tries to obey the traffic rules to the extent it is rewarded differently based on legal and illegal actions
2. the agent will learn to reach the destination to get the big reward

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.

Q: 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?

A: I made some improvements to the Q learning process by making the following adjustments:

1. **dynamic epsilon**: throughout the 100 learning trials, the epsilon will be linearly decreasing from 1.0 to 0, which is used for the balance between exploration and exploitation in Q learning. This will lead to the agent learning as much as it can by relying on exploration during the early trials and then exploiting the learnings by using the optimal action from the Q table in the later trials.
2. **value iteration step**: to incorporate the optimal future Q value into the Q learning formula, the Q learning step is set up in such a way that Q value update will happen on the (s_old, a_old) pairs so that the optimal future Q value can be easily calculated based on the current state. It is worth noting that because of the way this is set up in code, there needs to be one

last Q learning iteration for the reward of the last step, which is implemented under the ‘reset’ method.

3. **Learning rate and discount factor**: I played around with these parameters to find a good fit and came up with a learning rate of 0.4 and a discount factor of 0.7.

Q: 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?

A: By making the abovementioned adjustments, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive. However, it does not necessarily reach the destination in the minimum possible time without any penalties. I believe in order to achieve that, changing of the reward structure should be allowed in this project mainly because if we are trying to optimize total reward (as the goal of Q learning), the agent wouldn’t necessarily want to reach the destination in the minimum amount of time if, as an alternative, it can just collect rewards by driving around on the streets and still reach the destination in the allotted time.