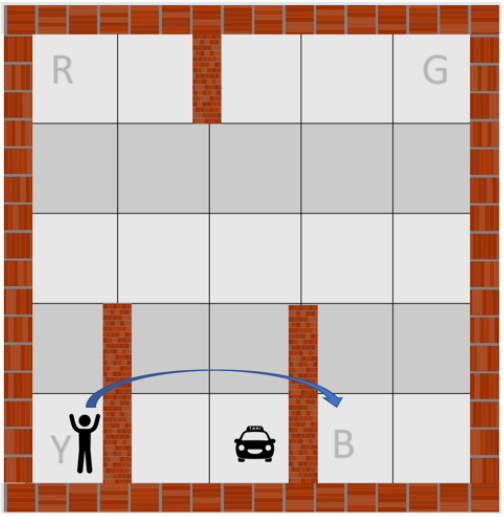
-> CAB DRIVER AGENT – INTRODUCTION TO THE ENVIRONMENT.



We are provided with the following information to create the environment for out agent to exploere :

State of the agent must be one hot encoded vector of size "**m + t + d",** where '**m'** is the number of cities the agent can be in, **'t'** the total number of hours that is 24 and '**d'** denotes the day of month.

Following the way we create the one hot encoded version for every state.

```
state_encod = [0 for _ in range(m+t+d)]
#city
state_encod[state[0]] = 1
#time
state_encod[m+state[1]] = 1
#day
state_encod[m+t+state[2]] = 1
return state_encod
```

So there can be m * t * d number of states in total and here's how we define the state and action space for our environment.

```
#initialize the total action space
self.action_space = [(0,0)] + list(permutations([i for i in range(m)] ,2))
# initialize the stte space also
self.state_space = [[city, time, day] for city in range(m) for time in range(t) for day in range(d)]
# get he initial state
self.state_init = random.choice(self.state_space)
```

The first action which [0, 0] denotes that the agent refuses to take the ride request. Each action is a 2 dimensional vector, of which first index is the pickup locaiton and the second index has the drop location. Time to reach from location to another location is discribed in the TM.pkl file provided to us.

So That was all the information about the environment we are provided with, next we wil discuss some of the functions that are to be defined for out agent to interact with and explore the environment.

Our agent is a cab driver so it needs all the ride requests it has at the current state. Following is the function that will return the number of requests and possible actions that the agent can take once given the requests.

```
def get_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
    #randomly initilize the number of reqeusts for each city.
    location = state[0]
    requests = None
    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)
    #max number of requests cannot exceed 15
    if requests > 15:
        requests = 15
    #get all possible actions indexes
    possible_actions_index = random.sample(range(1, (m-1)*m + 1), requests) +

    actions = [self.action_space[i] for i in possible_actions_index]

    return possible_actions_index,actions
```

Now upon arrival of a request one of the three things can happen

1) **Agent is at the pickup location** and has no need to traval to the pickup location, we only need to current time to reflect the ride time and there will be no wait time, current location of agent will eventually be updated to the drop location.

2) **Agent has decided to reject the ride request** here wait time will be increased by one and agent current location will remain the same.

3) **Agent is not at the pickup location** here agent will have to travel from current_location to pickup_location so current time will be udpated first with this travel_time and current_location will be updated to pickup locaiton will eventually be updated to drop locaiton as agent has decided to accept the ride. Reach time will be updated to the travel time of agent from current_location to pickup location that will cause reward to drop a bit, as the total time will be increased so the cost penality per unit time will be.

Following is the reward function that will give reward informaion to agent provided the wait_time, reach_time and ride_time.

```
#return the reward
def reward_func(self, khali_time, reach_time, ride_time):
    usefull_time = ride_time
    khali_time += reach_time
    total_time = khali_time + usefull_time;
    #use full time * reward - total time * Cost penalty
    reward = (R * usefull_time) - (C * (total_time))
    return reward
```

khali_time is the wait_time, **reach_time** is the time agent took to reach from the current_location to the pickup_location which will be added to khali_time to get overall wait time and usefull time will be calculated after subtracting the wait time from total time.

Reward formula is reward = (R * usefull_time) – (C * total_time).

So that’s everything we need in our ENV.py file, lets train the model and analyze the results.

To train our model we will be using the Replay buffer which will store experiances. And Deep learning model that we will be using is shown below.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense (Dense)                (None, 32)                1184

 dense_1 (Dense)              (None, 32)                1056

 dense_2 (Dense)              (None, 21)                693

=================================================================
Total params: 2,933
Trainable params: 2,933
Non-trainable params: 0
_________________________________________________________________
```

Following is the DQN algorithm that we have implemented.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

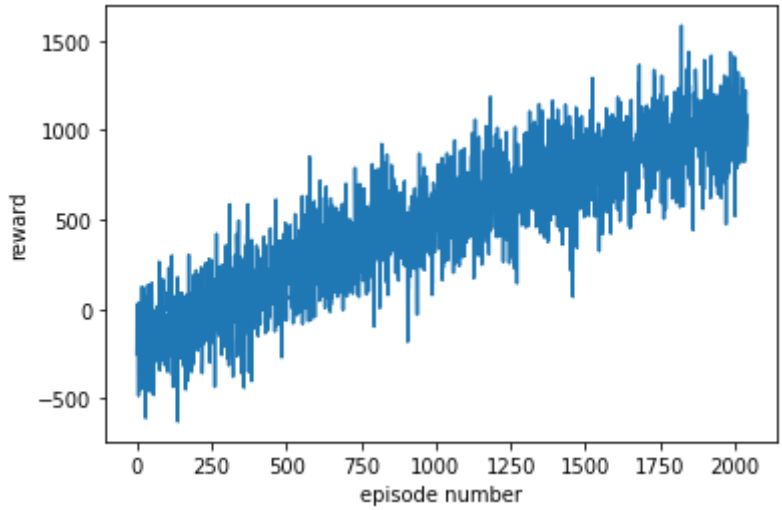
We first define some hyperparameters starting from the epsilon, discount_factor etc.

We will interact with the environment and record the state, action, reward and next_states in the replay buffer.

Once the size of replay buffer gets above 2000 we will sample the batch of train exmples of size 32 and make an array update_input that will store the one hot encoded sates that corresponds to the mini batch samles and will have the
size = 32 + m*t*d. Same we will create an array update_target that will store one hot encoded next_states corresponding to the mini batch. We will predict the q_value estimates for both tht update_input and update_output and make the supervised train samples using the following code.

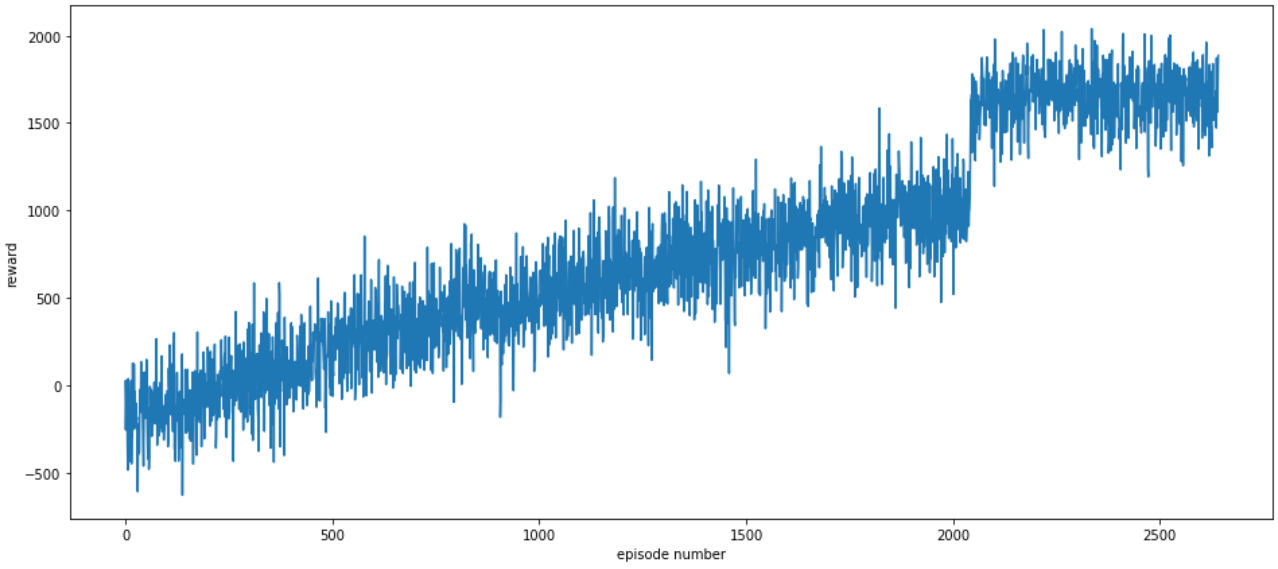
```
#predict the qvalue estimates for input
target = model.predict(update_input)
#prdict the target qval estimates for the ouptut array
target_qval = model.predict(update_output)

for i in range(self.batch_size):
    if done[i]:
        target[i][actions[i]] = rewards[i]
    else:
        #here sum up the current reward with the Qvalue predictions
        target[i][actions[i]] = rewards[i] + self.discount_facter * np.max(target_qval[i])
#fit the model with the calculated mini batch and target outputs!
model.fit(update_input, target, batch_size = self.batch_size, epochs = 1, verbose = 0)
```

Training the DQN for 2000 episodes gave us the following reward graph.

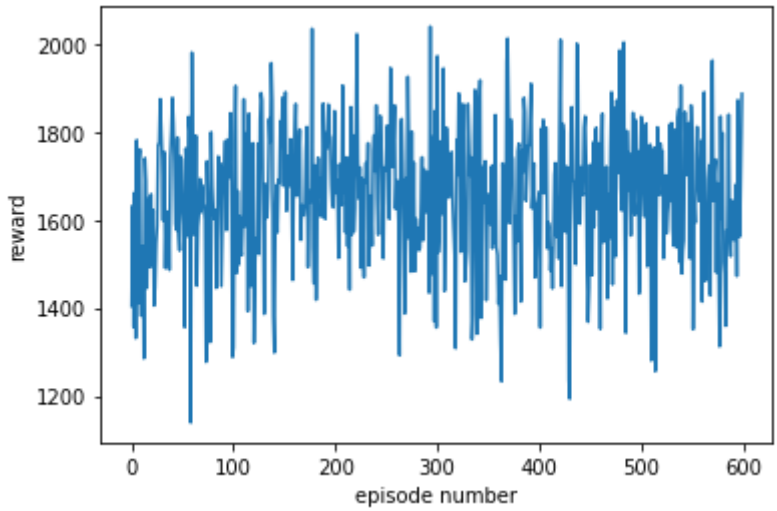


You can see that the agent has been learning and more learning is possible as the epsilon decreases as each time stamp progresses, we wil get more stable rewards. And the jitterness in rewards is because of the exploration that the agent does with epsolon greedy policy.

I trained the model more longer and longer to 5500 episodes and combined the final reward graph with the current 2000 episode reward graph following was the converged agent rewards graph looked like.



If we Zoom in the 4900 – 5500 episodes rewards we get the following graph.



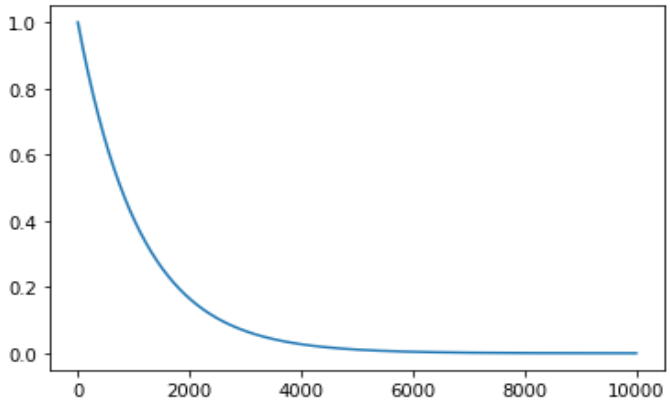
You can see that the agent converged with the reards having a common mean of 1600 approx..

Epilon Decay funcition.

Following code and graph shows how epsilon has been decaying as the learning progresses.

In [18]:
```
time = np.arange(0,10000)
epsilon = []
for i in range(0,10000):
    epsilon.append(0 + (1 - 0) * np.exp(-0.0009*i))
```

In [19]:
```
plt.plot(time, epsilon)
plt.show()
```



Thank You for going thourgh the complete report.

Prepared By : MT2021037 - DEEPAK NANDWANI
Mtech. CSE 23’