Udacity Machine Learning Project 4

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 agent makes move at random. That is, the agent at each iteration chooses one of "None", "forward", "left", or "right" uniformly at random. The agent always seems to make it to the destination before reaching the maximum number of iterations, 100, however it often has reached a negative deadline before it gets there. Additionally, the agent accrues lots of negative rewards along its way.

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 chose to use the light (red/green), traffic from each direction (oncoming, right, left), as well as the waypoint direction as my state tuple – (light, oncoming, right, left, waypoint). The light and the waypoint are very important in that they directly affect the reward for every action at every state. Traffic is not as important because there is a relatively low chance of there being other cars at any given stoplight. For this reason, it is likely also reasonable to not take into account the improbable presence of other cars. I chose to include the traffic into the state because, although it increases dimensionality, it will lead to a higher success rate after more learning. I felt that the deadline is not very useful to us and therefore did not include it. Although the deadline makes this a finite horizons problem, we do not know the cars position relative to the destination. If we knew the relative distance and our model did NOT give us traffic, there might be some cases where we would want to simply go directly to the waypoint without considering the risks of disobeying traffic laws.

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?

> The agent starts out by making lots of actions resulting in negative rewards. Over time, it makes less of these mistakes in general. On occasion, it will still make mistakes. Sometimes the agent will also get stuck in sort of circular patterns where it only turns right. The agent is certainly reaching the destination more than with the simple random move algorithm outlined earlier.

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?

> I started out having gamma = 0.8 with no randomization of action choices in the case that a max Q value existed. This performed reasonably well but sometimes got stuck in local maxima and consistently made suboptimal decisions. I first implemented an epsilon greedy exploration strategy with epsilon = .1. This seemed to increase performance by a good amount by avoiding the aforementioned local maxima. The other change I made was lowering gamma to .5. By lessening the value of future rewards, I was able to improve the algorithms success rate by over 20%.
> My algorithm, with n=10000, achieves about a 95% success rate. With epsilon greedy, the agent should avoid suboptimal policy for the most part. However, because epsilon is not being decayed over time (there is always a .1 chance of taking a random action), there will sometimes be suboptimal moves that the agent makes. It is worthwhile to consider, however, that the agent can generally afford to make a few suboptimal moves without running out of time.