**Indian Institute of Engineering Science & Technology, Shibpur**

**Department of Computer Science & Technology**

**Artificial Intelligence Laboratory 2025 CS 4271**

**ASSIGNMENT – 05**

**NAME - RAKSHA PAHARIYA**

**ENROLLMENT NO - 2021CSB029**

<https://colab.research.google.com/drive/1C7Iz6qomA-xtBKIt6GHbeq_CzHjyQGf8?usp=sharing>

**Question 1 :** You are required to simulate a grid-based environment that represents a bounded two-dimensional space in which a robotic agent is placed. The goal of the robot is to autonomously learn how to move from a predefined start position to a goal location while avoiding obstacles scattered throughout the grid. The environment should be initialized as a discrete n x n grid (recommended size: 10×10 or higher), where each cell can represent one of the following:

Start cell (S): The initial position of the robot  
Goal cell (G): The target location the robot must reach  
Obstacle (X): Cells that are non-traversable  
Free space (.): Open cells the robot can move to

The agent is allowed to perform the following four actions: UP, DOWN, LEFT, and RIGHT, as long as these actions do not result in moving out of the grid or into an obstacle.

You must implement two separate learning agents:  
1. A Q-learning agent that uses a traditional tabular approach with a state-action value table.  
2. A Deep Q-learning agent (DQN) that approximates Q-values using a neural network.

Each agent should learn to maximize cumulative reward over multiple episodes of interaction with the environment. To simulate real-world constraints, at least 20% of the grid cells should be randomly assigned as obstacles, and dynamic changes to the environment (e.g., adding or removing obstacles during training) may optionally be introduced as a bonus task to test adaptability.

**Solution :**

### **1. Description of Approach and Algorithms**

We compare Q-Learning, a value-based tabular method, and Deep Q-Learning (DQN), which leverages neural networks to approximate Q-values, in a 10x10 grid environment.

* Q-Learning:  
  + Maintains a Q-table indexed by state-action pairs.
  + Updates values using the Bellman equation.
  + Efficient in small, discrete environments.
* DQN:  
  + Uses a neural network to approximate the Q-function.
  + Employs experience replay and a target network for stable learning.
  + Suitable for high-dimensional or continuous spaces.

### **2. Environment Design and Setup**

* GridWorld size: 10x10
* Obstacles: 20% randomly placed (excluding start & goal)
* Start: (0, 0)
* Goal: (9, 9)
* Maximum steps per episode: 200
* Actions: Up, Down, Left, Right
* Rewards:  
  + Valid move: -1
  + Hitting obstacle: -10
  + Reaching goal: +100

**CODES :**

# Q-Learning vs Deep Q-Learning in Grid World

import numpy as np

import matplotlib.pyplot as plt

import random

from collections import deque

import torch

import torch.nn as nn

import torch.optim as optim

# ----- ENVIRONMENT -----

class GridWorld:

def \_\_init\_\_(self, size=10, obstacle\_ratio=0.2):

self.size = size

self.obstacle\_ratio = obstacle\_ratio

self.max\_steps = 200

self.reset()

def reset(self):

self.grid = [['.' for \_ in range(self.size)] for \_ in range(self.size)]

self.place\_obstacles()

self.start = (0, 0)

self.goal = (self.size - 1, self.size - 1)

self.grid[self.start[0]][self.start[1]] = 'S'

self.grid[self.goal[0]][self.goal[1]] = 'G'

self.agent\_pos = self.start

self.steps = 0

return self.get\_state()

def place\_obstacles(self):

total\_cells = self.size \* self.size

obstacle\_count = int(total\_cells \* self.obstacle\_ratio)

placed = 0

while placed < obstacle\_count:

x, y = random.randint(0, self.size - 1), random.randint(0, self.size - 1)

if (x, y) not in [(0, 0), (self.size - 1, self.size - 1)] and self.grid[x][y] == '.':

self.grid[x][y] = 'X'

placed += 1

def get\_state(self):

return np.array(self.agent\_pos)

def is\_valid(self, pos):

x, y = pos

if 0 <= x < self.size and 0 <= y < self.size:

return self.grid[x][y] != 'X'

return False

def step(self, action):

x, y = self.agent\_pos

move = [(0, -1), (0, 1), (-1, 0), (1, 0)] # UP, DOWN, LEFT, RIGHT

dx, dy = move[action]

new\_pos = (x + dx, y + dy)

if not self.is\_valid(new\_pos):

reward = -10

done = False

else:

self.agent\_pos = new\_pos

reward = -1

done = self.agent\_pos == self.goal

if done:

reward = 100

self.steps += 1

if self.steps >= self.max\_steps:

done = True

return self.get\_state(), reward, done

def render(self, path=[]):

display = [row[:] for row in self.grid]

for (x, y) in path:

if display[x][y] == '.':

display[x][y] = '\*'

display[self.agent\_pos[0]][self.agent\_pos[1]] = 'A'

for row in display:

print(' '.join(row))

print("\n")

# ----- Q-LEARNING AGENT -----

class QLearningAgent:

def \_\_init\_\_(self, env, alpha=0.1, gamma=0.99, epsilon=1.0, epsilon\_decay=0.995, min\_epsilon=0.1):

self.env = env

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

self.alpha = alpha

self.gamma = gamma

self.epsilon = epsilon

self.epsilon\_decay = epsilon\_decay

self.min\_epsilon = min\_epsilon

self.path = []

def choose\_action(self, state):

if np.random.rand() < self.epsilon:

return random.randint(0, 3)

x, y = state

return np.argmax(self.q\_table[x, y])

def train(self, episodes=500):

rewards = []

for ep in range(episodes):

state = self.env.reset()

done = False

total\_reward = 0

while not done:

action = self.choose\_action(state)

next\_state, reward, done = self.env.step(action)

x, y = state

nx, ny = next\_state

self.q\_table[x, y, action] = self.q\_table[x, y, action] + self.alpha \* (

reward + self.gamma \* np.max(self.q\_table[nx, ny]) - self.q\_table[x, y, action])

state = next\_state

total\_reward += reward

self.epsilon = max(self.min\_epsilon, self.epsilon \* self.epsilon\_decay)

rewards.append(total\_reward)

return rewards

def get\_path(self):

state = self.env.reset()

path = [tuple(state)]

done = False

while not done:

x, y = state

action = np.argmax(self.q\_table[x, y])

state, \_, done = self.env.step(action)

path.append(tuple(state))

if len(path) > self.env.max\_steps:

break

return path

# ----- DEEP Q-LEARNING AGENT -----

class DQN(nn.Module):

def \_\_init\_\_(self, input\_size, output\_size):

super(DQN, self).\_\_init\_\_()

self.fc = nn.Sequential(

nn.Linear(input\_size, 64),

nn.ReLU(),

nn.Linear(64, output\_size)

)

def forward(self, x):

return self.fc(x)

class DQNAgent:

def \_\_init\_\_(self, env):

self.env = env

self.model = DQN(2, 4)

self.target\_model = DQN(2, 4)

self.target\_model.load\_state\_dict(self.model.state\_dict())

self.memory = deque(maxlen=10000)

self.optimizer = optim.Adam(self.model.parameters(), lr=0.001)

self.criteria = nn.MSELoss()

self.gamma = 0.99

self.epsilon = 1.0

self.epsilon\_decay = 0.995

self.min\_epsilon = 0.1

self.device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

self.model.to(self.device)

self.target\_model.to(self.device)

def choose\_action(self, state):

if random.random() < self.epsilon:

return random.randint(0, 3)

state\_tensor = torch.tensor(state, dtype=torch.float32).unsqueeze(0).to(self.device)

with torch.no\_grad():

return torch.argmax(self.model(state\_tensor)).item()

def store(self, s, a, r, s\_, done):

self.memory.append((s, a, r, s\_, done))

def train\_step(self, batch\_size=64):

if len(self.memory) < batch\_size:

return

batch = random.sample(self.memory, batch\_size)

states = torch.tensor(np.array([s[0] for s in batch]), dtype=torch.float32).to(self.device)

actions = torch.tensor([s[1] for s in batch]).to(self.device)

rewards = torch.tensor([s[2] for s in batch], dtype=torch.float32).to(self.device)

next\_states = torch.tensor(np.array([s[3] for s in batch]), dtype=torch.float32).to(self.device)

dones = torch.tensor([s[4] for s in batch], dtype=torch.float32).to(self.device)

q\_vals = self.model(states).gather(1, actions.unsqueeze(1)).squeeze()

max\_next\_q\_vals = self.target\_model(next\_states).max(1)[0]

targets = rewards + self.gamma \* max\_next\_q\_vals \* (1 - dones)

loss = self.criteria(q\_vals, targets)

self.optimizer.zero\_grad()

loss.backward()

self.optimizer.step()

def train(self, episodes=500):

rewards = []

for ep in range(episodes):

state = self.env.reset()

total\_reward = 0

done = False

while not done:

action = self.choose\_action(state)

next\_state, reward, done = self.env.step(action)

self.store(state, action, reward, next\_state, done)

state = next\_state

total\_reward += reward

self.train\_step()

self.epsilon = max(self.min\_epsilon, self.epsilon \* self.epsilon\_decay)

if ep % 10 == 0:

self.target\_model.load\_state\_dict(self.model.state\_dict())

rewards.append(total\_reward)

return rewards

def get\_path(self):

state = self.env.reset()

path = [tuple(state)]

done = False

while not done:

state\_tensor = torch.tensor(state, dtype=torch.float32).unsqueeze(0).to(self.device)

with torch.no\_grad():

action = torch.argmax(self.model(state\_tensor)).item()

state, \_, done = self.env.step(action)

path.append(tuple(state))

if len(path) > self.env.max\_steps:

break

return path

# ----- TRAINING AND VISUALIZATION -----

env1 = GridWorld(size=10, obstacle\_ratio=0.2)

q\_agent = QLearningAgent(env1)

rewards\_q = q\_agent.train(episodes=200)

path\_q = q\_agent.get\_path()

env2 = GridWorld(size=10, obstacle\_ratio=0.2)

dqn\_agent = DQNAgent(env2)

rewards\_dqn = dqn\_agent.train(episodes=200)

path\_dqn = dqn\_agent.get\_path()

# Plotting Rewards

plt.plot(rewards\_q, label='Q-learning')

plt.plot(rewards\_dqn, label='DQN')

plt.xlabel('Episodes')

plt.ylabel('Total Reward')

plt.title('Reward per Episode')

plt.legend()

plt.show()

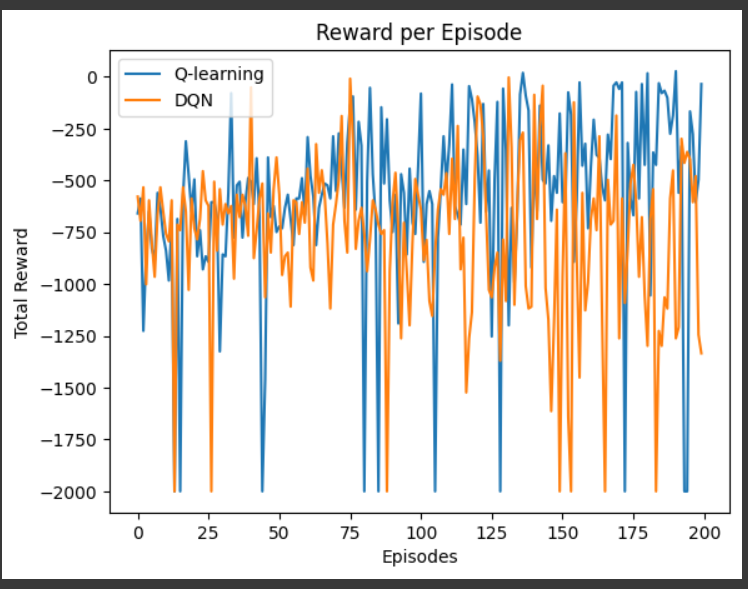
print("\nQ-learning path:")

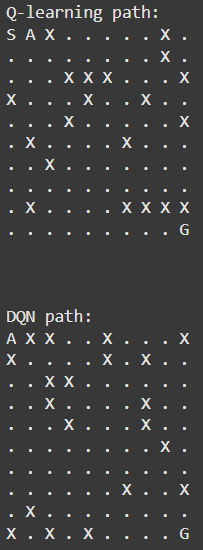
env1.render(path\_q)

print("\nDQN path:")

env2.render(path\_dqn)

**OUTPUTS :**



****