Solving CartPole with Deep Q-Network (DQN) in PyTorch

Below is a complete implementation of a DQN agent to solve the CartPole-v1 environment from OpenAI Gym.

1. Import Required Libraries

```
import gym
import numpy as np
import random
from collections import deque
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
```

2. Define the Q-Network

```
class DQN(nn.Module):
    def __init__(self, state_size, action_size):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(state_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, action_size)

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

PROF

3. Define the DQN Agent

```
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=10000)
        self.gamma = 0.95    # discount rate
        self.epsilon = 1.0   # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
```

```
        self.learning_rate = 0.001
        self.model = DQN(state_size, action_size)
        self.optimizer = optim.Adam(self.model.parameters(),
lr=self.learning_rate)

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        state = torch.FloatTensor(state)
        act_values = self.model(state)
        return torch.argmax(act_values).item()

    def replay(self, batch_size):
        if len(self.memory) < batch_size:
            return

        minibatch = random.sample(self.memory, batch_size)

        states = torch.FloatTensor(np.array([t[0] for t in minibatch]))
        actions = torch.LongTensor(np.array([t[1] for t in minibatch]))
        rewards = torch.FloatTensor(np.array([t[2] for t in minibatch]))
        next_states = torch.FloatTensor(np.array([t[3] for t in
minibatch]))
        dones = torch.FloatTensor(np.array([t[4] for t in minibatch]))

        # Current Q values
        current_q = self.model(states).gather(1, actions.unsqueeze(1))

        # Next Q values
        next_q = self.model(next_states).max(1)[0].detach()
        target_q = rewards + (1 - dones) * self.gamma * next_q

        # Compute loss and update
        loss = F.mse_loss(current_q.squeeze(), target_q)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # Decay epsilon
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def save(self, filename):
        torch.save(self.model.state_dict(), filename)

    def load(self, filename):
        self.model.load_state_dict(torch.load(filename))
```

PROF

4. Training Loop

```python
def train_dqn(episodes=1000, batch_size=32):
    env = gym.make('CartPole-v1')
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    agent = DQNAgent(state_size, action_size)
    scores = []

    for e in range(episodes):
        state = env.reset()
        state = np.array(state)
        total_reward = 0

        for time in range(500):  # Max steps per episode
            # env.render()  # Uncomment to visualize training
            action = agent.act(state)
            next_state, reward, done, _ = env.step(action)
            next_state = np.array(next_state)

            # Custom reward shaping can be added here if needed
            reward = reward if not done else -10

            agent.remember(state, action, reward, next_state, done)
            state = next_state
            total_reward += reward

            if done:
                break

        scores.append(total_reward)
        agent.replay(batch_size)

        # Print progress
        avg_score = np.mean(scores[-100:])  # Last 100 episodes
        print(f"Episode: {e+1}/{episodes}, Score: {total_reward}, Avg Score: {avg_score:.2f}, Epsilon: {agent.epsilon:.2f}")

        # Early stopping if solved
        if avg_score >= 195:  # CartPole-v1 solved condition
            print(f"Solved in {e+1} episodes!")
            agent.save('cartpole_dqn.pth')
            break

    env.close()
    return scores
```


5. Run Training and Plot Results

```python
if __name__ == "__main__":
    # Train the agent
    scores = train_dqn(episodes=500)

    # Plot the learning progress
    plt.figure(figsize=(10, 5))
    plt.plot(scores)
    plt.title('DQN Training Progress')
    plt.xlabel('Episode')
    plt.ylabel('Score')
    plt.grid(True)

    # Add moving average
    window_size = 100
    moving_avg = np.convolve(scores, np.ones(window_size)/window_size,
mode='valid')
    plt.plot(moving_avg, color='red', label=f'{window_size}-episode
average')
    plt.legend()

    plt.show()
```

6. Evaluate the Trained Agent

```python
def evaluate_agent():
    env = gym.make('CartPole-v1')
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    agent = DQNAgent(state_size, action_size)
    agent.load('cartpole_dqn.pth')
    agent.epsilon = 0  # No exploration during evaluation

    total_rewards = []

    for e in range(10):  # Run 10 evaluation episodes
        state = env.reset()
        state = np.array(state)
        total_reward = 0

        for time in range(500):
            env.render()  # Visualize the trained agent
            action = agent.act(state)
            next_state, reward, done, _ = env.step(action)
            state = np.array(next_state)
            total_reward += reward

            if done:
                break
```

PROF

```
        total_rewards.append(total_reward)
        print(f"Evaluation Episode {e+1}, Score: {total_reward}")

    env.close()
    print(f"\nAverage score over 10 episodes:
{np.mean(total_rewards):.2f}")

# Uncomment to evaluate
# evaluate_agent()
```

Key Components Explained:

1. **Q-Network**: A simple 3-layer neural network that approximates the Q-function.
2. **Experience Replay**: Stores transitions (state, action, reward, next_state) in a replay buffer for more efficient learning.
3. **Epsilon-Greedy Policy**: Balances exploration (random actions) and exploitation (actions with highest Q-values).
4. **Target Network**: While not implemented here (for simplicity), a common DQN improvement is to use a separate target network that's updated less frequently.
5. **Training Loop**: The agent interacts with the environment, stores experiences, and learns from random batches of these experiences.

Potential Improvements:

1. **Target Network**: Add a separate target network that's updated periodically to stabilize training.
2. **Double DQN**: Decouple action selection from value estimation to reduce overestimation bias.
3. **Prioritized Experience Replay**: Sample important transitions more frequently.
4. **Dueling DQN**: Separate the network into value and advantage streams.
5. **Hyperparameter Tuning**: Adjust learning rate, batch size, network architecture, etc.

This implementation should solve CartPole-v1 (reach an average score of 195+ over 100 episodes) within 200-300 episodes. The evaluation script lets you watch your trained agent balance the pole.