Faculty of Technology and Engineering

Chandubhai S. Patel Institute of Technology

Date: / /

Practical Performa

Academic Year	:	2025-26	Semester	:	7th
Course code	:	OCCSE4001	Course name	:	Reinforcement Learning

Practical- No. 4

Aim: To implement the Deep Q-Network (DQN) algorithm using neural networks for Q-value approximation and train an agent with experience replay and target network in the CartPole-v1 environment.

Code:

RL_Prac4.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

[7]

```python
import gymnasium as gym
import numpy as np
import random, collections
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

[3]

```python
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = collections.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)
        state, action, reward, next_state, done = map(np.array, zip(*batch))
        return state, action, reward, next_state, done

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

[4]

```python
class QNetwork(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.fc1 = nn.Linear(state_dim, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, action_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)
```

RL_Prac4.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

[5] 0s

```python
class DQNAgent:
    def __init__(self, state_dim, action_dim, lr=1e-3, gamma=0.99, tau=0.005, buffer_size=10000, batch_size=64):
        self.q_net = QNetwork(state_dim, action_dim).to(device)
        self.target_net = QNetwork(state_dim, action_dim).to(device)
        self.target_net.load_state_dict(self.q_net.state_dict())
        self.optimizer = optim.Adam(self.q_net.parameters(), lr=lr)

        self.gamma = gamma
        self.tau = tau
        self.batch_size = batch_size
        self.replay = ReplayBuffer(buffer_size)
        self.action_dim = action_dim

    def select_action(self, state, epsilon):
        if random.random() < epsilon:
            return random.randint(0, self.action_dim - 1)
        state = torch.FloatTensor(state).unsqueeze(0).to(device)
        with torch.no_grad():
            q_vals = self.q_net(state)
        return q_vals.argmax().item()

    def update(self):
        if len(self.replay) < self.batch_size:
            return
        s, a, r, s2, d = self.replay.sample(self.batch_size)

        s = torch.FloatTensor(s).to(device)
        a = torch.LongTensor(a).unsqueeze(1).to(device)
        r = torch.FloatTensor(r).unsqueeze(1).to(device)
        s2 = torch.FloatTensor(s2).to(device)
        d = torch.FloatTensor(d).unsqueeze(1).to(device)

        # Compute Q targets
        with torch.no_grad():
            max_next_q = self.target_net(s2).max(1, keepdim=True)[0]
            target_q = r + (1 - d) * self.gamma * max_next_q
```

RL_Prac4.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

[5] 0s

```python
        # Compute Q targets
        with torch.no_grad():
            max_next_q = self.target_net(s2).max(1, keepdim=True)[0]
            target_q = r + (1 - d) * self.gamma * max_next_q

        # Current Q estimates
        curr_q = self.q_net(s).gather(1, a)

        # Loss
        loss = nn.MSELoss()(curr_q, target_q)

        # Optimize
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # Soft update target network
        for target_param, param in zip(self.target_net.parameters(), self.q_net.parameters()):
            target_param.data.copy_(target_param.data * (1.0 - self.tau) + param.data * self.tau)
```

[8] 4m

```python
env = gym.make("CartPole-v1")

state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
agent = DQNAgent(state_dim, action_dim)

MAX_EPISODES = 300
MAX_STEPS = 500
epsilon_start = 1.0
epsilon_end = 0.01
epsilon_decay = 500  # steps

rewards = []
steps_done = 0
```

RL_Prac4.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

[8] 4m

```
    steps_done = 0

    for ep in range(1, MAX_EPISODES + 1):
        state, _ = env.reset()
        ep_reward = 0
        for t in range(MAX_STEPS):
            epsilon = epsilon_end + (epsilon_start - epsilon_end) * np.exp(-1.0 * steps_done / epsilon_decay)
            action = agent.select_action(state, epsilon)
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            agent.replay.push(state, action, reward, next_state, done)
            agent.update()

            state = next_state
            ep_reward += reward
            steps_done += 1

            if done:
                break

        rewards.append(ep_reward)
        print(f"Episode {ep}, Reward: {ep_reward}, Epsilon: {epsilon:.3f}")
```

RL_Prac4.ipynb

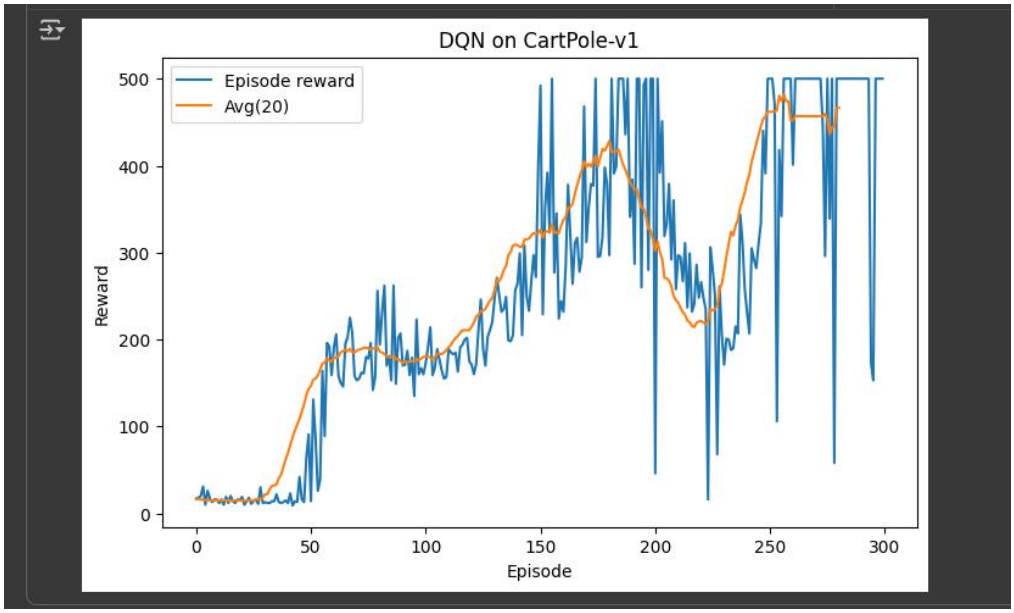
File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

[9] 1s

```
plt.figure(figsize=(8,5))
plt.plot(rewards, label="Episode reward")
plt.plot(np.convolve(rewards, np.ones(20)/20, mode='valid'), label="Avg(20)")
plt.xlabel("Episode")
plt.ylabel("Reward")
plt.title("DQN on CartPole-v1")
plt.legend()
plt.show()
```

Output:



Grade/Marks **Sign of Lab Teacher with Date**

(_____ / 10)