Mahfuz Rahman

Student No.: 217847518

EECS4421 Lab 6

```
import sys
import numpy as np
import cv2
import matplotlib.pyplot as plt

# using this library for running on Colab
from google.colab.patches import cv2_imshow
```

```
n_image = 1000

def save_Q(Q):
    global n_image
    image = np.zeros((700, 700))  # Updated to 700x700 for a 7x7 maze
    for idx in range(49):  # 7x7 = 49 states
        i = int(idx / 7)
        j = idx % 7
        v = max(Q[idx])
        z = (v + 1000) / 1500
        z = max(0, min(1, z))  # Normalize value to [0, 1]
        for k in range(100):
            for l in range(100):
                image[i * 100 + k][j * 100 + l] = z
    image = 255 * image
    img = image.astype(np.uint8)

    # Uncomment if running the code on local machine and not colab
    #cv2.imshow("image", img)
    cv2_imshow(img)

    cv2.waitKey(10)
    # Uncomment for output
    # cv2.imwrite(f"pic_{n_image}.jpg", img)
    n_image += 1

def init_Q():
    Q = []
    for i in range(49):  # 7x7 = 49 states
        x = [0, 0, 0, 0]
        Q.append(x)
    return Q

def reset():
    return 0

def state_to_row_col(s):
    return [int(s / 7), s % 7]
```

```
def execute_action(maze, s, a):
    sp = [s[0] + a[0], s[1] + a[1]]
    isp = sp[0] * 7 + sp[1]
    idxs = s[0] * 7 + s[1]

    if (sp[0] < 0) or (sp[1] < 0) or (sp[0] >= 7) or (sp[1] >= 7):
        return idxs, -100, False
    if maze[sp[0]][sp[1]] == 0:
        return isp, -1, False
    elif maze[sp[0]][sp[1]] == 1:
        return idxs, -100, False
    elif maze[sp[0]][sp[1]] == 10:
        return isp, 500, True
    print(f"Should not happen {s} {sp} {a} {maze}")
    sys.exit(1)
```

```
# Fixed parameters
n_episodes = 300
max_episode_length = 50
alpha = 0.2
gamma = 0.7
```

```
def q_learning(maze, actions, epsilon):
    rewards = []
    Q = init_Q()
    for e in range(n_episodes):
        s = reset()
        reward = 0

        for t in range(max_episode_length):
            if np.random.uniform(0, 1) < epsilon:
                a = np.random.randint(4)
            else:
                a = np.argmax(Q[s])

            sp, R, done = execute_action(maze, state_to_row_col(s), actions[a])
            Q[s][a] = (1 - alpha) * Q[s][a] + alpha * (R + gamma * max(Q[sp]))
            reward += R
            if done:
                break
            s = sp
        rewards.append(reward)

        # save_Q(Q)
    return rewards, Q
```

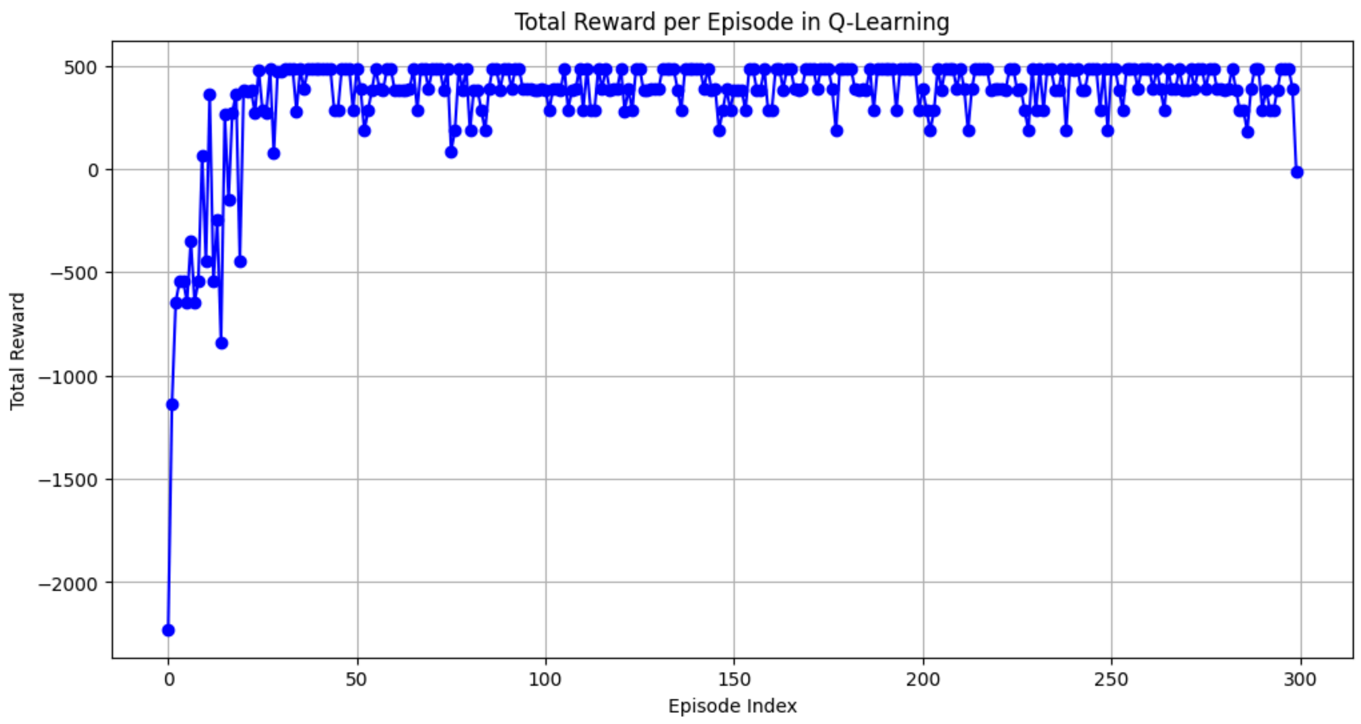
```
# Define the 7x7 maze
maze = np.array([
    [0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 0, 1],
    [0, 0, 0, 0, 1, 0, 1],
    [1, 1, 1, 0, 1, 0, 1],
    [1, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 0, 1, 1],
    [0, 10, 0, 0, 0, 0, 0]
])

actions = [[0, 1], [1, 0], [0, -1], [-1, 0]]
```

```
rewards, q = q_learning(maze, actions, epsilon=0.1)
```

```
for i, r in enumerate(rewards):
    print(f"{i}, {r}")
print(q)
```

Show hidden output

```
episode_indices = list(range(300))
plt.figure(figsize=(12, 6))
plt.plot(episode_indices, rewards, marker='o', linestyle='-', color='b')
plt.title('Total Reward per Episode in Q-Learning')
plt.xlabel('Episode Index')
plt.ylabel('Total Reward')
plt.grid()
plt.show()
```

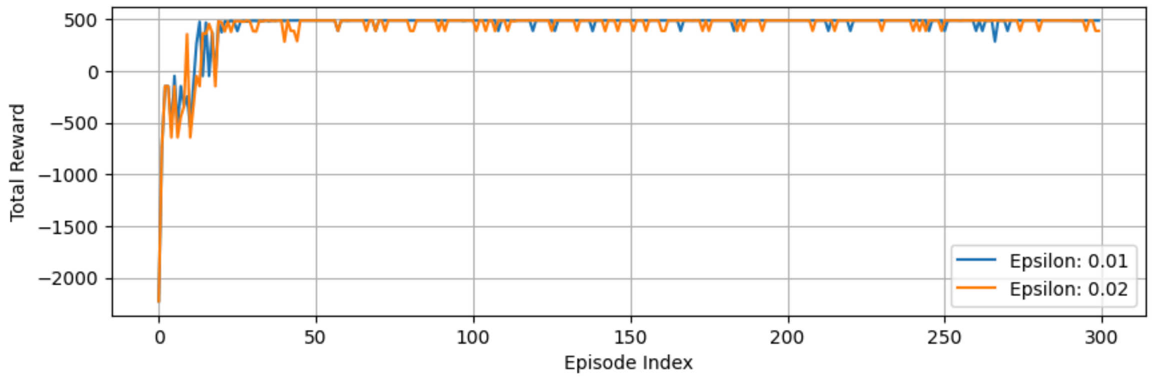


---

1. Using an epsilon greedy exploration strategy and with fixed learning and future discount reward parameters, for a given map world, solve the task. Keep the number of interaction steps per interaction and number of interactions fixed. Now choose different fixed epsilon values 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64 and run your code. If you plot the reward as a function of step size, what do you see?

```
def plotDifferentEpsilon(epsilon_values):

  # Plotting rewards for each epsilon value
  plt.figure(figsize=(10, 3))

  # Running Q-learning for each epsilon value and collecting rewards
  for epsilon in epsilon_values:
      rewards, q = q_learning(maze, actions, epsilon)
      plt.plot(range(300), rewards, label=f'Epsilon: {epsilon}')

  plt.xlabel('Episode Index')
  plt.ylabel('Total Reward')
  plt.legend()
  plt.grid()
  plt.show()
```

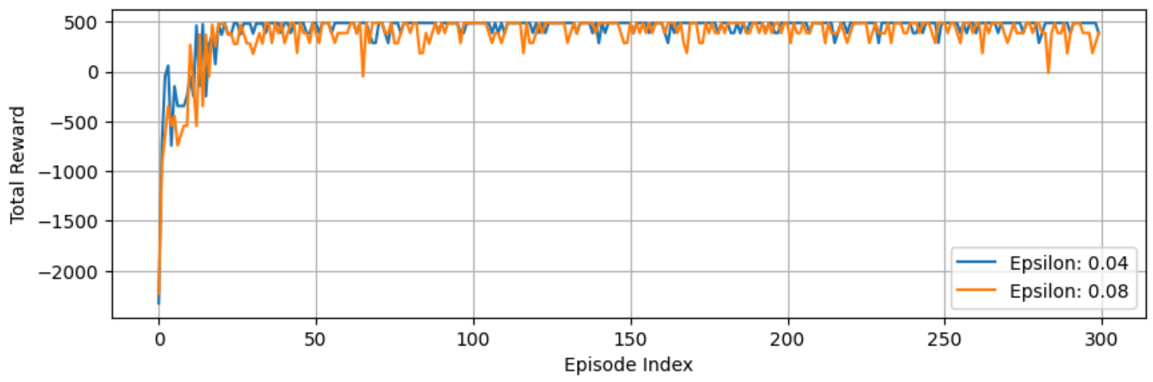
```
epsilon_values = [0.01, 0.02]
plotDifferentEpsilon(epsilon_values)
```



Low Epsilon Values (e.g., 0.01, 0.02):

These values represent a high level of exploitation. So the the agent relies mostly on its learned Q-values and explores less. Therefore, it might take longer to discover optimal paths, leading to lower initial rewards but potentially higher stability in later episodes as it learns.

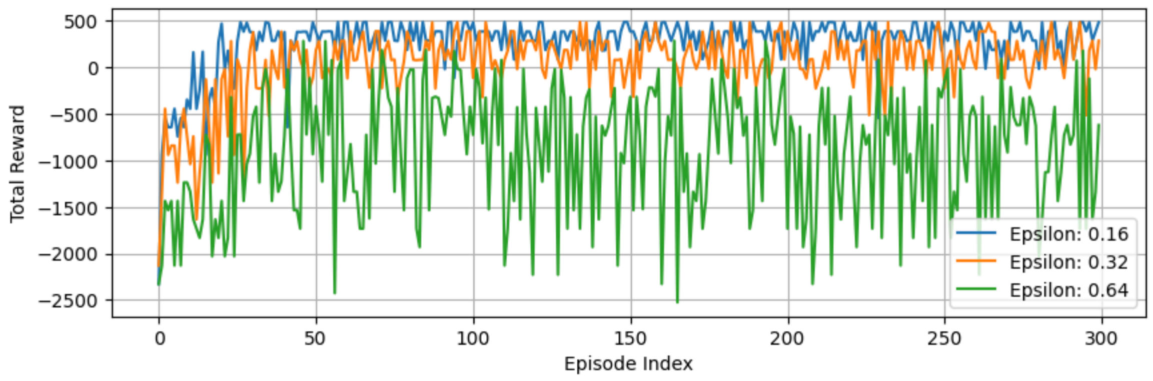
```
epsilon_values = [0.04, 0.08]
plotDifferentEpsilon(epsilon_values)
```



Medium Epsilon Values (e.g., 0.04, 0.08):

These provide a balanced approach between exploration and exploitation. The agent shows a gradual increase in rewards as it explores different actions and learns more about the maze.

```
epsilon_values = [0.16, 0.32, 0.64]
plotDifferentEpsilon(epsilon_values)
```



High Epsilon Values (0.16, 0.32, 0.64):

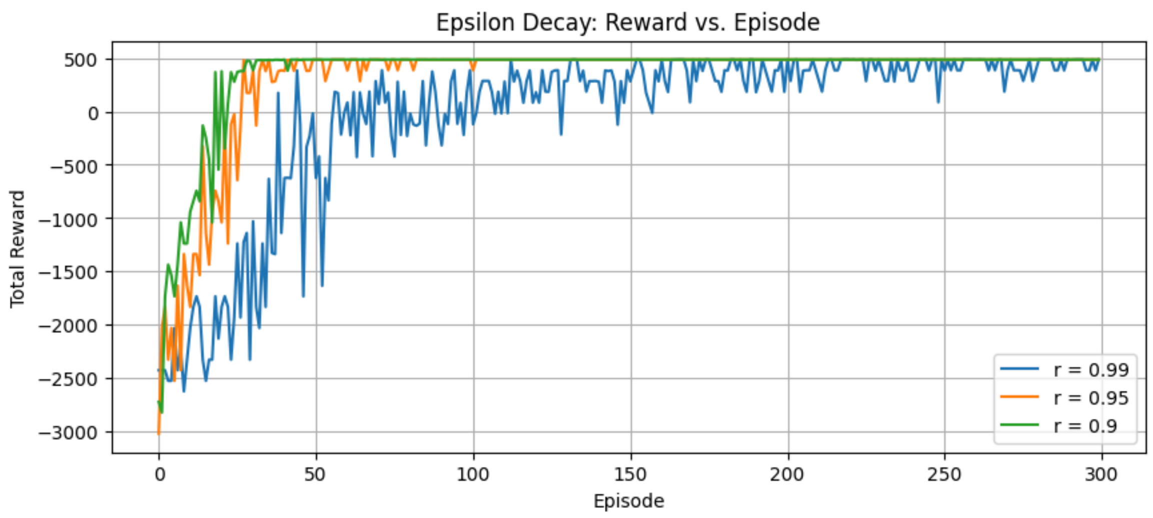
At these levels, the agent explores a lot, which is leading to a more erratic learning curve as seen above. The agent frequently takes suboptimal actions, leading to lower total rewards in the initial episodes.

---

2. `A common strategy to deal with a fixed epsilon is known as 'epsilon decay' where the value of epsilon is fixed for a given episode but then is made to decay to zero over multiple episodes. One can do this in many ways. A simple way is to multiply epsilon by a number r<1 after each episode (although this can decrease epsilon too quickly). Start at some epsilon0 (say 0.9) and try different r's. Plot some representative reward curves over your baseline from above.`

```python
def q_learning_1(maze, actions, epsilon, r):
    rewards = []
    Q = init_Q()

    for e in range(n_episodes):
        s = reset()
        reward = 0

        for t in range(max_episode_length):
            if np.random.uniform(0, 1) < epsilon:
                a = np.random.randint(4)
            else:
                a = np.argmax(Q[s])

            sp, R, done = execute_action(maze, state_to_row_col(s), actions[a])
            Q[s][a] = (1 - alpha) * Q[s][a] + alpha * (R + gamma * max(Q[sp]))
            reward += R

            if done:
                break
            s = sp

        rewards.append(reward)

        epsilon *= r
    return rewards, Q
```

```python
epsilon = 0.9
r_values = [0.99, 0.95, 0.9]

plt.figure(figsize=(10, 4))

for r in r_values:
    rewards, q = q_learning_1(maze, actions, epsilon, r)
    plt.plot(range(n_episodes), rewards, label=f'r = {r}')

plt.title('Epsilon Decay: Reward vs. Episode')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend()
plt.grid(True)
plt.show()
```



The effects of Decay Rates:

For Decay Rate 0.99, the agent starts with a relatively high epsilon (0.9) and decays it slowly. As a result, it maintains a good level of exploration throughout training, leading to a more stable and generally higher total reward over episodes.

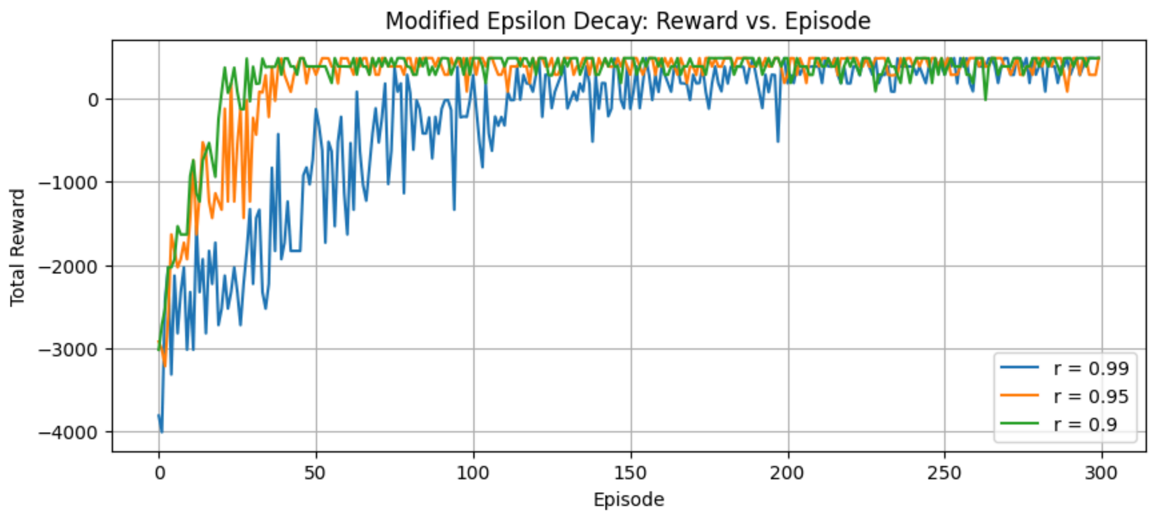
Decay Rate 0.95 results in a slightly faster decay of epsilon compared to 0.99. The agent still manages to explore effectively but may begin to stabilize its rewards sooner, reflecting less exploration towards the end of the training.

Lastly, for Decay Rate 0.90, the agent transitions to exploitation more quickly and with a more rapid decay of epsilon. This can lead to a suboptimal policy if the agent has not fully explored the maze, resulting in lower total rewards overall.

3. One fix to the epsilon value would be to use an epsilon value = r^n * epsilon1 + epsilon0. [never let epsilon decay below epsilon0]. Try this. How does it change performance

```
def q_learning_2(maze, actions, epsilon1, epsilon0, r):
    rewards = []
    Q = init_Q()

    for e in range(n_episodes):
        epsilon = (r**e) * epsilon1 + epsilon0 # r^n * epsilon1 + epsilon0

        s = reset()
        reward = 0

        for t in range(max_episode_length):
            if np.random.uniform(0, 1) < epsilon:
                a = np.random.randint(4)
            else:
                a = np.argmax(Q[s])

            sp, R, done = execute_action(maze, state_to_row_col(s), actions[a])
            Q[s][a] = (1 - alpha) * Q[s][a] + alpha * (R + gamma * max(Q[sp]))
            reward += R

            if done:
                break
            s = sp

        rewards.append(reward)
    return rewards, Q
```

```
epsilon1 = 0.9  # Initial exploration rate
epsilon0 = 0.1  # Minimum exploration rate
r_values = [0.99, 0.95, 0.9]

plt.figure(figsize=(10, 4))

for r in r_values:
    rewards, q = q_learning_2(maze, actions, epsilon1, epsilon0, r)
    plt.plot(range(n_episodes), rewards, label=f'r = {r}')

plt.title('Modified Epsilon Decay: Reward vs. Episode')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend()
plt.grid(True)
plt.show()
```

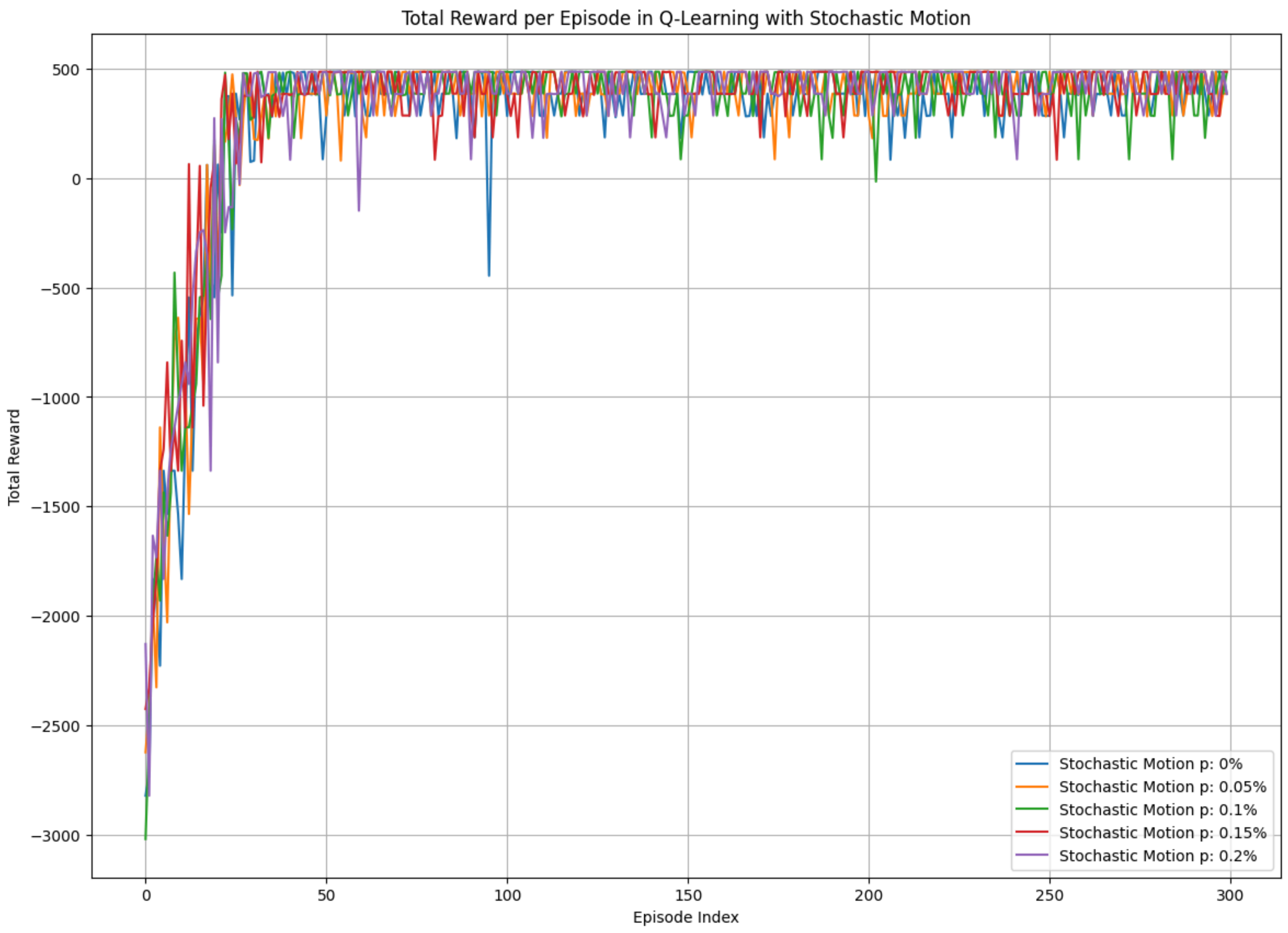


Upon not letting the epsilon value getting below epsilon0, the agent maintains some level of exploration throughout training, which can help it discover optimal strategies more effectively. So with the ability to continue exploring new actions, the agent may improve its learning and converge further later to optimal policies.

---

4. `Q-learning for fixed motion and fixed rewards does not highlight the full range of the performance of the algorithm. Introduce stochastic motion to your agent by having the agent do what is required (move up, down, left, right) p % if the time, but (100-p) % of the time it moves randomly over all other possible motions. Returning to your baseline, train on a given p (say 10%) and then performance with p=0,5,10,15,20.`

```
# Implementing stochastic motion for the Q-learning agent

def stochastic_execute_action(maze, s, a, p):
    """
    Executes the specified action with probability p, otherwise moves randomly.
    """
    if np.random.uniform(0, 1) < p:  # With probability p, take the specified action
        return execute_action(maze, s, a)
    else:  # With probability (100 - p), move randomly
        random_action = np.random.randint(4)
        return execute_action(maze, s, actions[random_action])

def q_learning_with_stochastic_motion(maze, actions, epsilon1, epsilon0, r, p):
    rewards = []
    Q = init_Q()  # Initialize Q-table
    for e in range(n_episodes):
        s = reset()  # Start from initial state
        reward = 0

        epsilon = (r**e) * epsilon1 + epsilon0 # r^n * epsilon1 + epsilon0

        for t in range(max_episode_length):

            if np.random.uniform(0, 1) < epsilon:
                # Stochastic action selection based on probability p
                if np.random.uniform(0, 1) < p:  # Move as intended
                    a = np.argmax(Q[s])
                else:  # Move randomly
                    a = np.random.randint(4)
            else:
                # Exploit the learned policy
                a = np.argmax(Q[s])  # Best action

            sp, R, done = execute_action(maze, state_to_row_col(s), actions[a])

            # Update Q-values
            Q[s][a] = (1 - alpha) * Q[s][a] + alpha * (R + gamma * max(Q[sp]))
            reward += R
            if done:
                break
            s = sp  # Move to the new state

        rewards.append(reward)

        #save_Q(Q)
    return rewards, Q



epsilon1 = 0.9
epsilon0 = 0.1
r_values = [0.99, 0.95, 0.9]
p_values = [0, 0.05, 0.10, 0.15, 0.20]

plt.figure(figsize=(14, 10))

for p in p_values:
    rewards, q = q_learning_with_stochastic_motion(maze, actions, epsilon1, epsilon0, r, p)
    plt.plot(range(n_episodes), rewards, label=f'Stochastic Motion p: {p}%')

plt.title('Total Reward per Episode in Q-Learning with Stochastic Motion')
plt.xlabel('Episode Index')
plt.ylabel('Total Reward')
plt.legend()
plt.grid()
plt.show()
```



I introduced stochastic action based on the following strategy:

If rand < ϵ, use stochastic motion:

- Decide to perform the intended action with probability
- With probability 1 - p, select a random action

If rand >= ϵ, follow the learned policy by selecting the action with the highest Q-value.

The results show a much more realistic learning strategies.