Project Report

Udacity Machine Learning Nanodegree
Thalles Santos Silva

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

Running the smartcab simulator with a policy that just picks one of the four possible positions (None, left, right, forward) at random and returns it as an action shows some interesting behavior from the agent. First off, we can see the agent moving around the grid world in a very erratic fashion. Also, we notice the rewards it gets when it performs some correct moves (2 points) at the intersections as well as the penalties it suffers when it violates some of the rules of world, in this case traffic rules (-1.0 points) and when it make a valid move (does not violate any traffic rule) but it turned out be the wrong one (it turned right when it should have gone forward) -0.5 points for this situation. However, even moving in a random way, it does, eventually, get to the destination, and when it happens, we can see a larger reward (12). In fact, for the hundred trial project setup, using this random policy, the agent accomplishes its goal with a roughly 20% accuracy.

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

Based on the problem setup, I believe the best way to choose the possible states that the agent can be is by looking at the set of inputs available and make some combinations that model both the agent and the environment with the most description but without too much complexity. According to the problem specification, the agent has an egocentric view of the grid intersection it is at being able to see if there are other agents in front of itself, or in one of the both roadways side. The agent can also sense the state of the semaphore, thus, being able to identify it as green or red along with the next waypoint that it should take in order to reach the desired destination. Based on these data, we can map these input to possible states the agent can be in, by combining these different input variables. For instance, suppose the agent is at some intersection where the semaphore is **green**, and there are no agents in any of the incoming roadways. This state is represented as:

light: green, left: None, oncoming: None, nextwaypoint: forward

As another example, suppose the agent is at an intersection and there is another agent in front of it that wants to turn right, and in this case the semaphore is **red**.

light: red, left: None, oncoming: right, nextwaypoint: forward

Note also that in both cases the agent should go forward to comply with the destination goal.

With this structure, we can model the necessary number of states that can differentiate the various situations that the agent can be based on the variety of input possibilities that exist. One thing that is

worth noting is that there is no indication in this state representation that there could be an agent coming from the road in the right. That is because according to the traffic rules stated for this project, the fact that there is a car coming from the right or not does not affect the current agent at all. In other words, regardless of the semaphore color, the addition of an agent coming from the right does not create a new situation or traffic violation that is not already defined, and that also helps the model to keep a concise Q-Learning table which in turn, benefits the agent's performance.

Another worth mentioning detail is the fact that to keep the QTable as small and concise as possible, I decided to not hard code all of the possible states at the beginning, but incrementally adding the states to the QTable as it is needed.

3. What changes do you notice in the agent's behavior?

After implementing a basic version of Q-Learning and programming the agent to use, instead of a random selected action, the best action from the Q-Learning table, some aspects regarding the behavior of the agent have changed. Firstly and most importantly, this scenario brings to the light the trade-off between exploration and exploitation. It is pretty clear that only choosing the best action from the QTable from the beginning, hurts the model in the sense that depending on what action it will choose to break ties, the result can be very different.

Because the QTable starts with 0s all over it at the beginning, for any action the agent asks the QTable for, the result will be the same because of the way the Q-Learning equation works. Since the agent has not seen most of the states at the beginning of the process, it did not learn anything about the environment and therefore, when the Q-Learning tries to pick the best option from the QTable, there is no best option yet, so one of the 4 possible actions must be chosen to break ties. In my experiments, when the None option is the one chosen in this situation, the agent simply does not move and consequently does not learn anything at all, since the None option has 0 reward. However, when the forward option is chosen, the agent surprisingly acts quite well reaching the destination in about 70% of the hundred trials.

I believe that is the core of the exploration exploitation dilemma, in order to use what it knows (exploitation) the agent must first learn the environment (exploration), and it is clear in the first case, when the action is None (for breaking ties) there is no exploration and consequently the agent is not able to learn anything about environment. On the other hand, in the case of the 'forward' action for breaking ties, the agent gets the chance to learn the environment when there is no best option from the QTable yet available, which in turn makes the agent able to fill the table with optimum values that helps in future decisions.

To be more specific about the results, an experiment was done using this basic Q-Learning implementation with no discount factor and a constant learning rate of 0.5. Over the 100 trials, the agent was able to accomplish the goal with an average of 74% accuracy. In this scenario, the forward action is chosen every time there is no best (maximum) value to pick.

4. **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?**

In order to reach the final version of this agent's implementation, some more advanced techniques such as learning rate decay, discount factor, and a new policy was implemented using techniques to better deal with the exploration and exploitation dilemma.

First off, to calculate the Q-Learning function:

$$Q(s,a) = (1.0 - \alpha)Q(s,a) + \alpha[\Upsilon + \gamma \max_{a'} Q(s')]$$

Where (**s** is the state, **a** is the action, $\boldsymbol{\alpha}$ is the learning rate, $\boldsymbol{\Upsilon}$ is the immediate reward, and s' is the new state), as described in the class's videos, we introduced the learning rate decay parameter ($\boldsymbol{\alpha}$), that basically controls how important this value the agent is learning at step t is. However, in order to reach the maximum learning, the values of $\boldsymbol{\alpha}$, that decreases over time (t), must come from a function f_t that generates values between 0 and 1 in such a way that the following rules apply:

$$\sum_{t=1}^{\infty} f_t = \infty \; and \; \sum_{t=1}^{\infty} (f_t)^2 < \infty$$

And one of the equations that turn out to follow this reasoning is the equation: $f = \frac{1}{t}$ that is used as the learning rate decay equation in this implementation.

Another improvement is the discount factor variable. According to the Wikipedia, the discount factory (**ɣ**) determines the importance of future rewards. A factor of 0 will make the agent "myopic" (or short-sighted) by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward. As you may see in the results below, various combinations of gamma and alpha were tried in order to find the most optimum parameter combination possible.

In order to make the best decisions at each time step, the random policy used by the agent was firstly changed to a policy that always picked up the best value from the QTable. However, as mentioned in the question 3, this approach showed some limitations mainly with the exploration/exploitation dilemma. Basically, in order to use what it has learned, the agent must first explore the environment, and that is the key point added to the new current policy. At each time-step, the agent will ask the QTable for the best action to take but it will take this action with probability (1-ε), where ε is a very small value that will control when the agent must simply take an action at random for the sake of exploring (learning) the environment. Also, this value of ε is set to decrease over time exponentially. This guarantees that at the beginning of the game, the agent will be allowed to explore/learn more from the environment and while its knowledge from the world grows, it will rely more and more in the values that it learned instead of taking random decisions. That added some exploration to the policy. Nevertheless, another problem

stated in question 3 was the fact that when the agent asks for the best action, the QTable might actually have more than one optimum actions to deliver, and now we have to deal with the problem of breaking these ties in the best way possible. From my experiments, I found the best I could do would be simply to add more exploration (take some random action), still, in some cases, we might do not know which is the best action to take but we might possibly know that some specific action(s) should not be taken at all. Based on the reasoning, when the QTable returns more than one best action, the agent will select randomly, only among these possible actions, the best one to take excluding the None action since this action does not add valuable information to the agent.

Finally, in order to access how good this agent is doing in finding the best policy in the time box defined to this project, a series of simulations were performed with a variety of parameter combinations for the learning rate, discount factor and epsilon. The ranges of values for each parameter are:

```
discount_factors = [0.6, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0]
starting_learning_rates = [0.6, 0.7, 0.8, 0.9, 1.0]
epsilon_greedy_policy = [0.1, 0.2]
```

Each run followed the rules stated in the project – 100 trials for each parameter combination with the enforce deadline option setup to true. The best set of parameters found was:

```
Learning rate: 0.9
Discount factor: 0.75
Greedy Policy: 0.1
Percentage completed:  0.89
```

Of count, because it is a stochastic algorithm, the results might change. It is worth noting that the percentage completed is computed as: $\frac{\#\ of\ completed\ travels\ with\ positive\ net\ reward}{\#\ of\ trials}$.

In addition, knowing that in order to learn the optimal policy, the agent is supposed to pass for each state an infinite number of times, this agent learns, based on the policy, in just a few steps a strategy that makes him reach the goal 89 times out of 100, which shows that the agent is able to consistently reach the destination within the defined number of steps.