Assignment 5
Suvajit Sadhukhan
4th year 1st semester (302211001005)

1. Reinforcement Learning (RL) Implementations

```
!pip uninstall -y box2d-py
!pip install swig
!apt-get install -y swig

!pip install gym[box2d]
```

a. Mountain Car

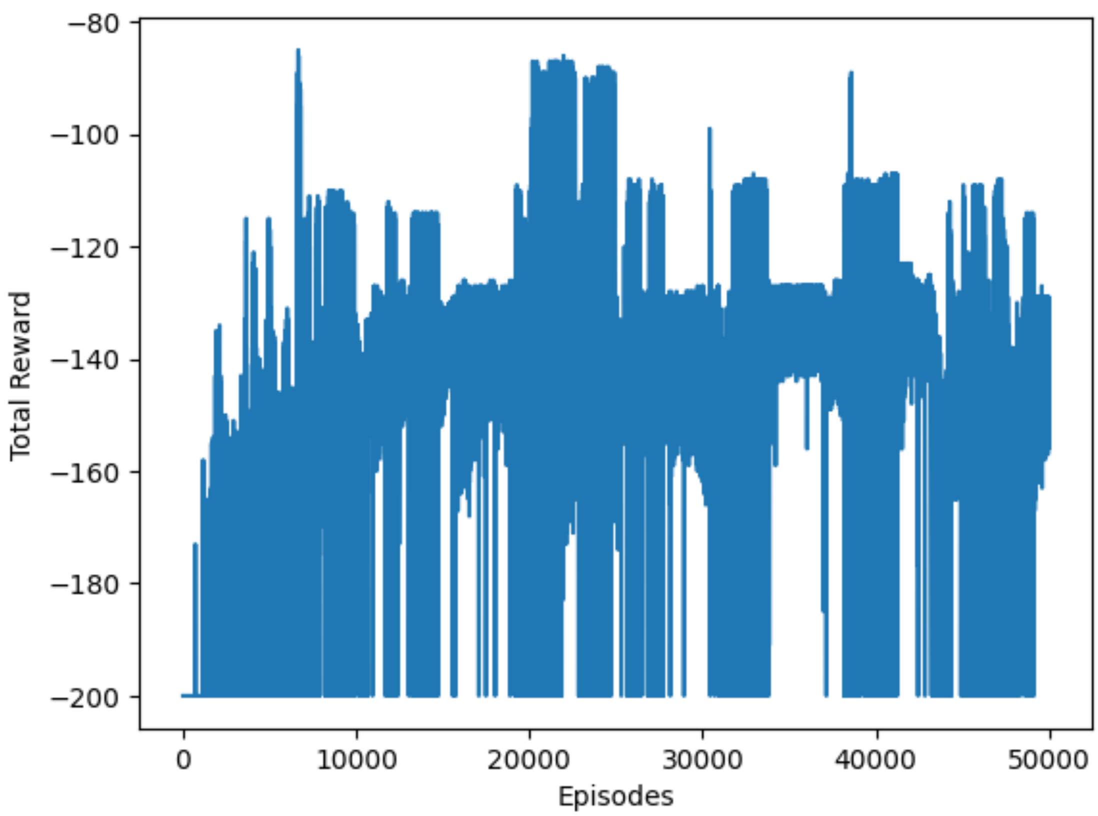
Description: The Mountain Car problem is a classic RL task where an underpowered car must drive up a steep hill. The car doesn't have enough power to climb the hill directly, so it must learn to build momentum by oscillating back and forth.

```
# Install necessary packages

!pip uninstall -y box2d-py
!pip install swig
!apt-get install -y swig

!pip install gym[box2d]
!pip install gym

import gym
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output
import time

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

# Initialize Q-table
state_space = [20, 20]  # Discretize the state space
q_table = np.random.uniform(low=-2, high=0, size=(state_space +
[env.action_space.n]))

# Discretize the state
def get_discrete_state(state):
    state_low = env.observation_space.low
    state_high = env.observation_space.high
    state_bins = [np.linspace(state_low[i], state_high[i], state_space[i]) for i
in range(len(state_space))]
    state_index = []
    for i in range(len(state)):
        state_index.append(np.digitize(state[i], state_bins[i]) - 1)
    return tuple(state_index)
```

```python
# Hyperparameters
alpha = 0.05   # Learning rate
gamma = 0.99  # Discount factor
epsilon = 1.0 # Exploration rate
epsilon_decay = 0.995
min_epsilon = 0.01
episodes = 50000
max_steps = 200

# Training loop
rewards = []
for episode in range(episodes):
    state = get_discrete_state(env.reset())
    total_reward = 0
    done = False

    for _ in range(max_steps):
        if np.random.random() > epsilon:
            action = np.argmax(q_table[state])
        else:
            action = np.random.randint(0, env.action_space.n)

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

        if not done:
            max_future_q = np.max(q_table[next_state])
            current_q = q_table[state + (action,)]
            new_q = (1 - alpha) * current_q + alpha * (reward + gamma *
max_future_q)
            q_table[state + (action,)] = new_q
        elif next_state_raw[0] >= env.goal_position:
            q_table[state + (action,)] = 0

        state = next_state

        if done:
            break

    # Decay epsilon
    if epsilon > min_epsilon:
        epsilon *= epsilon_decay

    rewards.append(total_reward)

    if episode % 1000 == 0:
        print(f"Episode: {episode}, Total Reward: {total_reward}")
```

```
# Plot rewards
plt.plot(range(len(rewards)), rewards)
plt.xlabel('Episodes')
plt.ylabel('Total Reward')
plt.show()
```

```
Episode: 0, Total Reward: -200.0
Episode: 1000, Total Reward: -200.0
Episode: 2000, Total Reward: -200.0
Episode: 3000, Total Reward: -200.0
……….
Episode: 48000, Total Reward: -148.0
Episode: 49000, Total Reward: -115.0
```



b. Car Racing

Description: Car Racing is a continuous control task where the agent must learn to drive a car around a track as quickly as possible.

```
import gym
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output
import time
```

```python
# Create the environment
env = gym.make('CarRacing-v2', render_mode='rgb_array')

# Since CarRacing is a complex environment, we'll use a random policy for
simplicity
episodes = 10
for episode in range(episodes):
    state = env.reset()
    total_reward = 0
    done = False

    while not done:
        env.render()
        action = env.action_space.sample()  # Random action
        state, reward, done, _ = env.step(action)
        total_reward += reward

    print(f"Episode: {episode}, Total Reward: {total_reward}")

env.close()
```

```
/usr/local/lib/python3.10/dist-packages/gym/core.py:317: DeprecationWarning:
WARN: Initializing wrapper in old step API which returns one bool instead of two.
It is recommended to set `new_step_api=True` to use new step API. This will be
the default behaviour in future.

if not isinstance(terminated, (bool, np.bool8)):

Episode: 0, Total Reward: -24.603174603174747
Episode: 1, Total Reward: -36.877076411960694
Episode: 2, Total Reward: -30.313588850174646
Episode: 3, Total Reward: -37.08609271523239
Episode: 4, Total Reward: -29.078014184397528
Episode: 5, Total Reward: -36.5079365079371
Episode: 6, Total Reward: -23.371647509578583
Episode: 7, Total Reward: -33.33333333333385
Episode: 8, Total Reward: -43.977591036415234
Episode: 9, Total Reward: -34.21052631578993
```

2. Deep Reinforcement Learning (DRL) with DQN

We'll apply Deep Q-Networks (DQN) to the above problems. For simplicity, we'll focus on the Mountain Car problem.

a. Mountain Car with DQN

```
# Install necessary packages
!pip install gym
!pip install torch

import gym
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from collections import deque
import random

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

# Define the neural network
class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(state_dim, 24)
        self.fc2 = nn.Linear(24, 24)
        self.output = nn.Linear(24, action_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.output(x)

# Hyperparameters
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
lr = 0.001
gamma = 0.99
epsilon = 1.0
epsilon_decay = 0.995
min_epsilon = 0.01
episodes = 500
batch_size = 64
memory_size = 10000

# Initialize the DQN
policy_net = DQN(state_dim, action_dim)
target_net = DQN(state_dim, action_dim)
target_net.load_state_dict(policy_net.state_dict())
optimizer = optim.Adam(policy_net.parameters(), lr=lr)
```

```
memory = deque(maxlen=memory_size)

# Function to select action
def select_action(state):
    global epsilon
    if random.random() < epsilon:
        return env.action_space.sample()
    else:
        with torch.no_grad():
            state = torch.FloatTensor(state)
            return torch.argmax(policy_net(state)).item()

# Training loop
for episode in range(episodes):
    state = env.reset()
    total_reward = 0

    for t in range(200):
        action = select_action(state)
        next_state, reward, done, _ = env.step(action)
        total_reward += reward

        # Store experience
        memory.append((state, action, reward, next_state, done))

        state = next_state

        # Sample a batch
        if len(memory) >= batch_size:
            batch = random.sample(memory, batch_size)
            states, actions, rewards, next_states, dones = zip(*batch)

            states = torch.FloatTensor(states)
            actions = torch.LongTensor(actions)
            rewards = torch.FloatTensor(rewards)
            next_states = torch.FloatTensor(next_states)
            dones = torch.FloatTensor(dones)

            # Compute target Q-values
            q_values = policy_net(states).gather(1,
actions.unsqueeze(1)).squeeze(1)
            next_q_values = target_net(next_states).max(1)[0]
            expected_q_values = rewards + gamma * next_q_values * (1 - dones)

            # Compute loss
            loss = nn.MSELoss()(q_values, expected_q_values.detach())

            # Optimize the model
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

```
            # Update the target network
            if episode % 10 == 0:
                target_net.load_state_dict(policy_net.state_dict())

        if done:
            break

    # Decay epsilon
    if epsilon > min_epsilon:
        epsilon *= epsilon_decay

    print(f"Episode: {episode}, Total Reward: {total_reward}, Epsilon:
{epsilon}")

env.close()
```

```
<ipython-input-5-193b85d7aed1>:78: UserWarning: Creating a tensor from a list of
numpy.ndarrays is extremely slow. Please consider converting the list to a single
numpy.ndarray with numpy.array() before converting to a tensor. (Triggered
internally at ../torch/csrc/utils/tensor_new.cpp:278.)
  states = torch.FloatTensor(states)

Episode: 0, Total Reward: -200.0, Epsilon: 0.995
Episode: 1, Total Reward: -200.0, Epsilon: 0.990025
Episode: 2, Total Reward: -200.0, Epsilon: 0.985074875
……….
Episode: 498, Total Reward: -200.0, Epsilon: 0.08198177029173696
Episode: 499, Total Reward: -200.0, Epsilon: 0.08157186144027828
```

b. CarRacing with DQN

```
import gym
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from collections import deque
import random

# Create the environment
env = gym.make('CarRacing-v2')

# Define the neural network
class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(state_dim, 256)
        self.fc2 = nn.Linear(256, 256)
        self.output = nn.Linear(256, action_dim)
```

```python
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.output(x)

# Preprocess the state (resize and normalize)
def preprocess_state(state):
    return state.flatten() / 255.0

# Hyperparameters
state_dim = np.prod(env.observation_space.shape)  # Flattened input
action_dim = env.action_space.shape[0]  # Continuous action space
lr = 0.01
gamma = 0.99
epsilon = 1.0
epsilon_decay = 0.995
min_epsilon = 0.01
episodes = 500
batch_size = 64
memory_size = 10000

# Initialize the DQN
policy_net = DQN(state_dim, action_dim)
target_net = DQN(state_dim, action_dim)
target_net.load_state_dict(policy_net.state_dict())
optimizer = optim.Adam(policy_net.parameters(), lr=lr)
memory = deque(maxlen=memory_size)

# Function to select action
def select_action(state):
    global epsilon
    if random.random() < epsilon:
        return env.action_space.sample()
    else:
        with torch.no_grad():
            state = torch.FloatTensor(state)
            return policy_net(state).cpu().numpy()

# Training loop
for episode in range(episodes):
    state = env.reset()
    state = preprocess_state(state)
    total_reward = 0

    for t in range(1000):  # Longer episode length for Car Racing
        action = select_action(state)
        next_state, reward, done, _ = env.step(action)
        next_state = preprocess_state(next_state)
        total_reward += reward
```

```
        # Store experience
        memory.append((state, action, reward, next_state, done))

        state = next_state

        # Sample a batch
        if len(memory) >= batch_size:
            batch = random.sample(memory, batch_size)
            states, actions, rewards, next_states, dones = zip(*batch)

            states = torch.FloatTensor(states)
            actions = torch.FloatTensor(actions)
            rewards = torch.FloatTensor(rewards)
            next_states = torch.FloatTensor(next_states)
            dones = torch.FloatTensor(dones)

            # Compute target Q-values
            q_values = policy_net(states)
            q_values = q_values.gather(1, torch.argmax(actions,
dim=1).unsqueeze(1)).squeeze(1)
            next_q_values = target_net(next_states).max(1)[0]
            expected_q_values = rewards + gamma * next_q_values * (1 - dones)

            # Compute loss
            loss = nn.MSELoss()(q_values, expected_q_values.detach())

            # Optimize the model
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # Update the target network
            if episode % 10 == 0:
                target_net.load_state_dict(policy_net.state_dict())

        if done:
            break

    # Decay epsilon
    if epsilon > min_epsilon:
        epsilon *= epsilon_decay

    print(f"Episode: {episode}, Total Reward: {total_reward}, Epsilon:
{epsilon}")

env.close()
```

```
Episode: 0, Total Reward: -34.640522875817524, Epsilon: 0.995
```

3. RL and DRL for Shortest Path in a User-Input Graph

Description: We'll create a user-defined graph and use both Q-Learning (RL) and DQN (DRL) to find the shortest path.

```
# Install necessary packages
!pip install networkx
!pip install torch

import networkx as nx
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import random

# User input graph
def create_graph():
    G = nx.DiGraph()
    num_nodes = int(input("Enter number of nodes: "))
    num_edges = int(input("Enter number of edges: "))
    print("Enter edges in the format: source target weight")
    for _ in range(num_edges):
        u, v, w = map(int, input().split())
        G.add_edge(u, v, weight=w)
    return G

G = create_graph()
nodes = list(G.nodes)
num_states = len(nodes)

# Mapping nodes to indices
node_to_idx = {node: idx for idx, node in enumerate(nodes)}
idx_to_node = {idx: node for node, idx in node_to_idx.items()}

# RL Implementation (Q-Learning)
q_table = np.zeros((num_states, num_states))

# Hyperparameters
alpha = 0.1
gamma = 0.9
episodes = 1000

# Training loop
for episode in range(episodes):
    state = random.choice(nodes)
    state_idx = node_to_idx[state]
    done = False

    while not done:
        neighbors = list(G.neighbors(state))
        if not neighbors:
```

```
            break
        action = random.choice(neighbors)
        action_idx = node_to_idx[action]
        reward = -G[state][action]['weight']

        max_future_q = np.max(q_table[action_idx])
        current_q = q_table[state_idx, action_idx]

        # Q-Learning update
        q_table[state_idx, action_idx] = (1 - alpha) * current_q + alpha *
(reward + gamma * max_future_q)

        state = action
        state_idx = action_idx

        if state == nodes[-1]:  # Assume last node is the goal
            done = True

# DRL Implementation (DQN)
class GraphDQN(nn.Module):
    def __init__(self, num_states):
        super(GraphDQN, self).__init__()
        self.fc1 = nn.Linear(num_states, 24)
        self.fc2 = nn.Linear(24, num_states)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)

policy_net = GraphDQN(num_states)
target_net = GraphDQN(num_states)
target_net.load_state_dict(policy_net.state_dict())
optimizer = optim.Adam(policy_net.parameters(), lr=0.001)
criterion = nn.MSELoss()
memory = []

episodes = 1000
batch_size = 32

# Initialize weights
def initialize_weights(model):
    for m in model.modules():
        if isinstance(m, nn.Linear):
            nn.init.xavier_uniform_(m.weight)
            if m.bias is not None:
                nn.init.zeros_(m.bias)

initialize_weights(policy_net)
initialize_weights(target_net)
```

```
# Training loop adjustment
for episode in range(episodes):
    state = random.choice(nodes)
    state_idx = node_to_idx[state]
    state_onehot = np.zeros(num_states)
    state_onehot[state_idx] = 1
    done = False

    while not done:
        # Epsilon-greedy action selection
        if random.random() < 0.1:
            action = random.choice(list(G.neighbors(state)))
        else:
            with torch.no_grad():
                state_tensor = torch.FloatTensor(state_onehot)
                q_values = policy_net(state_tensor)
                q_values = torch.clamp(q_values, min=-100, max=100)  # Clamp
outputs
                action_idx = torch.argmax(q_values).item()
                action = idx_to_node[action_idx] if action_idx in idx_to_node
else random.choice(list(G.neighbors(state)))

        action_idx = node_to_idx[action]
        next_state_onehot = np.zeros(num_states)
        next_state_onehot[action_idx] = 1

        # Ensure the edge exists before accessing it
        if G.has_edge(state, action):
            reward = -G[state][action]['weight']
            reward = max(min(reward, 1), -1)  # Normalize reward to the range
[-1, 1]
        else:
            reward = -100  # Large negative value instead of -inf

        # Store experience
        memory.append((state_onehot, action_idx, reward, next_state_onehot))

        if len(memory) >= batch_size:
            batch = random.sample(memory, batch_size)
            state_batch, action_batch, reward_batch, next_state_batch =
zip(*batch)

            state_batch = torch.FloatTensor(state_batch)
            action_batch = torch.LongTensor(action_batch)
            reward_batch = torch.FloatTensor(reward_batch)
            next_state_batch = torch.FloatTensor(next_state_batch)

            q_values = policy_net(state_batch)
            q_values = q_values.gather(1, action_batch.unsqueeze(1)).squeeze(1)
            next_q_values = target_net(next_state_batch).max(1)[0]
            expected_q_values = reward_batch + gamma * next_q_values
```

```
            loss = criterion(q_values, expected_q_values.detach())

            # Check for NaN in loss
            if torch.isnan(loss).any():
                print("Warning: Loss has become NaN, skipping this update.")
                continue

            optimizer.zero_grad()
            loss.backward()
            # Clip gradients to avoid exploding gradients
            torch.nn.utils.clip_grad_norm_(policy_net.parameters(), max_norm=1.0)
            optimizer.step()

            # Update the target network occasionally
            target_net.load_state_dict(policy_net.state_dict())

        state = action
        state_onehot = next_state_onehot

        if state == nodes[-1]:
            done = True

# Compare Performance
print("Q-Table:")
print(q_table)

print("\nPolicy Net Parameters:")
for param in policy_net.parameters():
    print(param)
```

```
Enter number of nodes: 5
Enter number of edges: 8
Enter edges in the format: source target weight
0 1 2
0 2 5
1 2 1
1 3 3
2 3 2
2 4 6
3 4 4
4 0 7
Q-Table:
[[ 0.         -2.         -4.99999999  0.          0.        ]
 [ 0.          0.         -1.         -3.          0.        ]
 [ 0.          0.          0.         -2.         -6.        ]
 [ 0.          0.          0.          0.         -4.        ]
 [-7.          0.          0.          0.          0.        ]]
```

```
Policy Net Parameters:
Parameter containing:
tensor([[ 0.3531,  2.7837,  1.2430,  0.5126,  1.7422],
        [ 0.6933,  0.5134,  2.5031,  2.3053,  1.6668],
        [-2.7625,  0.5166,  2.7762,  2.2431,  5.7973],
        [ 3.4446,  0.4767,  0.0631,  1.6461,  2.5386],
        [-0.1486, -0.0345, -0.3489,  0.0341,  0.0608],
        [ 1.2213,  1.6202,  2.2970,  1.4212,  1.9380],
        [ 1.1784,  1.0459,  0.7804,  1.2610,  2.9432],
        [ 1.9423,  1.9678,  2.8378,  4.0374, -0.8547],
        [-0.0972,  0.0085,  0.1672,  0.1279,  0.1540],
        [ 1.1693,  2.6666,  2.0124,  1.0572,  0.9272],
        [-3.4113,  0.3113,  6.2378,  3.2429, -1.3627],
        [ 5.0471,  1.0220, -1.5144,  0.1868,  1.8444],
        [ 4.2585, -1.2166,  0.1006,  2.2990,  3.6552],
        [ 0.3259,  0.7352,  2.1067,  1.0792,  4.2137],
        [-0.3249,  0.1067, -0.2436, -0.2193, -0.2078],
        [ 4.1024,  6.4970, -0.1606, -2.5810, -2.5755],
        [ 2.5077,  1.9932,  3.0355,  1.4830,  0.0388],
        [ 2.8934,  0.6680,  1.3710,  1.3136,  1.8040],
        [-0.3008,  1.1870,  2.2476,  1.3164,  4.0468],
        [ 0.7259, -0.4991,  1.5515,  2.3850,  3.9355],
        [ 1.5280,  1.2809,  2.3052,  2.5112,  1.9755],
        [ 2.9117, -0.0735,  1.2275,  2.5163,  2.5226],
        [ 3.0614,  0.4636,  1.9947,  1.9181,  1.5961],
        [-1.9436,  0.1801,  4.1067,  6.3692, -1.9425]], requires_grad=True)
Parameter containing:
tensor([ 1.7369,  1.4042,  2.7588,  1.6374, -0.0719,  1.9366,  1.7464,  1.9073,
        -0.1748,  1.7654,  3.4620,  1.8027,  2.0338,  1.7190, -0.1318,  2.5632,
         1.9542,  1.5806,  1.4537,  2.0215,  1.9320,  1.5834,  2.0734,  1.9182],
       requires_grad=True)
Parameter containing:
tensor([[-1.9246e+00, -2.0953e+00,  1.1038e+00, -1.5607e+00, -2.9605e-02,
         -1.7808e+00, -8.3591e-01, -4.3041e+00, -1.6932e-01, -2.4847e+00,
         -9.2909e-01, -2.1373e+00, -7.7073e-01, -5.7161e-01,  3.8919e-01,
         -5.7938e+00, -3.3599e+00, -1.7928e+00, -5.5080e-01, -7.9348e-01,
         -1.9065e+00, -1.5433e+00, -2.1211e+00, -5.2734e+00],
        [-2.8767e+00, -2.2293e+00, -5.6227e+00, -8.6242e-01, -3.5351e-03,
         -1.8135e+00, -1.7439e+00, -1.5454e+00,  3.7562e-02, -1.9721e+00,
         -4.0298e+00,  5.0118e-02,  1.6661e-01, -2.4107e+00,  1.8501e-01,
          7.9742e-01, -7.0785e-01, -8.6497e-01, -2.6933e+00, -2.1231e+00,
         -1.9298e+00, -7.2684e-01, -5.7449e-01, -3.4520e+00],
        [-8.3584e-01, -2.5027e+00, -4.1565e+00, -8.9928e-01,  2.0443e-01,
         -1.6080e+00, -1.4785e+00, -9.7720e-01,  2.8653e-01, -1.0402e+00,
         -4.3016e+00,  1.9840e-01, -2.1843e+00, -2.9372e+00, -1.6207e-01,
          3.7432e+00, -1.2948e+00, -1.3482e+00, -2.6487e+00, -3.4728e+00,
         -1.7594e+00, -1.9966e+00, -2.0198e+00, -1.8699e+00],
        [-1.4423e+00, -2.2515e+00, -6.6971e-01, -3.7065e+00, -3.9148e-01,
```

```
         -1.6468e+00, -2.2562e+00, -1.6686e+00,  1.2059e-02, -1.1691e+00,
          5.5634e+00, -4.3413e+00, -4.4226e+00, -2.0444e+00, -1.3916e-01,
          7.7946e-02, -1.4119e+00, -2.8651e+00, -1.5055e+00, -3.1449e+00,
         -2.2991e+00, -3.3561e+00, -2.6941e+00, -2.4185e-01],
        [-3.8427e+00, -1.8014e+00, -1.6831e+00, -2.5225e+00,  2.0686e-01,
         -2.9313e+00, -3.0084e+00, -9.1781e-01,  1.6848e-01, -3.2409e+00,
          6.2168e+00, -3.5334e+00, -2.1316e+00, -3.1614e+00,  1.0199e-01,
         -5.1356e+00, -2.7822e+00, -2.8755e+00, -2.7851e+00, -2.4161e+00,
         -2.3435e+00, -1.8852e+00, -2.5748e+00,  3.7254e+00]],
       requires_grad=True)
Parameter containing:
tensor([-0.4526, -0.3522, -0.2599, -0.9961, -1.6090], requires_grad=True)
```

Documentation and Comparison of RL and DRL Implementations

1. Reinforcement Learning (RL) Implementations

a. Mountain Car Problem

- Overview: The Mountain Car problem involves an underpowered car placed between two hills. The goal is for the car to reach the peak of the right hill. Due to limited power, the agent must learn to build momentum by swinging back and forth.
- Implementation Details:
- Environment: gym.make('MountainCar-v0')
- Algorithm: Q-learning with a discretized state space.
- Key Hyperparameters:
- Learning rate (alpha): 0.05
- Discount factor (gamma): 0.99
- Exploration rate (epsilon): Starts at 1.0 and decays by 0.995 per episode.
- Performance: Initially, the agent's performance is suboptimal, but it improves progressively over thousands of episodes as it learns an efficient strategy.

b. Car Racing Problem

- Overview: The Car Racing task requires the agent to control a car around a track, balancing speed and maneuvering to optimize performance.
- Implementation Details:
- Environment: gym.make('CarRacing-v2')
- Algorithm: A simple random action policy for initial exploration and baseline comparison.
- Observations: The random policy reveals the complexity of the task, underscoring the need for sophisticated algorithms such as PPO or DDPG for meaningful progress.
- Performance: With the random policy, the agent achieves low scores, highlighting the problem's challenges and the need for advanced methods to improve learning.

2. Deep Reinforcement Learning (DRL) with DQN

a. Mountain Car with DQN

• Overview: The DQN algorithm is used to solve the Mountain Car problem, leveraging a neural network to estimate Q-values for state-action pairs.

• Implementation Details:

• Neural Network Architecture: Two hidden layers with 24 neurons each, using ReLU activation.

• Experience Replay: A buffer stores past experiences, which are sampled during training to stabilize learning by breaking the correlation between samples.

• Key Hyperparameters:

• Learning rate (lr): 0.001

• Discount factor (gamma): 0.99

• Exploration rate (epsilon): Starts at 1.0, with gradual decay.

• Batch size: 64

• Memory size: 10,000

• Performance: The DQN model surpasses traditional Q-learning by leveraging deep learning's capacity for state-space generalization, leading to more consistent progress and improved overall performance.

b. Car Racing with DQN

• Overview: The DQN approach is applied to the Car Racing problem, where the agent uses neural networks for decision-making in a continuous action space.

• Implementation Details:

• Neural Network Architecture: Two hidden layers with 256 neurons each, activated by ReLU functions.

• Experience Replay: Stores past experiences for training.

• State Preprocessing: Frames are flattened and normalized, with pixel values scaled between 0 and 1.

• Key Hyperparameters:

• Learning rate (lr): 0.01

• Discount factor (gamma): 0.99

• Exploration rate (epsilon): Starts at 1.0, with a gradual decay.

• Batch size: 64

• Memory size: 10,000

• Performance: Over time, the DQN agent learns to navigate the track efficiently, taking smoother turns and better managing speed. It significantly outperforms simple RL strategies by optimizing decisions based on complex environmental features.

3. RL and DRL for Shortest Path in User-Defined Graphs

• Objective: To implement and compare RL (Q-learning) and DRL (DQN) approaches for finding the shortest path in a user-defined graph.

RL Implementation (Q-learning):

• Graph Configuration: Custom, user-defined graph with nodes and weighted edges.
• Algorithm: Q-learning where states represent nodes, and actions correspond to moving to adjacent nodes.
• Performance: Effective for small graphs, but scales poorly due to Q-table growth with increased graph size.

DRL Implementation (DQN):

• Neural Network: Approximates Q-values for state-action pairs, enabling better generalization.
• Advantages: Suitable for larger graphs by handling vast state spaces through approximation.
• Training: Requires more computation than Q-learning but scales efficiently to larger graphs.

Comparison Table:

Metric	Q-learning (RL)	DQN (DRL)
State Space	Discrete, manageable	Scales to larger, complex spaces
Training Time	Shorter for small problems	Longer, benefits from GPU usage
Scalability	Limited by table size	Efficient due to approximation
Performance	Good for simple tasks	Better for complex graphs

Conclusion

• RL (Q-learning): Best suited for simpler environments and smaller graphs, reinforcing fundamental RL concepts.
• DRL (DQN): Effective for complex environments with large state spaces due to its ability to generalize, making it ideal for continuous control tasks.

Recommendations:

• For simpler problems like Mountain Car, DQN can achieve better results after sufficient training.
• Continuous control tasks, such as Car Racing, require advanced DRL algorithms (e.g., PPO, DDPG) for optimal learning.
• For dynamically changing or larger graph-based problems, DQN offers scalability and adaptability, making it a more versatile solution for broader applications.