OpenAI –Taxi v3 Environment

A Graduate Student Project

Lamar University, Computer Science Department, Artificial Intelligence (CPSC 5370)

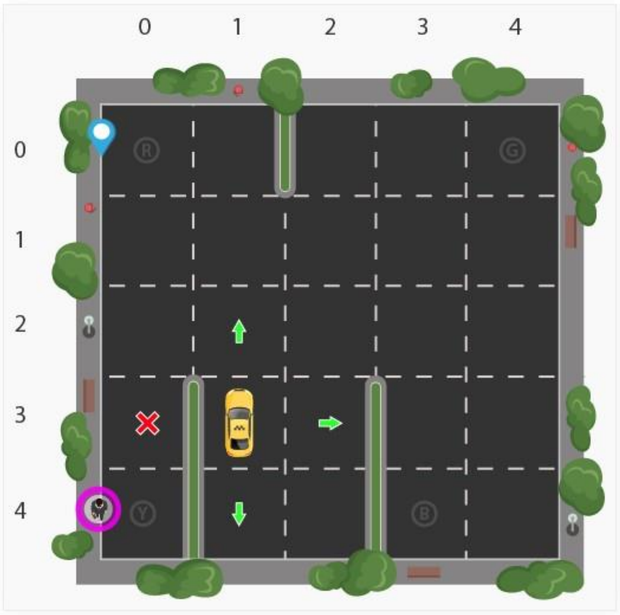
Group Members: Samuel E. Casco, Prashant R. Ghimire, Rezwanul Islam Rajib, Ashish Joshi

Instructor: Dr. Peggy Doerschuk

Background: The environment we are going to consider for this project is that of the self-driving taxi, utilized by the authors Satwik Kansal and Brendan Martin. We are recreating the experiment to test the performance in two scenarios: without reinforcement learning and then with reinforcement learning (Q-Learning Algorithm). In this experiment, the goal of the taxi is to pick up the passenger at one location and then drop them off in another location. The taxi needs to drop off the passenger at the correct location and should do so in minimal time.

Now that we have the background, the next step is to formulate the problem to come up with the States, the Actions, and the possible Rewards, as these are very important for our Q-Learning Algorithm. Mentioned below are the information selected by the authors for this project.

1. STATES: The State Space, for this example, is the set of all the possible situations (states) that our taxi could inhabit. In this example, we have considered the environment to be a 5x5 grid, with the rows and columns going from 0 to 4. The environment also consists of 4 different drop-off locations denoted by R, G, Y, and B.



Now, an assumption that has been made here is that the taxi is the only vehicle in this environment. So looking at the structure above, we have 5 x 5 = 25 possible taxi locations; this becomes part of our State Space.

Also, we have four passenger locations: R(ed), G(reen), Y(ellow), B(lue). If the passenger is inside the taxi then that is also considered a passenger location. Thus, we have a total of 5 possible passenger locations, represented as follows:

0 – R(ed)

1 – G(reen)

2 – Y(ellow)

3 – B(lue)

4 – in taxi

Similarly, we have four destination locations as well, also represented in a similar manner as above:

0 – R(ed)

1 – G(reen)

2 – Y(ellow)

3 – B(lue)

Now, all these possible locations are combined since the taxi location, the passenger location, and the destination location are independent of each other. So, we have a total of: 5 x 5 x 5 x 4 = 500 possible states.

<u>2. ACTIONS</u>: The next thing that is important for reinforcement learning, and especially for Q-Learning, is to figure out the actions that the agent can possibly take. The actions mentioned in this example are as follows:

1. south
2. north
3. east
4. west
5. pickup
6. drop-off

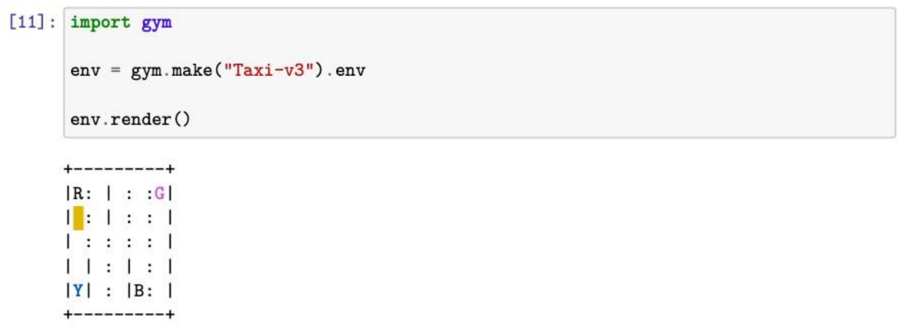
Now, in the example, certain actions cannot be taken at certain states due to the presence of a wall. For such actions, a simple -1 penalty is provided to the taxi and its actions are halted. Enough penalties will cause the taxi to move around the wall.

3. REWARDS: Perhaps the most important aspect of reinforcement learning is the rewards, because based on the rewards the agents are expected to prefer or avoid certain actions. The authors for this example have considered the following information while trying to decide the rewards:

- A high positive reward is given to the agent for a successful drop-off so that this behavior is more likely to be repeated.
- A penalization is given to the agent if the drop-off for the passenger is at a wrong location.
- A slight negative reward is given for every time-step until the agent reaches its desired location. The negative reward is only slight because it is preferred that the agent reaches the location late, but makes the correct moves as opposed to the agent making wrong moves to reach to the destination as fast as possible.

4. EXPERIMENTATION: The experiment for this environment is run in two different scenarios: 1) Without using Reinforcement Learning – the taxi keeps on making moves till it takes the passenger to the correct drop-off. 2) Using Reinforcement Learning (Q-Learning) – the taxi utilizes the Q-table to perform an action which would generate the desired output.

1. Without using Reinforcement Learning



```
[11]: import gym

      env = gym.make("Taxi-v3").env

      env.render()
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

→ We simply import the ***gym*** toolkit for our experiment and set the environment as shown.

→ ***env.render()*** helps us to visualize the problem and makes it easier to understand the states.

```python
env.reset() # reset environment to a new, random state
env.render()

print("Action Space {}".format(env.action_space))
print("State Space {}".format(env.observation_space))
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+

Action Space Discrete(6)
State Space Discrete(500)
```

→ We reset the environment to a new, random state and render that environment.

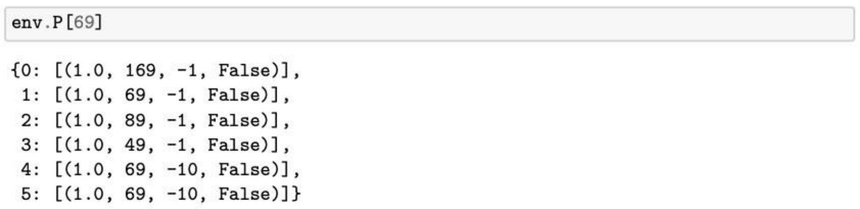
→ The output with ***action_space*** and ***observation_space*** helps confirm the total number of actions and states respectively.

```python
state = env.encode(0, 3, 2, 1) # (taxi row, taxi column, passenger index,␣
 ↪destination index)
print("State:", state)

env.s = state
env.render()
```

```
State: 69
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

→ We can pass the position of the taxi and the index locations of the pickup and drop-off utilizing the visualization of the environment. Based on these values, we get the current state.

→ So based on these current positions of all of these, the current state is 69.

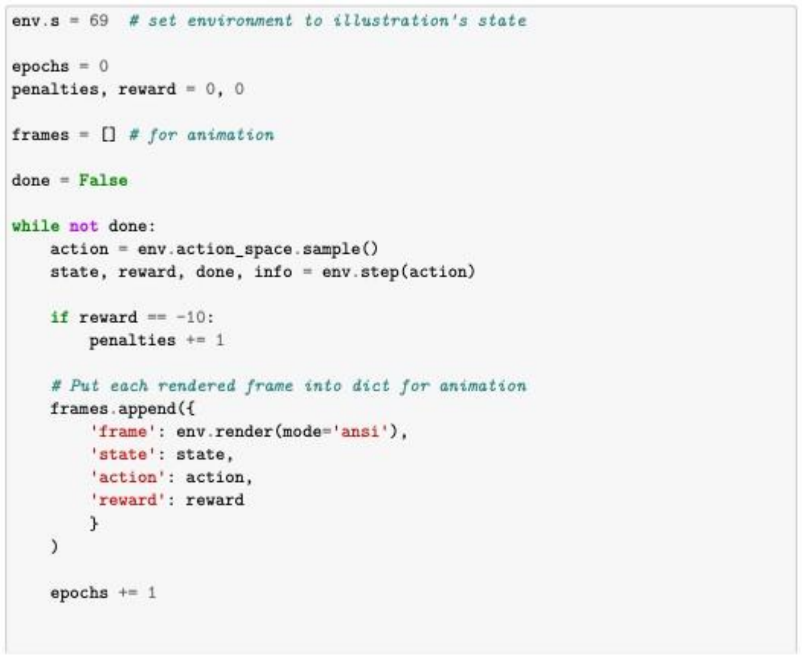


```
env.P[69]
```

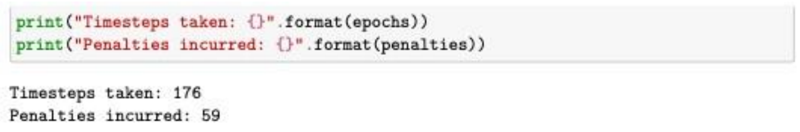
```
{0: [(1.0, 169, -1, False)],
 1: [(1.0, 69, -1, False)],
 2: [(1.0, 89, -1, False)],
 3: [(1.0, 49, -1, False)],
 4: [(1.0, 69, -10, False)],
 5: [(1.0, 69, -10, False)]}
```

→ We can utilize the ***env.P[69]*** command to display the default rewards associated with our current state.

→ Here, the above information can be generalized as – ***{action: [(probability, nextstate, reward, done)]}***. The probability for all the actions is 1.



```python
env.s = 69  # set environment to illustration's state

epochs = 0
penalties, reward = 0, 0

frames = [] # for animation

done = False

while not done:
    action = env.action_space.sample()
    state, reward, done, info = env.step(action)

    if reward == -10:
        penalties += 1

    # Put each rendered frame into dict for animation
    frames.append({
        'frame': env.render(mode='ansi'),
        'state': state,
        'action': action,
        'reward': reward
        }
    )

    epochs += 1
```



```python
print("Timesteps taken: {}".format(epochs))
print("Penalties incurred: {}".format(penalties))
```

```
Timesteps taken: 176
Penalties incurred: 59
```

→ Now we run this program, in sort of a brute-force method, until the agent (taxi) performs one successful pickup and drop-off.

→ We can see that the taxi takes 176 timesteps and in that time, incurs the penalty a total of 59 times, which means it performed a total of 59 incorrect pickup/drop-off.

2. With using Reinforcement Learning (Q-Learning)

Now, to start off with Q-Learning, most of the steps involved are the same, such as loading the environment, rendering it and assigning the state information. Once we are done with that, we start with reinforcement learning by first loading the Q-table.

```
qtable = np.zeros((state_size, action_size))
print(qtable)
```

```
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 ...
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]]
```

Scenario 1-1:

- Training the agent

Learning Rate = 0.1
Gamma = 0.25

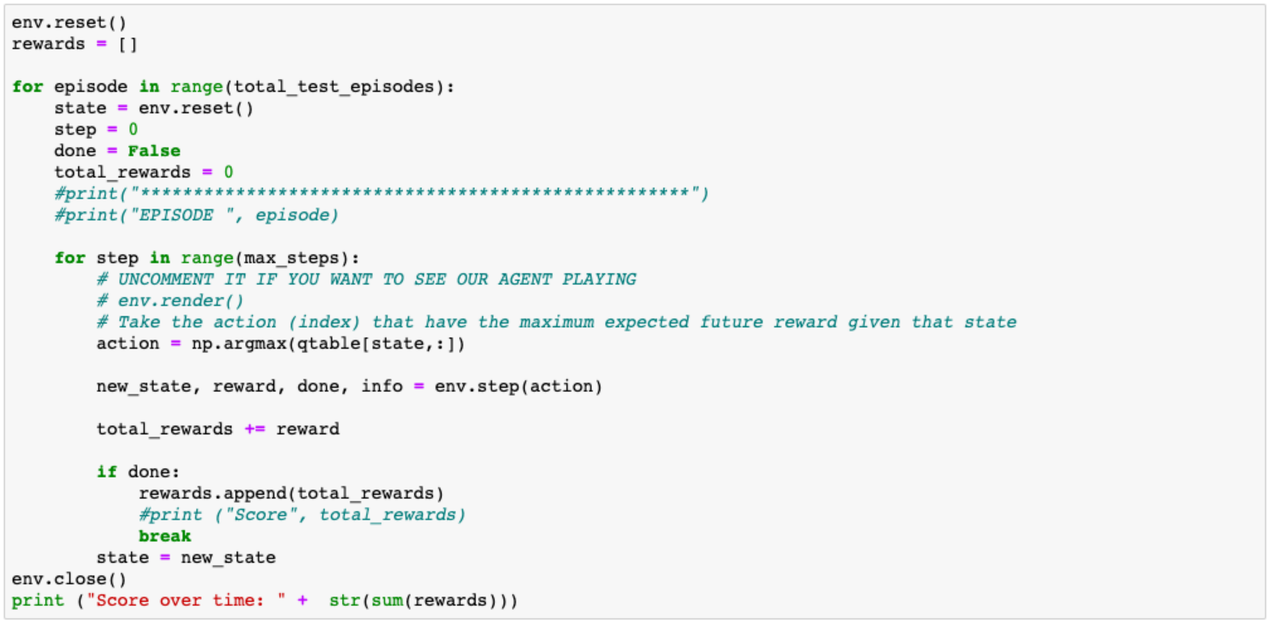
```
total_episodes = 50000        # Total episodes
total_test_episodes = 100     # Total test episodes
max_steps = 99                # Max steps per episode

learning_rate = 0.1           # Learning rate
gamma = 0.25                  # Discounting rate

# Exploration parameters
epsilon = 1.0                 # Exploration rate
max_epsilon = 1.0             # Exploration probability at start
min_epsilon = 0.01            # Minimum exploration probability
decay_rate = 0.01             # Exponential decay rate for exploration prob
```

→ All the parameters are kept the same except for two things: the *learning rate* and the value of *gamma*, which is the *discounting rate*.
→ For our first-first scenario, we use the value of *learning rate* as 0.1 and the value of *discount rate* as 0.25.

```
# 2 For life or until learning is stopped
for episode in range(total_episodes):
    # Reset the environment
    state = env.reset()
    step = 0
    done = False

    for step in range(max_steps):
        # 3. Choose an action a in the current world state (s)
        ## First we randomize a number
        exp_exp_tradeoff = random.uniform(0,1)

        ## If this number > greater than epsilon --> exploitation (taking the biggest Q value for this state)
        if exp_exp_tradeoff > epsilon:
            action = np.argmax(qtable[state,:])

        # Else doing a random choice --> exploration
        else:
            action = env.action_space.sample()

        # Take the action (a) and observe the outcome state(s') and reward (r)
        new_state, reward, done, info = env.step(action)

        # Update Q(s,a):= Q(s,a) + lr [R(s,a) + gamma * max Q(s',a') - Q(s,a)]
        qtable[state, action] = qtable[state, action] + learning_rate * (reward + gamma *
                                    np.max(qtable[new_state, :]) - qtable[state, action])

        # Our new state is state
        state = new_state

        # If done : finish episode
        if done == True:
            break

    # Reduce epsilon (because we need less and less exploration)
    epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*episode)
```

→ We train the agent by having the agent follow the Epsilon-Greedy Policy. The agent follows exploration when a randomly generated value is less than the epsilon value and follows exploitation otherwise.

- Evaluating the agent



```
env.reset()
rewards = []

for episode in range(total_test_episodes):
    state = env.reset()
    step = 0
    done = False
    total_rewards = 0
    #print("****************************************************")
    #print("EPISODE ", episode)

    for step in range(max_steps):
        # UNCOMMENT IT IF YOU WANT TO SEE OUR AGENT PLAYING
        # env.render()
        # Take the action (index) that have the maximum expected future reward given that state
        action = np.argmax(qtable[state,:])

        new_state, reward, done, info = env.step(action)

        total_rewards += reward

        if done:
            rewards.append(total_rewards)
            #print ("Score", total_rewards)
            break
        state = new_state
env.close()
print ("Score over time: " +  str(sum(rewards)))
```

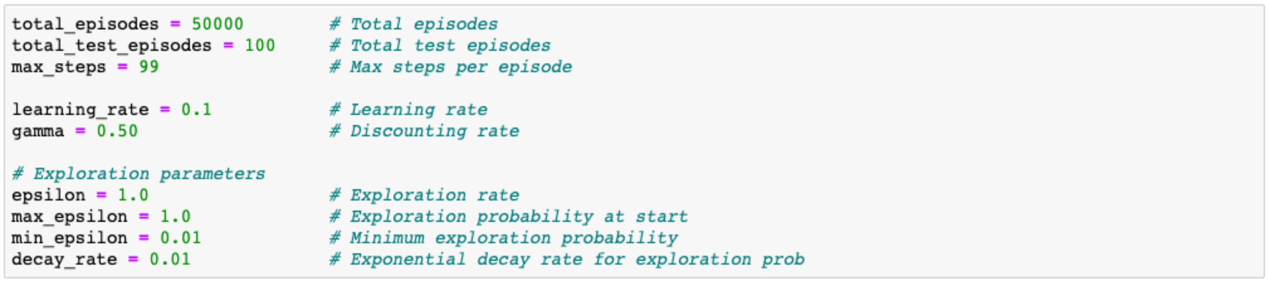
```
Score over time: 315
```

→ After the training, we evaluate our agent's performance over 100 episodes.
→ As we can see, the sum of the *rewards* received by our agent over a total episode of 100 is equal to 315.

Scenario 1-2:

- Training the agent

Learning Rate = 0.1
Gamma = 0.50



```
total_episodes = 50000        # Total episodes
total_test_episodes = 100     # Total test episodes
max_steps = 99                # Max steps per episode

learning_rate = 0.1           # Learning rate
gamma = 0.50                  # Discounting rate

# Exploration parameters
epsilon = 1.0                 # Exploration rate
max_epsilon = 1.0             # Exploration probability at start
min_epsilon = 0.01            # Minimum exploration probability
decay_rate = 0.01             # Exponential decay rate for exploration prob
```

→ All the parameters are kept the same except for two things: the *learning rate* and the value of *gamma*, which is the *discounting rate*.
→ For our first-second scenario, we use the same value of *learning rate* as 0.1 and change the value of *discount rate* to 0.50.

→ We train the agent again by having the agent follow the Epsilon-Greedy Policy. The agent follows exploration when a randomly generated value is less than the epsilon value and follows exploitation otherwise.

- Evaluating the agent

```
env.reset()
rewards = []

for episode in range(total_test_episodes):
    state = env.reset()
    step = 0
    done = False
    total_rewards = 0
    #print("****************************************************")
    #print("EPISODE ", episode)

    for step in range(max_steps):
        # UNCOMMENT IT IF YOU WANT TO SEE OUR AGENT PLAYING
        # env.render()
        # Take the action (index) that have the maximum expected future reward given that state
        action = np.argmax(qtable[state,:])

        new_state, reward, done, info = env.step(action)

        total_rewards += reward

        if done:
            rewards.append(total_rewards)
            #print ("Score", total_rewards)
            break
        state = new_state
env.close()
print ("Score over time: " +  str(sum(rewards)))
```

```
Score over time: 739
```

→ After the training, we evaluate our agent's performance over 100 episodes.
→ As we can see, the sum of the *rewards* received by our agent over a total episode of 100 is equal to 739.

<u>Scenario 1-3</u>:

- Training the agent

Learning Rate = 0.1
Gamma = 0.75

```
total_episodes = 50000        # Total episodes
total_test_episodes = 100     # Total test episodes
max_steps = 99                # Max steps per episode

learning_rate = 0.1           # Learning rate
gamma = 0.75                  # Discounting rate

# Exploration parameters
epsilon = 1.0                 # Exploration rate
max_epsilon = 1.0             # Exploration probability at start
min_epsilon = 0.01            # Minimum exploration probability
decay_rate = 0.01             # Exponential decay rate for exploration prob
```

→ For our first-third scenario, we use the same value of *learning rate* as 0.1 and change the value of *discount rate* to 0.75.
→ We train the agent again following the same policy as mentioned before.

- Evaluating the agent

```
env.reset()
rewards = []

for episode in range(total_test_episodes):
    state = env.reset()
    step = 0
    done = False
    total_rewards = 0
    #print("****************************************************")
    #print("EPISODE ", episode)

    for step in range(max_steps):
        # UNCOMMENT IT IF YOU WANT TO SEE OUR AGENT PLAYING
        # env.render()
        # Take the action (index) that have the maximum expected future reward given that state
        action = np.argmax(qtable[state,:])

        new_state, reward, done, info = env.step(action)

        total_rewards += reward

        if done:
            rewards.append(total_rewards)
            #print ("Score", total_rewards)
            break
        state = new_state
env.close()
print ("Score over time: " +  str(sum(rewards)))
```

```
Score over time: 853
```

→ After the training, we evaluate our agent's performance over 100 episodes.
→ As we can see, the sum of the *rewards* received by our agent over a total episode of 100 is equal to 853.

<u>**Scenario 2-1**</u>:

- Training the agent

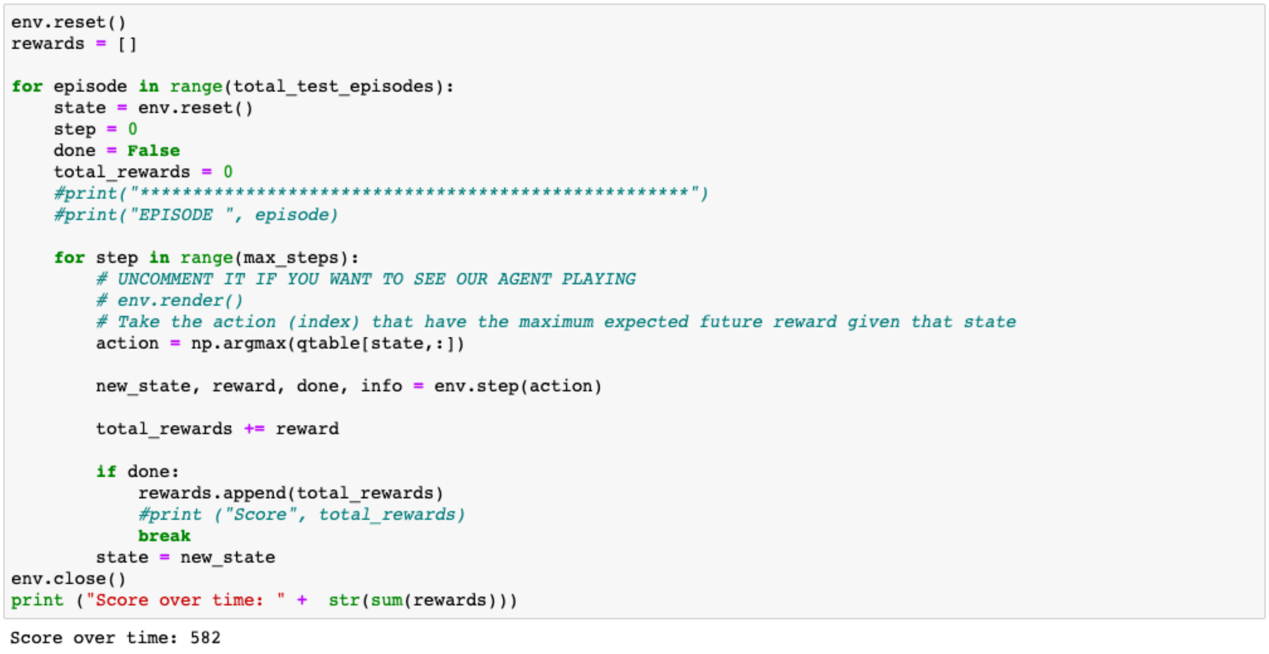
Learning Rate = 0.5
Gamma = 0.25

```
total_episodes = 50000        # Total episodes
total_test_episodes = 100     # Total test episodes
max_steps = 99                # Max steps per episode

learning_rate = 0.5           # Learning rate
gamma = 0.25                  # Discounting rate

# Exploration parameters
epsilon = 1.0                 # Exploration rate
max_epsilon = 1.0             # Exploration probability at start
min_epsilon = 0.01            # Minimum exploration probability
decay_rate = 0.01             # Exponential decay rate for exploration prob
```

→ All the parameters are kept the same except for two things: the *learning rate* and the value of *gamma*, which is the *discounting rate*.
→ For our second-first scenario, we use the value of *learning rate* as 0.5 and the value of *discount rate* as 0.25.

→ We train the agent by having the agent follow the Epsilon-Greedy Policy. The agent follows exploration when a randomly generated value is less than the epsilon value and follows exploitation otherwise.

- Evaluating the agent



```
env.reset()
rewards = []

for episode in range(total_test_episodes):
    state = env.reset()
    step = 0
    done = False
    total_rewards = 0
    #print("****************************************************")
    #print("EPISODE ", episode)

    for step in range(max_steps):
        # UNCOMMENT IT IF YOU WANT TO SEE OUR AGENT PLAYING
        # env.render()
        # Take the action (index) that have the maximum expected future reward given that state
        action = np.argmax(qtable[state,:])

        new_state, reward, done, info = env.step(action)

        total_rewards += reward

        if done:
            rewards.append(total_rewards)
            #print ("Score", total_rewards)
            break
        state = new_state
env.close()
print ("Score over time: " +  str(sum(rewards)))
```

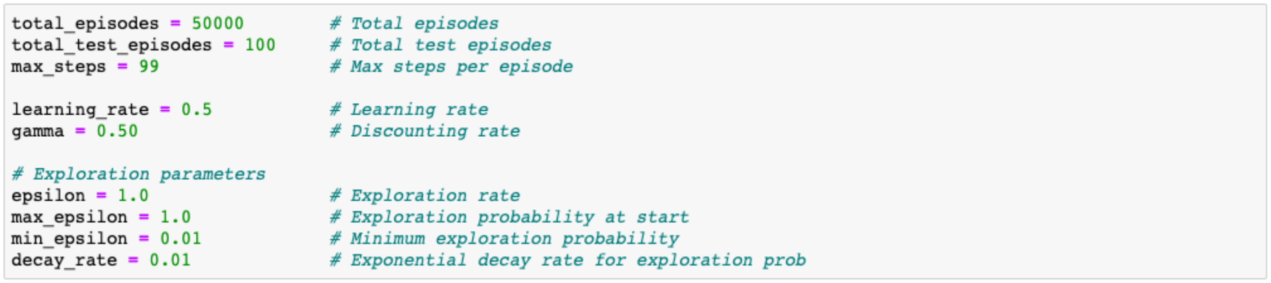
```
Score over time: 582
```

→ After the training, we evaluate our agent's performance over 100 episodes.
→ As we can see, the sum of the *rewards* received by our agent over a total episode of 100 is equal to 582.

<u>Scenario 2-2</u>:

- Training the agent

Learning Rate = 0.5
Gamma = 0.50

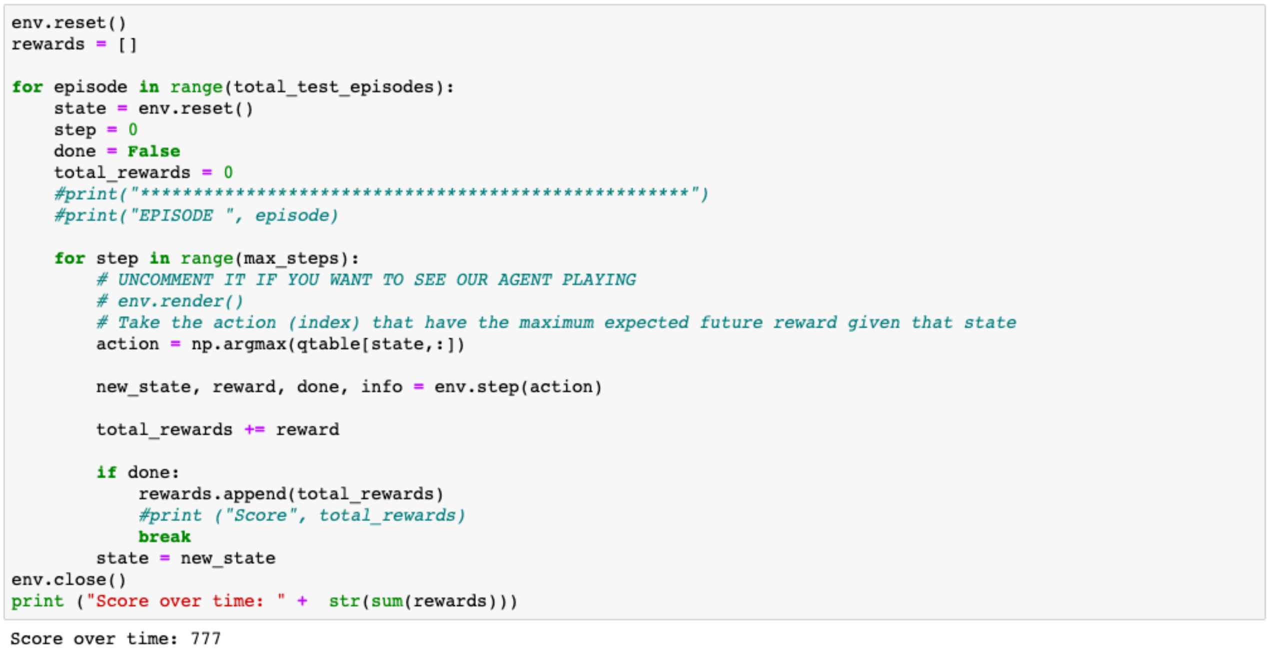


```
total_episodes = 50000        # Total episodes
total_test_episodes = 100     # Total test episodes
max_steps = 99                # Max steps per episode

learning_rate = 0.5           # Learning rate
gamma = 0.50                  # Discounting rate

# Exploration parameters
epsilon = 1.0                 # Exploration rate
max_epsilon = 1.0             # Exploration probability at start
min_epsilon = 0.01            # Minimum exploration probability
decay_rate = 0.01             # Exponential decay rate for exploration prob
```

→ All the parameters are kept the same except for two things: the *learning rate* and the value of *gamma*, which is the *discounting rate*.
→ For our second-second scenario, we use the same value of *learning rate* as 0.5 and change the value of *discount rate* to 0.50.

→ We train the agent again by having the agent follow the Epsilon-Greedy Policy. The agent follows exploration when a randomly generated value is less than the epsilon value and follows exploitation otherwise.

- Evaluating the agent



```
env.reset()
rewards = []

for episode in range(total_test_episodes):
    state = env.reset()
    step = 0
    done = False
    total_rewards = 0
    #print("****************************************************")
    #print("EPISODE ", episode)

    for step in range(max_steps):
        # UNCOMMENT IT IF YOU WANT TO SEE OUR AGENT PLAYING
        # env.render()
        # Take the action (index) that have the maximum expected future reward given that state
        action = np.argmax(qtable[state,:])

        new_state, reward, done, info = env.step(action)

        total_rewards += reward

        if done:
            rewards.append(total_rewards)
            #print ("Score", total_rewards)
            break
        state = new_state
env.close()
print ("Score over time: " +  str(sum(rewards)))
```

```
Score over time: 777
```

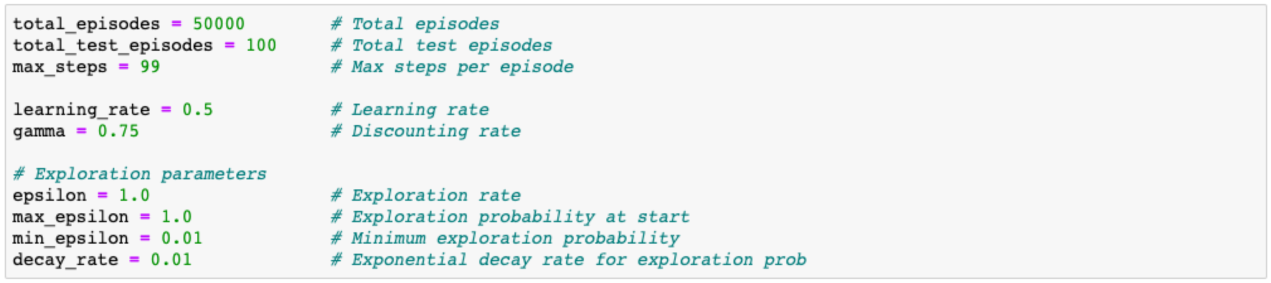
→ After the training, we evaluate our agent's performance over 100 episodes.

→ As we can see, the sum of the *rewards* received by our agent over a total episode of 100 is equal to 777.

<u>Scenario 2-3</u>:

- Training the agent

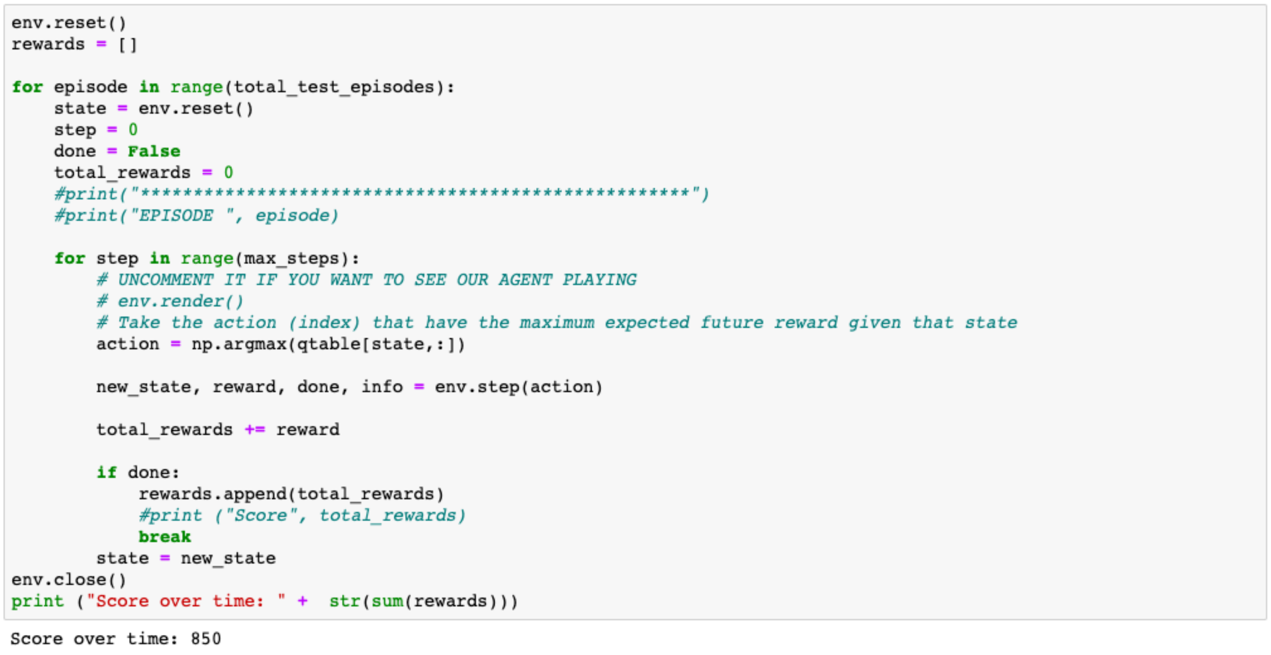
Learning Rate = 0.5
Gamma = 0.75



```
total_episodes = 50000        # Total episodes
total_test_episodes = 100     # Total test episodes
max_steps = 99                # Max steps per episode

learning_rate = 0.5           # Learning rate
gamma = 0.75                  # Discounting rate

# Exploration parameters
epsilon = 1.0                 # Exploration rate
max_epsilon = 1.0             # Exploration probability at start
min_epsilon = 0.01            # Minimum exploration probability
decay_rate = 0.01             # Exponential decay rate for exploration prob
```

→ For our second-third scenario, we use the same value of *learning rate* as 0.5 and change the value of *discount rate* to 0.75.
→ We train the agent again following the same policy as mentioned before.

- Evaluating the agent



```
env.reset()
rewards = []

for episode in range(total_test_episodes):
    state = env.reset()
    step = 0
    done = False
    total_rewards = 0
    #print("****************************************************")
    #print("EPISODE ", episode)

    for step in range(max_steps):
        # UNCOMMENT IT IF YOU WANT TO SEE OUR AGENT PLAYING
        # env.render()
        # Take the action (index) that have the maximum expected future reward given that state
        action = np.argmax(qtable[state,:])

        new_state, reward, done, info = env.step(action)

        total_rewards += reward

        if done:
            rewards.append(total_rewards)
            #print ("Score", total_rewards)
            break
        state = new_state
env.close()
print ("Score over time: " +  str(sum(rewards)))
```

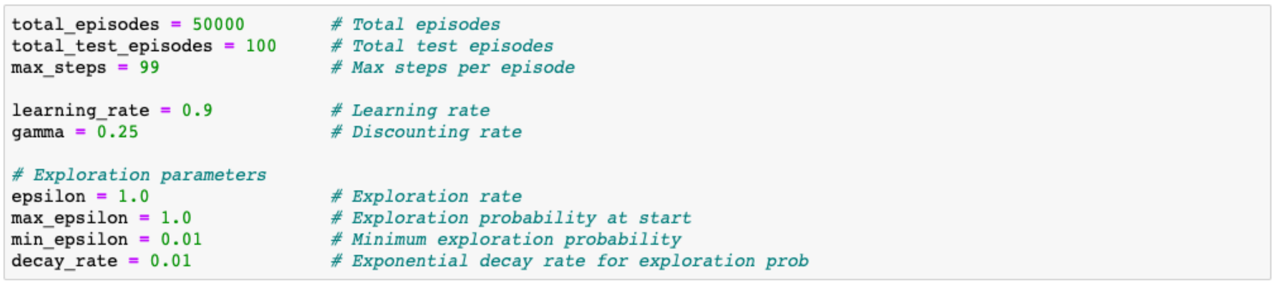
```
Score over time: 850
```

→ After the training, we evaluate our agent's performance over 100 episodes.
→ As we can see, the sum of the *rewards* received by our agent over a total episode of 100 is equal to 850.

<u>Scenario 3-1</u>:

- Training the agent

Learning Rate = 0.9
Gamma = 0.25

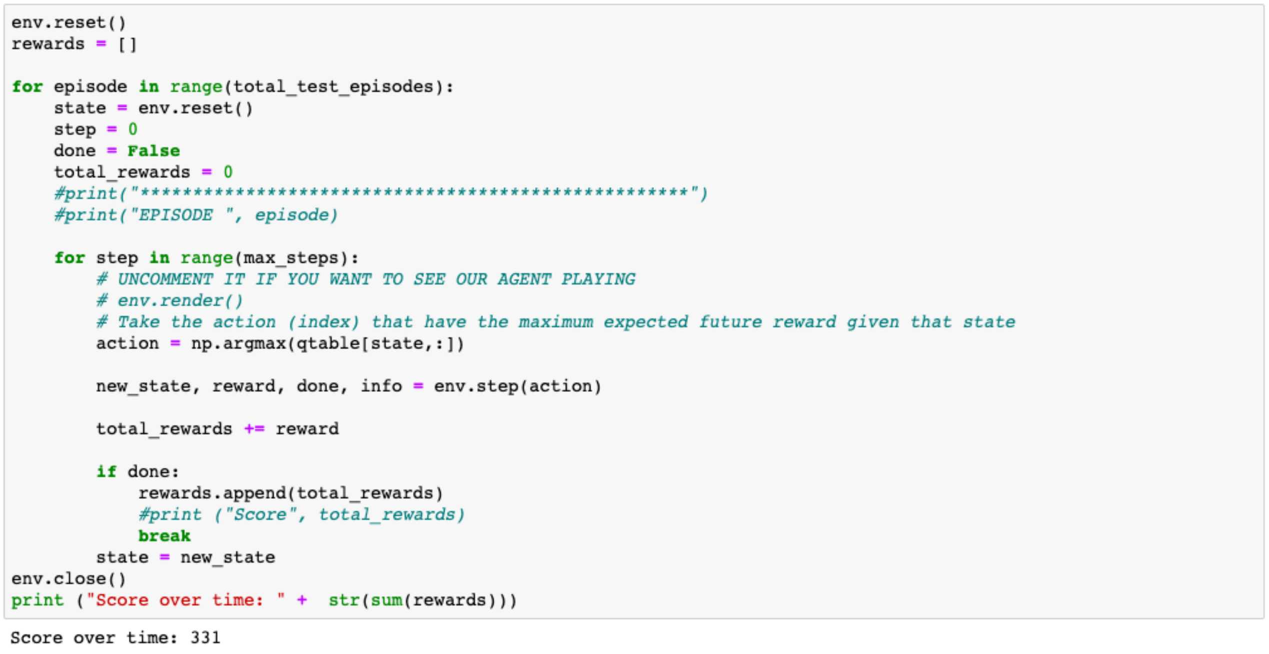


```
total_episodes = 50000        # Total episodes
total_test_episodes = 100     # Total test episodes
max_steps = 99                # Max steps per episode

learning_rate = 0.9           # Learning rate
gamma = 0.25                  # Discounting rate

# Exploration parameters
epsilon = 1.0                 # Exploration rate
max_epsilon = 1.0             # Exploration probability at start
min_epsilon = 0.01            # Minimum exploration probability
decay_rate = 0.01             # Exponential decay rate for exploration prob
```

→ All the parameters are kept the same except for two things: the *learning rate* and the value of *gamma*, which is the *discounting rate*.
→ For our third-first scenario, we use the value of *learning rate* as 0.9 and the value of *discount rate* as 0.25.

→ We train the agent by having the agent follow the Epsilon-Greedy Policy. The agent follows exploration when a randomly generated value is less than the epsilon value and follows exploitation otherwise.

- *Evaluating the agent*



```
env.reset()
rewards = []

for episode in range(total_test_episodes):
    state = env.reset()
    step = 0
    done = False
    total_rewards = 0
    #print("****************************************************")
    #print("EPISODE ", episode)

    for step in range(max_steps):
        # UNCOMMENT IT IF YOU WANT TO SEE OUR AGENT PLAYING
        # env.render()
        # Take the action (index) that have the maximum expected future reward given that state
        action = np.argmax(qtable[state,:])

        new_state, reward, done, info = env.step(action)

        total_rewards += reward

        if done:
            rewards.append(total_rewards)
            #print ("Score", total_rewards)
            break
        state = new_state
env.close()
print ("Score over time: " +  str(sum(rewards)))
```

```
Score over time: 331
```

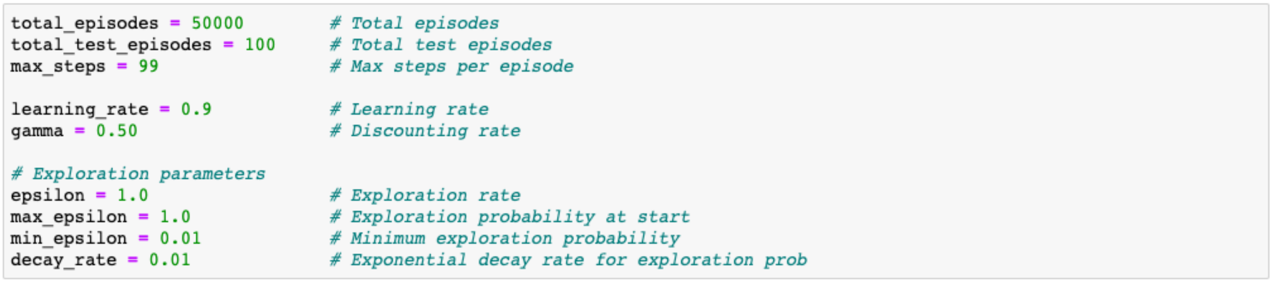
→ After the training, we evaluate our agent's performance over 100 episodes.

→ As we can see, the sum of the *rewards* received by our agent over a total episode of 100 is equal to 331.

Scenario 3-2:

- *Training the agent*

Learning Rate = 0.9
Gamma = 0.50



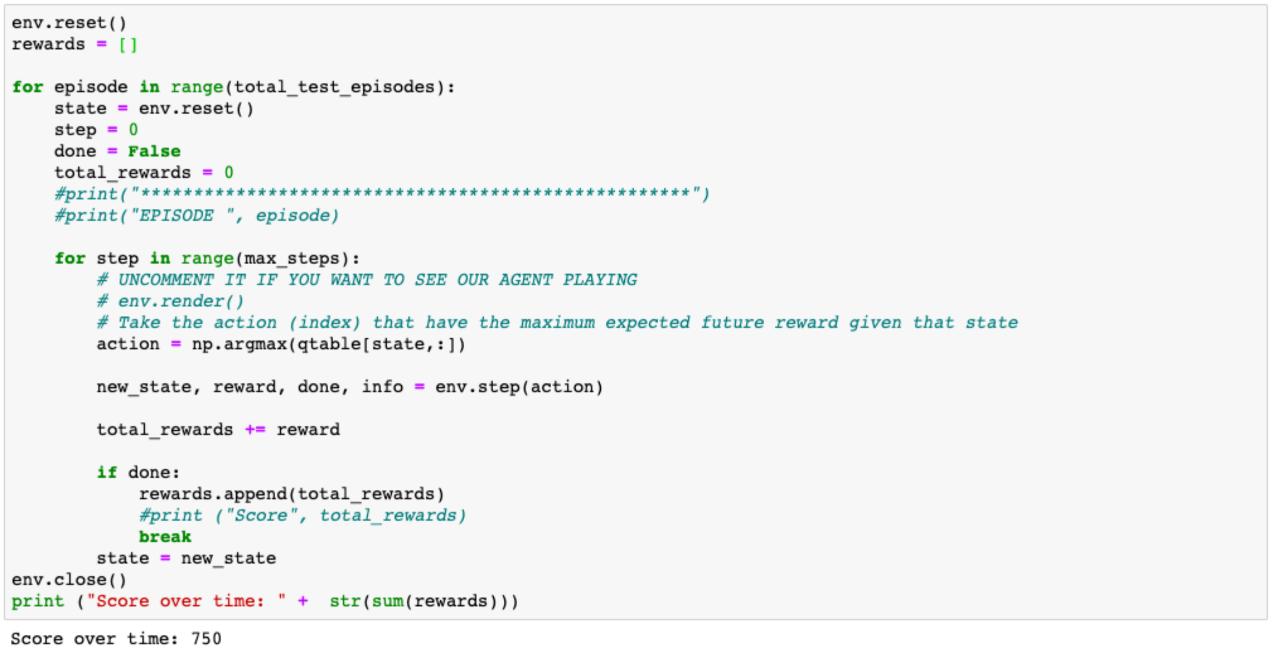
```
total_episodes = 50000        # Total episodes
total_test_episodes = 100     # Total test episodes
max_steps = 99                # Max steps per episode

learning_rate = 0.9           # Learning rate
gamma = 0.50                  # Discounting rate

# Exploration parameters
epsilon = 1.0                 # Exploration rate
max_epsilon = 1.0             # Exploration probability at start
min_epsilon = 0.01            # Minimum exploration probability
decay_rate = 0.01             # Exponential decay rate for exploration prob
```

→ All the parameters are kept the same except for two things: the *learning rate* and the value of *gamma*, which is the *discounting rate*.

→ For our third-second scenario, we use the same value of *learning rate* as 0.9 and change the value of *discount rate* to 0.50.

→ We train the agent again by having the agent follow the Epsilon-Greedy Policy. The agent follows exploration when a randomly generated value is less than the epsilon value and follows exploitation otherwise.

- Evaluating the agent



```
env.reset()
rewards = []

for episode in range(total_test_episodes):
    state = env.reset()
    step = 0
    done = False
    total_rewards = 0
    #print("****************************************************")
    #print("EPISODE ", episode)

    for step in range(max_steps):
        # UNCOMMENT IT IF YOU WANT TO SEE OUR AGENT PLAYING
        # env.render()
        # Take the action (index) that have the maximum expected future reward given that state
        action = np.argmax(qtable[state,:])

        new_state, reward, done, info = env.step(action)

        total_rewards += reward

        if done:
            rewards.append(total_rewards)
            #print ("Score", total_rewards)
            break
        state = new_state
env.close()
print ("Score over time: " +  str(sum(rewards)))
```

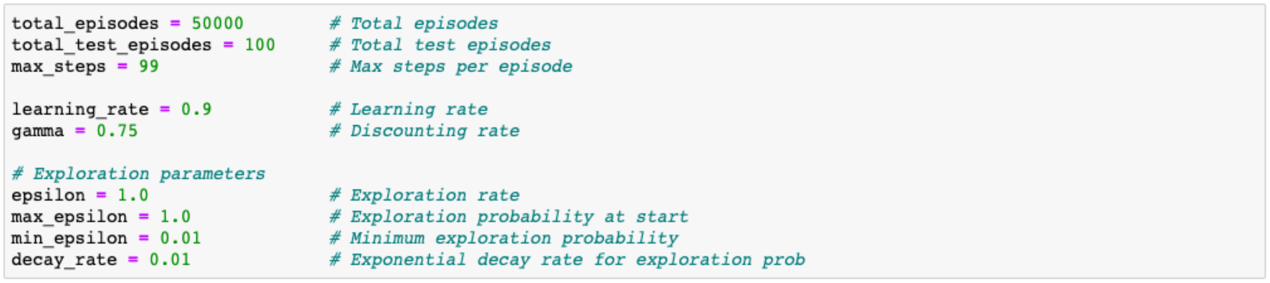
```
Score over time: 750
```

→ After the training, we evaluate our agent's performance over 100 episodes.
→ As we can see, the sum of the *rewards* received by our agent over a total episode of 100 is equal to 750.

<u>Scenario 3-3</u>:

- Training the agent

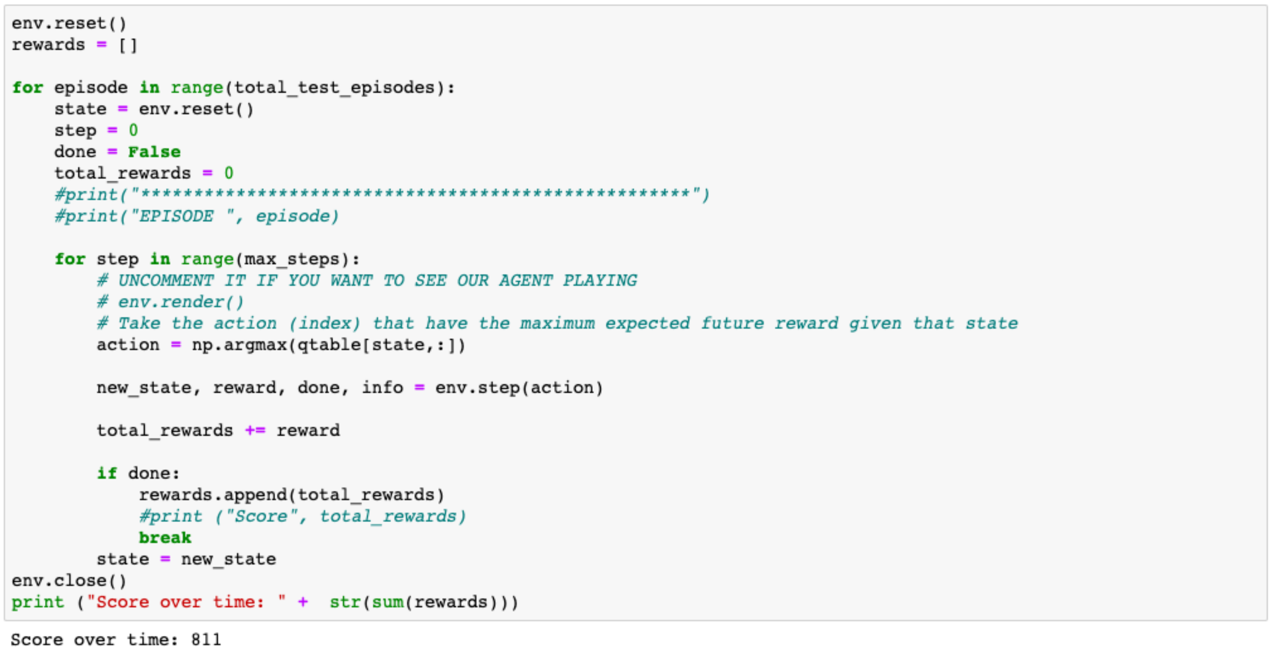
Learning Rate = 0.9
Gamma = 0.75



```
total_episodes = 50000        # Total episodes
total_test_episodes = 100     # Total test episodes
max_steps = 99                # Max steps per episode

learning_rate = 0.9           # Learning rate
gamma = 0.75                  # Discounting rate

# Exploration parameters
epsilon = 1.0                 # Exploration rate
max_epsilon = 1.0             # Exploration probability at start
min_epsilon = 0.01            # Minimum exploration probability
decay_rate = 0.01             # Exponential decay rate for exploration prob
```

→ For our third-third scenario, we use the same value of *learning rate* as 0.9 and change the value of *discount rate* to 0.75.
→ We train the agent again following the same policy as mentioned before.

- Evaluating the agent



```
env.reset()
rewards = []

for episode in range(total_test_episodes):
    state = env.reset()
    step = 0
    done = False
    total_rewards = 0
    #print("****************************************************")
    #print("EPISODE ", episode)

    for step in range(max_steps):
        # UNCOMMENT IT IF YOU WANT TO SEE OUR AGENT PLAYING
        # env.render()
        # Take the action (index) that have the maximum expected future reward given that state
        action = np.argmax(qtable[state,:])

        new_state, reward, done, info = env.step(action)

        total_rewards += reward

        if done:
            rewards.append(total_rewards)
            #print ("Score", total_rewards)
            break
        state = new_state
env.close()
print ("Score over time: " +  str(sum(rewards)))
```

```
Score over time: 811
```

→ After the training, we evaluate our agent's performance over 100 episodes.

→ As we can see, the sum of the *rewards* received by our agent over a total episode of 100 is equal to 811.

4. RESULTS:

	Score		
Gamma	Learning Rate = 0.1	Learning Rate = 0.5	Learning Rate = 0.9
0.25	315	582	331
0.50	739	777	750
0.75	853	850	811

→ As we can notice from our table above, for *gamma = 0.25* and *gamma = 0.*50, the algorithm performs better when we increase the *learning rate* from *0.1* to *0.5* but performs slightly less when the *learning rate* is *0.9*.

→ This is not the scenario when *gamma = 0.75* as we can see the score is reduced through each iteration of the different values of the *learning rate*.

4. REFERENCES:

1) https://github.com/simoninithomas/Deep_reinforcement_learning_Course/blob/master/Q%20Learning/Taxi-v2/Q%20Learning%20with%20OpenAI%20Taxi-v2%20video%20version.ipynb

2) https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/

***5. Contributions*:**

1. Research on the paper was performed by the entire group.
2. The documentation of the paper was taken care of by Prashant R. Ghimire and Samuel E. Casco up till the Rewards and then by Ashish Joshi and Rezwanul Islam Rajib for the remaining portion.
3. The total of 9 variations of the experiment were divided between the 4, with Prashant R. Ghimire doing the final combination for the report.