Reward:
The objective is to maximize the profit of a driver. It is assumed that both the cost and the revenue are purely functions of time, i.e. for every hour of driving, the cost (of battery and other costs) and the revenue (from the customer) is the same - irrespective of the traffic conditions, speed of the car etc.

So, the reward function will be (revenue earned from pickup point p to drop point q) - (Cost of battery used in moving from pickup point p to drop point q) - (Cost of battery used in moving from current point i to pick-up point p).

Activity 1: Environment Summary

The script is performed on the windows 10, 64-bit system with 32GB RAM. The script can be run on Google Colab, however due to their usage limitation, it is not feasible to run the script longer than 3 hours. Therefore, it is being run locally on using Anaconda. A total of 5000 episodes were executed to train the agent, which takes more than 10 hours to complete.

An environment class (ENV) is used for running the DQN, either using architecture 1 or 2. Within the ENV there are a number of methods (function) that which formulate the data to be fed into the NN. This defines the various state of the driver as he start or end a journey together with the reward from the job. It also defines the duration of the journey and the prepares the next state.

The environment are imported with each of the methods below and the brief explanation of their purpose;

The overall environment is to the Class CabDriver which is imported to the main file.

Hyperparameters are defined at the initial.

__init__ : For initializing the state, action space, and state space.

state_encode_arch1 : This method converts a given state into vector format. Eg, (m + t +d). This is used for Architecture 1 of the DQN.

state_encod_arch2 : This method converts a given state and action into vector format. Eg, (m + t + d + m + m). This is used for Architecture 2 of the DQN

requests : This method will determine the number of requests from the locations. (0,0) will also be added to denote where driver is not picking up customers.

reward_funct : It formulates the reward based on the state, action and time-matrix.

update_time_day : Calculate the current state and time taken for the journey.

next_state_func : Taking the current state and action as input and returns the next state. It considers situation whereby the driver is resting or calculating the duration of the journey. Then it returns the next state.

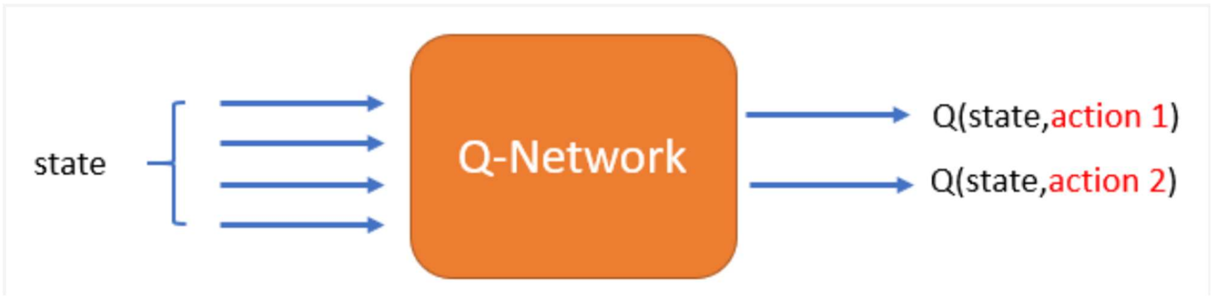
reset : resetting all the variables or action and state

step : It takes the next state and various time durations, calculating the reward based on the time durations.

The remaining methods are for assigning the various states and actions for different scenarios.

Activity 2: DQN Overview

As stated, we will be implementing Reinforcement Learning system to emulate the business problem. After setting up the proper environment, I will train my agent using Deep Q-Learning. Within the model, I'll apply Architecture 1 where the current states are fed into the Q-Network and it will churn out the different state and action.



The Q-Network underlying principle is the epsilon-greedy function. It will do some exploration to gauge which rewards are better before deciding on the most rewarding route and continues. It is a better option than the normal greedy function as it limits the exploration, where the reward are likely to be sub-optimal.

The epsilon and epsilon-decay are to be set so to allow the agent to explore the states maximally at start. It will continue to do so until it settles down to a lower exploration rate. Together with the epsilon, discount factors and learning rate which are the hyperparameters that will be used for the DQN as listed below;

discount_factor = 0.95

learning_rate = 0.01

epsilon = 1

epsilon_max = 1

epsilon_decay = -0.0005

epsilon_min = 0.00001

To reach the optimal policy, we should be running 15,000 (or more) episodes so the agent can pick up the best rewards over sufficient time periods. However, due to resource limitation (time and hardware), 5k episodes were used for the training. Our result may not be the best, however it does allow us to gain an insight on the final possible outcome.

In the DQNAgent Class, the core method is built using the neural network. 3 hidden layers were inserted using the relu function. The Adam optimizer function is used to compile. Other methods such as get_action, append_Sample and train_model works on the different action and state of the agent during the episodes. Lastly, the state is saved to track the agent, and later for plotting charts.

Activity 3: Convergence Overview

Ideally, the Q-values (rewards) will reach the optimal level when the agent learns to pick the best route as the epsilon decays after each episode. This is referred to as convergence, where the Q-values reaches a stable point.

At the initial start, the rewards are randomised and hence the agent go through iterations after iterations of the states. Initial stage where the epsilon is one, the chances of the agent exploring will be high. As the epsilon decays, the probability of exploration will reduce, and when it doesn't explore, it picks the most rewarding route. As the epsilon reduces to a point where exploitation will become the default choice of the agent. This is the point where the Q-values will experience a convergence to the optimal Q-value.

Figure. 1

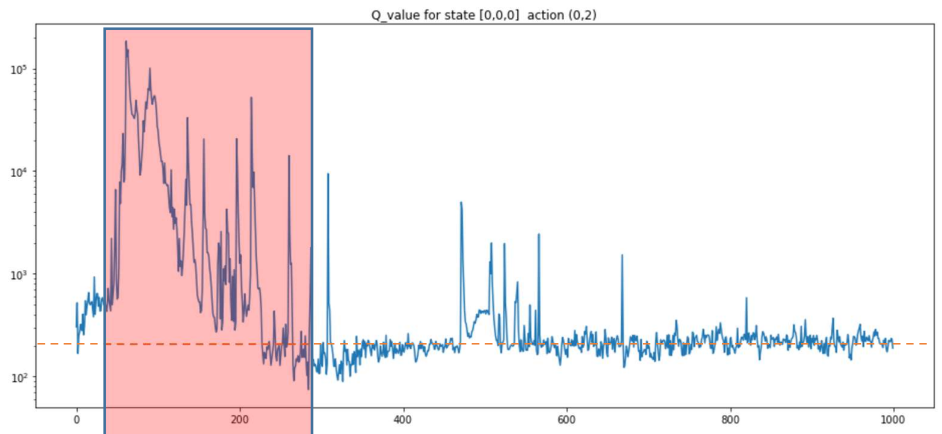
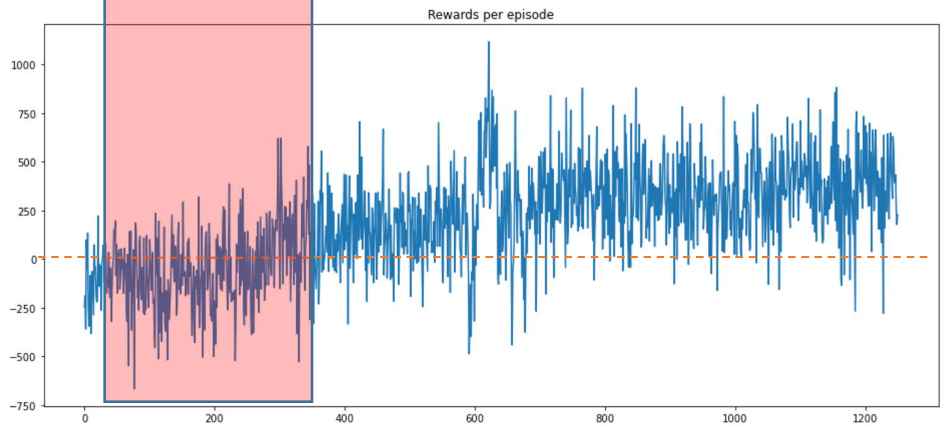


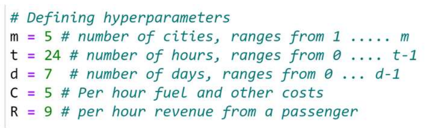
Figure. 2



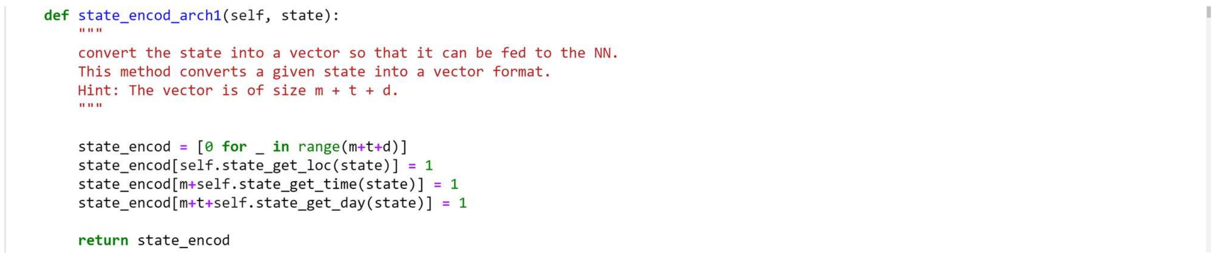
From Figure.1 we can see the Q-values will stabilise at about 1300 level. It is interesting to note that the Q-value can reach high level of over a million at the initial stage. It tapers down towards the 1300 level about halfway through the training. The tapering down to 1300 make economical sense by looking at Figure. 2, where the rewards per episode was mostly negative (below red line) at the beginning. It mean to say that even though the Q-values are high at start, the total rewards for the drivers are low or negative after each 30 days.

It can be seen here that the DQN is learning to maximise the reward for the long term. Given more time (episodes) to train the agent, there is a likelihood the agent will be able to find the optimal revenue for the drivers, which in turn will retain good drivers for the company.

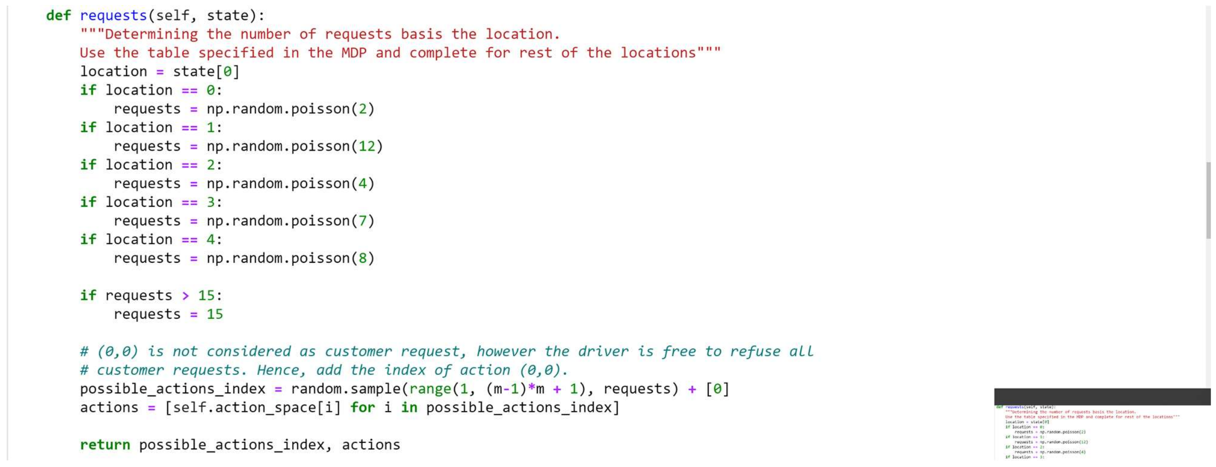
Screenshots of each task of Activity 1 and its explanation



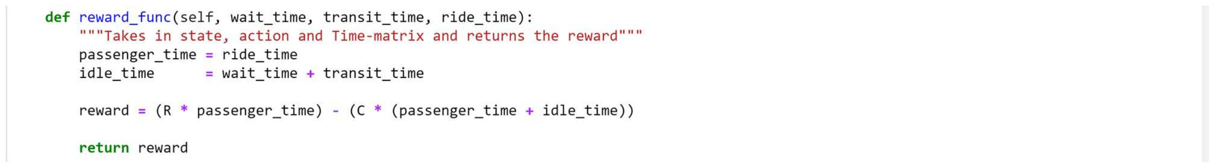
```
# Defining hyperparameters
m = 5 # number of cities, ranges from 1 ..... m
t = 24 # number of hours, ranges from 0 .... t-1
d = 7  # number of days, ranges from 0 ... d-1
C = 5 # Per hour fuel and other costs
R = 9 # per hour revenue from a passenger
```



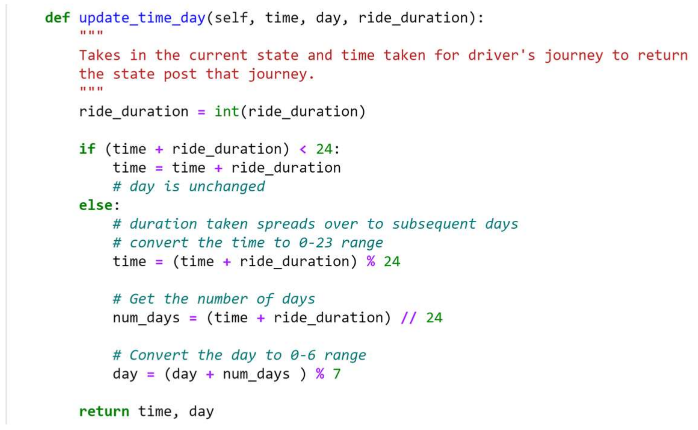
```
    def state_encod_arch1(self, state):
        """
        convert the state into a vector so that it can be fed to the NN.
        This method converts a given state into a vector format.
        Hint: The vector is of size m + t + d.
        """

        state_encod = [0 for _ in range(m+t+d)]
        state_encod[self.state_get_loc(state)] = 1
        state_encod[m+self.state_get_time(state)] = 1
        state_encod[m+t+self.state_get_day(state)] = 1

        return state_encod
```



```
    def requests(self, state):
        """Determining the number of requests basis the location.
        Use the table specified in the MDP and complete for rest of the locations"""
        location = state[0]
        if location == 0:
            requests = np.random.poisson(2)
        if location == 1:
            requests = np.random.poisson(12)
        if location == 2:
            requests = np.random.poisson(4)
        if location == 3:
            requests = np.random.poisson(7)
        if location == 4:
            requests = np.random.poisson(8)

        if requests > 15:
            requests = 15

        # (0,0) is not considered as customer request, however the driver is free to refuse all
        # customer requests. Hence, add the index of action (0,0).
        possible_actions_index = random.sample(range(1, (m-1)*m + 1), requests) + [0]
        actions = [self.action_space[i] for i in possible_actions_index]

        return possible_actions_index, actions
```



```
    def reward_func(self, wait_time, transit_time, ride_time):
        """Takes in state, action and Time-matrix and returns the reward"""
        passenger_time = ride_time
        idle_time      = wait_time + transit_time

        reward = (R * passenger_time) - (C * (passenger_time + idle_time))

        return reward
```



```
    def update_time_day(self, time, day, ride_duration):
        """
        Takes in the current state and time taken for driver's journey to return
        the state post that journey.
        """
        ride_duration = int(ride_duration)

        if (time + ride_duration) < 24:
            time = time + ride_duration
            # day is unchanged
        else:
            # duration taken spreads over to subsequent days
            # convert the time to 0-23 range
            time = (time + ride_duration) % 24

            # Get the number of days
            num_days = (time + ride_duration) // 24

            # Convert the day to 0-6 range
            day = (day + num_days ) % 7

        return time, day
```

```python
    def next_state_func(self, state, action, Time_matrix):
        """Takes state and action as input and returns next state"""

        next_state = []

        # Initialize various times
        total_time   = 0
        transit_time = 0    # to go from current  location to pickup location
        wait_time    = 0    # in case driver chooses to refuse all requests
        ride_time    = 0    # from Pick-up to drop

        # Derive the current location, time, day and request locations
        curr_loc = self.state_get_loc(state)
        pickup_loc = self.action_get_pickup(action)
        drop_loc = self.action_get_drop(action)
        curr_time = self.state_get_time(state)
        curr_day = self.state_get_day(state)
        if ((pickup_loc== 0) and (drop_loc == 0)):
            # Refuse all requests, so wait time is 1 unit, next location is current location
            wait_time = 1
            next_loc = curr_loc
        elif (curr_loc == pickup_loc):
            # means driver is already at pickup point, wait and transit are both 0 then.
            ride_time = Time_matrix[curr_loc][drop_loc][curr_time][curr_day]

            # next location is the drop location
            next_loc = drop_loc
        else:
            # Driver is not at the pickup point, he needs to travel to pickup point first
            # time take to reach pickup point
            transit_time      = Time_matrix[curr_loc][pickup_loc][curr_time][curr_day]
            new_time, new_day = self.update_time_day(curr_time, curr_day, transit_time)

            # The driver is now at the pickup point
            # Time taken to drop the passenger
            ride_time = Time_matrix[pickup_loc][drop_loc][new_time][new_day]
            next_loc  = drop_loc
        # Calculate total time as sum of all durations
        total_time = (wait_time + transit_time + ride_time)
        next_time, next_day = self.update_time_day(curr_time, curr_day, total_time)

        # Construct next_state using the next_loc and the new time states.
        next_state = [next_loc, next_time, next_day]

        return next_state, wait_time, transit_time, ride_time


    def reset(self):
        return self.action_space, self.state_space, self.state_init


    def step(self, state, action, Time_matrix):
        """
        Take a trip as cabby to get rewards next step and total time spent
        """
        # Get the next state and the various time durations
        next_state, wait_time, transit_time, ride_time = self.next_state_func(
            state, action, Time_matrix)

        # Calculate the reward based on the different time durations
        rewards = self.reward_func(wait_time, transit_time, ride_time)
        total_time = wait_time + transit_time + ride_time

        return rewards, next_state, total_time
```