Machine Learning Engineer Nanodegree

Reinforcement Learning

Project 4: Train Smart Cab to Drive

Welcome to the fourth project of the Machine Learning Engineer Nanodegree! In this notebook, some template code has already been provided for you, and it will be your job to implement the additional functionality necessary to successfully complete this project. Sections that begin with **'Implementation'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section and the specifics of the implementation are marked in the code block with a `'TODO'` statement. Please be sure to read the instructions carefully!

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. In addition, Markdown cells can be edited by typically double-clicking the cell to enter edit mode.

Project Overview

In this project you will apply **reinforcement learning techniques** for a **self-driving agent** in a simplified world to aid it in effectively reaching its destinations in the allotted time. You will first investigate the environment the agent operates in by constructing a very basic driving implementation. Once your agent is successful at operating within the environment, you will then identify each possible state the agent can be in when considering such things as traffic lights and oncoming traffic at each intersection. With states identified, you will then implement a Q-Learning algorithm for the self-driving agent to guide the agent towards its destination within the allotted time. Finally, you will improve upon the Q-Learning algorithm to find the best configuration of learning and exploration factors to ensure the self-driving agent is reaching its destinations with consistently positive results.

Description

In the not-so-distant future, taxicab companies across the United States no longer employ human drivers to operate their fleet of vehicles. Instead, the taxicabs are operated by self-driving agents — known as smartcabs — to transport people from one location to another within the cities those

companies operate. In major metropolitan areas, such as **Chicago, New York City, and San Francisco**, an increasing number of people have come to rely on smartcabs to get to where they need to go as safely and efficiently as possible. Although smartcabs have become the transport of choice, concerns have arose that a self-driving agent might not be as safe or efficient as human drivers, particularly when considering city traffic lights and other vehicles. To alleviate these concerns, your task as an employee for a national taxicab company is to use reinforcement learning techniques to construct a demonstration of a smartcab operating in real-time to prove that both safety and efficiency can be achieved.

Starting the Project

In /smartcab/ are the following four files:

- **Modify :**
  - agent.py: This is the main Python file where you will be performing your work on the project.
- **Do not modify : **
  - environment.py: This Python file will create the smartcab environment.
  - planner.py: This Python file creates a high-level planner for the agent to follow towards a set goal.
  - simulation.py: This Python file creates the simulation and graphical user interface.

Environment

The smartcab operates in an ideal, grid-like city (similar to New York City), with roads going in the North-South and East-West directions. Other vehicles will certainly be present on the road, but there will be no pedestrians to be concerned with. At each intersection is a traffic light that either allows traffic in the North-South direction or the East-West direction. U.S. Right-of-Way rules apply:

On a green light, a left turn is permitted if there is no oncoming traffic making a right turn or coming straight through the intersection. On a red light, a right turn is permitted if no oncoming traffic is approaching from your left through the intersection. To understand how to correctly yield to oncoming traffic when turning left, you may refer to this official drivers' education video, or this passionate exposition.

Inputs and Outputs

Assume that the smartcab is assigned a route plan based on the passengers' starting location and destination. The route is split at each intersection into waypoints, and you may assume that the smartcab, at any instant, is at some intersection in the world. Therefore, the next waypoint to the destination, assuming the destination has not already been reached, is one intersection away in one direction (North, South, East, or West). The smartcab has only an egocentric view of the intersection it is at: It can determine the state of the traffic light for its direction of movement, and whether there is a vehicle at the intersection for each of the oncoming directions. For each action, the smartcab may either idle at the intersection, or drive to the next intersection to the left, right, or

ahead of it. Finally, each trip has a time to reach the destination which decreases for each action taken (the passengers want to get there quickly). If the allotted time becomes zero before reaching the destination, the trip has failed.

Rewards and Goal

The smartcab receives a reward for each successfully completed trip, and also receives a smaller reward for each action it executes successfully that obeys traffic rules. The smartcab receives a small penalty for any incorrect action, and a larger penalty for any action that violates traffic rules or causes an accident with another vehicle. Based on the rewards and penalties the smartcab receives, the self-driving agent implementation should learn an optimal policy for driving on the city roads while obeying traffic rules, avoiding accidents, and reaching passengers' destinations in the allotted time.

Implement a Basic Driving Agent

To begin, your only task is to get the smartcab to move around in the environment. At this point, you will not be concerned with any sort of optimal driving policy. Note that the driving agent is given the following information at each intersection:

The next waypoint location relative to its current location and heading. The state of the traffic light at the intersection and the presence of oncoming vehicles from other directions. The current time left from the allotted deadline. To complete this task, simply have your driving agent choose a random action from the set of possible actions (None, 'forward', 'left', 'right') at each intersection, disregarding the input information above. Set the simulation deadline enforcement, enforce_deadline to False and observe how it performs.

QUESTION: Observe what you see with the agent's behavior as it takes random actions. Does the smartcab eventually make it to the destination? Are there any other interesting observations to note?

ANSWER:

The observation shown the agent's behavior takes the random action (self.state = Random) from the valid action such as None, left, forward and right. The target is a red car and the final destination is the dot in the grid with the circle in the outer side. I observed it actually does rarely get to the destination (reward: 2.0) but I can't determine the timing when it will get to the target.

Interesting observation to note, we can use this analyze the reward to train the agent's to have better performance.

The comparison for better performance we can do by forcing enforce_deadline to True and report the results of 100 tries and adjust the delay. We also can examine the random agent success rate in this case. The result of this test can be found in output_random1.txt

Inform the Driving Agent

Now that your driving agent is capable of moving around in the environment, your next task is to identify a set of states that are appropriate for modeling the smartcab and environment. The main source of state variables are the current inputs at the intersection, but not all may require representation. You may choose to explicitly define states, or use some combination of inputs as an implicit state. At each time step, process the inputs and update the agent's current state using the self.state variable. Continue with the simulation deadline enforcement **enforce_deadline** being set to **False**, and observe how your driving agent now reports the change in state as the simulation progresses.

QUESTION: What states have you identified that are appropriate for modeling the smartcab and environment? Why do you believe each of these states to be appropriate for this problem?

OPTIONAL: How many states in total exist for the smartcab in this environment? Does this number seem reasonable given that the goal of Q-Learning is to learn and make informed decisions about each state? Why or why not?

ANSWER:

1- I identified there are three states and they are; - Light - Oncoming - Next Waypoint

I reverted the code and observed the state with **deadline**. It gets really bad performance. My thought on this is because the Q matrix got too scattered and it affects the prediction to fail the required value in Q matrix. The best way to describe this is the Q value does not tend to meet at a point when we add the deadline into the state.

My intuition led me into the information given from the video. I believe each of these states are appropriate for this problem. This knowledge needs to have because the agent needs to know where to go through the intersection without disrupting the traffic. The agent also needs to know what LIGHT is turning. Green to pass, Yellow to be ready and Red to STOP. If the LIGHT is green then the agent can go straight. The left turn only happens when the green arrow light is on and state of the on-coming cars. Red LIGHT, then the agent needs to turn right. This happens in order not to disrupt the ONCOMING vehicle from different sides. The agent should check the ONCOMING vehicle before deciding to go step. The NEXTWAYPOINT will lead the agent reach to the destination which this can be found from the planner instruction.

2- Total states exsist for the agent with this environment is 2x4x4=32 states with 4 methods (None, FWD, Left, Right). This came from mixed combination from right, left, light, oncoming and next waypoint. From the information given, we can conclude that our Q-learning matrix will be 32x4 =128. This number should be reasonable to perform because the states are close enough for the agent to learn and make decision in each state.

Implement a Q-Learning Driving Agent

With your driving agent being capable of interpreting the input information and having a mapping of environmental states, your next task is to implement the Q-Learning algorithm for your driving agent to choose the best action at each time step, based on the Q-values for the current state and action. Each action taken by the smartcab will produce a reward which depends on the state of the environment. The Q-Learning driving agent will need to consider these rewards when updating the Q-values. Once implemented, set the simulation deadline enforcement enforce_deadline to True. Run the simulation and observe how the smartcab moves about the environment in each trial.

QUESTION: What changes do you notice in the agent's behavior when compared to the basic driving agent when random actions were always taken? Why is this behavior occurring?

ANSWER:

I have implemented Q-learning algorithm by selecting the agent's action per Q values. The discount will be gamma and the step size will be alpha. Some preconditions apply in this case

- The discount factor is Gamma which this is the Bellman equation formula and will be tested (range 0.1 to 1.0)
- The step size is alpha or simply justify this as learning rate (0.0 - 1.0 Range)
- The state variables are light, oncoming, next waypoint (this action will let the agent's try new path)

When we tested with random order, the agent really takes many steps to get the destination with the lowest rewards as observed. The agent will reach the destination with fewer steps and higher rewards after several trials.

I have taken deeper look from this link and found the Q-table updating rules and as well with updating the learning rate then finally updated with "simulated annealing" approach for choosing best action.

I have observed the result and found out the agent gets into circles or loop actions and rarely reached to the target location. This happens beacuse Q-learning tries to maximize the reward when the initial rewards for all environment actions was set to 0. From the link given about epsilon decay, I came to the conclusion that the agent should be able to get rid from infinte loop by implementing Epsilon Greedy Exploration algorithm and in the end reached the destination. I believe in this case exploring refers to the infinite loop to find the destination and the policy was to avoid the negative award.

Enhance the driving agent

Your final task for this project is to enhance your driving agent so that, after sufficient training, the smartcab is able to reach the destination within the allotted time safely and efficiently. Parameters in the Q-Learning algorithm, such as the learning rate (alpha), the discount factor (gamma) and the exploration rate (epsilon) all contribute to the driving agent’s ability to learn the best action for each state. To improve on the success of your smartcab:

- Set the number of trials, n_trials, in the simulation to 100.

- Run the simulation with the deadline enforcement enforce_deadline set to True (you will need to reduce the update delay update_delay and set the display to False).
- Observe the driving agent's learning and smartcab's success rate, particularly during the later trials.
- Adjust one or several of the above parameters and iterate this process.

This task is complete once you have arrived at what you determine is the best combination of parameters required for your driving agent to learn successfully.

QUESTION: Report the different values for the parameters tuned in your basic implementation of Q-Learning. For which set of parameters does the agent perform best? How well does the final driving agent perform?

**ANSWER: ** The different values for the parameters tuned (per learning rate: alpha and discount rate : Gamma). I can see that when alpha is closer to zero, the agent won't perform well. However when the agent alpha's sets closer to 1, the agent tends to forget what it has already learnt. The Gamma value is randomize from 0.1 to 1. The closer to 0.1, I found that the next states will be reduced and when it gets closer to 1 the next states will be larger.

QUESTION: 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? How would you describe an optimal policy for this problem?

**ANSWER: ** The agent actually gets close to finding an optimal policy to reach the destination in the minimum possible time without any penalties. We can observe this from the three states (green - None - Forward, green - left - forward and green, forward - forward). Well the optimal policy I have found by setting the gamma to 0.33 through many trials as well with the epsilon closer to 0 (in this case, epsilon sets to 0.1). I might look for second opinion about my results for defining my policy.

Here are snippet the results for the best results

('green', None, 'forward') ['2.13', '1.00', '1.00', '1.00']

('green', 'left', 'forward') ['2.01', '1.00', '1.00', '1.00']

('green', 'forward', 'forward') ['2.67', '1.00', '1.00', '1.00']

I have printed out the statistic per code review suggestion. I found out with the hardcoded (some trials and errors) brute forcing method I can come with conclusion that the success rate with alpha 0.9, gamma 0.33 and epsilon 0.1 is 97/99 of 1301 trials. This is the last run from the latest generated output with the total reward of 2230.5. With this tuning parameter, I could get 0.03 penalty rate which consider really low for improved Learning Agent.

This part, I will discuss the performance tuning. I also will explain why the conclusion comes with alpha 0.9, gamma 0.33 and epsilon 0.1. The different parameters tried and their corresponding performances are:

1) Alpha : 0.5, Gamma: 0.3, Epsilon: 0.2 (First Trial)

```
The parameters give me results of 97/99 over 1421 trials with total reward of
2269.5. However, we can see that the penalty rate is considered high. It is 10% of
the sucessful moves. Observed best policies are:
```

- ('green', 'right', 'left') ['1.00', '1.00', '1.00', '1.00'] - ('green', 'left', 'forward') ['2.12', '1.00', '1.00', '1.00'] - ('green', None, 'right') ['0.40', '0.09', '2.81', '0.85'] - ('green', 'right', 'forward') ['2.32', '1.00', '1.00', '1.00'] - ('green', None, 'left') ['0.25', '2.45', '0.41', '0.69']

2) Alpha : 0.6, Gamma: 0.1, Epsilon: 0.2 (Observe Alpha and Gamma)

```
The parameters give me results of 98/100 over 1552 trials with total reward of
2286.5. Barely, from this statistic, we can see that the penalty rate is way
smaller than the previous parameters. It is 8% of penalty from 131/1552 trials
observation. Observed best policies are:
```

- ('green', 'left', 'forward') ['2.01', '1.00', '1.00', '1.00'] - ('green', 'right', 'forward') ['1.72', '1.00', '1.00', '1.00'] - ('green', 'forward', 'forward') ['1.73', '1.00', '1.00', '1.00']

3) Alpha: 0.9, Gamma: 0.1, Epsilon: 0.1 (Observe Alpha)

```
The parameters give me results of 98/99 over 1379 trials with total reward of
2197.5. The total reward is smaller but look over the number of trials. We can
conclude this is close to the results from parameters (2) because the penalty rate
is the same which it is 8%. Observed best policies are:
```

- ('green', 'right', 'left') ['1.00', '1.00', '1.00', '1.00']
- ('red', 'right', 'left') ['1.00', '1.00', '1.00', '1.00']
- **('green', 'left', 'forward') ['2.01', '1.00', '1.00', '1.00']**
- ('green', 'left', 'right') ['1.00', '1.00', '1.00', '1.00']
- ('green', 'right', 'right') ['1.00', '1.00', '1.00', '1.00']
- ('red', 'right', 'right') ['1.00', '1.00', '1.00', '1.00']
- ('red', 'left', 'right') ['1.00', '1.00', '1.00', '1.00']
- ('red', 'left', 'forward') ['1.00', '1.00', '1.00', '1.00']
- ('green', 'right', 'forward') ['1.00', '1.00', '1.00', '1.00']
- **('green', 'forward', 'forward') ['2.00', '1.00', '1.00', '1.00']**I want to address the majority of the policies here, which one example is ('green', 'left', 'right') ['1.00', '1.00', '1.00', '1.00']. The policy starts to predict the involved upcoming cars. In simple ways, it's expecting oncoming and the agent intends to learn before the state occurs. We also can observe that the situation of oncoming doesn't happen often but the agent still tries to learn the policy. In sum, this is the main reason, I have seen several observation results with the ['1.00', '1.00', '1.00', '1.00'] metric.

4) Alpha: 0.9, Gamma: 0.33, Epsilon: 0.2 (Tune Epsilon)

```
The parameters give me success rate 94/98 over 1497 trials with reward of 2253.5.
This is the highest penalty rate over the observation. We observe thhat the penalty
reate is higher than the previous observation which is 11%. The best policies with
this parameters observed are:
```

- **('green', 'right', 'left') ['0.51', '1.00', '1.00', '1.00']** - ('green', 'left', 'left') ['1.00', '1.00', '1.00', '1.00'] - ('red', 'left', 'left') ['1.00', '1.00', '1.00', '1.00'] - ('red', 'forward', 'right') ['1.00', '1.00', '1.00', '1.00'] - ('red', 'right', 'left') ['1.00', '1.00', '1.00', '1.00'] - **('green', None, 'forward') ['2.81', '-0.41', '0.21', '0.70']** - ('red', 'right', 'forward') ['1.00', '1.00', '1.00', '1.00'] - ('green', 'forward', 'left') ['1.00', '1.00', '1.00', '1.00'] - **('green', 'left', 'forward') ['2.77', '1.00', '1.00', '1.00']** - ('green', 'right', 'right') ['1.00', '1.00', '1.00', '1.00'] - ('red', 'right', 'right') ['1.00', '1.00', '1.00', '1.00'] - **('red', None, 'right') ['-0.05', '-0.07', '2.09', '0.71']** - **('green', None, 'right') ['0.44', '-0.12', '2.91', '0.97']** - **('red', 'left', 'right') ['-0.50', '-0.50', '2.78', '1.00']** - ('green', 'right', 'forward') ['1.00', '1.00', '1.00', '1.00'] - **('green', 'forward', 'forward') ['1.90', '1.00', '0.27', '1.00']** - **('green', None, 'left') ['-0.05', '2.07', '0.19', '0.94']**

5) Alpha:0.9, Gamma: 0.33, Epsilon: 0.1 (Final Observation, Tune Gamma)

```
The parameters give me result of 100/100 over 1257 trials with total reward: 2222.0.
This is the best result we can generate though its 100/100 success, we still get a
lower penalty rate which 2%. The best observed policies are:
```

- ('green', 'right', 'left') ['1.00', '1.00', '1.00', '1.00'] - ('green', 'forward', 'right') ['1.00', '1.00', '1.00', '1.00'] - ('red', 'right', 'left') ['1.00', '1.00', '1.00', '1.00'] - **('green', None, 'forward') ['2.80', '1.00', '1.00', '1.00']** - ('red', 'right', 'forward') ['1.00', '1.00', '1.00', '1.00'] - **('green', 'left', 'forward') ['1.90', '1.00', '1.00', '1.00']** - ('green', 'left', 'right') ['1.00', '1.00', '1.00', '1.00'] - ('green', 'right', 'right') ['1.00', '1.00', '1.00', '1.00'] - ('red', 'right', 'right') ['1.00', '1.00', '1.00', '1.00'] - **('green', 'right', 'forward') ['2.49', '1.00', '1.00', '1.00']** - **('green', 'forward', 'forward') ['1.90', '1.00', '1.00', '1.00']** - **('green', None, 'left') ['-0.05', '2.92', '1.00', '1.00']**

The optimal policy I can observe so far is the agent always tries to follow the rules of the road and get positive rewards when the Alpha gets closer to 1 with lower Epsilon. I compare this based on the tuning parameters and the policies. The agent intends to have short memory when Alpha gets to closer to 1 however it intends to give positive rewards after the exploration step. In this case, the agent still to observes the best route to reach the next_waypoint/ destination. Another good things I have observed, the agent tries to get within the shortest path to the next_waypoint without getting negative rewards. If we take a look closely to the discount factor (gamma), we can see that the agent learns from the previous state when the value is closer to 0. By looking at the success rate 100/100 and lowest penalty error, I can summarize the best tuning parameters are Alpha: 0.9, Gamma: 0.33 and Epsilon: 0.1. The best state is Green, None, Left.