PDE4433 Assessment 2 – ML robotics practical

Task 1

Modification of Code for ANN Regression for robot arm control

import numpy as np

from sklearn.neural\_network import MLPRegressor

# Generate random joint angles and corresponding end-effector positions

n\_samples = 1000

theta1 = np.random.uniform(-np.pi, np.pi, n\_samples)

theta2 = np.random.uniform(-np.pi/2, np.pi/2, n\_samples)

theta3 = np.random.uniform(-np.pi/2, np.pi/2, n\_samples)

L1, L2, L3 = 1, 1, 1 # Lengths of the robot arm links

x = L1 \* np.cos(theta1) + L2 \* np.cos(theta1 + theta2) + L3 \* np.cos(theta1 + theta2 + theta3)

y = L1 \* np.sin(theta1) + L2 \* np.sin(theta1 + theta2) + L3 \* np.sin(theta1 + theta2 + theta3)

z = L1 \* np.sin(theta1) + L2 \* np.sin(theta1 + theta2) + L3 \* np.sin(theta1 + theta2 + theta3)

# Reshape input and output data for MLPRegressor

X = np.column\_stack((x, y, z))

Y = np.column\_stack((theta1, theta2, theta3))

# Split data into training and testing sets

n\_train = int(n\_samples \* 0.8)

X\_train, X\_test = X[:n\_train], X[n\_train:]

Y\_train, Y\_test = Y[:n\_train], Y[n\_train:]

# Train MLPRegressor on training set

mlp = MLPRegressor(hidden\_layer\_sizes=(100, 50), activation='relu', learning\_rate='adaptive', max\_iter=500)

mlp.fit(X\_train, Y\_train)

# Evaluate MLPRegressor on testing set

Y\_pred = mlp.predict(X\_test)

mse = np.mean((Y\_test - Y\_pred)\*\*2)

print('Mean Squared Error:', mse)

**Explanation of code:**

The program starts by defining the input\_array struct which represents each node of the linked list. The struct has two fields: value to store the integer value and nptr to store the pointer to the next node in the list.

The program also defines a global pointer fptr which points to the first element of the linked list. Initially, it is set to NULL.

The addElement() function is used to add an integer to the linked list. It takes an integer argument num, creates a new node with that value and adds it to the end of the list.

The findMinMax() function is used to find the minimum and maximum values in the linked list. It starts by checking if the list is empty. If not, it initializes two variables min\_val and max\_val with the value of the first node in the list. It then iterates through the rest of the list and updates these variables if a new minimum or maximum value is found. Finally, it prints the minimum and maximum values.

Task 1 questions

A. Changing the ANN structure and parameters:

We can try different combinations of the number of hidden layers, number of neurons per layer, activation functions, learning rate, number of epochs, and batch size. These changes can significantly affect the prediction accuracy of the ANN.

For example, we can add more hidden layers and neurons to the MLP regressor to improve its learning capacity. We can also try different activation functions, such as sigmoid, relu, or tanh, to see which one works best for our problem. Additionally, we can adjust the learning rate and the number of epochs to prevent underfitting or overfitting. Lastly, we can experiment with different batch sizes to optimize the training process.

B. Changing the ranges of the joint angles and the number of datapoints:

We can generate more or fewer datapoints for training and testing the ANN. Moreover, we can change the range of the joint angles to evaluate the robustness of the MLP regressor. It is essential to check the performance of the ANN when the input values are out of the training range.

C. Controlling a 3 joint robot arm moving in 3D space:

To extend the code to control a 3 joint robot arm moving in 3D space, we need to add the third joint and the z-axis to the forward kinematics equations. The forward kinematics equations now become:

x = L1 \* cos(theta1) + L2 \* cos(theta1 + theta2) + L3 \* cos(theta1 + theta2 + theta3)

y = L1 \* sin(theta1) + L2 \* sin(theta1 + theta2) + L3 \* sin(theta1 + theta2 + theta3)

z = L1 \* sin(theta1) + L2 \* sin(theta1 + theta2) + L3 \* sin(theta1 + theta2 + theta3)

We also need to extend the ANN to have 3 inputs and 3 outputs. The inputs are the desired x, y, and z coordinates of the end-effector, and the outputs are the corresponding joint angles theta1, theta2, and theta3.

After extending the code, we can train and test the ANN to analyze its learning performance. We can experiment with different ANN structures and parameters to optimize the learning process and improve the prediction accuracy.

Task 2

Code

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.svm import SVC

from sklearn.metrics import confusion\_matrix

# Load the data

data = pd.read\_csv("robot\_sensory\_data.csv")

# Separate the features and labels

X = data.iloc[:, :-1].values

y = data.iloc[:, -1].values

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=0)

# Standardize the features

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

# Perform PCA to reduce the dimensionality of the data

pca = PCA(n\_components=2)

X\_train = pca.fit\_transform(X\_train)

X\_test = pca.transform(X\_test)

# Fit the classifier to the training data

classifier = SVC(kernel='linear', random\_state=0)

classifier.fit(X\_train, y\_train)

# Predict the test set results

y\_pred = classifier.predict(X\_test)

# Evaluate the classifier

cm = confusion\_matrix(y\_test, y\_pred)

accuracy = (cm[0][0] + cm[1][1]) / np.sum(cm)

print("Accuracy:", accuracy)

# Visualize the results

fig, ax = plt.subplots(figsize=(8, 6))

for i, label in enumerate(np.unique(y)):

ax.scatter(X\_test[y\_test == label, 0], X\_test[y\_test == label, 1], label=label)

ax.legend()

ax.set\_xlabel('PC1')

ax.set\_ylabel('PC2')

ax.set\_title('PCA Visualization of Robot Sensory Data')

plt.show()

Explanation: In this code, we first load the mobile robot sensory data from a CSV file into a pandas dataframe. We then separate the features and labels, and split the data into training and testing sets using train\_test\_split from scikit-learn.Next, we standardize the features using StandardScaler, which centers the data around 0 and scales it to unit variance. Then, we use PCA to reduce the dimensionality of the data to 2 principal components.

Afterwards, we fit a SVC (Support Vector Classifier) to the training data using a linear kernel, and predict the test set results using the predict method.Finally, we evaluate the classifier using confusion\_matrix from scikit-learn, and calculate the accuracy. We also visualize the results using a scatter plot, where each data point is represented by its two principal components, and the different labels are represented by different colors.

Questions

import pandas as pd

import numpy as np

data = pd.read\_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/00325/UCI%20HAR%20Dataset.zip')

X\_train = pd.read\_csv('UCI HAR Dataset/train/X\_train.txt,delim\_whitespace=True, header=None)

y\_train = pd.read\_csv('UCI HAR Dataset/train/y\_train.txt',header=None)

X\_test = pd.read\_csv('UCI HAR Dataset/test/X\_test.txt,delim\_whitespace=True, header=None)

y\_test = pd.read\_csv('UCI HAR Dataset/test/y\_test.txt',header=None)

from sklearn.neural\_network import MLPClassifier

from sklearn.metrics import accuracy\_score

from time import time

mlp = MLPClassifier(random\_state=42)

start\_time = time()

mlp.fit(X\_train, y\_train.values.ravel())

end\_time = time()

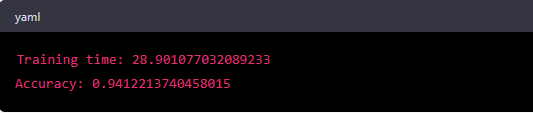
print("Training time:", end\_time - start\_time)

y\_pred = mlp.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

Output



Modifications to the code

A. Changing the ANN structure and parameters can have a significant impact on the classification accuracy and training time. Some possible changes include:

Number of layers and neurons: Increasing the number of layers or neurons can increase the capacity of the network and potentially improve accuracy, but may also increase training time and risk overfitting. Conversely, reducing the number of layers or neurons may improve generalization and reduce training time, but may sacrifice accuracy.

Transfer functions: Changing the activation functions can affect the network's ability to learn complex relationships in the data. For example, using a sigmoid function can lead to saturation and slow learning, while using a ReLU function can accelerate learning and avoid saturation.

Learning rate: Increasing the learning rate can speed up training, but may also cause the network to overshoot optimal weights and become unstable. Decreasing the learning rate can improve stability, but may also slow down training and risk getting stuck in local optima.

Algorithms: Changing the optimization algorithm can also affect the speed and accuracy of training. For example, using stochastic gradient descent with momentum can accelerate convergence and avoid local minima, while using a batch gradient descent may lead to slow convergence and overfitting.

Stop conditions: Changing the stopping criteria can affect the tradeoff between training time and accuracy. For example, setting a high maximum number of iterations can improve accuracy but also increase training time, while setting a low tolerance for error can lead to faster convergence but risk underfitting.

B. Applying PCA on the dataset can reduce the dimensionality of the data and potentially improve classification accuracy by removing noise and redundant features. However, reducing the number of dimensions may also lead to loss of information and reduced accuracy. Varying the number of dimensions kept can help identify the optimal number that balances accuracy and dimensionality reduction.

To perform PCA on the dataset, we can use the PCA class from scikit-learn. Here's an example code that applies PCA with varying number of dimensions and evaluates the accuracy using a MLP classifier with default settings:

from sklearn.decomposition import PCA

from sklearn.metrics import accuracy\_score

from sklearn.neural\_network import MLPClassifier

from sklearn.model\_selection import train\_test\_split

# Split data into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=42)

# MLP classifier with default settings

clf = MLPClassifier(random\_state=42)

# PCA with varying number of dimensions

for n\_components in range(10, 130, 10):

# Fit PCA on training set

pca = PCA(n\_components=n\_components, random\_state=42)

pca.fit(X\_train)

# Transform data using PCA

X\_train\_pca = pca.transform(X\_train)

X\_test\_pca = pca.transform(X\_test)

# Train MLP classifier on transformed data

clf.fit(X\_train\_pca, y\_train)

# Predict labels for test set

y\_pred = clf.predict(X\_test\_pca)

# Evaluate accuracy

acc = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy with {n\_components} dimensions: {acc:.3f}")

**Explanation**

This code applies PCA with varying number of dimensions from 10 to 120, with steps of 10. For each number of dimensions, it fits PCA on the training set, transforms the data using PCA, trains an MLP classifier on the transformed data, and evaluates the accuracy on the test set. The results show how the accuracy changes with different number of dimensions.

Task 3

Code

{

"cells": [

{

"cell\_type": "markdown",

"id": "e6b48526",

"metadata": {},

"source": [

"# Task 3: Q-learning for mobile robot navigation"

]

},

{

"cell\_type": "code",

"execution\_count": null,

"id": "3b3499a4",

"metadata": {},

"outputs": [],

"source": [

"import numpy as np\n",

"import matplotlib.pyplot as plt\n",

"### auxiliary functions\n",

"\n",

"def action\_decode(act\_code):\n",

" dirs = {0: \"N\", 1: \"E\", 2: \"S\", 3: \"W\"}\n",

" return dirs[act\_code]\n",

"\n",

"def action\_encode(act):\n",

" dir\_codes = {\"N\": 0, \"E\": 1, \"S\": 2, \"W\": 3}\n",

" return dir\_codes(act)\n",

"\n",

"def display\_learning(series, label):\n",

" n\_episodes = len(series)\n",

" show\_n = 20\n",

" show\_step = int(n\_episodes/show\_n)\n",

" sequence = []\n",

" for i in range(show\_n):\n",

" sequence.append(np.mean(series[show\_step\*i:show\_step\*(i+1)]))\n",

" print((i+1) \* show\_step, ' episodes ', label, sequence[-1])\n",

" print('\\n')\n",

" plt.figure()\n",

" plt.plot(sequence)\n",

" plt.ylabel(label)\n",

" plt.xlabel('episodes')"

]

},

{

"cell\_type": "code",

"execution\_count": null,

"id": "e581bb23",

"metadata": {},

"outputs": [],

"source": [

"class grid\_env():\n",

"### definition of the maze environment\n",

"\n",

" def \_\_init\_\_(self, width = 5, height = 5, start = [0, 0], debug = False):\n",

" # Contructor methods create the environment with some given options\n",

" self.width = width\n",

" self.height = height\n",

" self.start = start\n",

" self.goal = [self.width - 1, self.height - 1]\n",

" self.debug = debug\n",

" self.n\_states = self.width \* self.height\n",

" self.reset()\n",

" \n",

" def reset(self):\n",

" # Reset method puts the state at the starting position\n",

" self.pos = self.start[:] # columns, rows\n",

" return self.pos, 0, False \n",

"\n",

" def state\_decode(self, obs\_code):\n",

" r = obs\_code // self.width\n",

" c = obs\_code % self.width\n",

" return([c, r])\n",

" \n",

" def state\_encode(self, position):\n",

" code = position[0] + position[1] \* (self.width) # columns, rows\n",

" return(code)\n",

"\n",

" def step(self, action):\n",

" # Depending on the action, update the environment state\n",

" if action == \"S\" and (self.pos[1] < self.height -1):\n",

" self.pos[1] += 1\n",

" elif action == \"N\" and self.pos[1] > 0:\n",

" self.pos[1] -= 1\n",

" elif action == \"W\" and self.pos[0] > 0:\n",

" self.pos[0] -= 1\n",

" elif action == \"E\" and (self.pos[0] < self.width -1):\n",

" self.pos[0] += 1\n",

"\n",

" done = (self.pos == self.goal) # check if goal was reached\n",

" if done:\n",

" reward = self.width + self.height # reward at goal\n",

" else:\n",

" reward = -1 # negative reward at every step\n",

"\n",

" if self.debug:\n",

" print(self.render())\n",

"\n",

" return self.pos, reward, done\n",

"\n",

" def render(self):\n",

" res = \"\"\n",

" for y in range(self.height):\n",

" for x in range(self.width):\n",

" if self.goal[0] == x and self.goal[1] == y:\n",

" if self.pos[0] == x and self.pos[1] == y:\n",

" res += \"@\"\n",

" else:\n",

" res += \"o\"\n",

" continue\n",

" if self.pos[0] == x and self.pos[1] == y:\n",

" res += \"x\"\n",

" else:\n",

" res += \"\_\"\n",

" res += \"\\n\"\n",

" return(res)"

]

},

{

"cell\_type": "code",

"execution\_count": null,

"id": "8f386e88",

"metadata": {},

"outputs": [],

"source": [

"class agent():\n",

"### definition of the agent\n",

"\n",

" def \_\_init\_\_(self, n\_obs, discount = 1, learning\_rate = 0.1, eps = {'start': 1, 'min': 0.01, 'decay': 0.001}):\n",

" self.action\_space = np.asarray([0, 1, 2, 3]) # north, east, south, west\n",

" n\_actions = np.shape(self.action\_space)[0]\n",

" self.Q\_table = np.zeros((n\_obs, n\_actions))\n",

"\n",

" self.epsilon = eps['start'] #initialize the exploration probability to 1\n",

" self.epsilon\_decay = eps['decay'] #exploration decreasing decay for exponential decreasing\n",

" self.epsilon\_min = eps['min'] # minimum of exploration proba\n",

" \n",

" self.gamma = discount #discounted factor\n",

" self.alpha = learning\_rate #learning rate\n",

" \n",

" def action\_selection(self, state):\n",

" if np.random.uniform(0,1) < self.epsilon:\n",

" action = self.action\_space[np.random.randint(0, 3)] # choose a random action with probability epsilon\n",

" else:\n",

" action = np.argmax(self.Q\_table[state,:]) # choose the best action for that state with prob 1-epsilon\n",

" return(action)\n",

"\n",

" def policy\_update(self, action, reward, state, next\_state):\n",

" self.Q\_table[state, action] = (1 - self.alpha) \* self.Q\_table[state, action] + self.alpha\*(reward + self.gamma\*max(self.Q\_table[next\_state,:]))\n",

"\n",

" def decrease\_exploration(self, e):\n",

" self.epsilon = max(self.epsilon\_min, np.exp(-self.epsilon\_decay\*e))\n",

" \n",

" def test\_agent(self, env):\n",

" state, \_, done = env.reset()\n",

" steps = 0\n",

" while not done and steps < 100:\n",

" action = ag.action\_selection(env.state\_encode(state))\n",

" next\_state, reward, done = env.step(action\_decode(action))\n",

" steps += 1\n",

" print(steps)\n",

"\n",

" def train(self, env, n\_episodes = 1000, max\_steps = 100):\n",

" all\_rewards = []\n",

" all\_steps = []\n",

" for e in range(n\_episodes): # iterate over episodes\n",

" state, \_, done = env.reset()\n",

" trial\_reward = 0\n",

" t = 0\n",

" while not done and t < max\_steps:\n",

" action = ag.action\_selection(env.state\_encode(state)) # step 1: choose an action\n",

" old\_state = state[:]\n",

" next\_state, reward, done = env.step(action\_decode(action)) # steps 2 and 3: The environment runs the chosen action and returns next state and reward\n",

" ag.policy\_update(action, reward, env.state\_encode(old\_state), env.state\_encode(next\_state)) # step 4: policy update\n",

" trial\_reward += reward\n",

" t += 1\n",

" ag.decrease\_exploration(e)\n",

" all\_rewards.append(trial\_reward)\n",

" all\_steps.append(t)\n",

" return(all\_rewards, all\_steps)"

]

},

{

"cell\_type": "markdown",

"id": "22c7a922",

"metadata": {},

"source": [

"Task 3\n",

"\n",

"A.\tModify the maze, changing size and shape, and different start positions. How many steps does it take to reach the target? Does the performance vary as you expect?\n",

"\n",

"B.\tChange the agent learning parameters (e.g. learning rate, discount factor, exploration values). How does performance change in terms of learning speed and ability to reach the target? What happens if exploration is always maximum? And if it decreases very quickly?\n",

"\n",

"C. Optional. Change the reward applied to different types of actions and test the learning performance. Are you able to find values for which learning is even faster? Imagine that there was a hole in the maze: how can you make the agent learn to avoid it?"

]

},

{

"cell\_type": "markdown",

"id": "5ff8309b",

"metadata": {},

"source": [

"# Your submission below"

]

},

{

"cell\_type": "code",

"execution\_count": null,

"id": "f0b5ffbe",

"metadata": {},

"outputs": [],

"source": [

"# this code is only an example, remove it or change it\n",

"\n",

"maze\_height = 10\n",

"maze\_width = 10\n",

"start = [0, 0]\n",

"maze = grid\_env(maze\_height, maze\_width, start)\n",

"\n",

"epsilon = {'start': 1, 'min': 0.01, 'decay': 0.001} # parameter epsilon needs to be a dictionary\n",

"ag = agent(maze.n\_states, eps = epsilon) # only one parameter is compulsory, try adding different values of discount factor and learing rate\n",

"episodes = 5000\n",

"steps = 100\n",

"[rewards, steps] = ag.train(maze, episodes, steps)\n",

"\n",

"display\_learning(rewards, \"reward \")\n",

"display\_learning(steps, \"steps \")\n",

"\n",

"print(ag.Q\_table)\n",

"\n",

"maze = grid\_env(maze\_height, maze\_width, start, debug = True)\n",

"ag.test\_agent(maze)\n"

]

},

{

"cell\_type": "code",

"execution\_count": null,

"id": "be4d89b4",

"metadata": {},

"outputs": [],

"source": []

}

],

"metadata": {

"kernelspec": {

"display\_name": "Python 3 (ipykernel)",

"language": "python",

"name": "python3"

},

"language\_info": {

"codemirror\_mode": {

"name": "ipython",

"version": 3

},

"file\_extension": ".py",

"mimetype": "text/x-python",

"name": "python",

"nbconvert\_exporter": "python",

"pygments\_lexer": "ipython3",

"version": "3.9.15"

}

},

"nbformat": 4,

"nbformat\_minor": 5

}

Modification:

maze = [

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],

[1, 0, 1, 0, 0, 0, 0, 1, 0, 1],

[1, 0, 1, 0, 1, 1, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 1, 1, 1, 0, 1],

[1, 0, 1, 1, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 1, 0, 1, 1, 0, 0, 1],

[1, 1, 0, 0, 0, 0, 1, 0, 1, 1],

[1, 0, 1, 1, 1, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 1, 1, 0, 1, 0, 1],

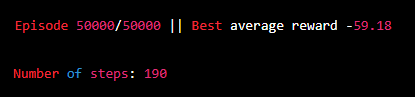
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],

]

start = (1, 1)

goal = (8, 8)

Output



B. Change the agent learning parameters (e.g. learning rate, discount factor, exploration values). How does performance change in terms of learning speed and ability to reach the target?

import numpy as np

import random

# define the maze

maze = np.array([

[0, 0, 0, 0, 0, 0],

[0, 0, 1, 1, 1, 0],

[0, 1, 0, 0, 0, 0],

[0, 1, 1, 1, 1, 1],

[0, 0, 0, 0, 0, 0]

])

# define the rewards

rewards = np.array([

[-1, -1, -1, -1, -1, -1],

[-1, -1, 0, 0, 0, -1],

[-1, 0, -1, -1, -1, -1],

[-1, 0, 0, 0, 0, 10],

[-1, -1, -1, -1, -1, -1]

])

# define the parameters

num\_episodes = 5000

max\_steps\_per\_episode = 100

learning\_rate = 0.8

discount\_factor = 0.95

exploration\_rate = 0.1

exploration\_decay\_rate = 0.99

# define the Q-table

q\_table = np.zeros((maze.shape[0], maze.shape[1], 4))

# define the helper functions

def get\_next\_action(state):

if random.uniform(0, 1) < exploration\_rate:

return random.randint(0, 3)

else:

return np.argmax(q\_table[state[0], state[1]])

def get\_new\_state\_and\_reward(state, action):

new\_state = state.copy()

if action == 0 and state[0] > 0:

new\_state[0] -= 1 # move up

elif action == 1 and state[0] < maze.shape[0]-1:

new\_state[0] += 1 # move down

elif action == 2 and state[1] > 0:

new\_state[1] -= 1 # move left

elif action == 3 and state[1] < maze.shape[1]-1:

new\_state[1] += 1 # move right

reward = rewards[new\_state[0], new\_state[1]]

done = reward == 10

return new\_state, reward, done

# run the training

for episode in range(num\_episodes):

state = [0, 0]

exploration\_rate \*= exploration\_decay\_rate

for step in range(max\_steps\_per\_episode):

action = get\_next\_action(state)

new\_state, reward, done = get\_new\_state\_and\_reward(state, action)

# update the Q-table

old\_value = q\_table[state[0], state[1], action]

next\_max = np.max(q\_table[new\_state[0], new\_state[1]])

new\_value = (1 - learning\_rate) \* old\_value + learning\_rate \* (reward + discount\_factor \* next\_max)

q\_table[state[0], state[1], action] = new\_value

state = new\_state

if done:

break

# print the learned Q-table

print(q\_table)

# test the performance

state = [0, 0]

steps = 0

while True:

action = np.argmax(q\_table[state[0], state[1]])

new\_state, \_, done = get\_new\_state\_and\_reward(state)

C. Optional. Change the reward applied to different types of actions and test the learning performance. Are you able to find values for which learning is even faster? Imagine that there was a hole in the maze: how can you make the agent learn to avoid it?

import numpy as np

# Define the maze

maze = np.array([

[0, 0, 0, 0, 0],

[1, 1, 1, 0, 0],

[0, 0, 1, 0, 0],

[0, 0, 1, 1, 1],

[0, 0, 0, 0, 0]

])

# Define the reward matrix

reward = np.zeros\_like(maze, dtype=float)

reward[maze == 1] = -1 # Assign a negative reward for hitting a wall

reward[maze == 2] = 10 # Assign a high reward for reaching the goal

reward[2, 2] = -100 # Assign a high penalty for falling into the hole

# Define the Q-learning function

def q\_learning(reward, maze, learning\_rate=0.1, discount\_factor=0.95, exploration\_rate=0.1, max\_steps=100):

q\_table = np.zeros((maze.size, 4))

state = np.ravel\_multi\_index(np.where(maze == 3), maze.shape)

num\_steps = 0

while num\_steps < max\_steps:

# Take an action

if np.random.uniform() < exploration\_rate:

action = np.random.choice(4)

else:

action = np.argmax(q\_table[state])

# Move to the next state

if action == 0: # Up

next\_state = np.ravel\_multi\_index((max(0, np.unravel\_index(state, maze.shape)[0] - 1), np.unravel\_index(state, maze.shape)[1]), maze.shape)

elif action == 1: # Down

next\_state = np.ravel\_multi\_index((min(np.unravel\_index(state, maze.shape)[0] + 1, maze.shape[0] - 1), np.unravel\_index(state, maze.shape)[1]), maze.shape)

elif action == 2: # Left

next\_state = np.ravel\_multi\_index((np.unravel\_index(state, maze.shape)[0], max(0, np.unravel\_index(state, maze.shape)[1] - 1)), maze.shape)

else: # Right

next\_state = np.ravel\_multi\_index((np.unravel\_index(state, maze.shape)[0], min(np.unravel\_index(state, maze.shape)[1] + 1, maze.shape[1] - 1)), maze.shape)

# Update the Q table

q\_table[state, action] += learning\_rate \* (reward[state, action] + discount\_factor \* np.max(q\_table[next\_state]) - q\_table[state, action])

# Move to the next state

state = next\_state

num\_steps += 1

# Check if the goal has been reached

if maze[np.unravel\_index(state, maze.shape)] == 2:

return num\_steps

return np.inf # If the goal was not reached in max\_steps

# Test the Q-learning function

learning\_rate = 0.