INM707 Deep Reinforcement Learning Coursework

Ho Yin, Tam

GitHub link: https://github.com/harrytam/inm707/blob/main/INM707-hoyintam.ipynb

1 Defining the Environment and Presenting the Problem

The growth in e-commerce and the requirement for fast fulfilment of customer orders drive companies to adopt automated warehouse robots in warehouses. The automated warehouse robots provide more efficiency than human labour as they can timelessly work in the warehouse [1]. In this report, the delivery from the starting point to the target destination of an automated warehouse robot is simulated in a grid-based environment.

Figure 1 displays the simulated warehouse environment in a grid format. As the grid is 10 x 10, the number of states in the environment is 100. The shelves which are the obstacles are presented as black cells while the aisles that the automated warehouse robots can freely travel are expressed as white cells. The left bottom corner is the starting point while the top right corner is the target destination. The automated warehouse robot acts as the agent and has four possible actions including up, down, left and right. To simplify the simulation, the assumption is that the agent already carries the inventory, and the goal of the agent is to deliver it to the target destination for packaging in the shortest path.

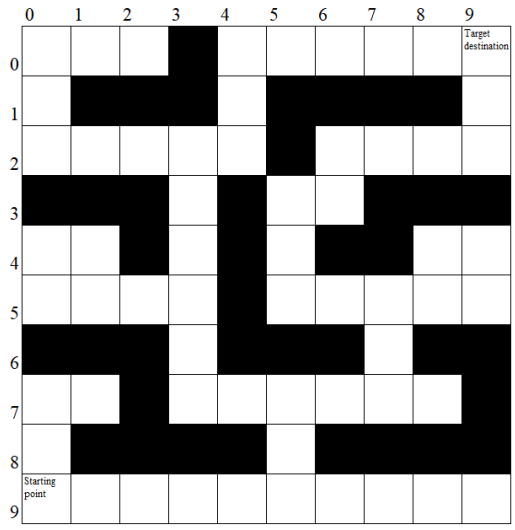

Figure 1: Simulated warehouse environment

2 State Transitions and Rewards

State transition refers to the process of moving from one state to another state in an environment in response to the action of the agent. In this case, the state transition

occurs when the automated warehouse robot moves from one grid to another grid in the warehouse environment based on its chosen action including up, down, left, and right. For instance, the automated warehouse robot starts at the position of (9,0) which is the current state. It can choose an action to move to the right and transit to another state (9, 1).

Rewards refer to the numerical value or immediate feedback which the agent receives from the environment in response to the action. In this case, if the automated warehouse robot moves between aisles or moves from the state (9, 0) to (9, 1), it receives a reward of -1. The reward is set to be negative so as to encourage it to find the shortest path. If it hits the shelve, it receives a reward of -100. The harsh penalty aims to discourage the automated warehouse robot from colliding with the shelve. Also, if it arrives at the target location (0, 9), it receives a reward of 100.

3 Q-Learning Parameter and Policies

Three parameters are applied in Q-Learning. The first parameter is the learning rate (alpha) which controls the step size in updating the Q-values. It is set to 0.1 so the automated warehouse robot can slowly adapt to the new information and is less likely to forget the past experience. The second parameter is the discount factor (gamma) which represents the importance of future rewards compared to the immediate reward. It is set to 0.9 since it is hoped that the agent emphasises the long-term benefits. The third parameter is epsilon which is used to balance the situation of exploration and exploitation. It is set to 0.1 so the agent has a 10% chance to explore a random action and a 90% chance to exploit the current knowledge and select the action with the highest estimated value.

The policy refers to the strategy which the agent follows to make decisions in the environment. In this case, using the epsilon greedy strategy, the optimal policy is the combination of minimizing the penalty by preventing colliding with the shelves and maximizing the rewards by reaching the target destination while also involving exploitation most of the time with the highest Q values and exploration to learn and improve the policy over time.

4 Q-Learning algorithm performance

In the first experiment, the parameters of learning rate, discount factor, and epsilon are set as 0.1, 0.9, and 0.1 respectively. Figure 2 depicts the total rewards in each episode for 1000 episodes. To start with, the total rewards in the first episode and in the last episode are approximately -800 and 80 respectively. For the first 100 episodes, the majority of the total rewards are negative. This indicates that the automated warehouse robot frequently collides with shelves. There is a gradual upward trend in the total rewards per episode which suggests that the agent is slowly learning to avoid crashing the shelves. It is clear that the total rewards converge when the episode is close to 200. This shows that the agent has successfully and consistently reached the target destination for packaging.

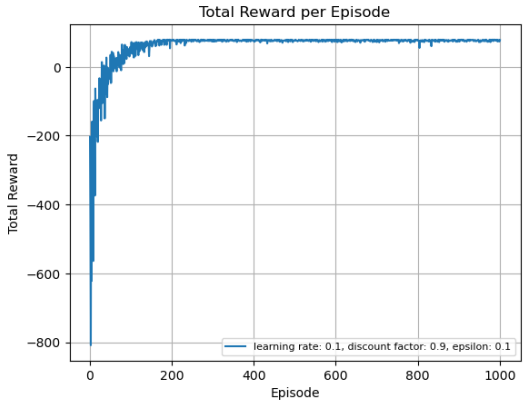

Figure 2: Total rewards per episode

5 Experiment with different parameters (learning rate, discount factor, and epsilon)

For better and clearer comparison, the experiment will be implemented in the way that only one parameter will change at one time while the other two parameters will remain constant. The default setting is the learning rate of 0.1, the discount factor of 0.9, and the epsilon of 0.1 in Figure 2.

The first experiment tests the learning rate of 0.01, 0.1, and 0.9 while the discounting factor and epsilon remain with the value of 0.9 and 0.1 respectively.

The second experiment tests the discount factor of 0.1, 0.5, and 0.9 while the learning rate and epsilon remain with the value of 0.1 and 0.1 respectively.

The third experiment tests the epsilon of 0.1, 0.5, and 0.9 while the learning rate and epsilon remain with the value of 0.1 and 0.9 respectively.

6 Analysis

Figure 3 illustrates the total reward per episode with different learning rates given the discount factor and epsilon remains unchanged. Three learning rates show that the convergence eventually occurs. The learning rate of 0.9 converges the fastest with only 25 episodes with the steepest slope as it allows larger updates to the policy. This may lead to the faster update and convergence. However, it also shows instability as the fluctuation after convergence is larger than the others. The learning rate of 0.01 converges the slowest with 400 episodes with a smoother slope. Despite the difference in convergence speed, all three learning rates eventually achieve convergence and have a positive total reward. This suggests that the automated warehouse robot eventually learns an effective policy to navigate the warehouse with the shortest path to the target destination and without colliding with the shelves

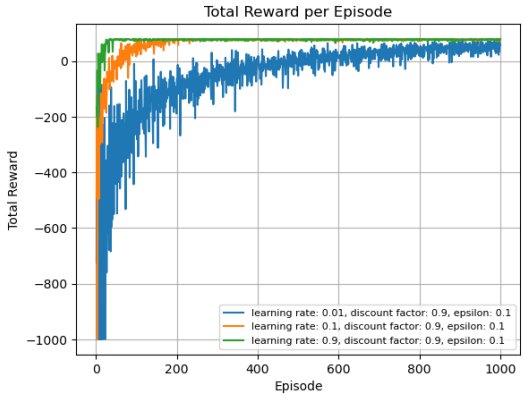

Figure 3: Experiment with different learning rates

Figure 4 exhibits the total reward per episode with different discount factors given the learning rate and epsilon remain unchanged. The line representing the discount factor of 0.5 and 0.9 converges at a similar episode of 150. On the other hand, the line representing the discount factor of 0.1 provides an interesting result. The initial negative rewards from the first 200 episodes show that the automated warehouse robot frequently hit the shelves. At the same time, for the first 200 episodes, the upward trend shows the robot is gradually learning and improving by less frequently bumping to the shelves. After the first 200 episodes, an obvious fluctuation is observed. This indicates that the performance of the robot is still unstable in which it may be exploring new paths in the warehouse and collide the shelves. A few positive rewards also reveal that the robot only successfully arrives at the target destination without crashing the shelves a few times. In addition, the discount factor of 0.1 suggests that the robot emphasises the immediate reward. It prioritises the move between aisles within the maximum 1000 steps in each episode and avoids colliding with the shelves over reaching out to the target destination. Therefore, the change in the discount factor may pose a significant impact on the convergence.

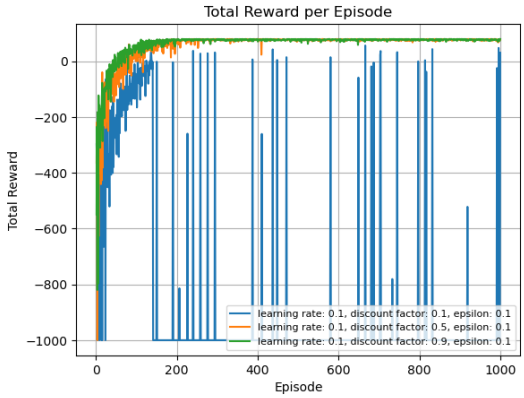

Figure 4: Experiment with different discount factors

Figure 5 exhibits the total reward per episode with different epsilon given the learning

rate and discount factor remain unchanged. The lines with the epsilons of 0.1 and 0.5 converge at the episode of 150. Though the epsilon of 0.1 has a higher reward than the epsilon of 0.5 most of the time, they both have positive total rewards indicating they successfully arrive at the target destination. A lower epsilon leads to a faster convergence as the automated warehouse robot emphasises exploitation over exploration and is more likely to choose the action that results in less penalty. Another notable observation is that the epsilon of 0.9 has fluctuation in the whole 1000 episodes. The high epsilon value means that the robot has a 90% chance to explore by choosing a random action and a 10% to exploit by selecting the action with the highest expected reward. This fluctuation and the negative total rewards most of the time suggest that the robot may frequently take random actions and hit the shelves. Also, a few positive total rewards indicate that the robot only successfully arrives at the target destination without crashing the shelves a few times.

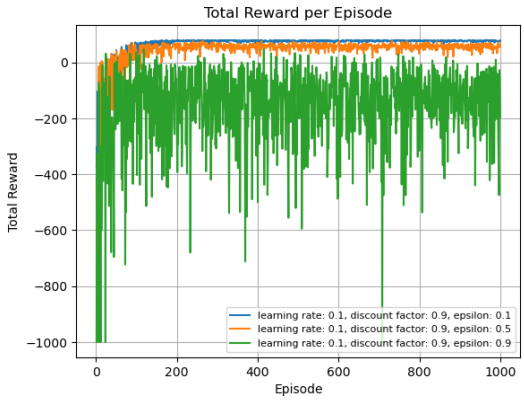

Figure 5: Experiment with different epsilons

Overall, the initial parameters of the learning rate of 0.1, the discount factor of 0.9 and the epsilon of 0.1 provide a satisfactory result as the convergence occurs with the positive total rewards of approximately 75 indicating the robot succeeds in reaching the target destination without colliding the shelves. The total rewards converge in an acceptable range at the 200 episodes which is moderate allowing the robot to explore and exploit. The less fluctuation shows the stability of the performance of the robot in the long term.

7 Deep Q-Network Implementations and Two Improvements

Deep Q-Network on Frozen Lake

In this task, a deep Q-network will be implemented first. A deep Q-network is an algorithm that utilizes the neural network to approximate the state-value function in Q-learning [2]. Instead of using the same simulated warehouse environment, the frozen lake environment is employed. In the environment, the action space is 4 including moving up, down, left, and right, while the observation space is 16 as it is a

4 x 4 grid. The goal of the agent is to cross the frozen lake from the starting position to the target destination without falling holes. If the agent arrives at the target destination, it receives a reward of positive one, else, the agent receives no reward or penalty. The interesting setting compared to the simulated warehouse environment is that the frozen lake is slippery in which the agent may have a 33.3% probability of moving in one direction and at the same time the agent also has a 33.3% probability of moving in the perpendicular directions of the chosen direction.

8 Performance of the Deep Q Networks

Improvements with Double Deep Q-Network

Double deep Q-network (DDQN) used a single Q-network for the target Q-values and another Q-network for the current Q-values. As the deep Q-network may potentially overestimate the Q-values, the double deep Q-network employs two Q-networks to provide a more accurate estimation of Q-values. Hence, it is expected that the performance of double deep Q-network is better than the deep Q-network.

Figure 6 displays the average rewards per episode of deep Q-network and double deep Q-network. Overall, it is clear that the average reward per episode of the double deep Q-network is higher than the deep Q-network. The highest average reward of the double deep Q-network is 10 which is the double to the deep Q-network. As the deep Q-network may overestimate the Q-value, the agent may make suboptimal decisions and slow down the learning process. The double deep Q-network introduces a separate target network to estimate the Q-values and provides higher and more stable Q-values. This indicates that the agent in double deep Q-network performs better in navigating the frozen lake environment. This result is the same as expected.

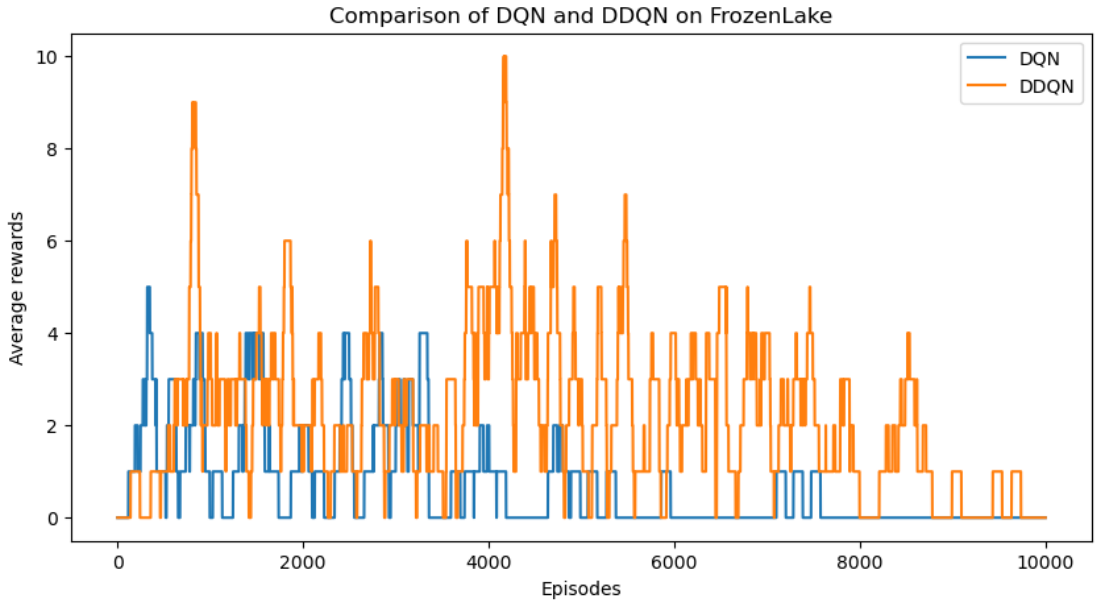

Figure 6: DQN vs DDQN

Improvements with Experience Replay

Experience replay introduces the memory buffer in which the function is to store the past experience of the agent interacting with the environment. In the training process, the agent samples a small batch of experience from the memory to update the policy network. The experience replay can improve learning efficiency and increase stability by utilizing a broader range of training data. Therefore, it is expected that the deep Q-network with experience replay performs better than those without experience replay.

Figure 7 illustrates the total rewards per episode of the deep Q-network with experience replay and without experience replay. It is clear that the total reward of the network without experience replay performs better than with the experience replay. This shows that it has more episodes to have the total reward of 1 as the agent successfully arrives at the target destination. Also, the total reward without experience replay shows frequent fluctuation while the total reward with experience replay shows less fluctuation, always maintaining at the level of zero. However, this is not the same as expected. This is probably due to the limited number of training episodes as the network without experience replay may be lucky to reach the target destination.

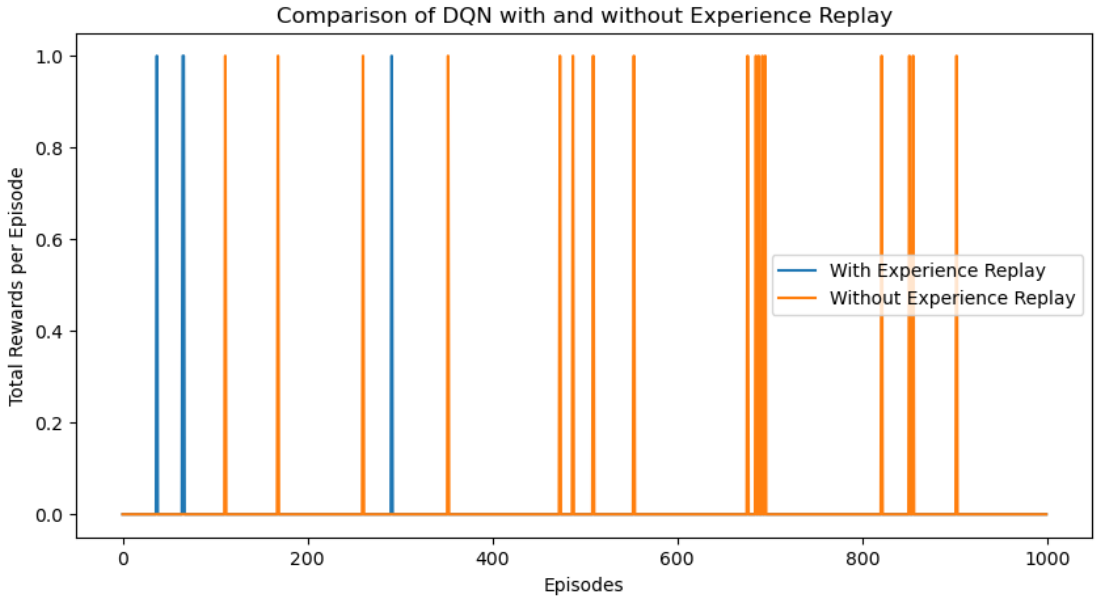

Figure 7: With and without experience replay

9 Proximal Policy Optimization

In this task, the environment of a cart pole is employed. The cart pole environment consists of a pole attached to a cart by a joint and the goal is to balance the inverted pendulum in an upright and vertical direction. The algorithm of proximal policy optimization is used. It is an algorithm that excels in balancing exploration and exploitation. The agent continuously learns by interacting with the environment,

collecting data and updating the policy network. The clipping mechanism allows the agent to learn stably and prevent the policy from becoming overly greedy and neglecting the exploration opportunities. The advantages of this algorithm are easy to implement and computationally inexpensive. Hence, it is suitable to apply on the cart pole environment.

10 Analysis

Figure 8 displays the average reward per episode of the proximal policy optimization algorithm. Overall, a clear increasing trend can be observed which indicates the agent is learning to control and balance the pole on the cart. At the episode of approximately 40, there is a slight drop from the average reward of 42 to 32. This may be possibly due to the agent becoming overly exploitative and neglecting exploration. In general, the convergence occurs around the episode of 60 having the average reward of 35.

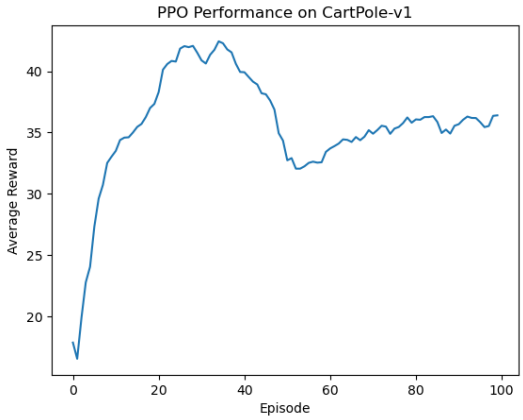

Figure 8: PPO performance

Reference:

[1] Dhaliwal, A. (2020). The rise of automation and robotics in warehouse management. In Transforming Management Using Artificial Intelligence Techniques (pp. 63-72). CRC Press.

[2] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, .

Contribution:

I did the work individually.

 # INM707 Coursework

Ho Yin Tam

Basic

Import the library

```
import numpy as np
import random
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import gymnasium as gym
import pygame
from collections import deque
import ray
from ray.rllib.algorithms.ppo import PPOConfig
from ray.rllib.env import EnvContext
from ray.rllib.algorithms.sac.sac_tf_policy import SACTFPolicy
```

Define the environment and perform Q learning.

```
simulated_warehouse_environment = np.full((10,10), -1)

# Set the top right corner as the target destination with a reward of 100
simulated_warehouse_environment[0,9] = 100
```

```
# Set the shelve with the reward of -100
simulated_warehouse_environment[0,3] = -100

simulated_warehouse_environment[1,1] = -100
simulated_warehouse_environment[1,2] = -100
simulated_warehouse_environment[1,3] = -100
simulated_warehouse_environment[1,5] = -100
simulated_warehouse_environment[1,6] = -100
simulated_warehouse_environment[1,7] = -100
simulated_warehouse_environment[1,8] = -100

simulated_warehouse_environment[2,5] = -100

simulated_warehouse_environment[3,0] = -100
simulated_warehouse_environment[3,1] = -100
simulated_warehouse_environment[3,2] = -100
simulated_warehouse_environment[3,4] = -100
simulated_warehouse_environment[3,9] = -100
simulated_warehouse_environment[3,8] = -100
simulated_warehouse_environment[3,7] = -100

simulated_warehouse_environment[4,2] = -100
simulated_warehouse_environment[4,4] = -100
simulated_warehouse_environment[4,6] = -100
simulated_warehouse_environment[4,7] = -100

simulated_warehouse_environment[5,4] = -100

simulated_warehouse_environment[6,0] = -100
```

```
simulated_warehouse_environment[6,1] = -100
simulated_warehouse_environment[6,2] = -100
simulated_warehouse_environment[6,4] = -100
simulated_warehouse_environment[6,5] = -100
simulated_warehouse_environment[6,6] = -100
simulated_warehouse_environment[6,8] = -100
simulated_warehouse_environment[6,9] = -100

simulated_warehouse_environment[7,2] = -100
simulated_warehouse_environment[7,9] = -100

simulated_warehouse_environment[8,1] = -100
simulated_warehouse_environment[8,2] = -100
simulated_warehouse_environment[8,3] = -100
simulated_warehouse_environment[8,4] = -100
simulated_warehouse_environment[8,6] = -100
simulated_warehouse_environment[8,7] = -100
simulated_warehouse_environment[8,8] = -100
simulated_warehouse_environment[8,9] = -100

print(simulated_warehouse_environment)

# Define constants variable
start_position = (9, 0)
target_destination = (0, 9)
num_episodes = 1000
max_steps_per_episode = 1000
learning_rate = 0.1
discount_factor = 0.9
epsilon = 0.1
```

```
# Define the foun actions which are up, down, left, right
# Up = (-1, 0) as moving up will decrease the row index
# Down = (1, 0) as moving up will increase the row index
# Left = (0, -1) as moving up will decrease the column index
# Right = (0, 1) as moving up will increase the column index
actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Initialize Q-table to store the estimated values of each action for each state in the warehouse
Q_table = np.zeros((10, 10, 4))

# Define function to choose action by using epsilon-greedy
def choose_action(state):
    if random.uniform(0, 1) < epsilon:
        # Choose a random action to explore
        return random.randint(0, len(actions) - 1)
    else:
        # Choose the action with the highest Q-value
        return np.argmax(Q_table[state[0]][state[1]])

# Define function to update Q-table using Q-Learning
def update_q_table(state, action, next_state, reward):
    best_next_action = np.argmax(Q_table[next_state[0]][next_state[1]])
    Q_table[state[0]][state[1]][action] += learning_rate * (reward + discount_factor * Q_table[next_state[0]][next_state[1]][best_next_action] - Q_table[state[0]][state[1]][action])

# Define function to perform Q-learning
def Q_learning():
```

```
# Initiate an empty list to store the total reward in each episode
rewards_in_each_episode = []

for episode in range(num_episodes):
    state = start_position
    total_reward = 0

    for step in range(max_steps_per_episode):
        action = choose_action(state)
        next_state = (state[0] + actions[action][0], state[1] + actions[action][1])

        # Check if the state is valid
        # If the chosen action will move outside the grid or hit the shelve, it is a
invalid state
        if next_state[0] < 0 or next_state[0] >= 10 or next_state[1] < 0 or
next_state[1] >= 10 or
simulated_warehouse_environment[next_state[0]][next_state[1]] == -100:

            # Stay in the same grid or state
            next_state = state

        # Check if the agent arrives to the target destination
        # If yes, give the agent a reward of 100
        if next_state == target_destination:
            reward = 100
            total_reward += reward
            update_q_table(state, action, next_state, reward)
            break

        else:
            # Transit to the next state
```

```
            reward = -1
            total_reward += reward
            update_q_table(state, action, next_state, reward)
            state = next_state

        rewards_in_each_episode.append(total_reward)
        print(f"Episode {episode + 1}: Total Reward = {total_reward}")

    return rewards_in_each_episode

# Run Q-learning and collect the rewards
rewards_in_each_episode = Q_learning()

# Plot the learning curve (total reward accumulated per episode)
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode, label = f'learning rate: {learning_rate} , discount factor: {discount_factor} , epsilon: {epsilon}')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend(fontsize = '8')
plt.title('Total Reward per Episode')
plt.grid(True)
plt.show()

### Experiment wth different learning rate or alpha.

#### Learning rate = 0.01

# Define constants variable
start_position = (9, 0)
target_destination = (0, 9)
num_episodes = 1000
```

```
max_steps_per_episode = 1000
learning_rate = 0.01
discount_factor = 0.9
epsilon = 0.1

# Define the foun actions which are up, down, left, right
# Up = (-1, 0) as moving up will decrease the row index
# Down = (1, 0) as moving up will increase the row index
# Left = (0, -1) as moving up will decrease the column index
# Right = (0, 1) as moving up will increase the column index
actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Initialize Q-table to store the estimated values of each action for each state in the warehouse
Q_table = np.zeros((10, 10, 4))

# Define function to choose action by using epsilon-greedy
def choose_action(state):
    if random.uniform(0, 1) < epsilon:
        # Choose a random action to explore
        return random.randint(0, len(actions) - 1)
    else:
        # Choose the action with the highest Q-value
        return np.argmax(Q_table[state[0]][state[1]])

# Define function to update Q-table using Q-Learning
def update_q_table(state, action, next_state, reward):
    best_next_action = np.argmax(Q_table[next_state[0]][next_state[1]])
    Q_table[state[0]][state[1]][action] += learning_rate * (reward + discount_factor * Q_table[next_state[0]][next_state[1]][best_next_action] - Q_table[state[0]][state[1]][action])
```

```
# Define function to perform Q-learning
def Q_learning():

    # Initiate an empty list to store the total reward in each episode
    rewards_in_each_episode = []

    for episode in range(num_episodes):
        state = start_position
        total_reward = 0

        for step in range(max_steps_per_episode):
            action = choose_action(state)
            next_state = (state[0] + actions[action][0], state[1] + actions[action][1])

            # Check if the state is valid
            # If the chosen action will move outside the grid or hit the shelve, it is a
invalid state
            if next_state[0] < 0 or next_state[0] >= 10 or next_state[1] < 0 or
next_state[1] >= 10 or
simulated_warehouse_environment[next_state[0]][next_state[1]] == -100:

                # Stay in the same grid or state
                next_state = state

            # Check if the agent arrives to the target destination
            # If yes, give the agent a reward of 100
            if next_state == target_destination:
                reward = 100
                total_reward += reward
                update_q_table(state, action, next_state, reward)
```

```
            break

        else:
            # Transit to the next state
            reward = -1
            total_reward += reward
            update_q_table(state, action, next_state, reward)
            state = next_state

        rewards_in_each_episode.append(total_reward)
        print(f"Episode {episode + 1}: Total Reward = {total_reward}")

    return rewards_in_each_episode

# Run Q-learning and collect the rewards
rewards_in_each_episode_alpha1 = Q_learning()

# Plot the learning curve (total reward accumulated per episode)
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_alpha1, label =
f'learning rate: {learning_rate} , discount factor: {discount_factor} , epsilon:
{epsilon}')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend(fontsize = '8')
plt.title('Total Reward per Episode')
plt.grid(True)
plt.show()

#### Learning rate = 0.1

# Define constants variable
```

```
start_position = (9, 0)
target_destination = (0, 9)
num_episodes = 1000
max_steps_per_episode = 1000
learning_rate = 0.1
discount_factor = 0.9
epsilon = 0.1

# Define the foun actions which are up, down, left, right
# Up = (-1, 0) as moving up will decrease the row index
# Down = (1, 0) as moving up will increase the row index
# Left = (0, -1) as moving up will decrease the column index
# Right = (0, 1) as moving up will increase the column index
actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Initialize Q-table to store the estimated values of each action for each state in the warehouse
Q_table = np.zeros((10, 10, 4))

# Define function to choose action by using epsilon-greedy
def choose_action(state):
    if random.uniform(0, 1) < epsilon:
        # Choose a random action to explore
        return random.randint(0, len(actions) - 1)
    else:
        # Choose the action with the highest Q-value
        return np.argmax(Q_table[state[0]][state[1]])

# Define function to update Q-table using Q-Learning
def update_q_table(state, action, next_state, reward):
    best_next_action = np.argmax(Q_table[next_state[0]][next_state[1]])
```

```
    Q_table[state[0]][state[1]][action] += learning_rate * (reward +
discount_factor * Q_table[next_state[0]][next_state[1]][best_next_action] -
Q_table[state[0]][state[1]][action])


# Define function to perform Q-learning
def Q_learning():


    # Initiate an empty list to store the total reward in each episode
    rewards_in_each_episode = []


    for episode in range(num_episodes):
        state = start_position
        total_reward = 0


        for step in range(max_steps_per_episode):
            action = choose_action(state)
            next_state = (state[0] + actions[action][0], state[1] + actions[action][1])


            # Check if the state is valid
            # If the chosen action will move outside the grid or hit the shelve, it is a
invalid state
            if next_state[0] < 0 or next_state[0] >= 10 or next_state[1] < 0 or
next_state[1] >= 10 or
simulated_warehouse_environment[next_state[0]][next_state[1]] == -100:


                # Stay in the same grid or state
                next_state = state


            # Check if the agent arrives to the target destination
            # If yes, give the agent a reward of 100
            if next_state == target_destination:
                reward = 100
```

```
            total_reward += reward
            update_q_table(state, action, next_state, reward)
            break

        else:
            # Transit to the next state
            reward = -1
            total_reward += reward
            update_q_table(state, action, next_state, reward)
            state = next_state

    rewards_in_each_episode.append(total_reward)
    print(f"Episode {episode + 1}: Total Reward = {total_reward}")

  return rewards_in_each_episode

# Run Q-learning and collect the rewards
rewards_in_each_episode_alpha2 = Q_learning()

# Plot the learning curve (total reward accumulated per episode)
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_alpha2, label =
f'learning rate: {learning_rate} , discount factor: {discount_factor} , epsilon:
{epsilon}')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend(fontsize = '8')
plt.title('Total Reward per Episode')
plt.grid(True)
plt.show()

#### Learning rate = 0.9
```

```
# Define constants variable
start_position = (9, 0)
target_destination = (0, 9)
num_episodes = 1000
max_steps_per_episode = 1000
learning_rate = 0.9
discount_factor = 0.9
epsilon = 0.1

# Define the foun actions which are up, down, left, right
# Up = (-1, 0) as moving up will decrease the row index
# Down = (1, 0) as moving up will increase the row index
# Left = (0, -1) as moving up will decrease the column index
# Right = (0, 1) as moving up will increase the column index
actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Initialize Q-table to store the estimated values of each action for each state in the warehouse
Q_table = np.zeros((10, 10, 4))

# Define function to choose action by using epsilon-greedy
def choose_action(state):
    if random.uniform(0, 1) < epsilon:
        # Choose a random action to explore
        return random.randint(0, len(actions) - 1)
    else:
        # Choose the action with the highest Q-value
        return np.argmax(Q_table[state[0]][state[1]])

# Define function to update Q-table using Q-Learning
```

```
def update_q_table(state, action, next_state, reward):
    best_next_action = np.argmax(Q_table[next_state[0]][next_state[1]])
    Q_table[state[0]][state[1]][action] += learning_rate * (reward + discount_factor * Q_table[next_state[0]][next_state[1]][best_next_action] - Q_table[state[0]][state[1]][action])


# Define function to perform Q-learning
def Q_learning():


    # Initiate an empty list to store the total reward in each episode
    rewards_in_each_episode = []


    for episode in range(num_episodes):
        state = start_position
        total_reward = 0


        for step in range(max_steps_per_episode):
            action = choose_action(state)
            next_state = (state[0] + actions[action][0], state[1] + actions[action][1])


            # Check if the state is valid
            # If the chosen action will move outside the grid or hit the shelve, it is a invalid state
            if next_state[0] < 0 or next_state[0] >= 10 or next_state[1] < 0 or next_state[1] >= 10 or simulated_warehouse_environment[next_state[0]][next_state[1]] == -100:


                # Stay in the same grid or state
                next_state = state


            # Check if the agent arrives to the target destination
            # If yes, give the agent a reward of 100
```

```
            if next_state == target_destination:
                reward = 100
                total_reward += reward
                update_q_table(state, action, next_state, reward)
                break

            else:
                # Transit to the next state
                reward = -1
                total_reward += reward
                update_q_table(state, action, next_state, reward)
                state = next_state

        rewards_in_each_episode.append(total_reward)
        print(f"Episode {episode + 1}: Total Reward = {total_reward}")

    return rewards_in_each_episode

# Run Q-learning and collect the rewards
rewards_in_each_episode_alpha3 = Q_learning()

# Plot the learning curve (total reward accumulated per episode)
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_alpha3, label =
f'learning rate: {learning_rate} , discount factor: {discount_factor} , epsilon:
{epsilon}')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend(fontsize = '8')
plt.title('Total Reward per Episode')
plt.grid(True)
plt.show()
```

Plot all these three together for the ease of comparison.

```
# Plot the learning curve (total reward accumulated per episode)
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_alpha1, label =
'learning rate: 0.01, discount factor: 0.9, epsilon: 0.1')
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_alpha2, label =
'learning rate: 0.1, discount factor: 0.9, epsilon: 0.1')
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_alpha3, label =
'learning rate: 0.9, discount factor: 0.9, epsilon: 0.1')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend(fontsize = '8')
plt.title('Total Reward per Episode')
plt.grid(True)
plt.show()
```

Experiment wth different discount factor or gamma.

Discount factor = 0.1.

```
# Define constants variable
start_position = (9, 0)
target_destination = (0, 9)
num_episodes = 1000
max_steps_per_episode = 1000
learning_rate = 0.1
discount_factor = 0.1
epsilon = 0.1
# Define the foun actions which are up, down, left, right
```

```
# Up = (-1, 0) as moving up will decrease the row index
# Down = (1, 0) as moving up will increase the row index
# Left = (0, -1) as moving up will decrease the column index
# Right = (0, 1) as moving up will increase the column index
actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Initialize Q-table to store the estimated values of each action for each state in the warehouse
Q_table = np.zeros((10, 10, 4))

# Define function to choose action by using epsilon-greedy
def choose_action(state):
    if random.uniform(0, 1) < epsilon:
        # Choose a random action to explore
        return random.randint(0, len(actions) - 1)
    else:
        # Choose the action with the highest Q-value
        return np.argmax(Q_table[state[0]][state[1]])

# Define function to update Q-table using Q-Learning
def update_q_table(state, action, next_state, reward):
    best_next_action = np.argmax(Q_table[next_state[0]][next_state[1]])
    Q_table[state[0]][state[1]][action] += learning_rate * (reward + discount_factor * Q_table[next_state[0]][next_state[1]][best_next_action] - Q_table[state[0]][state[1]][action])

# Define function to perform Q-learning
def Q_learning():

    # Initiate an empty list to store the total reward in each episode
    rewards_in_each_episode = []
```

```
    for episode in range(num_episodes):
        state = start_position
        total_reward = 0

        for step in range(max_steps_per_episode):
            action = choose_action(state)
            next_state = (state[0] + actions[action][0], state[1] + actions[action][1])

            # Check if the state is valid
            # If the chosen action will move outside the grid or hit the shelve, it is a invalid state
            if next_state[0] < 0 or next_state[0] >= 10 or next_state[1] < 0 or next_state[1] >= 10 or
simulated_warehouse_environment[next_state[0]][next_state[1]] == -100:

                # Stay in the same grid or state
                next_state = state

            # Check if the agent arrives to the target destination
            # If yes, give the agent a reward of 100
            if next_state == target_destination:
                reward = 100
                total_reward += reward
                update_q_table(state, action, next_state, reward)
                break

            else:
                # Transit to the next state
                reward = -1
                total_reward += reward
```

```
            update_q_table(state, action, next_state, reward)
            state = next_state

        rewards_in_each_episode.append(total_reward)
        print(f"Episode {episode + 1}: Total Reward = {total_reward}")

    return rewards_in_each_episode

# Run Q-learning and collect the rewards
rewards_in_each_episode_gamma1 = Q_learning()

# Plot the learning curve (total reward accumulated per episode)
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_gamma1, label =
f'learning rate: {learning_rate} , discount factor: {discount_factor} , epsilon:
{epsilon}')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend(fontsize = '8')
plt.title('Total Reward per Episode')
plt.grid(True)
plt.show()
```

Discount factor = 0.5.

```
# Define constants variable
start_position = (9, 0)
target_destination = (0, 9)
num_episodes = 1000
max_steps_per_episode = 1000
learning_rate = 0.1
discount_factor = 0.5
```

```
epsilon = 0.1


# Define the foun actions which are up, down, left, right
# Up = (-1, 0) as moving up will decrease the row index
# Down = (1, 0) as moving up will increase the row index
# Left = (0, -1) as moving up will decrease the column index
# Right = (0, 1) as moving up will increase the column index
actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]


# Initialize Q-table to store the estimated values of each action for each state in the warehouse
Q_table = np.zeros((10, 10, 4))


# Define function to choose action by using epsilon-greedy
def choose_action(state):
    if random.uniform(0, 1) < epsilon:
        # Choose a random action to explore
        return random.randint(0, len(actions) - 1)
    else:
        # Choose the action with the highest Q-value
        return np.argmax(Q_table[state[0]][state[1]])


# Define function to update Q-table using Q-Learning
def update_q_table(state, action, next_state, reward):
    best_next_action = np.argmax(Q_table[next_state[0]][next_state[1]])
    Q_table[state[0]][state[1]][action] += learning_rate * (reward + discount_factor * Q_table[next_state[0]][next_state[1]][best_next_action] - Q_table[state[0]][state[1]][action])


# Define function to perform Q-learning
def Q_learning():
```

```
    # Initiate an empty list to store the total reward in each episode
    rewards_in_each_episode = []

    for episode in range(num_episodes):
        state = start_position
        total_reward = 0

        for step in range(max_steps_per_episode):
            action = choose_action(state)
            next_state = (state[0] + actions[action][0], state[1] + actions[action][1])

            # Check if the state is valid
            # If the chosen action will move outside the grid or hit the shelve, it is a
invalid state
            if next_state[0] < 0 or next_state[0] >= 10 or next_state[1] < 0 or
next_state[1] >= 10 or
simulated_warehouse_environment[next_state[0]][next_state[1]] == -100:

                # Stay in the same grid or state
                next_state = state

            # Check if the agent arrives to the target destination
            # If yes, give the agent a reward of 100
            if next_state == target_destination:
                reward = 100
                total_reward += reward
                update_q_table(state, action, next_state, reward)
                break

            else:
```

```
            # Transit to the next state
            reward = -1
            total_reward += reward
            update_q_table(state, action, next_state, reward)
            state = next_state

        rewards_in_each_episode.append(total_reward)
        print(f"Episode {episode + 1}: Total Reward = {total_reward}")

    return rewards_in_each_episode

# Run Q-learning and collect the rewards
rewards_in_each_episode_gamma2 = Q_learning()

# Plot the learning curve (total reward accumulated per episode)
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_gamma2, label =
f'learning rate: {learning_rate} , discount factor: {discount_factor} , epsilon:
{epsilon}')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend(fontsize = '8')
plt.title('Total Reward per Episode')
plt.grid(True)
plt.show()
```

Discount factor = 0.9.

```
# Define constants variable
start_position = (9, 0)
target_destination = (0, 9)
num_episodes = 1000
```

```
max_steps_per_episode = 1000
learning_rate = 0.1
discount_factor = 0.9
epsilon = 0.1

# Define the foun actions which are up, down, left, right
# Up = (-1, 0) as moving up will decrease the row index
# Down = (1, 0) as moving up will increase the row index
# Left = (0, -1) as moving up will decrease the column index
# Right = (0, 1) as moving up will increase the column index
actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Initialize Q-table to store the estimated values of each action for each state in the warehouse
Q_table = np.zeros((10, 10, 4))

# Define function to choose action by using epsilon-greedy
def choose_action(state):
    if random.uniform(0, 1) < epsilon:
        # Choose a random action to explore
        return random.randint(0, len(actions) - 1)
    else:
        # Choose the action with the highest Q-value
        return np.argmax(Q_table[state[0]][state[1]])

# Define function to update Q-table using Q-Learning
def update_q_table(state, action, next_state, reward):
    best_next_action = np.argmax(Q_table[next_state[0]][next_state[1]])
    Q_table[state[0]][state[1]][action] += learning_rate * (reward + discount_factor * Q_table[next_state[0]][next_state[1]][best_next_action] - Q_table[state[0]][state[1]][action])
```

```
# Define function to perform Q-learning
def Q_learning():

    # Initiate an empty list to store the total reward in each episode
    rewards_in_each_episode = []

    for episode in range(num_episodes):
        state = start_position
        total_reward = 0

        for step in range(max_steps_per_episode):
            action = choose_action(state)
            next_state = (state[0] + actions[action][0], state[1] + actions[action][1])

            # Check if the state is valid
            # If the chosen action will move outside the grid or hit the shelve, it is a
invalid state
            if next_state[0] < 0 or next_state[0] >= 10 or next_state[1] < 0 or
next_state[1] >= 10 or
simulated_warehouse_environment[next_state[0]][next_state[1]] == -100:

                # Stay in the same grid or state
                next_state = state

            # Check if the agent arrives to the target destination
            # If yes, give the agent a reward of 100
            if next_state == target_destination:
                reward = 100
                total_reward += reward
                update_q_table(state, action, next_state, reward)
```

```
            break

        else:
            # Transit to the next state
            reward = -1
            total_reward += reward
            update_q_table(state, action, next_state, reward)
            state = next_state

    rewards_in_each_episode.append(total_reward)
    print(f"Episode {episode + 1}: Total Reward = {total_reward}")

  return rewards_in_each_episode

# Run Q-learning and collect the rewards
rewards_in_each_episode_gamma3 = Q_learning()

# Plot the learning curve (total reward accumulated per episode)
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_gamma3, label =
f'learning rate: {learning_rate} , discount factor: {discount_factor} , epsilon:
{epsilon}')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend(fontsize = '8')
plt.title('Total Reward per Episode')
plt.grid(True)
plt.show()

#### Plot all these three together for the ease of comparison.

# Plot the learning curve (total reward accumulated per episode)
```

```
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_gamma1, label = 'learning rate: 0.1, discount factor: 0.1, epsilon: 0.1')

plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_gamma2, label = 'learning rate: 0.1, discount factor: 0.5, epsilon: 0.1')

plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_gamma3, label = 'learning rate: 0.1, discount factor: 0.9, epsilon: 0.1')

plt.xlabel('Episode')

plt.ylabel('Total Reward')

plt.legend(fontsize = '8', loc = 'lower right')

plt.title('Total Reward per Episode')

plt.grid(True)

plt.show()


### Experiment wth different epsilon.


#### Epsilon = 0.1.


# Define constants variable

start_position = (9, 0)

target_destination = (0, 9)

num_episodes = 1000

max_steps_per_episode = 1000

learning_rate = 0.1

discount_factor = 0.9

epsilon = 0.1


# Define the foun actions which are up, down, left, right

# Up = (-1, 0) as moving up will decrease the row index

# Down = (1, 0) as moving up will increase the row index

# Left = (0, -1) as moving up will decrease the column index

# Right = (0, 1) as moving up will increase the column index
```

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


# Initialize Q-table to store the estimated values of each action for each state in the warehouse
Q_table = np.zeros((10, 10, 4))


# Define function to choose action by using epsilon-greedy
def choose_action(state):
    if random.uniform(0, 1) < epsilon:
        # Choose a random action to explore
        return random.randint(0, len(actions) - 1)
    else:
        # Choose the action with the highest Q-value
        return np.argmax(Q_table[state[0]][state[1]])


# Define function to update Q-table using Q-Learning
def update_q_table(state, action, next_state, reward):
    best_next_action = np.argmax(Q_table[next_state[0]][next_state[1]])
    Q_table[state[0]][state[1]][action] += learning_rate * (reward + discount_factor * Q_table[next_state[0]][next_state[1]][best_next_action] - Q_table[state[0]][state[1]][action])


# Define function to perform Q-learning
def Q_learning():


    # Initiate an empty list to store the total reward in each episode
    rewards_in_each_episode = []


    for episode in range(num_episodes):
        state = start_position
        total_reward = 0
```

```
    for step in range(max_steps_per_episode):
        action = choose_action(state)
        next_state = (state[0] + actions[action][0], state[1] + actions[action][1])

        # Check if the state is valid
        # If the chosen action will move outside the grid or hit the shelve, it is a
invalid state
        if next_state[0] < 0 or next_state[0] >= 10 or next_state[1] < 0 or
next_state[1] >= 10 or
simulated_warehouse_environment[next_state[0]][next_state[1]] == -100:

            # Stay in the same grid or state
            next_state = state

        # Check if the agent arrives to the target destination
        # If yes, give the agent a reward of 100
        if next_state == target_destination:
            reward = 100
            total_reward += reward
            update_q_table(state, action, next_state, reward)
            break

        else:
            # Transit to the next state
            reward = -1
            total_reward += reward
            update_q_table(state, action, next_state, reward)
            state = next_state

    rewards_in_each_episode.append(total_reward)
```

```
        print(f"Episode {episode + 1}: Total Reward = {total_reward}")


    return rewards_in_each_episode

# Run Q-learning and collect the rewards
rewards_in_each_episode_eps1 = Q_learning()

# Plot the learning curve (total reward accumulated per episode)
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_eps1, label =
f'learning rate: {learning_rate} , discount factor: {discount_factor} , epsilon:
{epsilon}')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend(fontsize = '8')
plt.title('Total Reward per Episode')
plt.grid(True)
plt.show()

#### Epsilon = 0.5.

# Define constants variable
start_position = (9, 0)
target_destination = (0, 9)
num_episodes = 1000
max_steps_per_episode = 1000
learning_rate = 0.1
discount_factor = 0.9
epsilon = 0.5

# Define the foun actions which are up, down, left, right
# Up = (-1, 0) as moving up will decrease the row index
```

```
# Down = (1, 0) as moving up will increase the row index
# Left = (0, -1) as moving up will decrease the column index
# Right = (0, 1) as moving up will increase the column index
actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Initialize Q-table to store the estimated values of each action for each state in the warehouse
Q_table = np.zeros((10, 10, 4))

# Define function to choose action by using epsilon-greedy
def choose_action(state):
    if random.uniform(0, 1) < epsilon:
        # Choose a random action to explore
        return random.randint(0, len(actions) - 1)
    else:
        # Choose the action with the highest Q-value
        return np.argmax(Q_table[state[0]][state[1]])

# Define function to update Q-table using Q-Learning
def update_q_table(state, action, next_state, reward):
    best_next_action = np.argmax(Q_table[next_state[0]][next_state[1]])
    Q_table[state[0]][state[1]][action] += learning_rate * (reward + discount_factor * Q_table[next_state[0]][next_state[1]][best_next_action] - Q_table[state[0]][state[1]][action])

# Define function to perform Q-learning
def Q_learning():

    # Initiate an empty list to store the total reward in each episode
    rewards_in_each_episode = []
```

```
    for episode in range(num_episodes):
        state = start_position
        total_reward = 0

        for step in range(max_steps_per_episode):
            action = choose_action(state)
            next_state = (state[0] + actions[action][0], state[1] + actions[action][1])

            # Check if the state is valid
            # If the chosen action will move outside the grid or hit the shelve, it is a
invalid state
            if next_state[0] < 0 or next_state[0] >= 10 or next_state[1] < 0 or
next_state[1] >= 10 or
simulated_warehouse_environment[next_state[0]][next_state[1]] == -100:

                # Stay in the same grid or state
                next_state = state

            # Check if the agent arrives to the target destination
            # If yes, give the agent a reward of 100
            if next_state == target_destination:
                reward = 100
                total_reward += reward
                update_q_table(state, action, next_state, reward)
                break

            else:
                # Transit to the next state
                reward = -1
                total_reward += reward
                update_q_table(state, action, next_state, reward)
```

```
            state = next_state

        rewards_in_each_episode.append(total_reward)
        print(f"Episode {episode + 1}: Total Reward = {total_reward}")

    return rewards_in_each_episode

# Run Q-learning and collect the rewards
rewards_in_each_episode_eps2 = Q_learning()

# Plot the learning curve (total reward accumulated per episode)
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_eps2, label =
f'learning rate: {learning_rate} , discount factor: {discount_factor} , epsilon:
{epsilon}')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend(fontsize = '8')
plt.title('Total Reward per Episode')
plt.grid(True)
plt.show()

#### Epsilon = 0.9.

# Define constants variable
start_position = (9, 0)
target_destination = (0, 9)
num_episodes = 1000
max_steps_per_episode = 1000
learning_rate = 0.1
discount_factor = 0.9
epsilon = 0.9
```

```
# Define the foun actions which are up, down, left, right
# Up = (-1, 0) as moving up will decrease the row index
# Down = (1, 0) as moving up will increase the row index
# Left = (0, -1) as moving up will decrease the column index
# Right = (0, 1) as moving up will increase the column index
actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Initialize Q-table to store the estimated values of each action for each state in the warehouse
Q_table = np.zeros((10, 10, 4))

# Define function to choose action by using epsilon-greedy
def choose_action(state):
    if random.uniform(0, 1) < epsilon:
        # Choose a random action to explore
        return random.randint(0, len(actions) - 1)
    else:
        # Choose the action with the highest Q-value
        return np.argmax(Q_table[state[0]][state[1]])

# Define function to update Q-table using Q-Learning
def update_q_table(state, action, next_state, reward):
    best_next_action = np.argmax(Q_table[next_state[0]][next_state[1]])
    Q_table[state[0]][state[1]][action] += learning_rate * (reward + discount_factor * Q_table[next_state[0]][next_state[1]][best_next_action] - Q_table[state[0]][state[1]][action])

# Define function to perform Q-learning
def Q_learning():
```

```
# Initiate an empty list to store the total reward in each episode
rewards_in_each_episode = []

for episode in range(num_episodes):
    state = start_position
    total_reward = 0

    for step in range(max_steps_per_episode):
        action = choose_action(state)
        next_state = (state[0] + actions[action][0], state[1] + actions[action][1])

        # Check if the state is valid
        # If the chosen action will move outside the grid or hit the shelve, it is a
invalid state
        if next_state[0] < 0 or next_state[0] >= 10 or next_state[1] < 0 or
next_state[1] >= 10 or
simulated_warehouse_environment[next_state[0]][next_state[1]] == -100:

            # Stay in the same grid or state
            next_state = state

        # Check if the agent arrives to the target destination
        # If yes, give the agent a reward of 100
        if next_state == target_destination:
            reward = 100
            total_reward += reward
            update_q_table(state, action, next_state, reward)
            break

        else:
            # Transit to the next state
```

```
            reward = -1
            total_reward += reward
            update_q_table(state, action, next_state, reward)
            state = next_state

        rewards_in_each_episode.append(total_reward)
        print(f"Episode {episode + 1}: Total Reward = {total_reward}")

    return rewards_in_each_episode

# Run Q-learning and collect the rewards
rewards_in_each_episode_eps3 = Q_learning()

# Plot the learning curve (total reward accumulated per episode)
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_eps3, label =
f'learning rate: {learning_rate} , discount factor: {discount_factor} , epsilon:
{epsilon}')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend(fontsize = '8')
plt.title('Total Reward per Episode')
plt.grid(True)
plt.show()

#### Plot all these three together for the ease of comparison.

# Plot the learning curve (total reward accumulated per episode)
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_eps1, label =
'learning rate: 0.1, discount factor: 0.9, epsilon: 0.1')
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_eps2, label =
'learning rate: 0.1, discount factor: 0.9, epsilon: 0.5')
plt.plot(range(1, num_episodes + 1), rewards_in_each_episode_eps3, label =
```

```
'learning rate: 0.1, discount factor: 0.9, epsilon: 0.9')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend(fontsize = '8', loc = 'lower right')
plt.title('Total Reward per Episode')
plt.grid(True)
plt.show()

## Advance

#### Comparison of DQN and DDQN

# Define the neural network for both DQN and DDQN
class DQN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.out(x)
        return x

# Define experience replay memory
class ReplayMemory():
    def __init__(self, maxlen):
        self.memory = deque([], maxlen=maxlen)

    def append(self, transition):
```

```
        self.memory.append(transition)

    def sample(self, sample_size):
        return random.sample(self.memory, sample_size)

    def __len__(self):
        return len(self.memory)

# Define FrozenLake Deep Q-Learning class
class FrozenLakeDQN():

    # Initiate the parameters
    learning_rate_a = 0.1
    discount_factor_g = 0.9
    network_sync_rate = 10
    replay_memory_size = 1000
    mini_batch_size = 32
    loss_fn = nn.MSELoss()
    optimizer = None
    ACTIONS = ['L', 'D', 'R', 'U']

    def __init__(self, is_ddqn=False):
        self.is_ddqn = is_ddqn

    def train(self, episodes, render=False, is_slippery=False):
        env = gym.make('FrozenLake-v1', map_name="4x4",
is_slippery=is_slippery, render_mode='human' if render else None)
        num_states = env.observation_space.n
        num_actions = env.action_space.n
        epsilon = 1
        memory = ReplayMemory(self.replay_memory_size)
```

```
        policy_dqn = DQN(input_size=num_states, hidden_size=num_states,
output_size=num_actions)
        target_dqn = DQN(input_size=num_states, hidden_size=num_states,
output_size=num_actions)
        target_dqn.load_state_dict(policy_dqn.state_dict())
        self.optimizer = torch.optim.Adam(policy_dqn.parameters(),
lr=self.learning_rate_a)
        rewards_per_episode = np.zeros(episodes)
        epsilon_history = []
        step_count = 0

        for i in range(episodes):
            state = env.reset()[0]
            terminated = False
            truncated = False

            while not terminated and not truncated:
                if random.random() < epsilon:
                    action = env.action_space.sample()
                else:
                    with torch.no_grad():
                        action = policy_dqn(self.state_to_dqn_input(state,
num_states)).argmax().item()

                new_state, reward, terminated, truncated, _ = env.step(action)
                memory.append((state, action, new_state, reward, terminated))
                state = new_state
                step_count += 1

            if reward == 1:
                rewards_per_episode[i] = 1
```

```
        if len(memory) > self.mini_batch_size and np.sum(rewards_per_episode)
> 0:
            mini_batch = memory.sample(self.mini_batch_size)
            self.optimize(mini_batch, policy_dqn, target_dqn)
            epsilon = max(epsilon - 1/episodes, 0)
            epsilon_history.append(epsilon)

            if step_count > self.network_sync_rate:
                target_dqn.load_state_dict(policy_dqn.state_dict())
                step_count = 0

    env.close()
    torch.save(policy_dqn.state_dict(), "frozen_lake_dql.pt")
    sum_rewards = np.zeros(episodes)
    for x in range(episodes):
        sum_rewards[x] = np.sum(rewards_per_episode[max(0, x-100):(x+1)])
    plt.plot(sum_rewards)

  def optimize(self, mini_batch, policy_dqn, target_dqn):
    num_states = policy_dqn.fc1.in_features
    current_q_list = []
    target_q_list = []

    for state, action, new_state, reward, terminated in mini_batch:
        if terminated:
            target = torch.FloatTensor([reward])
        else:
            with torch.no_grad():
                target = torch.FloatTensor(reward + self.discount_factor_g *
target_dqn(self.state_to_dqn_input(new_state, num_states)).max())
```

```
            current_q = policy_dqn(self.state_to_dqn_input(state, num_states))
            current_q_list.append(current_q)
            target_q = target_dqn(self.state_to_dqn_input(state, num_states))
            target_q[action] = target
            target_q_list.append(target_q)

        loss = self.loss_fn(torch.stack(current_q_list), torch.stack(target_q_list))
        self.optimizer.zero_grad()
        loss.backward()

    def state_to_dqn_input(self, state:int, num_states:int)->torch.Tensor:
        input_tensor = torch.zeros(num_states)
        input_tensor[state] = 1
        return input_tensor

    def test(self, episodes, is_slippery=False):
        env = gym.make('FrozenLake-v1', map_name="4x4",
is_slippery=is_slippery, render_mode='human')
        num_states = env.observation_space.n
        num_actions = env.action_space.n
        policy_dqn = DQN(input_size=num_states, hidden_size=num_states,
output_size=num_actions)
        policy_dqn.load_state_dict(torch.load("frozen_lake_dql.pt"))
        policy_dqn.eval()

        for i in range(episodes):
            state = env.reset()[0]
            terminated = False
            truncated = False

            while not terminated and not truncated:
```

```
            with torch.no_grad():
                action = policy_dqn(self.state_to_dqn_input(state,
num_states)).argmax().item()

            state, reward, terminated, truncated, _ = env.step(action)

        env.close()

if __name__ == '__main__':
    frozen_lake_dqn = FrozenLakeDQN()
    frozen_lake_ddqn = FrozenLakeDQN(is_ddqn=True)

    is_slippery = False
    episodes = 10000

    plt.figure(figsize=(10, 5))
    plt.title('Comparison of DQN and DDQN on FrozenLake')
    plt.xlabel('Episodes')
    plt.ylabel('Average rewards')

    frozen_lake_dqn.train(episodes, is_slippery=is_slippery)
    frozen_lake_ddqn.train(episodes, is_slippery=is_slippery)

    plt.legend(['DQN', 'DDQN'])
    plt.show()

#### Comparison of DQN with and without experience replay

# Define the neural network for both DQN
class DQN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
```

```
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.out(x)
        return x

# Define experience replay memory
class ReplayMemory():
    def __init__(self, maxlen):
        self.memory = deque([], maxlen=maxlen)

    def append(self, transition):
        self.memory.append(transition)

    def sample(self, sample_size):
        return random.sample(self.memory, sample_size)

    def __len__(self):
        return len(self.memory)

class FrozenLakeDQN():

    learning_rate_a = 0.1
    discount_factor_g = 0.9
    network_sync_rate = 10
    replay_memory_size = 1000
    mini_batch_size = 32
```

```
    # Neural Network
    loss_fn = nn.MSELoss()
    optimizer = None
    ACTIONS = ['L','D','R','U']

    def __init__(self, use_experience_replay=True):
        self.use_experience_replay = use_experience_replay
        if self.use_experience_replay:
            self.replay_memory = ReplayMemory(self.replay_memory_size)
        self.rewards_per_episode = []

    def train(self, episodes, is_slippery=False):
        env = gym.make('FrozenLake-v1', map_name="4x4",
is_slippery=is_slippery)
        num_states = env.observation_space.n
        num_actions = env.action_space.n

        epsilon = 1

        policy_dqn = DQN(input_size=num_states, hidden_size=num_states,
output_size=num_actions)
        target_dqn = DQN(input_size=num_states, hidden_size=num_states,
output_size=num_actions)
        target_dqn.load_state_dict(policy_dqn.state_dict())

        self.optimizer = torch.optim.Adam(policy_dqn.parameters(),
lr=self.learning_rate_a)

        rewards_per_episode = np.zeros(episodes)
        step_count = 0
```

```
        for i in range(episodes):
            state = env.reset()[0]
            terminated = False
            truncated = False

            while not terminated and not truncated:
                if random.random() < epsilon:
                    action = env.action_space.sample()
                else:
                    with torch.no_grad():
                        action = policy_dqn(self.state_to_dqn_input(state,
num_states)).argmax().item()

                new_state, reward, terminated, truncated, _ = env.step(action)

                if self.use_experience_replay:
                    self.replay_memory.append((state, action, new_state, reward,
terminated))
                    if len(self.replay_memory) > self.mini_batch_size:
                        mini_batch = self.replay_memory.sample(self.mini_batch_size)
                        self.optimize(mini_batch, policy_dqn, target_dqn)

                else:
                    if terminated:
                        target = torch.FloatTensor([reward])
                    else:
                        with torch.no_grad():
                            target = torch.FloatTensor(
                                reward + self.discount_factor_g *
target_dqn(self.state_to_dqn_input(new_state, num_states)).max()
                            )
```

```
                current_q = policy_dqn(self.state_to_dqn_input(state, num_states))
                target_q = target_dqn(self.state_to_dqn_input(state, num_states))
                target_q[action] = target

                loss = self.loss_fn(current_q, target_q)

                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()

            state = new_state
            step_count += 1

        if reward == 1:
            rewards_per_episode[i] = 1

        if not self.use_experience_replay:
            if step_count > self.network_sync_rate:
                target_dqn.load_state_dict(policy_dqn.state_dict())
                step_count = 0

        epsilon = max(epsilon - 1/episodes, 0)
        self.rewards_per_episode.append(reward)

    env.close()

    return rewards_per_episode

def optimize(self, mini_batch, policy_dqn, target_dqn):
```

```
        num_states = policy_dqn.fc1.in_features

        current_q_list = []
        target_q_list = []

        for state, action, new_state, reward, terminated in mini_batch:
            if terminated:
                target = torch.FloatTensor([reward])
            else:
                with torch.no_grad():
                    target = torch.FloatTensor(
                        reward + self.discount_factor_g *
target_dqn(self.state_to_dqn_input(new_state, num_states)).max()
                    )

            current_q = policy_dqn(self.state_to_dqn_input(state, num_states))
            current_q_list.append(current_q)

            target_q = target_dqn(self.state_to_dqn_input(state, num_states))
            target_q[action] = target
            target_q_list.append(target_q)

        loss = self.loss_fn(torch.stack(current_q_list), torch.stack(target_q_list))

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    def state_to_dqn_input(self, state:int, num_states:int)->torch.Tensor:
        input_tensor = torch.zeros(num_states)
        input_tensor[state] = 1
```

```
        return input_tensor


if __name__ == '__main__':
    num_episodes = 1000


    frozen_lake_dqn_with_replay =
FrozenLakeDQN(use_experience_replay=True)
    frozen_lake_dqn_no_replay = FrozenLakeDQN(use_experience_replay=False)


    # Train DQN with experience replay
    rewards_with_replay = frozen_lake_dqn_with_replay.train(num_episodes,
is_slippery=False)
    # Train DQN without experience replay
    rewards_no_replay = frozen_lake_dqn_no_replay.train(num_episodes,
is_slippery=False)


    plt.figure(figsize=(10, 5))
    plt.plot(np.arange(num_episodes), rewards_with_replay, label='With
Experience Replay')
    plt.plot(np.arange(num_episodes), rewards_no_replay, label='Without
Experience Replay')
    plt.title('Comparison of DQN with and without Experience Replay')
    plt.xlabel('Episodes')
    plt.ylabel('Total Rewards per Episode')
    plt.legend()
    plt.show()



### PPO


# Create the environment
def create_env(config):
```

```
    return EnvContext(env_name="CartPole-v1")


# Train the agent with performance logging
ppo_config = PPOConfig()
ppo_config = ppo_config.training(gamma = 0.9, lr = 0.1, kl_coeff = 0.3,
train_batch_size = 128)
ppo_config = ppo_config.resources(num_gpus = 0)
ppo_config = ppo_config.env_runners(num_env_runners = 1)
ppo_algo = ppo_config.build(env = "CartPole-v1")


ppo_episode_rewards = []
for i in range(100):
    ppo_result = ppo_algo.train()
    ppo_episode_rewards.append(ppo_result["episode_reward_mean"])
    print(f"Episode {i+1}: Reward Mean =
{ppo_result['episode_reward_mean']}")


ppo_algo.stop()


plt.plot(ppo_episode_rewards)
plt.xlabel("Episode")
plt.ylabel("Average Reward")
plt.title("PPO Performance on CartPole-v1")
plt.show()
```