ARTIFICIAL INTELLIGENCE FINAL PROJECT

On

RL-BASED SYSTEM FOR ASSISTING CAB DRIVERS

Submitted By:

Sai Saran Naraharisetti

Ramya prathusha Paladugu

Jahnavi Mulpuri

Under the Guidance of

SHIVANJALI KARE (Assistant Professor)

Project Objective:-

The goal of our project is to build an RL-based algorithm which can help cab drivers maximize their profits by improving their decision-making process on the field.

In this highly competitive industry, retention of good cab drivers is crucial in business. Cab drivers, like most people are incentivized by a healthy growth in income. The Need for Choosing the 'Right' Requests Most drivers get a healthy number of ride requests from customers throughout the day. But with the recent hikes in electricity prices (all cabs are electric), many drivers complain that although their revenues are gradually increasing, their profits are almost flat. Thus, it is important that drivers choose the 'right' rides, i.e., choose the rides which are likely to maximize the total profit earned by the driver that day. The Need for Choosing the 'Right' Requests Most drivers get a healthy number of ride requests from customers throughout the day. But with the recent hikes in electricity prices (all cabs are electric), many drivers complain that although their revenues are gradually increasing, their profits are almost flat. Thus, it is important that drivers choose the 'right' rides, i.e., choose the rides which are likely to maximize the total profit earned by the driver that day.

For example, say a driver gets three ride requests at 10 PM. The first one is a long-distance ride guaranteeing high fare, but it will take him to a location which is unlikely to get him another ride for the next few hours. The second one ends in a better location, but it requires him to take a slight detour to pick the customer up, adding to fuel costs. Perhaps the best choice is to choose the third one, which although is medium distance, it will likely get him another ride subsequently and avoid most of the traffic.

APPROACH: -

In this project, we need to create the environment and an RL agent that learns to choose the best request. We need to train our agent using Deep Q-learning (DQN).

Goals: -

Create the environment:

The ‘Env.py’ file is the "environment class" - each method (function) of the class has a specific purpose.

Build an Agent:

Build an agent that learns to pick the best request using DQN. We can choose the hyperparameters (epsilon (decay rate), learning-rate, discount factor etc.) of our choice.

Training depends purely on the epsilon-function we choose. If it decays fast, it won't let our model explore much and the Q-values will converge early but to suboptimal values. If it decays slowly, our model will converge slowly.

Assumptions: -

1. The taxis are electric cars. It can run for 30 days non-stop, i.e., 24*30 hours.

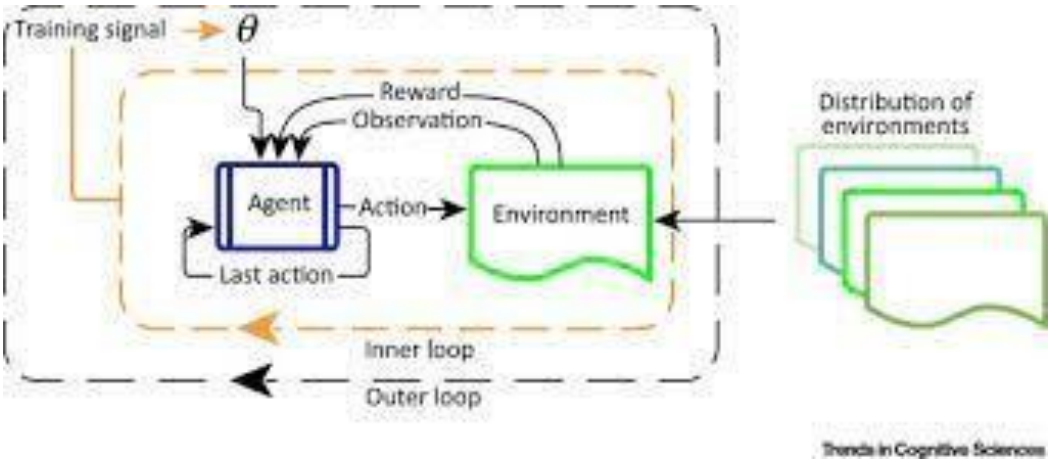
Then it needs to recharge itself. If the cab driver is completing his trip at that time, he will finish that trip and then stop for recharging. So, the terminal state is independent of the number of rides covered in a month, it is achieved as soon as the cab driver crosses 24*30 hours.

2. There are only 5 locations in the city where the cab can operate.

3. All decisions are made at hourly intervals. We won't consider minutes and seconds for this project. So for example, the cab driver gets request at 3 PM then at 4 PM and so on. He can decide to pick among the requests only at these times. A request cannot come at 3.30 PM.

4. The time taken to travel from one place to another is considered in integer hours only and is dependent on the traffic. Also, the traffic is dependent on the hour-of-the-day and the day-of-the-week.

Reinforcement Learning: -



Reinforcement Learning is a subset of machine learning. It enables an agent to learn through the consequences of actions in a specific environment.

It differs from other forms of supervised learning because the sample data set does not train the machine. Instead, it learns by trial and error. Therefore, a series of right decisions would strengthen the method as it better solves the problem.

Reinforced learning is similar to what we humans have when we are children. We all went through the learning reinforcement — when you started crawling and tried to get up, you fell over and over, but your parents were there to lift you and teach you.

Applications areas of RL: -

- Personalized Recommendations
- Robotics
- Games
- Deep Learning

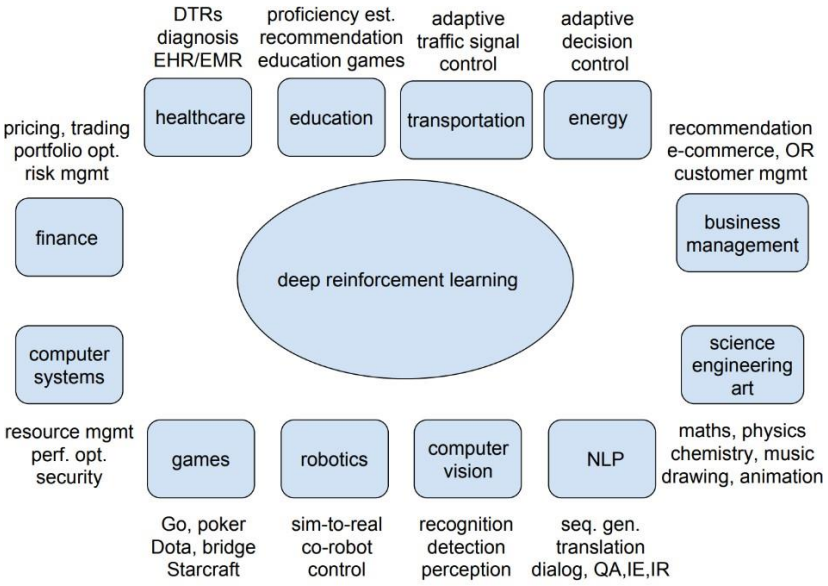


Figure 4: Deep Reinforcement Learning Applications

Challenges: -

- Reinforcement learning's key challenge is to plan the simulation environment, which relies heavily on the task to be performed.
- Transferring the model from the training setting to the real world becomes problematic.
- Scaling and modifying the agent's neural network is another problem.
- Another difficulty is reaching a great location — that is, the agent executes the mission as it is, but not in the ideal or required manner.

Reinforcement is done with rewards according to the decisions made; it is possible to always learn continuously from interactions with the environment. With each correct action, we will have positive rewards and penalties for incorrect decisions. In the industry, this type of learning can help optimize processes, simulations, monitoring, maintenance, and the control of autonomous systems.

Markov Decision Process :-

In mathematics, a Markov decision process (MDP) is a discrete-time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker. MDPs are useful for studying optimization problems solved via dynamic programming.

A Markov decision process is a 4-tuple **(S, A, P_a, R_a**), where:

- S is a set of states called the state space,
- A is a set of actions called the action space (alternatively, A_s is the set of actions available from state s),
- P(s, s')= Pr(s_t+1=s'|s_t=s, a_t=a) is the probability that action a in state s at time t will lead to state s' at time t+1,
- R_a(s, s') is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a

The state and action spaces may be finite or infinite, for example the set of real numbers. Some processes with countably infinite state and action spaces can be reduced to ones with finite state and action spaces.

A policy function pi is a (potentially probabilistic) mapping from state space to action space.

Deep Q-Learning: -

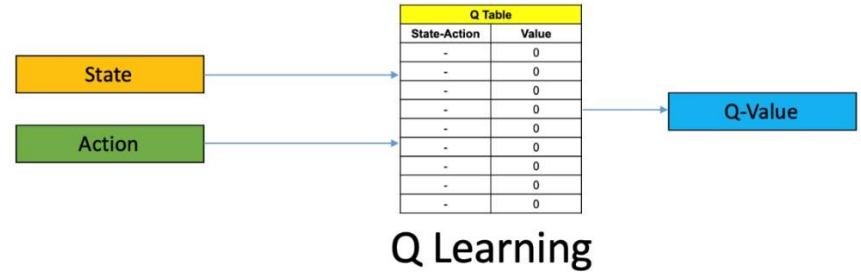
Q-learning is a simple yet quite powerful algorithm to create a cheat sheet for our agent. This helps the agent figure out exactly which action to perform.

But what if this cheatsheet is too long?

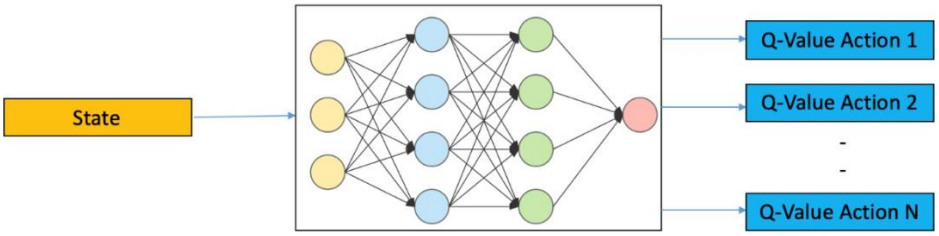
This presents two problems:

- First, the amount of memory required to save and update that table would increase as the number of states increases.
- Second, the amount of time required to explore each state to create the required Q-table would be unrealistic.

In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. The comparison between Q-learning & deep Q-learning is wonderfully illustrated as below.



Q Learning



Deep Q Learning

Code Explanation:-

Environment Class: -

A reinforcement learning task is about training an agent which interacts with its environment. The agent arrives at different scenarios known as states by performing actions. Actions lead to rewards which could be positive and negative.

The agent has only one purpose here – to maximize its total reward across an episode. This episode is anything and everything that happens between the first state and the last or terminal state within the environment. We reinforce the agent to learn to perform the best actions by experience. This is the strategy or policy.

This is the "environment class" - each method (function) of the class has a specific purpose.

Initialize the hyper parameters

- m =5, number of locations ranges from 0 ..... m-1
- t =24, number of hours, ranges from 0 .... t-1
- d =30, number of days, ranges from 0 ... d-1
- C =5, Cost Per hour for fuel and other costs
- R =9, Reward per hour revenue from a passenger

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. The vector is of size m + t + d

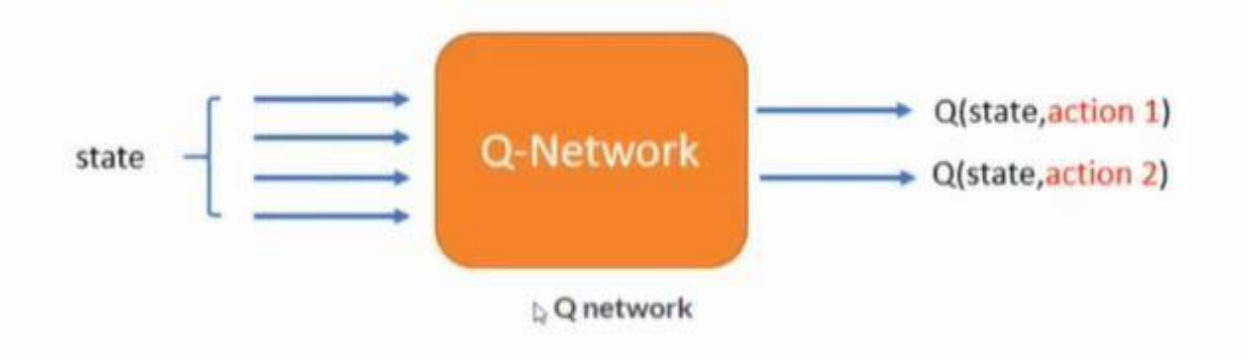
```
def state_encod_arch1(self, state):

    # creating the vector of size 5+24+7=36
    state_encod = np.zeros(m+t+d)
    #storing index of current state,time and day
    state_index=state[0]
    time_index = m + state[1]
    day_index = m + t + state[2]
     # encode location
    state_encod[state_index] = 1
     # encode hour of the day
    state_encod[time_index] = 1
     # encode day of the week
    state_encod[day_index] = 1

    return state_encod
```

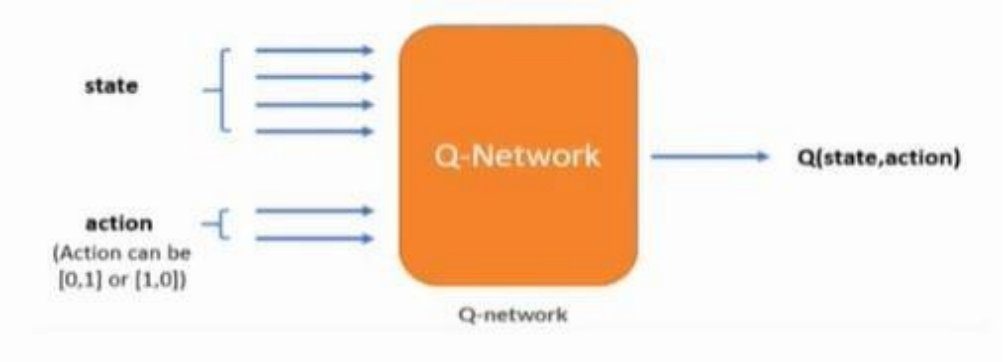
We have 2 architectures of DQN,

1. We pass only State as input

2. We pass State and Action as input

The Architecture 1 (only State as input) performs better than Architecture 2 because we will get Q(s, a) for each action, so you have to run the NN just once for every state. Take the action for which Q(s, a) is the maximum.





Next State function: -

Takes state and action as input and returns next state with considering below conditions.

1. driver refuse to request.

2. cab is already at pick up point.

3. cab is not at the pickup point.

```
# Calculate total time as sum of all durations
total_time = (waiting_time + pickup_time + drop_time)
next_time, next_day = self.new_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, waiting_time, pickup_time, drop_time
```

Reward function: -

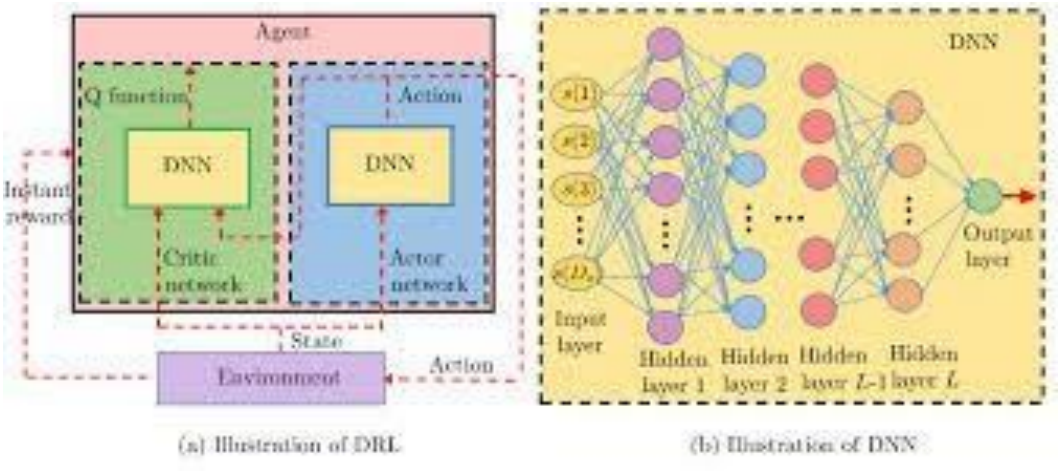
Assessment requires to determine what action is to be taken to minimize loss and maximize benefits. The reward, r(s, a), in our system for taking an action a ∈ A at a given state s ∈ S is computed as follows.

Your objective is to maximize the profit of a driver.

$$R(s = X_i T_j D_k) = \begin{cases} R_k * (Time(p,q)) - C_f * (Time(p,q) + Time(i,p)) & a = (p,q) \\ -C_f & a = (0,0) \end{cases}$$

```
def reward_func(self, waiting_time, pickup_time, drop_time):
    """Takes in state, action and Time-matrix and returns the reward"""
    # pickup time and waiting time we will not get revenue,only we loose
    passenger_time = drop_time
    idle_time      = waiting_time + pickup_time

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

    return reward
```

Cab Driver DQN Agent: -



(a) Illustration of DRL (b) Illustration of DNN

In Agent class we need to work on below functions are

- Assigning hyperparameters
- Creating a neural-network model.
- Define epsilon-greedy strategy.
- Appends the recent experience state, action, reward, new state to the memory.
- Build the DQN model using the updated input and output batch.

Hyperparameters: -

We can tweak these parameters for better performance.

```
# Write here: Specify you hyper parameters for the DQN
self.discount_factor = 0.95
self.learning_rate = 0.01
self.epsilon = 1
self.epsilon_max = 1
self.epsilon_decay = -0.0005
self.epsilon_min = 0.00001
```

Neural Network Model: -

```
def build_model(self):
    input_shape = self.state_size
    model = Sequential()
    # Write your code here: Add layers to your neural nets
    model.add(Dense(32, input_dim=self.state_size, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(32, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(self.action_size, activation='relu', kernel_initializer='he_uniform'))
    model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
    model.summary
    return model
```

Using keras we build a sequential model by adding dense layers.

We have provided state as input at the first layer with relu nonlinear activation function and then added hidden layers for better learning with relu activation function.

Here, we are using Adam optimizer which uses epsilon greedy policy and learning rate to improve the weights and bias to minimize mean square error.

Build the DQN model: -

Appends the recent experience state, action, reward, new state to the memory with updated input and output batch.

```
for episode in range(total_episodes):

    terminal_state = False
    score = 0
    track_reward = False

    # reset at the start of each episode
    env = CabDriver()
    action_space, state_space, state = env.reset()
    # Saving initial state
    initial_state = env.state_init

    # Total time
    total_time = 0
    while not terminal_state:
        # Getting requests
        possible_actions_indices, actions = env.requests(state)
        # 2. Pick epsilon-greedy action from possible actions for the current state.
        # Getting possible actions
        action = agent.get_action(state, possible_actions_indices, actions)

        # 3. Evaluate your reward and next state
        reward, next_state, step_time = env.step(state, env.action_space[action], Time_matrix)
        # Calculating total time
        total_time += step_time
        if (total_time > episode_time):
            terminal_state = True
        else:
            # 5. Append the experience to the memory
```

Evaluation : -

Model continuously update strategies to learn a strategy that maximizes long-term cumulative rewards.

Below two are the performance matrices for our model.

- Q-Value convergence.
- Rewards per episode.

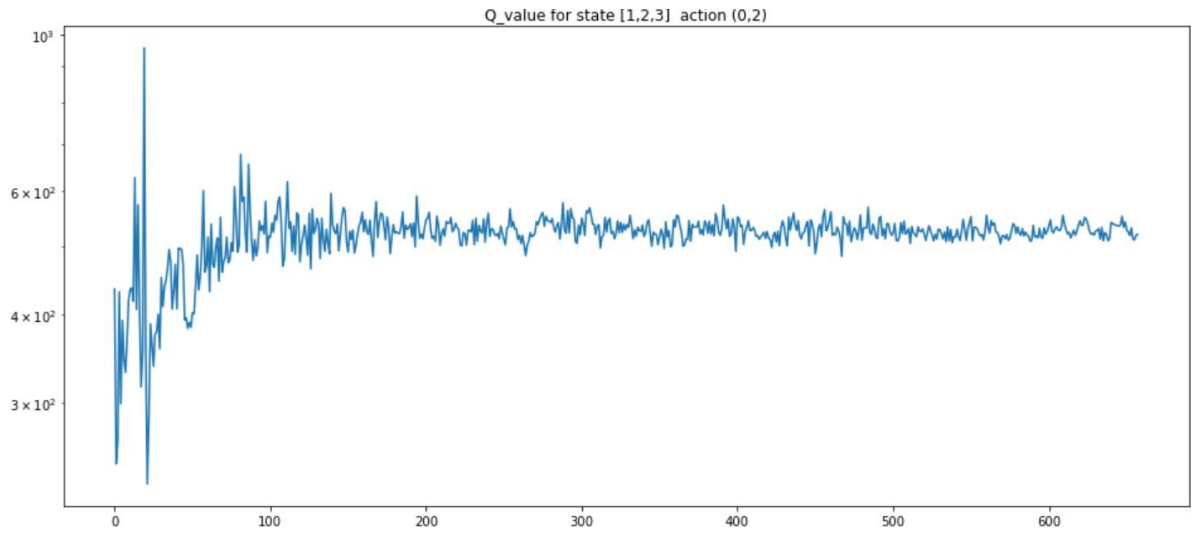
At initial stage, rewards are not better since our agent didn't learn since it's have less experience.

```
episode 0, reward -315.0, memory_length 130, epsilon 0.99999 total_time 729.0
Saving Model 0
episode 1, reward -369.0, memory_length 274, epsilon 0.9994901299779195 total_time 722.0
episode 2, reward 51.0, memory_length 425, epsilon 0.9989905098283768 total_time 730.0
episode 3, reward -296.0, memory_length 551, epsilon 0.9984911394264665 total_time 722.0
episode 4, reward -539.0, memory_length 689, epsilon 0.9979920186473464 total_time 721.0
episode 5, reward -72.0, memory_length 835, epsilon 0.9974931473662362 total_time 725.0
episode 6, reward -211.0, memory_length 972, epsilon 0.996994525458418 total_time 725.0
episode 7, reward -216.0, memory_length 1119, epsilon 0.9964961527992363 total_time 733.0
episode 8, reward -450.0, memory_length 1262, epsilon 0.9959980292640981 total_time 730.0
episode 9, reward 33.0, memory_length 1410, epsilon 0.9955001547284723 total_time 723.0
episode 10, reward -184.0, memory_length 1531, epsilon 0.9950025290678904 total_time 732.0
```

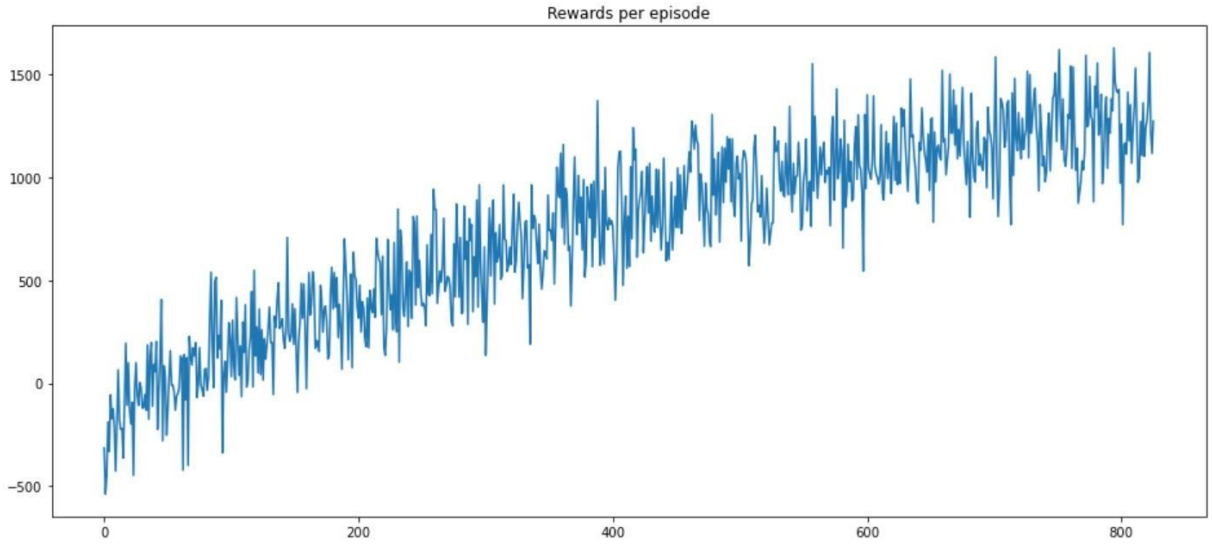
After few episodes, our agent learns to choose best request with an experience and provides better rewards.

```
episode 1738, reward 792.0, memory_length 2000, epsilon 0.41936651663540053 total_time 726.0
episode 1739, reward 732.0, memory_length 2000, epsilon 0.4191568857891617 total_time 722.0
episode 1740, reward 757.0, memory_length 2000, epsilon 0.41894735973214653 total_time 722.0
episode 1741, reward 766.0, memory_length 2000, epsilon 0.41873793841197343 total_time 726.0
episode 1742, reward 1114.0, memory_length 2000, epsilon 0.41852862177628714 total_time 721.0
episode 1743, reward 819.0, memory_length 2000, epsilon 0.41831940977275844 total_time 723.0
episode 1744, reward 1037.0, memory_length 2000, epsilon 0.4181103023490844 total_time 726.0
episode 1745, reward 902.0, memory_length 2000, epsilon 0.4179012994529881 total_time 726.0
episode 1746, reward 1054.0, memory_length 2000, epsilon 0.41769240103221883 total_time 731.0
episode 1747, reward 152.0, memory_length 2000, epsilon 0.417483607034552 total_time 722.0
episode 1748, reward 803.0, memory_length 2000, epsilon 0.41727491740778916 total_time 725.0
episode 1749, reward 863.0, memory_length 2000, epsilon 0.4170663320997578 total_time 723.0
episode 1750, reward 775.0, memory_length 2000, epsilon 0.41685785105831163 total_time 722.0
episode 1751, reward 1127.0, memory_length 2000, epsilon 0.41664947423133036 total_time 723.0
```

Q- Value Convergence



Rewards per episode:-



Conclusion: -

We have modeled a RL-Based system agent using Deep Q-Learning Network, which can help cab drivers to maximize their profits by improving agent's decision-making process. We have plotted Q-value convergence and rewards per episode to understand the model performance. As we increase the number of rides in the system, the collected reward increases.

References: -

https://arxiv.org/abs/2104.12000

https://arxiv.org/abs/2104.12226

https://towardsdatascience.com/about-reinforcement-learning-2ff0dafe9b75

https://www.kaggle.com/super-cabs-assignment

https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

https://towardsdatascience.com/the-complete-reinforcement-learning-dictionary-e16230b7d24e