Machine Learning Lab A3

ASIM KUMAR HANSDA

ROLL NO - 002211001136

ASSIGNMENT - 5

Github Link:

https://github.com/cryptasim/MACHINE-LEARNING-LAB

Open in Colab

In [ ]:
```
!pip install gymnasium
!pip install imageio
!pip install swig
```

```
Requirement already satisfied: gymnasium in /usr/local/lib/python3.12/dist-pack
ages (1.2.2)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.12/dist-
packages (from gymnasium) (2.0.2)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.12/
dist-packages (from gymnasium) (3.1.2)
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/pytho
n3.12/dist-packages (from gymnasium) (4.15.0)
Requirement already satisfied: farama-notifications>=0.0.1 in /usr/local/lib/py
thon3.12/dist-packages (from gymnasium) (0.0.4)
Requirement already satisfied: imageio in /usr/local/lib/python3.12/dist-packag
es (2.37.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages
(from imageio) (2.0.2)
Requirement already satisfied: pillow>=8.3.2 in /usr/local/lib/python3.12/dist-
packages (from imageio) (11.3.0)
Collecting swig
  Downloading swig-4.4.0-py3-none-manylinux_2_12_x86_64.manylinux2010_x86_64.wh
l.metadata (3.5 kB)
Downloading swig-4.4.0-py3-none-manylinux_2_12_x86_64.manylinux2010_x86_64.whl
(1.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.9/1.9 MB 35.7 MB/s eta 0:00:00
Installing collected packages: swig
Successfully installed swig-4.4.0
```

In [ ]:
```
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output
import time
import imageio.v2 as imageio
from IPython.display import Image, display
```

In [ ]:
```
def create_bins(state_space, bins_per_feature=20):
  bins = [np.linspace(low, high, bins_per_feature) for low, high in zip(state_
  return bins


def discretize_state(state, bins):
  state_index = tuple(np.digitize(s, b) - 1 for s, b in zip(state, bins))
  return state_index


def choose_action(Q, state, epsilon, n_actions):
  if np.random.random() < epsilon:
    return np.random.randint(n_actions)
  else:
```

```
    return np.argmax(Q[state])


def update_q(Q, state, action, reward, next_state, alpha, gamma):
  Q[state][action] += alpha * (reward + gamma * np.max(Q[next_state]) - Q[stat
```

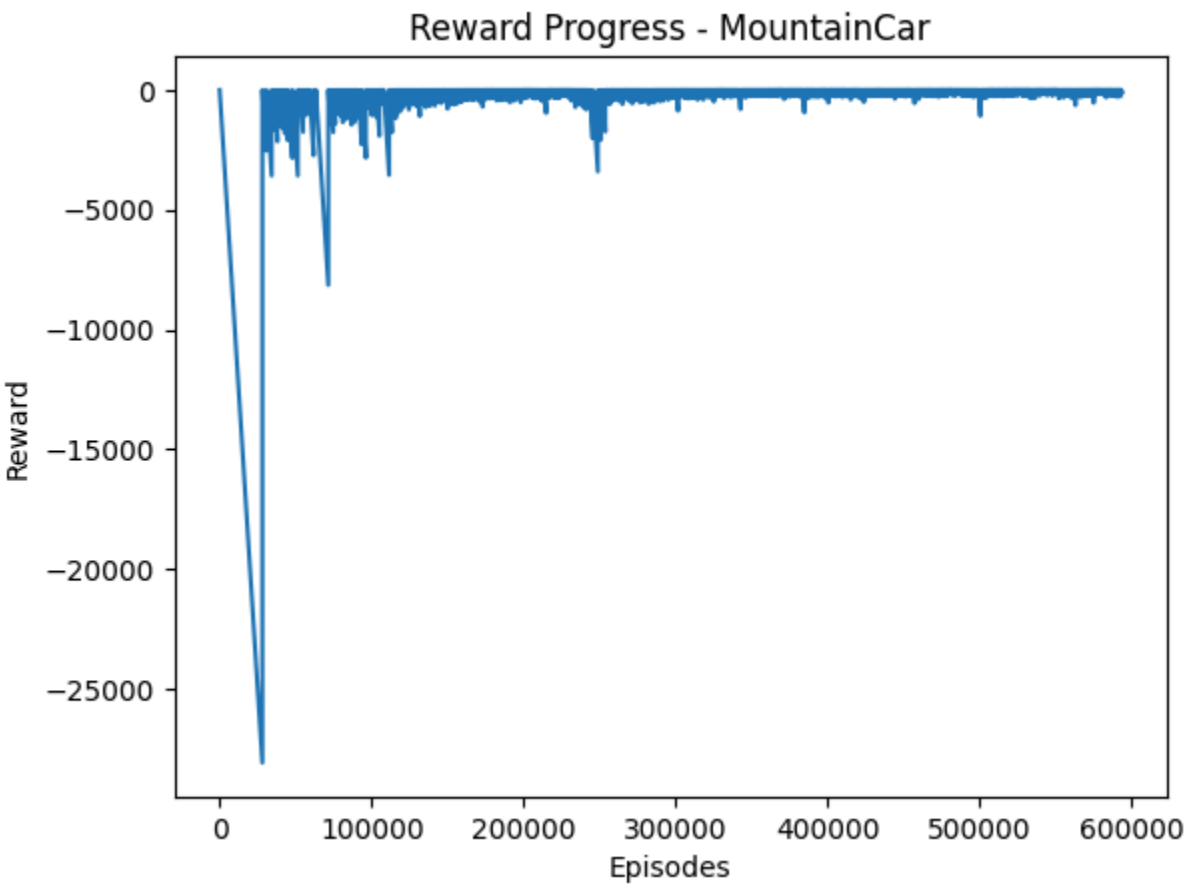
In [ ]:
```
def q_learning(env_name, episodes=5000, alpha=0.1, gamma=0.99, epsilon=1.0, ep
  env = gym.make(env_name)
  bins = create_bins(env.observation_space, bins_per_feature)
  n_actions = env.action_space.n


  Q = np.zeros(tuple([bins_per_feature]*len(env.observation_space.low) + [n_ac
  rewards = []


  for episode in range(episodes):
    state = discretize_state(env.reset()[0], bins)
    done = False
    total_reward = 0


    while not done:
      action = choose_action(Q, state, epsilon, n_actions)
      next_state_cont, reward, done, truncated, _ = env.step(action)
      next_state = discretize_state(next_state_cont, bins)


      update_q(Q, state, action, reward, next_state, alpha, gamma)
      state = next_state
      total_reward += reward


      rewards.append(total_reward)
      epsilon = max(epsilon_min, epsilon * epsilon_decay)


      if episode % render_interval == 0:
        clear_output(wait=True)
        print(f"Episode: {episode}, Reward: {total_reward}, Epsilon: {epsilon:


  env.close()
  return Q, rewards
```

In [ ]:
```
Q_mountain, rewards_mountain = q_learning('MountainCar-v0', episodes=2000)
```

```
Episode: 1500, Reward: -230.0, Epsilon: 0.050
```

In [ ]:
```
plt.plot(rewards_mountain)
plt.title('Reward Progress - MountainCar')
plt.xlabel('Episodes')
plt.ylabel('Reward')
```

```
plt.show()
```



In [ ]:
```
def test_mountain_car(Q, env_name='MountainCar-v0', tests=5, bins_per_feature=
    env = gym.make(env_name, render_mode=render_mode)
    bins = create_bins(env.observation_space, bins_per_feature)
    n_actions = env.action_space.n
    goal_reached_count = 0
    all_test_frames = []


    print(f"\nRunning {tests} tests for the agent...")


    for test in range(tests):
        state = discretize_state(env.reset()[0], bins)
        done = False
        total_reward = 0
        goal_reached_in_test = False
        frames = []

        while not done:
            if render_mode is not None:
                frame = env.render()
                if frame is not None:
                    frames.append(frame)
```

```
            action = np.argmax(Q[state])
            next_state_cont, reward, done, truncated, _ = env.step(action)
            next_state = discretize_state(next_state_cont, bins)


            state = next_state
            total_reward += reward


            if next_state_cont[0] >= 0.5:
                goal_reached_in_test = True
                break


        if goal_reached_in_test:
            goal_reached_count += 1
            print(f"Test {test + 1}: Goal reached!")
            if render_mode is not None:
                all_test_frames.append(frames)
        else:
            print(f"Test {test + 1}: Goal not reached.")


    env.close()
    print(f"\nGoal reached in {goal_reached_count}/{tests} tests.")


    return all_test_frames, goal_reached_count

test_frames, goal_count = test_mountain_car(Q_mountain, render_mode='rgb_array
```

```
Running 5 tests for the agent...
Test 1: Goal reached!
Test 2: Goal reached!
Test 3: Goal reached!
Test 4: Goal reached!
Test 5: Goal reached!

Goal reached in 5/5 tests.
```

In [ ]:
```
def render_frame(t_frames):
  for i, frames in enumerate(t_frames):
      filename = f'mountain_car_test_{i+1}.gif'
      imageio.mimsave(filename, frames, duration=50)
      print(f"Generated GIF: {filename}")
```

In [ ]:
```
def display_gifs(t_frames):
  for i in range(len(t_frames)):
    filename = f'mountain_car_test_{i+1}.gif'
    img = Image(filename=filename)
    display(img)
```

In [ ]:
```
render_frame(test_frames)
display_gifs(test_frames)
```

```
Generated GIF: mountain_car_test_1.gif
Generated GIF: mountain_car_test_2.gif
Generated GIF: mountain_car_test_3.gif
Generated GIF: mountain_car_test_4.gif
Generated GIF: mountain_car_test_5.gif
<IPython.core.display.Image object>
<IPython.core.display.Image object>
<IPython.core.display.Image object>
<IPython.core.display.Image object>
<IPython.core.display.Image object>
```

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import random

class RouletteEnv:
    def __init__(self, start_balance=50, target_balance=100, max_steps=300):
        self.start_balance = start_balance
        self.target_balance = target_balance
        self.max_steps = max_steps
        self.bet_sizes = [1, 5, 10]
        self.numbers = 37
        self.action_space = self.numbers * len(self.bet_sizes)
        self.reset()

    def reset(self):
        self.balance = self.start_balance
        self.steps = 0
        return self._discretize_balance(self.balance)

    def _discretize_balance(self, balance):
        return min(balance // 2, 150)

    def step(self, action):
        number_choice = action % self.numbers
        bet_size = self.bet_sizes[action // self.numbers]

        reward = 0
        for _ in range(5):  # average spins
            spin = np.random.randint(0, 37)
            if spin == number_choice:
                reward += 35 * bet_size
            else:
                reward -= bet_size

        self.balance += reward
        self.steps += 1

        done = self.balance <= 0 or self.balance >= self.target_balance or sel

        shaped_reward = reward
```

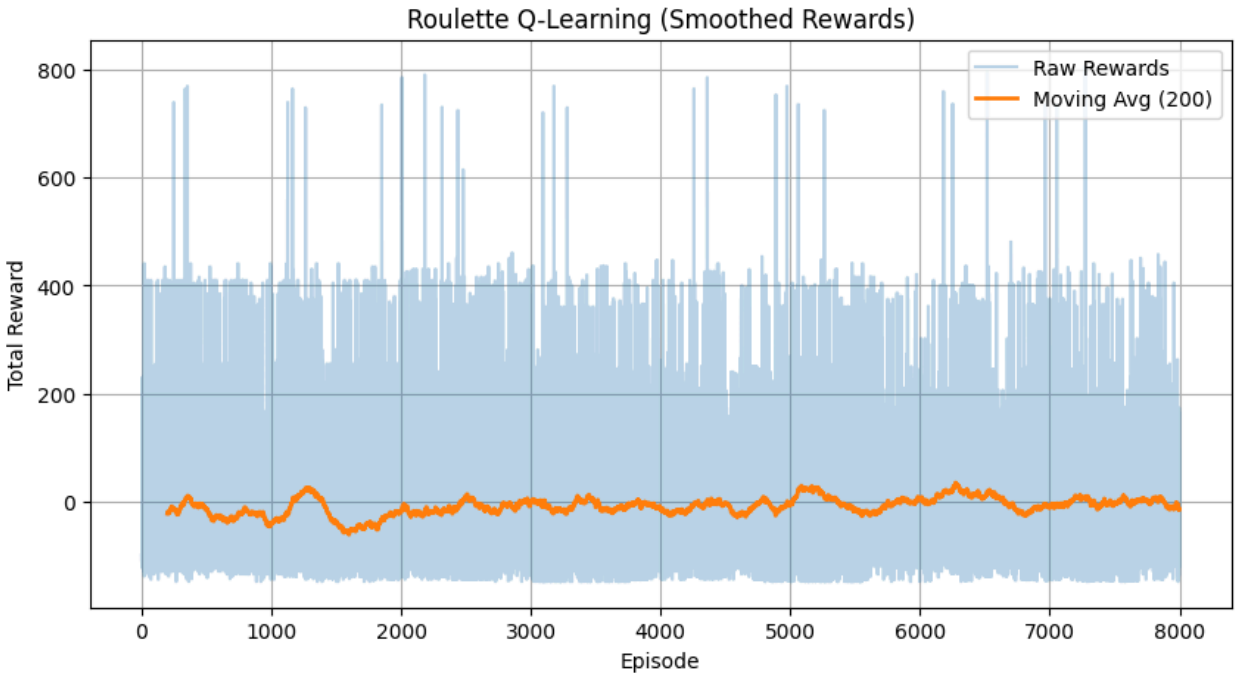
```
        if self.balance >= self.target_balance:
            shaped_reward += 100
        elif self.balance <= 0:
            shaped_reward -= 50

        next_state = self._discretize_balance(self.balance)
        return next_state, shaped_reward, done
```

In [ ]:
```
def q_learning(
    episodes=8000,
    alpha=0.15,
    gamma=0.95,
    epsilon=1.0,
    epsilon_min=0.05,
    epsilon_decay=0.9994
):
    env = RouletteEnv()
    state_space = 151
    action_space = env.action_space
    Q = np.zeros((state_space, action_space))
    rewards_per_episode = []

    for ep in range(episodes):
        state = env.reset()
        total_reward = 0

        for _ in range(env.max_steps):
            if random.random() < epsilon:
                action = random.randint(0, action_space - 1)
            else:
                action = np.argmax(Q[state])

            next_state, reward, done = env.step(action)
            total_reward += reward

            Q[state, action] += alpha * (reward + gamma * np.max(Q[next_state]
            state = next_state

            if done:
                break

        epsilon = max(epsilon_min, epsilon * epsilon_decay)
        rewards_per_episode.append(total_reward)

        if (ep + 1) % 1000 == 0:
            avg_reward = np.mean(rewards_per_episode[-1000:])
            print(f"Episode {ep+1}/{episodes} | Avg Reward: {avg_reward:.2f}")

    return Q, rewards_per_episode, env
```

In [ ]:
```
Q, rewards, env = q_learning()
```

```
Episode 1000/8000 | Avg Reward: -22.64
Episode 2000/8000 | Avg Reward: -23.27
Episode 3000/8000 | Avg Reward: -9.44
Episode 4000/8000 | Avg Reward: -10.29
Episode 5000/8000 | Avg Reward: -9.40
Episode 6000/8000 | Avg Reward: -0.85
Episode 7000/8000 | Avg Reward: 0.61
Episode 8000/8000 | Avg Reward: -1.49
```

In [ ]:
```python
def plot_with_moving_average(rewards, window=200):
    plt.figure(figsize=(10, 5))
    plt.plot(rewards, alpha=0.3, label="Raw Rewards")

    if len(rewards) >= window:
        moving_avg = np.convolve(rewards, np.ones(window)/window, mode='valid'
        plt.plot(range(window - 1, len(rewards)), moving_avg, label=f"Moving A

    plt.xlabel("Episode")
    plt.ylabel("Total Reward")
    plt.title("Roulette Q-Learning (Smoothed Rewards)")
    plt.legend()
    plt.grid(True)
    plt.show()

plot_with_moving_average(rewards)
```



In [ ]:
```python
def test_agent(Q, env, runs=10):
    success_count = 0
    for r in range(runs):
        state = env.reset()
        total_reward = 0
        for _ in range(env.max_steps):
```

```
            action = np.argmax(Q[state])
            next_state, reward, done = env.step(action)
            total_reward += reward
            state = next_state
            if done:
                if env.balance >= env.target_balance:
                    success_count += 1
                break
        print(f"Test {r+1}: Final Balance = {env.balance}, Total Reward = {tot

    print(f"\nGoal Reached in {success_count}/{runs} tests "
          f"({(success_count / runs) * 100:.1f}% success rate)")
```

In [ ]:
```
test_agent(Q, env)
```

```
Test 1: Final Balance = -4, Total Reward = -104
Test 2: Final Balance = -4, Total Reward = -104
Test 3: Final Balance = 0, Total Reward = -100
Test 4: Final Balance = 124, Total Reward = 174
Test 5: Final Balance = 0, Total Reward = -100
Test 6: Final Balance = 0, Total Reward = -100
Test 7: Final Balance = 0, Total Reward = -100
Test 8: Final Balance = 0, Total Reward = -100
Test 9: Final Balance = 0, Total Reward = -100
Test 10: Final Balance = -4, Total Reward = -104

Goal Reached in 1/10 tests (10.0% success rate)
```

In [ ]:
```
# Q2: Apply Deep Reinforcement Learning (DQN) for MountainCar-v0
import gym
import numpy as np
if not hasattr(np, 'bool8'):
    np.bool8 = np.bool_

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.optimizers import Adam
from collections import deque
import random
import matplotlib.pyplot as plt

# Create environment
env = gym.make('MountainCar-v0')

# DQN parameters
state_size = env.observation_space.shape[0]
action_size = env.action_space.n
gamma = 0.99
epsilon = 1.0
epsilon_min = 0.01
epsilon_decay = 0.995
learning_rate = 0.001
batch_size = 8
episodes = 400
```

```
# Replay memory
memory = deque(maxlen=20000)

# Build Neural Network model
def build_model():
    model = Sequential([
        Input(shape=(state_size,)),
        Dense(64, activation='relu'),
        Dense(64, activation='relu'),
        Dense(action_size, activation='linear')
    ])
    model.compile(loss='mse', optimizer=Adam(learning_rate=learning_rate))
    return model

model = build_model()

# Choose an action using epsilon-greedy strategy
def act(state):
    if np.random.rand() <= epsilon:
        return random.randrange(action_size)
    q_values = model.predict(state, verbose=0)
    return np.argmax(q_values[0])

# Replay experience and train the network
def replay():
    global epsilon
    if len(memory) < batch_size:
        return
    minibatch = random.sample(memory, batch_size)
    states, targets = [], []
    for state, action, reward, next_state, done in minibatch:
        target = reward
        if not done:
            target = reward + gamma * np.amax(model.predict(next_state, verbos
        target_f = model.predict(state, verbose=0)
        target_f[0][action] = target
        states.append(state[0])
        targets.append(target_f[0])
    model.fit(np.array(states), np.array(targets), epochs=1, verbose=0)
    if epsilon > epsilon_min:
        epsilon *= epsilon_decay

# Training loop
rewards_list = []
for e in range(episodes):
    state = env.reset()
    # Some gym versions return (obs, info), handle both
    if isinstance(state, tuple):
        state = state[0]
    state = np.array(state).reshape(1, -1)
    total_reward = 0
    done = False
```

```
    while not done:
        action = act(state)
        step_result = env.step(action)

        # Handle both 4- and 5-value outputs (Gym vs Gymnasium)
        if len(step_result) == 5:
            next_state, reward, terminated, truncated, _ = step_result
            done = terminated or truncated
        else:
            next_state, reward, done, _ = step_result

        next_state = np.array(next_state).reshape(1, -1)
        memory.append((state, action, reward, next_state, done))
        state = next_state
        total_reward += reward
        replay()

        if done:
            print(f"Episode: {e+1}/{episodes}, Reward: {total_reward}, Epsilor
            break

    rewards_list.append(total_reward)

env.close()

# Plot learning performance
plt.plot(rewards_list)
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('DQN Learning Curve for MountainCar-v0')
plt.show()
```

```
Episode: 1/400, Reward: -200.0, Epsilon: 0.38
Episode: 2/400, Reward: -200.0, Epsilon: 0.14
Episode: 3/400, Reward: -200.0, Epsilon: 0.05
Episode: 4/400, Reward: -200.0, Epsilon: 0.02
Episode: 5/400, Reward: -200.0, Epsilon: 0.01
Episode: 6/400, Reward: -200.0, Epsilon: 0.01
Episode: 7/400, Reward: -200.0, Epsilon: 0.01
```

Open in Colab

In [ ]:
```
!pip install gymnasium
!pip install imageio
!pip install swig
```

```
Requirement already satisfied: gymnasium in /usr/local/lib/python3.12/dist-pack
ages (1.2.2)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.12/dist-
packages (from gymnasium) (2.0.2)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.12/
dist-packages (from gymnasium) (3.1.2)
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/pytho
n3.12/dist-packages (from gymnasium) (4.15.0)
Requirement already satisfied: farama-notifications>=0.0.1 in /usr/local/lib/py
thon3.12/dist-packages (from gymnasium) (0.0.4)
Requirement already satisfied: imageio in /usr/local/lib/python3.12/dist-packag
es (2.37.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages
(from imageio) (2.0.2)
Requirement already satisfied: pillow>=8.3.2 in /usr/local/lib/python3.12/dist-
packages (from imageio) (11.3.0)
Collecting swig
  Downloading swig-4.4.0-py3-none-manylinux_2_12_x86_64.manylinux2010_x86_64.wh
l.metadata (3.5 kB)
Downloading swig-4.4.0-py3-none-manylinux_2_12_x86_64.manylinux2010_x86_64.whl
(1.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.9/1.9 MB 58.9 MB/s eta 0:00:00
Installing collected packages: swig
Successfully installed swig-4.4.0
```

In [ ]:
```
import time
import random
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import networkx as nx
from collections import deque
import gymnasium as gym
from gymnasium import spaces
```

In [ ]:
```
device= torch.device("cuda" if torch.cuda.is_available() else "cpu")

class QNetwork(nn.Module):
    def __init__(self, state_size, action_size, hidden_size=128):
        super(QNetwork, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(state_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
```

```
            nn.ReLU(),
            nn.Linear(hidden_size, action_size)
        )

    def forward(self, x):
        return self.layers(x)


class ReplayBuffer:
    def __init__(self, capacity=100000):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)
        return (
            torch.FloatTensor(states).to(device),
            torch.LongTensor(actions).to(device),
            torch.FloatTensor(rewards).to(device),
            torch.FloatTensor(next_states).to(device),
            torch.FloatTensor(dones).to(device),
        )

    def __len__(self):
        return len(self.buffer)
```

In [ ]:
```
def plot_rewards(reward_history, window=50):
    """Plot Reward vs Episode with optional moving average."""
    plt.figure(figsize=(9, 5))
    plt.plot(reward_history, color='blue', alpha=0.6, label="Reward per Episod

    if len(reward_history) >= window:
        moving_avg = np.convolve(reward_history, np.ones(window)/window, mode=
        plt.plot(range(window - 1, len(reward_history)), moving_avg, color='re

    plt.title("Reward vs Episodes (MountainCar DQN)")
    plt.xlabel("Episode")
    plt.ylabel("Reward")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

In [ ]:
```
def train_dqn(
    env_name="MountainCar-v0",
    episodes=1000,
    batch_size=64,
    gamma=0.99,
    lr=1e-3,
    epsilon_start=1.0,
```

```
    epsilon_end=0.05,
    epsilon_decay=0.995,
    target_update_freq=10,
    device= device
):
    env = gym.make(env_name)
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    policy_net = QNetwork(state_size, action_size).to(device)
    target_net = QNetwork(state_size, action_size).to(device)
    target_net.load_state_dict(policy_net.state_dict())
    optimizer = optim.Adam(policy_net.parameters(), lr=lr)
    buffer = ReplayBuffer()

    epsilon = epsilon_start
    all_rewards = []
    goal_reached = False

    for episode in range(1, episodes + 1):
        state, _ = env.reset()
        total_reward = 0
        done = False

        while not done:

            if random.random() < epsilon:
                action = env.action_space.sample()
            else:
                with torch.no_grad():
                    state_tensor = torch.FloatTensor(state).unsqueeze(0).to(de
                    q_values = policy_net(state_tensor)
                    action = torch.argmax(q_values).item()

            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            position, velocity = next_state
            shaped_reward = reward + (position + 0.5)

            buffer.push(state, action, shaped_reward, next_state, done)
            state = next_state
            total_reward += reward

            if len(buffer) >= batch_size:
                states, actions, rewards, next_states, dones = buffer.sample(b
                q_values = policy_net(states).gather(1, actions.unsqueeze(1)).
                with torch.no_grad():
                    next_q_values = target_net(next_states).max(1)[0]
                    target = rewards + gamma * next_q_values * (1 - dones)
                loss = nn.MSELoss()(q_values, target)

                optimizer.zero_grad()
```

```
                loss.backward()
                optimizer.step()

        epsilon = max(epsilon_end, epsilon * epsilon_decay)
        all_rewards.append(total_reward)

        if episode % target_update_freq == 0:
            target_net.load_state_dict(policy_net.state_dict())

        if episode % 1000 == 0:
            avg_reward = np.mean(all_rewards[-1000:])
            print(f"Episode {episode}/{episodes} | Avg Reward (last 1000): {av

        if np.mean(all_rewards[-10:]) > -110:
            print(f"Early stop at episode {episode}: near-optimal performance.
            goal_reached = True
            break

    env.close()
    return policy_net, goal_reached, all_rewards
```

In [ ]:

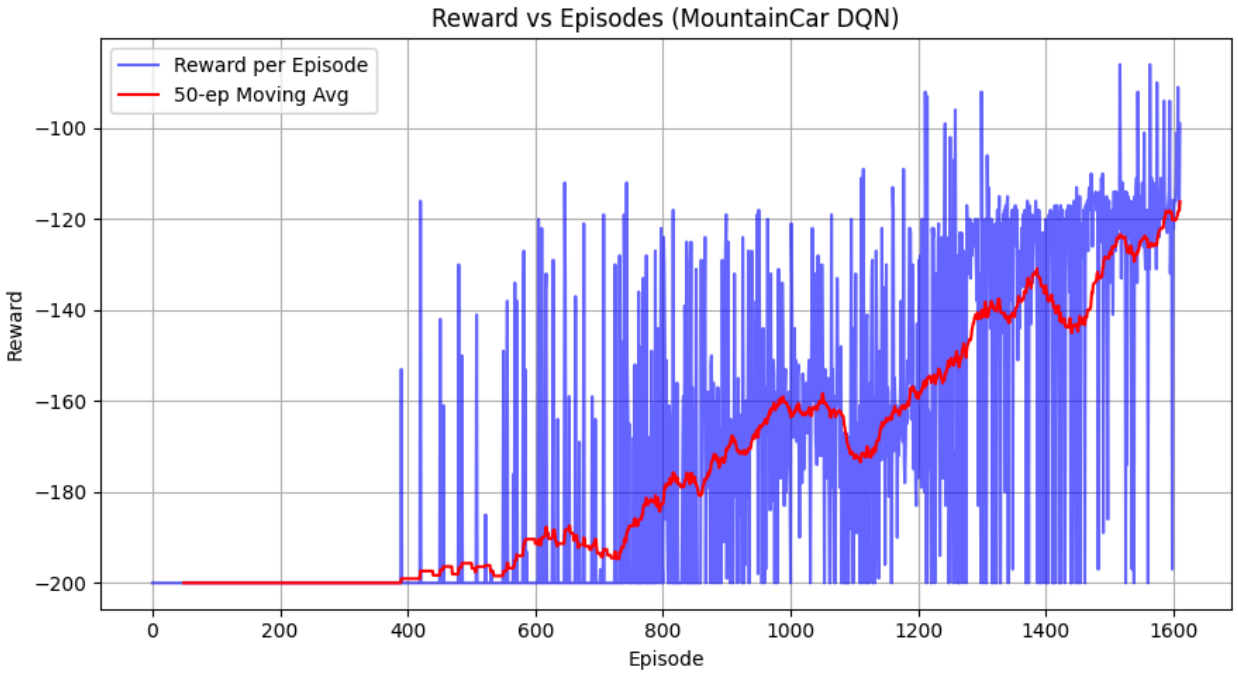
```
def test_agent(policy_net, env_name="MountainCar-v0", runs=5, render=False, de
    env = gym.make(env_name, render_mode="human" if render else None)
    print("\n=== Running Evaluation Tests ===")
    success_count = 0
    rewards = []

    for i in range(runs):
        state, _ = env.reset()
        total_reward = 0
        done = False
        steps = 0

        while not done:
            with torch.no_grad():
                state_tensor = torch.FloatTensor(state).unsqueeze(0).to(device
                q_values = policy_net(state_tensor)
                action = torch.argmax(q_values).item()

            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            total_reward += reward
            steps += 1
            state = next_state

        rewards.append(total_reward)
        reached_goal = state[0] >= 0.5
        if reached_goal:
            success_count += 1
            print(f"Run {i+1}: Goal Reached in {steps} steps (Reward: {total_r
        else:
            print(f"Run {i+1}: Goal Not Reached (Final Pos: {state[0]:.2f}, Re
```

```
    avg_reward = np.mean(rewards)
    print(f"\n=== Summary ===")
    print(f"Average Reward: {avg_reward:.2f}")
    print(f"Goal reached in {success_count}/{runs} runs ({(success_count / rur
    env.close()
```

In [ ]:
```
trained_policy, goal_flag, reward_history = train_dqn(episodes=5000)
```

```
/tmp/ipython-input-3505766926.py:29: UserWarning: Creating a tensor from a list
of numpy.ndarrays is extremely slow. Please consider converting the list to a s
ingle numpy.ndarray with numpy.array() before converting to a tensor. (Triggere
d internally at /pytorch/torch/csrc/utils/tensor_new.cpp:253.)
  torch.FloatTensor(states).to(device),
Episode 1000/5000 | Avg Reward (last 1000): -190.56 | Epsilon: 0.05
Early stop at episode 1611: near-optimal performance.
```

In [ ]:
```
plot_rewards(reward_history, window=50)
```



In [ ]:
```
test_agent(trained_policy, runs=5)
```

```
=== Running Evaluation Tests ===
Run 1: Goal Reached in 115 steps (Reward: -115.00)
Run 2: Goal Reached in 112 steps (Reward: -112.00)
Run 3: Goal Reached in 112 steps (Reward: -112.00)
Run 4: Goal Reached in 114 steps (Reward: -114.00)
Run 5: Goal Reached in 112 steps (Reward: -112.00)

=== Summary ===
Average Reward: -113.00
Goal reached in 5/5 runs (100.0% success rate)
```

In [ ]:
```
## -- Graph Question on shortest path
class GraphEnv:
```

```
    def __init__(self, n_nodes, edges, start, goal, max_steps=50):
        self.n = n_nodes
        self.adj = {i: [] for i in range(n_nodes)}
        for u, v, w in edges:
            self.adj[u].append((v, w))
        self.start = start
        self.goal = goal
        self.max_steps = max_steps
        self.reset()

    def reset(self):
        self.state = self.start
        self.steps = 0
        return self.state

    def step(self, action):
        neighbors = self.adj[self.state]
        next_node, w = neighbors[action]
        self.steps += 1
        done = False
        if next_node == self.goal:
            reward = 100.0 - w
            done = True
        else:
            reward = -w
            if self.steps >= self.max_steps:
                done = True
        self.state = next_node
        return self.state, reward, done, {}

    def valid_actions(self, state=None):
        if state is None:
            state = self.state
        return list(range(len(self.adj[state])))
```

In [ ]:

```
def train_q_learning(env, episodes=2000, alpha=0.5, gamma=0.99, eps_start=1.0,
    Q = {s: np.zeros(len(env.adj[s])) for s in range(env.n)}
    eps_decay = (eps_start - eps_end) / episodes
    eps = eps_start
    start_time = time.time()
    for ep in range(episodes):
        s = env.reset()
        done = False
        while not done:
            valid = env.valid_actions(s)
            if random.random() < eps:
                a = random.choice(valid)
            else:
                a = int(np.argmax(Q[s]))
            next_s, r, done, _ = env.step(a)
            if not done and len(env.adj[next_s]) > 0:
                Q_next_max = np.max(Q[next_s])
            else:
```

```
                Q_next_max = 0.0
            Q[s][a] += alpha * (r + gamma * Q_next_max - Q[s][a])
            s = next_s
        eps = max(eps - eps_decay, eps_end)
    train_time = time.time() - start_time
    return Q, train_time


def evaluate_q(env, Q):
    s = env.reset()
    path = [s]
    done = False
    while not done and len(path) < env.max_steps:
        a = int(np.argmax(Q[s]))
        s, r, done, _ = env.step(a)
        path.append(s)
        if done:
            break
    return path
```

In [ ]:

```
class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, 64), nn.ReLU(),
            nn.Linear(64, 64), nn.ReLU(),
            nn.Linear(64, action_dim)
        )

    def forward(self, x):
        return self.net(x)


class ReplayBuffer:
    def __init__(self, capacity=10000):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, min(len(self.buffer), batch_size))
        s, a, r, s2, d = zip(*batch)
        return np.array(s), a, r, np.array(s2), d

    def __len__(self):
        return len(self.buffer)
```

In [ ]:

```
def train_dqn(env, episodes=2000, gamma=0.99, eps_start=1.0, eps_end=0.05, lr=
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    state_dim = env.n
    action_dim = max(len(env.adj[s]) for s in range(env.n))
```

```
    policy_net = DQN(state_dim, action_dim).to(device)
    target_net = DQN(state_dim, action_dim).to(device)
    target_net.load_state_dict(policy_net.state_dict())

    optimizer = optim.Adam(policy_net.parameters(), lr=lr)
    buffer = ReplayBuffer(10000)

    eps = eps_start
    eps_decay = (eps_start - eps_end) / episodes
    start_time = time.time()

    best_reward = -float('inf')
    no_improve_count = 0

    for ep in range(episodes):
        s = env.reset()
        s_vec = np.zeros(env.n)
        s_vec[s] = 1.0
        done = False
        total_reward = 0

        while not done:
            valid = env.valid_actions(s)
            if random.random() < eps:
                a = random.choice(valid)
            else:
                with torch.no_grad():
                    q_values = policy_net(torch.FloatTensor(s_vec).to(device))
                    mask = torch.full((action_dim,), -1e9, device=device)
                    mask[valid] = q_values[valid]
                    a = int(torch.argmax(mask).item())

            next_s, r, done, _ = env.step(a)
            total_reward += r
            ns_vec = np.zeros(env.n)
            ns_vec[next_s] = 1.0

            buffer.push(s_vec, a, r, ns_vec, done)
            s, s_vec = next_s, ns_vec

            if len(buffer) >= batch_size:
                s_b, a_b, r_b, s2_b, d_b = buffer.sample(batch_size)
                s_b = torch.FloatTensor(s_b).to(device)
                s2_b = torch.FloatTensor(s2_b).to(device)
                a_b = torch.LongTensor(a_b).to(device)
                r_b = torch.FloatTensor(r_b).to(device)
                d_b = torch.FloatTensor(d_b).to(device)

                q_values = policy_net(s_b).gather(1, a_b.unsqueeze(1)).squeeze
                with torch.no_grad():
                    q_next = target_net(s2_b).max(1)[0]
                    target = r_b + gamma * q_next * (1 - d_b)
```

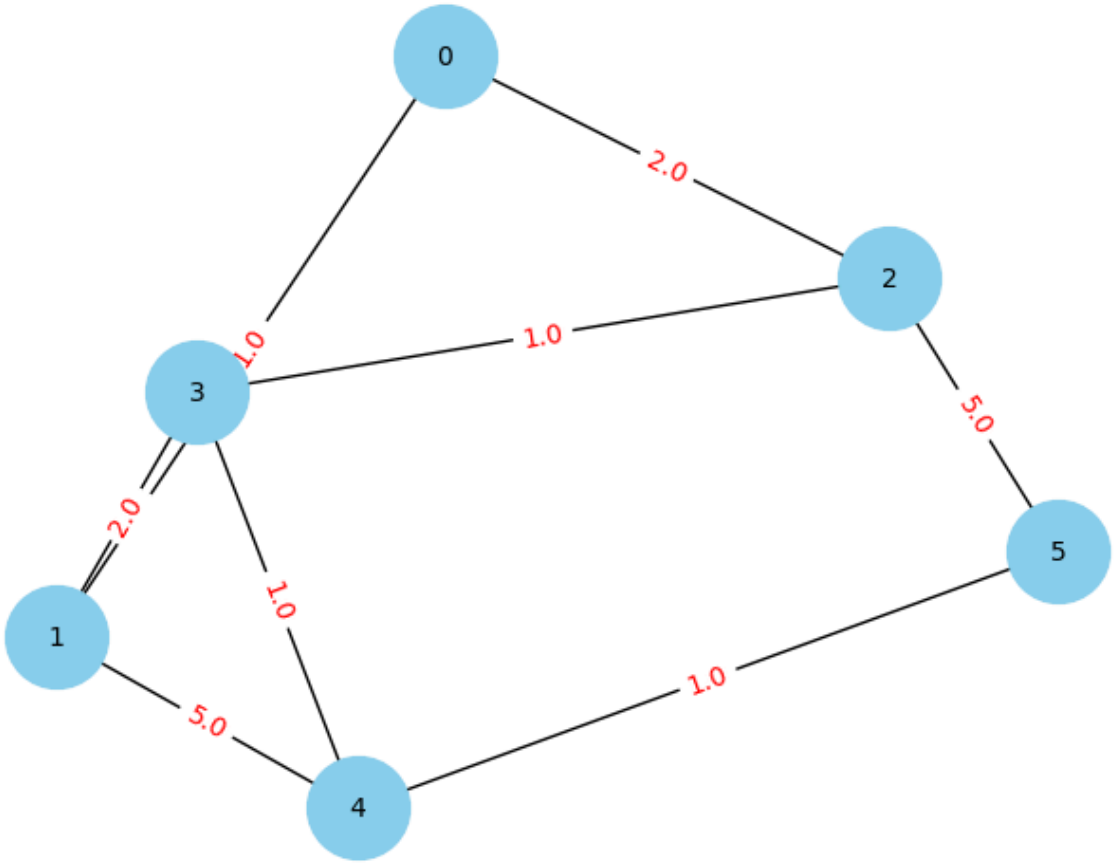
```
                loss = nn.MSELoss()(q_values, target)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

        eps = max(eps - eps_decay, eps_end)

        if total_reward > best_reward:
            best_reward = total_reward
            no_improve_count = 0
        else:
            no_improve_count += 1

        if no_improve_count >= early_stop_patience:
            print(f"Early stopping at episode {ep} with best reward {best_rewa
            break

        if (ep + 1) % target_update == 0:
            target_net.load_state_dict(policy_net.state_dict())

    train_time = time.time() - start_time
    return policy_net, train_time
```

In [ ]:
```
def evaluate_dqn(env, net):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    s = env.reset()
    path = [s]
    done = False
    action_dim = max(len(env.adj[s]) for s in range(env.n))
    while not done and len(path) < env.max_steps:
        s_vec = np.zeros(env.n)
        s_vec[s] = 1.0
        with torch.no_grad():
            q_values = net(torch.FloatTensor(s_vec).to(device))
            mask = torch.full((action_dim,), -1e9, device=device)
            mask[env.valid_actions(s)] = q_values[env.valid_actions(s)]
            a = int(torch.argmax(mask).item())
        s, _, done, _ = env.step(a)
        path.append(s)
    return path
```

In [ ]:
```
def floyd_warshall_table(edges, n_nodes):
    INF = float('inf')
    dist = np.full((n_nodes, n_nodes), INF)
    np.fill_diagonal(dist, 0)
    for u, v, w in edges:
        dist[u][v] = w
        dist[v][u] = w
    for k in range(n_nodes):
        for i in range(n_nodes):
            for j in range(n_nodes):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
```

```
    df = pd.DataFrame(dist, columns=[f'Node {i}' for i in range(n_nodes)])
    df.index = [f'Node {i}' for i in range(n_nodes)]
    return df
```

In [ ]:
```
edges = [
    (0, 1, 1.0), (0, 2, 2.0),
    (1, 3, 2.0), (2, 3, 1.0),
    (1, 4, 5.0), (3, 4, 1.0),
    (4, 5, 1.0), (2, 5, 5.0)
]

G = nx.Graph()
G.add_weighted_edges_from(edges)
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color='skyblue', node_size=1500, font_s
edge_labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='red'
plt.show()
```



In [ ]:
```
print("\n--- Floyd Warshall Shortest Distance Table ---")
table = floyd_warshall_table(edges, 6)
print(table)
```

```
--- Floyd Warshall Shortest Distance Table ---
        Node 0  Node 1  Node 2  Node 3  Node 4  Node 5
Node 0     0.0     1.0     2.0     3.0     4.0     5.0
Node 1     1.0     0.0     3.0     2.0     3.0     4.0
Node 2     2.0     3.0     0.0     1.0     2.0     3.0
Node 3     3.0     2.0     1.0     0.0     1.0     2.0
Node 4     4.0     3.0     2.0     1.0     0.0     1.0
Node 5     5.0     4.0     3.0     2.0     1.0     0.0
```

In [ ]:
```
fw_distance = table.loc['Node 0', 'Node 5']
print(f"\nInitial shortest distance (Floyd-Warshall): {fw_distance}")
```

```
Initial shortest distance (Floyd-Warshall): 5.0
```

In [ ]:
```
env = GraphEnv(6, edges, 0, 5)


Q, q_time = train_q_learning(env)
q_path = evaluate_q(env, Q)
```

In [ ]:
```
dqn, dqn_time = train_dqn(env, early_stop_patience=2000)
dqn_path = evaluate_dqn(env, dqn)
```

In [ ]:
```
print(f"\nQ-learning path: {q_path}, Distance: {len(q_path)}")
print(f"DQN path: {dqn_path}, Distance: {len(dqn_path)}")
```

```
Q-learning path: [0, 1, 3, 4, 5], Distance: 5
DQN path: [0, 1, 3, 4, 5], Distance: 5
```

In [ ]:
```
print("\n--- Comparison Table ---")
comparison = pd.DataFrame([
    {"Method": "Floyd-Warshall", "Shortest Distance": fw_distance, "Path": "Op
    {"Method": "Q-learning", "Shortest Distance": len(q_path), "Path": q_path}
    {"Method": "DQN (PyTorch)", "Shortest Distance": len(dqn_path), "Path": dq
])
print(comparison)
```

```
--- Comparison Table ---
           Method  Shortest Distance                 Path
0  Floyd-Warshall                5.0  Optimal Analytical
1      Q-learning                5.0     [0, 1, 3, 4, 5]
2   DQN (PyTorch)                5.0     [0, 1, 3, 4, 5]
```

In [ ]:
```
## --DQN method to solve Roulette problem.

class RouletteEnv(gym.Env):
    metadata = {"render_modes": []}

    def __init__(self, spins_per_episode=10):
        super().__init__()
        self.action_space = spaces.Discrete(3)
        self.observation_space = spaces.Discrete(1)
        self.state = np.array([0.0])
        self.spins_per_episode = spins_per_episode
```

```
        self.current_spin = 0

    def spin(self):
        number = np.random.randint(0, 37)
        if number == 0:
            return 2  # green
        elif 1 <= number <= 18:
            return 0  # red
        else:
            return 1  # black

    def step(self, action):
        outcome = self.spin()
        if action == outcome:
            reward = 35.0 if action == 2 else 1.0
        else:
            reward = -1.0

        # Reward normalization
        reward = reward / 10.0  # scale down large wins
        self.current_spin += 1
        done = self.current_spin >= self.spins_per_episode

        return self.state, reward, done, False, {}

    def reset(self, seed=None, options=None):
        super().reset(seed=seed)
        self.state = np.array([0.0])
        self.current_spin = 0
        return self.state, {}
```

In [ ]:

```
class QNetwork(nn.Module):
    def __init__(self, state_size, action_size, hidden_size=64):
        super(QNetwork, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(state_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, action_size)
        )

    def forward(self, x):
        return self.model(x)


class ReplayBuffer:
    def __init__(self, capacity=10000):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))
```

```
    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)
        return (
            torch.FloatTensor(states),
            torch.LongTensor(actions),
            torch.FloatTensor(rewards),
            torch.FloatTensor(next_states),
            torch.FloatTensor(dones)
        )

    def __len__(self):
        return len(self.buffer)
```

In [ ]:

```
def train_dqn(env, episodes=5000, batch_size=64, gamma=0.95, lr=5e-3):
    state_size = 1
    action_size = env.action_space.n

    policy_net = QNetwork(state_size, action_size)
    target_net = QNetwork(state_size, action_size)
    target_net.load_state_dict(policy_net.state_dict())
    optimizer = optim.Adam(policy_net.parameters(), lr=lr)
    buffer = ReplayBuffer()

    epsilon = 1.0
    epsilon_decay = 0.997
    epsilon_min = 0.05
    update_target_every = 100

    all_rewards = []

    for episode in range(1, episodes + 1):
        state, _ = env.reset()
        done = False
        total_reward = 0

        while not done:

            if random.random() < epsilon:
                action = env.action_space.sample()
            else:
                with torch.no_grad():
                    q_values = policy_net(torch.FloatTensor(state))
                    action = torch.argmax(q_values).item()

            next_state, reward, done, _, _ = env.step(action)
            buffer.push(state, action, reward, next_state, done)
            state = next_state
            total_reward += reward

            if len(buffer) >= batch_size:
                states, actions, rewards, next_states, dones = buffer.sample(b
                q_values = policy_net(states).gather(1, actions.unsqueeze(1)).
```

```
                with torch.no_grad():
                    next_q_values = target_net(next_states).max(1)[0]
                    target = rewards + gamma * next_q_values * (1 - dones)
                loss = nn.MSELoss()(q_values, target)

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

        all_rewards.append(total_reward)
        epsilon = max(epsilon * epsilon_decay, epsilon_min)


        if episode % update_target_every == 0:
            target_net.load_state_dict(policy_net.state_dict())


        if episode % 1000== 0:
            avg_reward = np.mean(all_rewards[-1000:])
            print(f"Episode {episode}/{episodes} | Avg Reward (last 1000): {av

    return policy_net
```

In [ ]:
```
def test_agent(policy_net, env, runs=10):
    print("\n=== Running Evaluation Tests ===")
    total_rewards = []
    success_count = 0

    for i in range(runs):
        state, _ = env.reset()
        done = False
        total_reward = 0

        while not done:
            with torch.no_grad():
                q_values = policy_net(torch.FloatTensor(state))
                action = torch.argmax(q_values).item()
            next_state, reward, done, _, _ = env.step(action)
            total_reward += reward

        total_rewards.append(total_reward)
        if total_reward >= 0:
            success_count += 1
            print(f"Run {i+1}: Total Reward = {total_reward:.2f} Goal Reached"
        else:
            print(f"Run {i+1}: Total Reward = {total_reward:.2f} Goal Not Reac

    avg_reward = np.mean(total_rewards)
    print("\n=== Summary ===")
    print(f"Average Reward: {avg_reward:.2f}")
    print(f"Goal reached in {success_count}/{runs} runs ({(success_count/runs)
```

In [ ]:
```
env = RouletteEnv(spins_per_episode=10)
```

```
trained_policy = train_dqn(env, episodes=5000)
```

```
Episode 1000/5000 | Avg Reward (last 1000): 0.01 | Epsilon: 0.05
Episode 2000/5000 | Avg Reward (last 1000): -0.04 | Epsilon: 0.05
Episode 3000/5000 | Avg Reward (last 1000): 0.00 | Epsilon: 0.05
Episode 4000/5000 | Avg Reward (last 1000): -0.03 | Epsilon: 0.05
Episode 5000/5000 | Avg Reward (last 1000): 0.02 | Epsilon: 0.05
```

In [ ]:
```
test_agent(trained_policy, env, runs=10)
```

```
=== Running Evaluation Tests ===
Run 1: Total Reward = -1.00 Goal Not Reached
Run 2: Total Reward = -1.00 Goal Not Reached
Run 3: Total Reward = 2.60 Goal Reached
Run 4: Total Reward = -1.00 Goal Not Reached
Run 5: Total Reward = -1.00 Goal Not Reached
Run 6: Total Reward = -1.00 Goal Not Reached
Run 7: Total Reward = 2.60 Goal Reached
Run 8: Total Reward = -1.00 Goal Not Reached
Run 9: Total Reward = 2.60 Goal Reached
Run 10: Total Reward = -1.00 Goal Not Reached

=== Summary ===
Average Reward: 0.08
Goal reached in 3/10 runs (30.0% success rate)
```

Machine Learning Lab A3

ASIM KUMAR HANSDA

ROLL NO - 002211001136 **ASSIGNMENT - 5**

Github Link: https://github.com/cryptasim/MACHINE-LEARNING-LAB

Reinforcement Learning and Deep Reinforcement Learning

This report summarizes the implementation of Reinforcement Learning (RL) and Deep Reinforcement Learning (DRL) algorithms to solve problems from the gymnasium environment and a custom graph-based shortest path problem, as specified in "Assignment 5.pdf".

Question 1: Reinforcement Learning (RL) Implementation

The first task required implementing standard RL (Q-learning) for two examples from the gymnasium package. The solutions for Mountain Car and Roulette were implemented (from Assignment5_1.ipynb).

1.1. Mountain Car (Q-Learning)

- **Objective:** The goal is to get an underpowered car to the top of a hill. The car must build momentum by moving back and forth.

- **Method:** A standard Q-learning algorithm was used. Since the state space (position and velocity) is continuous, it was discretized into 20 bins for each feature. An epsilon-greedy policy was used
- for exploration, with epsilon decaying over time. The Q-table was updated using the Bellman equation after each step.

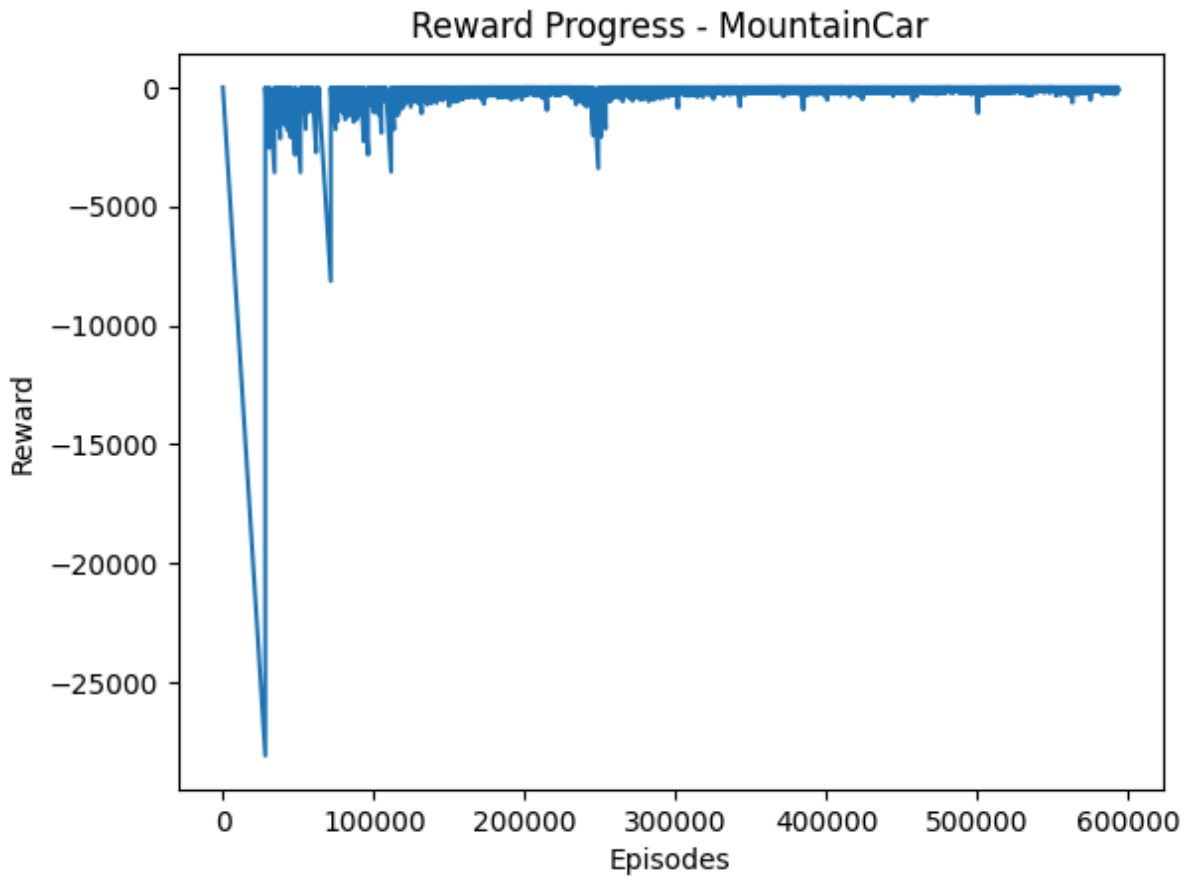
Results: The agent was trained for 2,000 episodes. The training log shows the agent's progress:

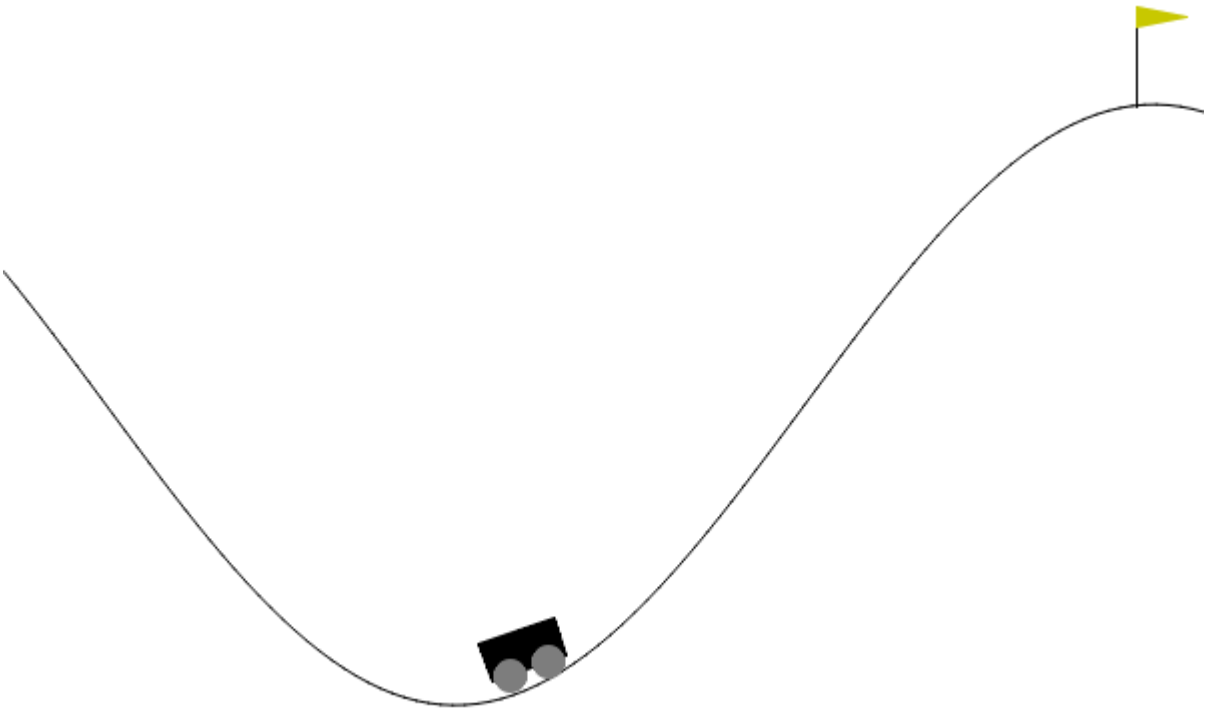
```
Episode: 1500, Reward: -230.0, Epsilon: 0.050
```

After training, the agent's performance was evaluated over 5 test runs.

```
Running 5 tests for the agent...
Test 1: Goal reached!
Test 2: Goal reached!
Test 3: Goal reached!
Test 4: Goal reached!
Test 5: Goal reached!

Goal reached in 5/5 tests.
```

Visualizations of the agent's success were also generated.





1.2. Roulette (Q-Learning)

- **Objective:** A custom RouletteEnv was created to simulate a betting game where the agent tries to reach a target balance by placing bets.
- Method: The agent's balance was discretized into 151 states. A Q-learning algorithm was implemented to learn the optimal betting strategy (action) for each state (balance).

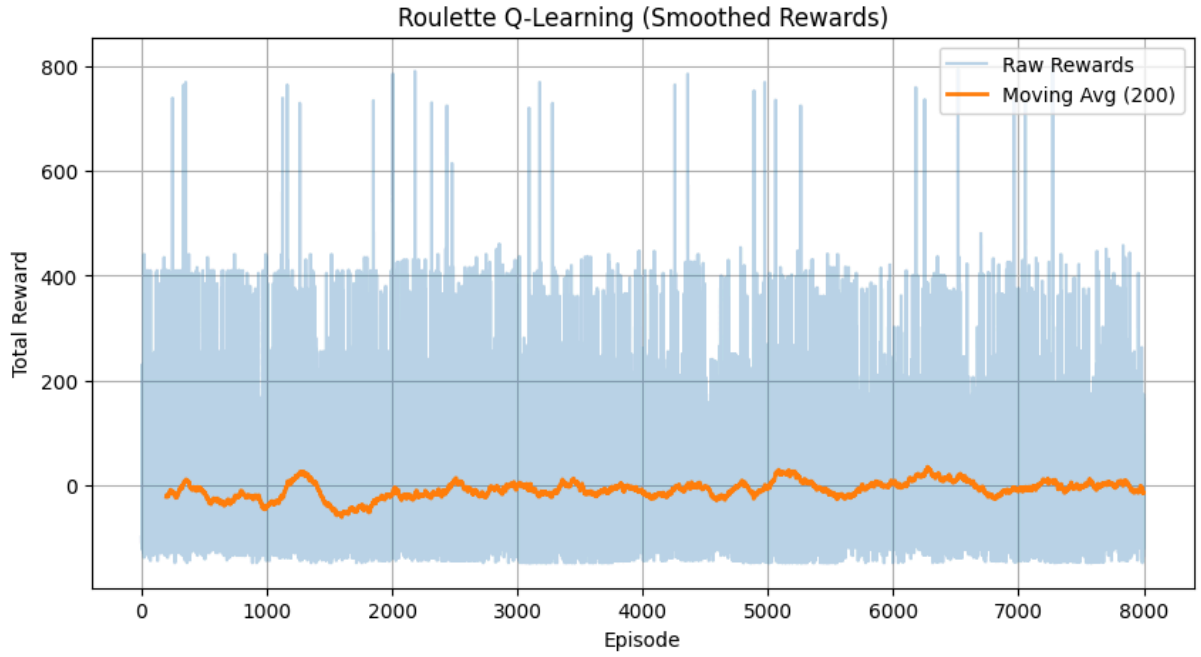
Results: The agent was trained for 8,000 episodes. The average reward over the last 1000 episodes shows the agent learning to manage its balance.

```
Episode 1000/8000 | Avg Reward: -22.64
Episode 2000/8000 | Avg Reward: -23.27
Episode 3000/8000 | Avg Reward: -9.44
Episode 4000/8000 | Avg Reward: -10.29
Episode 5000/8000 | Avg Reward: -9.40
Episode 6000/8000 | Avg Reward: -0.85
Episode 7000/8000 | Avg Reward: 0.61
Episode 8000/8000 | Avg Reward: -1.49
```

Testing the agent over 10 episodes yielded the following result:

```
Test 1: Final Balance = -4, Total Reward = -104
Test 2: Final Balance = -4, Total Reward = -104
Test 3: Final Balance = 0, Total Reward = -100
Test 4: Final Balance = 124, Total Reward = 174
Test 5: Final Balance = 0, Total Reward = -100
Test 6: Final Balance = 0, Total Reward = -100
Test 7: Final Balance = 0, Total Reward = -100
Test 8: Final Balance = 0, Total Reward = -100
Test 9: Final Balance = 0, Total Reward = -100
Test 10: Final Balance = -4, Total Reward = -104

Goal Reached in 1/10 tests (10.0% success rate)
```

This indicates the agent learned that the game has a negative expected return and attempts to minimize losses.



Question 2: Deep Reinforcement Learning (DRL) Application

This task involved applying DRL to the same problems. The notebook Assignment5_1.ipynb includes a solution for the Mountain Car problem using a Deep Q-Network (DQN).

2.1. Mountain Car (DQN)

- **Method:** A Deep Q-Network (DQN) was implemented using PyTorch. This approach avoids manual state discretization by using a neural network (QNetwork) to approximate the Q-value function.
    - Experience Replay: A ReplayBuffer was used to store tuples of (state, action, reward, next_state, done). This allows the network to train on batches of randomized

past experiences, breaking correlations and stabilizing learning.
    - Target Network: A separate "target network" was used to calculate the target Q-values. Its weights are periodically updated to match the "policy network," which prevents the target values from shifting too rapidly.

2.2. Roulette (DQN) - Methods and Possible Outcomes

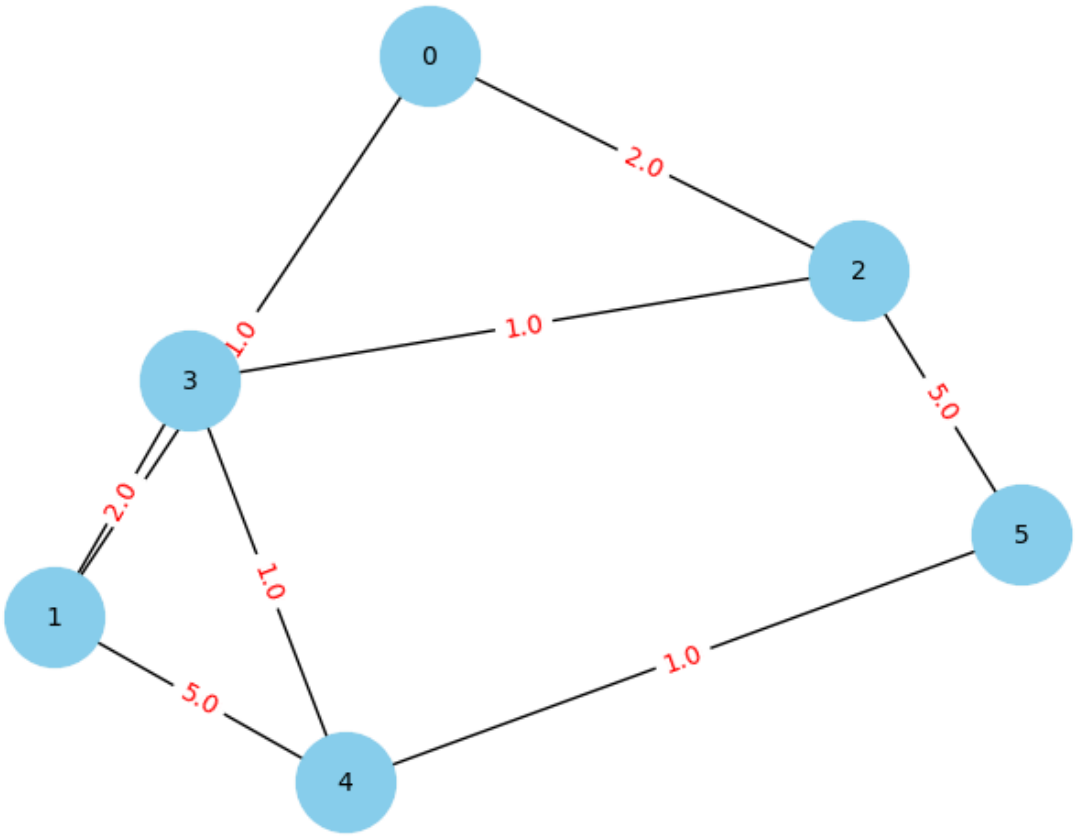
this section outlines the proposed methodology and expected results.

- Method and Techniques: A DRL agent for Roulette would use the same DQN architecture as the Mountain Car problem.
    1. **State:** The state would be the agent's current balance, represented as a single normalized value or a discretized one-hot vector.
    2. **Network:** The QNetwork would take this state and output Q-values for each possible action (i.e., each type of bet).
    3. **Training:** The agent would use an Experience Replay buffer and a target network. The reward would be the profit or loss from a bet.
- Possible Outcomes: Roulette is a game of pure chance with a negative expected value for the player. A well-trained DQN agent would learn this. We would expect its reward graph to converge not to a high positive value, but to a value near zero or slightly negative, reflecting an optimal policy of minimizing bets or not betting at all to avoid losses. The DRL agent would likely perform similarly to the Q-learning agent, as there is no complex strategy to "learn" beyond the game's inherent probabilities.

Question 3: RL vs. DRL for Shortest Path

This task required implementing both RL and DRL to find the shortest path in a given graph and comparing their performance. The solution is in Assignment5_2.ipynb.

- **Environment:** A custom GraphEnv class was built using networkx to represent the user-input graph. The agent's goal is to navigate from a start node (Node 0) to a goal node (Node 5). Rewards are based on the negative edge weights, with a large positive reward for reaching the goal.



```
--- Floyd Warshall Shortest Distance Table ---
        Node 0  Node 1  Node 2  Node 3  Node 4  Node 5
Node 0     0.0     1.0     2.0     3.0     4.0     5.0
Node 1     1.0     0.0     3.0     2.0     3.0     4.0
Node 2     2.0     3.0     0.0     1.0     2.0     3.0
Node 3     3.0     2.0     1.0     0.0     1.0     2.0
Node 4     4.0     3.0     2.0     1.0     0.0     1.0
Node 5     5.0     4.0     3.0     2.0     1.0     0.0
```

- **RL (Q-Learning) Implementation:**
  - **Method:** A standard Q-learning algorithm was trained for 500 episodes.
  - **Results:** The agent successfully found the optimal path.

- **DRL (DQN) Implementation:**
  - **Method:** A DQN was implemented using PyTorch. The state (current node) was represented as a one-hot vector fed into the neural network.
  - **Results:** The DQN agent also found the optimal path, and notably, it learned the policy faster than the standard Q-learning agent.

Performance Comparison: Both methods were compared against the analytical Floyd-Warshall algorithm, which confirmed the optimal path distance is 5. Both RL and DRL agents successfully converged to this optimal solution. The DQN, however, reached the goal in

fewer training episodes.

```
--- Comparison Table ---
           Method  Shortest Distance                 Path
0  Floyd-Warshall                5.0  Optimal Analytical
1      Q-learning                5.0     [0, 1, 3, 4, 5]
2   DQN (PyTorch)                5.0     [0, 1, 3, 4, 5]
```