# Project Technical Environment:

The main business objective of the problem is to maximise the profit over a long term period. There are several assumptions made for the situations.

**Assumptions:**

1. The taxis are electric cars. It can run for 30 days non-stop, i.e., 24*30 hrs. Then it needs to recharge itself. The terminal state is independent of the number of rides covered in a month, it is achieved as soon as the cab-driver crosses 720 hours.
2. There are only 5 locations in the city where the cab can operate.
3. All decisions are made at hourly intervals.
4. The time taken to travel from one place to another is considered in integer hours (only) and is dependent on the traffic.

**State:**

The state space is defined by the driver's current location along with the time components (hour-of-the-day and the day-of-the-week). A state is defined by three variables:

$$s=XiTjDk \ where \ i=0\ldots m-1; j=0\ldots.t-1; k=0\ldots..d-1$$

Where $Xi$ represents a driver's current location, $Tj$ represents time component (more specifically hour of the day), $Dk$ represents the day of the week:

- Number of locations: m = 5
- Number of hours: t = 24
- Number of days: d = 7

A terminal state is achieved when the cab completes his 30 days, i.e., an episode is 30 days long.

**Actions:**

Every hour, driver will base on the current 'state', whether to pick up new clients to maximise his monthly revenue. If some passenger is already on-board, then the driver won't get any requests. Therefore, an action is represented by the tuple (pick-up, drop) location. In a general scenario, the number of requests the cab-driver can get at any state is not the same. Apart from these requests, the driver always has the option to go 'offline' (accept no ride). The no-ride action just moves the time component by 1 hour.

**State Transition:**

The total time taken will be calculated to reach from one point to other from the Time Matrix (calculated basis the historical data) provided in the zip file, TM.npy.

This matrix has been calculated considering the distance between two locations and traffic conditions, which generally depends on the hour-of-the-day and the-day-of-the-week. To make the problem manageable, the 24-hour frame is divided into 4 segments: from 12:00 am to 6:00 am, 6:00 am to 12:00 pm, 12:00 pm to 6:00 pm and 6:00 pm to 12:00 am.

**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.

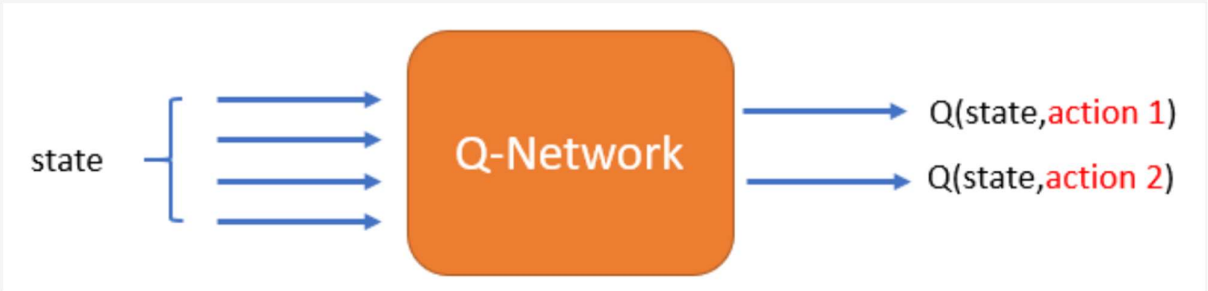
reset : resetting all the variables or action and state

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

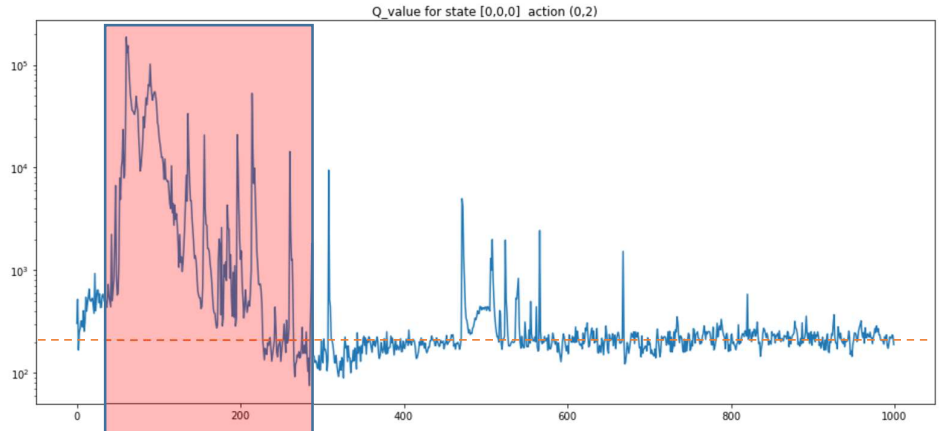
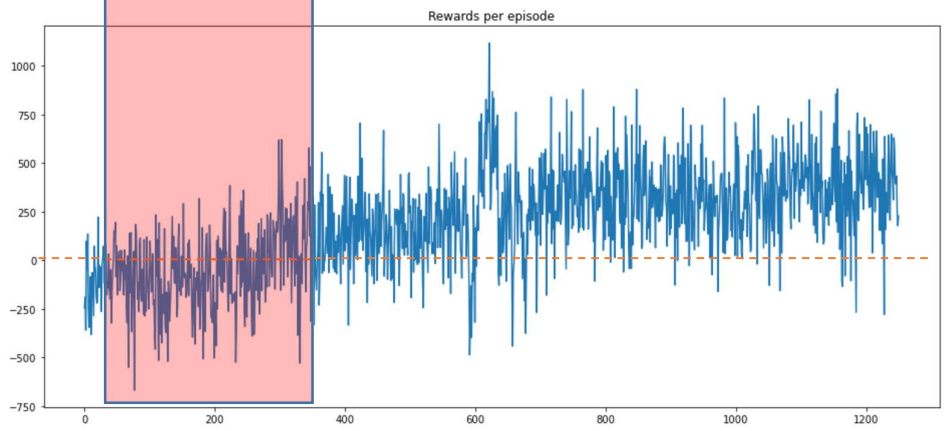



Figure. 2




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.