Lab 10

Daniel Mehta

In [3]:
```
#imports
import numpy as np
import random
from collections import deque
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

Setting Up the Environment

Grid Environment

In [6]:
```
# grid parameters
grid_size = 5
start_pos = (0,0)
goal_pos = (4,4)

# Initializing the grid
grid = np.zeros((grid_size,grid_size), dtype=int)
```

In [7]:
```
# Marking the start and goal
grid[start_pos]=1 # 1 is the Start
grid[goal_pos]=2 # 2 is the Goal
```

In [8]:
```
# Randomly placeing 3 obstacles (obstacles are being represented by -1)
obstacles = set()
while len(obstacles) <3:
    pos = (random.randint(0,4), random.randint(0,4))
    if pos !=start_pos and pos != goal_pos and pos not in obstacles:
```

```python
        obstacles.add(pos)
        grid[pos] = -1  # Obstacle
```

In [9]:
```python
print("Grid:\n", grid)

print("\nLegend:")
print("  0 = Empty")
print("  1 = Start")
print("  2 = Goal")
print(" -1 = Obstacle")
```

```
Grid:
 [[ 1  0  0 -1  0]
 [-1  0  0  0  0]
 [ 0  0  0  0  0]
 [-1  0  0  0  0]
 [ 0  0  0  0  2]]

Legend:
  0 = Empty
  1 = Start
  2 = Goal
 -1 = Obstacle
```

Rewards

In [11]:
```python
# Creating a reward matrix matching the grid
rewards = np.zeros((grid_size, grid_size),dtype=int)
```

In [12]:
```python
# Assigning the rewards
for i in range(grid_size):
    for j in range(grid_size):
        if (i,j) == goal_pos:
            rewards[i,j] =10
        elif (i, j) in obstacles:
            rewards[i,j] = -10
        else:
            rewards[i,j] = 0
```

In [13]:
```python
print("Reward Matrix:\n", rewards)
```

```
Reward Matrix:
 [[  0   0   0 -10   0]
 [-10   0   0   0   0]
 [  0   0   0   0   0]
 [-10   0   0   0   0]
 [  0   0   0   0  10]]
```

State Representation

In [15]:
```python
def state_to_coords(state):
    # Normalize grid the coordinates to [0, 1] range for NN input
    return [state[0] / (grid_size - 1), state[1] / (grid_size - 1)]
```

In [16]:
```python
sample_state = (2,3)
print("Normalized coords for", sample_state, ":", state_to_coords(sample_state))
```

```
Normalized coords for (2, 3) : [0.5, 0.75]
```

Implementing Deep Q-Learning

Neural Network Architecture

In [18]:
```python
# Deep Q Network
class DQN(nn.Module):
    def __init__(self):
        super(DQN, self).__init__()
        self.fc1=nn.Linear(2,64) # Input:(x, y)
        self.fc2=nn.Linear(64,64)
        self.out=nn.Linear(64,4) # Output: Q vals for 4 actions

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.out(x)
```

Algorithm Steps

In [20]:
```python
# experience replay buffer
replay_buffer=deque(maxlen=10000)
```

In [21]:
```
# Hyperparameters
epsilon = 1.0 # exploration rate
epsilon_min = 0.01
epsilon_decay = 0.995
gamma = 0.99 # discount factor
batch_size = 32
learning_rate = 0.001
```

In [22]:
```
# action space
action_space = [0,1,2,3]  # up, down, left, right
```

In [23]:
```
# Epsilon greedy policy
def select_action(model, state, epsilon):
    if random.random() <epsilon:
        return random.choice(action_space) # explore
    else:
        state_tensor = torch.FloatTensor(state).unsqueeze(0) # shape (1, 2)
        with torch.no_grad():
            q_values =model(state_tensor)
        return torch.argmax(q_values).item() # exploit
```

Steps

In [25]:
```
# Initialize model and target model
policy_net=DQN()
target_net=DQN()
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()

optimizer = optim.Adam(policy_net.parameters(), lr=learning_rate)
```

In [26]:
```
def step(state, action):
    x, y = state
    next_state = (x, y)

    if action == 0 and x > 0:
        next_state = (x - 1, y)  # up
    elif action == 1 and x < grid_size - 1:
        next_state = (x + 1, y)  # down
    elif action == 2 and y > 0:
        next_state = (x, y - 1)  # left
    elif action == 3 and y < grid_size - 1:
        next_state = (x, y + 1)  # right

    if next_state == goal_pos:
```

```
        return next_state, 10, True
    elif next_state in obstacles:
        return next_state, -10, False
    else:
        return next_state, 0, False
```

In [27]:

```
# Training loop

episode_rewards = []
num_episodes = 2000
replay_buffer.clear()

for episode in range(num_episodes):
    state = start_pos
    total_reward = 0
    done = False
    steps = 0
    max_steps = 100  # to avoid infinite loops

    while not done and steps < max_steps:
        # normalize current state
        state_norm = state_to_coords(state)

        # epsilon-greedy action selection
        action = int(select_action(policy_net, state_norm, epsilon))

        # take step in env
        next_state, reward, done = step(state, action)
        next_state_norm = state_to_coords(next_state)

        # store transition
        replay_buffer.append((state_norm, action, reward, next_state_norm, done))

        # update stats
        total_reward += reward
        steps += 1

        # train only if enough samples
        if len(replay_buffer) < batch_size:
            state = next_state
            continue

        # sample batch
        batch = random.sample(replay_buffer, batch_size)
        states, actions, rewards_, next_states, dones = zip(*batch)

        # convert to tensors
```

```python
        states = torch.FloatTensor(np.array(states))
        actions = torch.LongTensor(actions).unsqueeze(1)
        rewards_ = torch.FloatTensor(rewards_).unsqueeze(1)
        next_states = torch.FloatTensor(np.array(next_states))
        dones = torch.FloatTensor(dones).unsqueeze(1)

        # current Q-values
        q_values = policy_net(states).gather(1, actions)

        # max Q-values for next state (from target net)
        with torch.no_grad():
            max_next_q_values = target_net(next_states).max(1)[0].unsqueeze(1)

        # bellman target
        targets = rewards_ + gamma * max_next_q_values * (1 - dones)

        # loss + backprop
        loss = F.mse_loss(q_values, targets)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # move to next state
        state = next_state

    # save reward
    episode_rewards.append(total_reward)

    # decay epsilon
    if epsilon > epsilon_min:
        epsilon *= epsilon_decay

    # update target net
    if (episode + 1) % 10 == 0:
        target_net.load_state_dict(policy_net.state_dict())

    # print progress
    if (episode + 1) % 50 == 0:
        print(f"Episode {episode + 1}, Total Reward: {total_reward}, Epsilon: {epsilon:.3f}")
```

```
Episode 50, Total Reward: -30, Epsilon: 0.778
Episode 100, Total Reward: 10, Epsilon: 0.606
Episode 150, Total Reward: 10, Epsilon: 0.471
Episode 200, Total Reward: -30, Epsilon: 0.367
Episode 250, Total Reward: -30, Epsilon: 0.286
Episode 300, Total Reward: 0, Epsilon: 0.222
Episode 350, Total Reward: 0, Epsilon: 0.173
Episode 400, Total Reward: -20, Epsilon: 0.135
Episode 450, Total Reward: 0, Epsilon: 0.105
Episode 500, Total Reward: -10, Epsilon: 0.082
Episode 550, Total Reward: 0, Epsilon: 0.063
Episode 600, Total Reward: 0, Epsilon: 0.049
Episode 650, Total Reward: -10, Epsilon: 0.038
Episode 700, Total Reward: -10, Epsilon: 0.030
Episode 750, Total Reward: 0, Epsilon: 0.023
Episode 800, Total Reward: 0, Epsilon: 0.018
Episode 850, Total Reward: 0, Epsilon: 0.014
Episode 900, Total Reward: -10, Epsilon: 0.011
Episode 950, Total Reward: 0, Epsilon: 0.010
Episode 1000, Total Reward: -10, Epsilon: 0.010
Episode 1050, Total Reward: 0, Epsilon: 0.010
Episode 1100, Total Reward: 10, Epsilon: 0.010
Episode 1150, Total Reward: 10, Epsilon: 0.010
Episode 1200, Total Reward: 10, Epsilon: 0.010
Episode 1250, Total Reward: 10, Epsilon: 0.010
Episode 1300, Total Reward: 10, Epsilon: 0.010
Episode 1350, Total Reward: 10, Epsilon: 0.010
Episode 1400, Total Reward: 10, Epsilon: 0.010
Episode 1450, Total Reward: 10, Epsilon: 0.010
Episode 1500, Total Reward: 10, Epsilon: 0.010
Episode 1550, Total Reward: 10, Epsilon: 0.010
Episode 1600, Total Reward: 10, Epsilon: 0.010
Episode 1650, Total Reward: 10, Epsilon: 0.010
Episode 1700, Total Reward: 10, Epsilon: 0.010
Episode 1750, Total Reward: 10, Epsilon: 0.010
Episode 1800, Total Reward: 0, Epsilon: 0.010
Episode 1850, Total Reward: 0, Epsilon: 0.010
Episode 1900, Total Reward: 0, Epsilon: 0.010
Episode 1950, Total Reward: 0, Epsilon: 0.010
Episode 2000, Total Reward: 10, Epsilon: 0.010
```

In [28]:

```python
def get_greedy_path(model, start):
    path = [start]
    state = start
    visited = set()

    while state != goal_pos:
        visited.add(state)
```

```python
        state_tensor = torch.FloatTensor(state_to_coords(state)).unsqueeze(0)
        with torch.no_grad():
            q_values = model(state_tensor)
        action = torch.argmax(q_values).item()

        next_state, _, _ =step(state,action)

        if next_state in visited or next_state in obstacles:
            break

        path.append(next_state)
        state = next_state

    return path

# generate path after training
path = get_greedy_path(policy_net,start_pos)
```

In [29]:

```python
print(f"Total episodes trained: {num_episodes}")
print(f"Final epsilon: {epsilon:.3f}")
print(f"Path length from S to G: {len(path)}")
```

```
Total episodes trained: 2000
Final epsilon: 0.010
Path length from S to G: 9
```

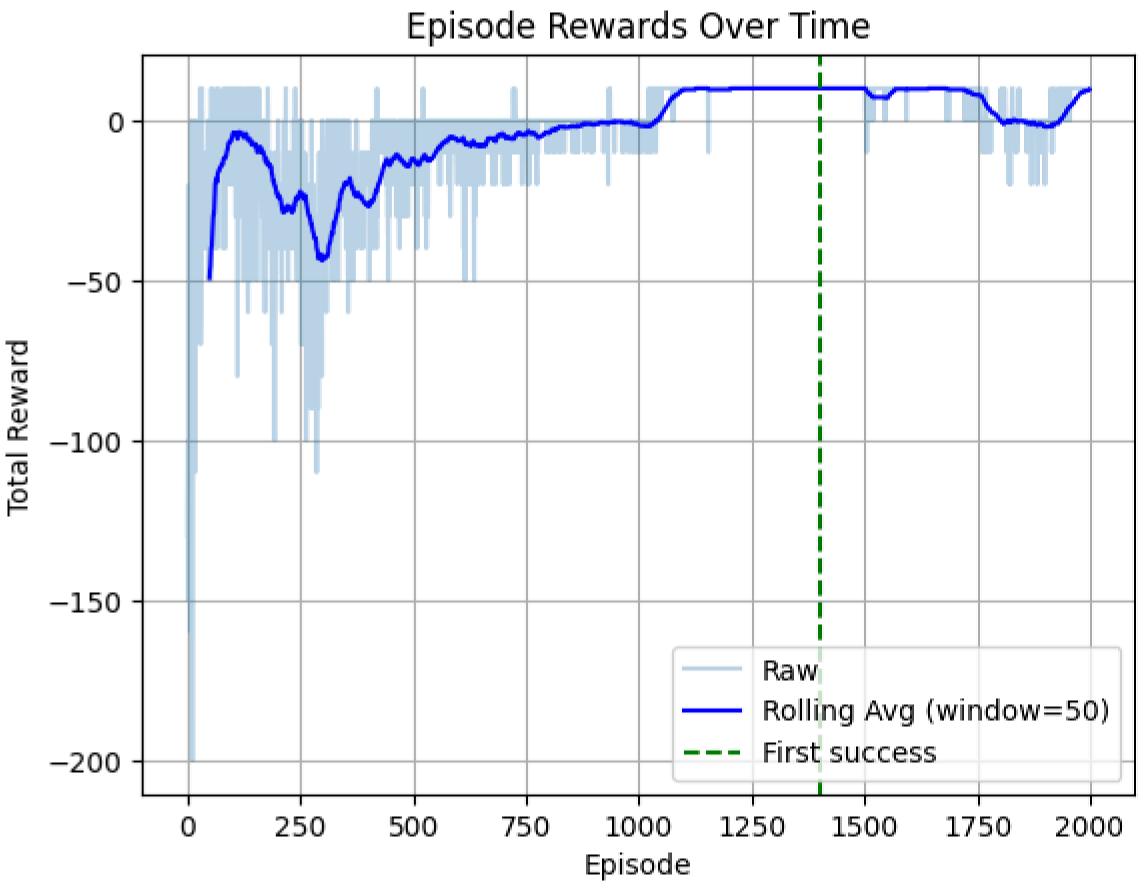
Visualizing Results

Performance Metrics

In [31]:

```python
# Episode Rewards Over Time
smoothed = pd.Series(episode_rewards).rolling(window=50).mean()

plt.plot(episode_rewards, alpha=0.3, label="Raw")
plt.plot(smoothed, color='blue', label="Rolling Avg (window=50)")
plt.axvline(x=1400, color='green', linestyle='--', label='First success')
plt.xlabel("Episode")
plt.ylabel("Total Reward")
plt.title("Episode Rewards Over Time")
plt.grid(True)
plt.legend()
plt.show()
```



In [32]:

```
def print_path_grid(grid, path, start, goal, obstacles):
    display = [['.' for _ in range(grid_size)] for _ in range(grid_size)]

    for (i,j) in path:
        display[i][j] = '*'

    for (i,j) in obstacles:
        display[i][j] = 'X'

    sx, sy = start
    gx, gy =goal
    display[sx][sy]='S'
    display[gx][gy]='G'

    for row in display:
        print(' '.join(row))

# Show the path from start to goal
print_path_grid(grid, path, start_pos, goal_pos, obstacles)
```

```
S * . X .
X * . . .
. * . . .
X * . . .
. * * * G
```

Q-Value Approximation

In [34]:
```
sample_states = [(0, 0), (1, 1), (2, 2), (3, 3)]
for s in sample_states:
    coords = torch.FloatTensor(state_to_coords(s)).unsqueeze(0)
    with torch.no_grad():
        q_vals = policy_net(coords)
    print(f"State {s} -> Q-values: {q_vals.numpy().flatten()}")
```

```
State (0, 0) -> Q-values: [ 9.259881  -0.6334461  9.258819   9.362408 ]
State (1, 1) -> Q-values: [ 9.369141   9.542203  -0.5945864  9.493502 ]
State (2, 2) -> Q-values: [9.387017 9.710793 9.553543 9.526387]
State (3, 3) -> Q-values: [9.518854 9.916376 9.750566 9.802436]
```

Reflection Questions

Q1: How does using neural networks improve over traditional Q-tables?

Neural networks improve over traditional Q-tables by allowing generalization across similar states. Instead of storing a value for every possible state action pair, the network learns patterns and can estimate Q vals for unseen inputs. This makes it more scalable and effective in environments with large or continuous state spaces.

Q2: What challenges did you face when implementing the Deep Q-Learning algorithm?

One challenge I faced was that the agent struggled to learn due to sparse rewards and early episode termination when hitting obstacles. I had to modify the environment to allow continued exploration after hitting an obstacle and extend training to 2000 episodes. It also took some experimentation to get the epsilon decay rate and reward structure right so that the agent could balance exploration and exploitation and eventually converge on a successful policy.

Q3. How do hyperparameters affect training?

Hyperparameters control how the agent learns. A high learning rate can make training unstable, while a low one slows it down. Epsilon affects exploration, and poor tuning can prevent the agent from finding the goal. The discount factor and batch size also influence how effectively the agent learns over time.

Q4. Suggest a real-world application of Deep Reinforcement Learning and explain its implementation.

A real world application of Deep Reinforcement Learning is traffic management for autonomous vehicles. In a past project, I used a Q-learning agent with a distance based BNART heuristic to optimize routing on a 10×10 grid. While that implementation used a Q table, the same idea can scale to larger or continuous environments using deep reinforcement learning, where a neural network approximates Q-values based on the vehicle's state and destination. This allows more complex and flexible routing strategies in real world traffic systems.

In [ ]:

```

```