深度强化学习在CartPole游戏中的应用案例

1 应用案例说明

CartPole是OpenAI Gym提供的一个标准强化学习测试环境。在这个简单的仿真中，目标是通过横向移动底部的小车来平衡上方的杆子，防止其倒下。这一任务是强化学习领域中经典的动态平衡问题，经常被用来测试不同算法的效果。

1.1 环境参数描述

该环境的状态由以下四个参数组成：

- **小车位置（Position）**：表示小车在一维轨道上的位置。
- **小车速度（Velocity）**：小车的移动速度。
- **杆的角度（Angle）**：杆子与垂直方向的偏离角度。
- **杆的角速度（Angular Velocity）**：杆子偏离角度变化的速度。

1.2 目标

开发一个基于深度强化学习的智能体，利用DQN（Deep Q-Network）算法学习控制策略，使得杆子能够尽可能长时间保持平衡。

2 算法实现核心思路

2.1 DQN算法详细介绍

DQN结合了传统的Q学习和现代的深度学习技术，通过一个深度神经网络来近似Q值函数。这种方法允许智能体在连续的、高维的状态空间中做出决策，是解决复杂问题的一种有效方式。

2.1.1 关键特性

1. **经验回放（Experience Replay）**：该技术通过保存智能体的经历（状态、动作、奖励等），并在训练过程中随机抽取这些经历来训练网络，有效地打破样本之间的时间相关性，增加训练的稳定性。
2. **目标网络（Target Network）**：DQN算法采用两个结构相同但参数更新频率不同的网络：一个快速更新的策略网络和一个缓慢更新的目标网络。这种设计减少了学习过程中目标Q值的波动，从而提高了算法的稳定性。

2.2 实现步骤详解

1. **网络设计**：使用全连接层构建神经网络，输入层接受四个状态参数，输出层根据状态输出两个可能动作的Q值（向左或向右推动小车）。
2. **经验回放机制**：构建经验池，用于存储智能体的状态转移，训练时从中随机抽取样本进行学习，以此增强数据的独立性和代表性。
3. **损失函数**：使用均方误差（MSE）衡量实际输出Q值和目标Q值之间的差异，指导网络参数的调整。
4. **epsilon-greedy策略**：在初期采取较高的探索率以发现更多可能的策略，随着学习的进行，逐步减少探索率，增加对已学习策略的利用，以此平衡探索和利用。

5. **模型训练与更新**：

   - 利用Adam优化器对网络进行参数更新，确保学习过程的高效和稳定。
   - 定期将策略网络的参数复制到目标网络，确保目标Q值的稳定性。

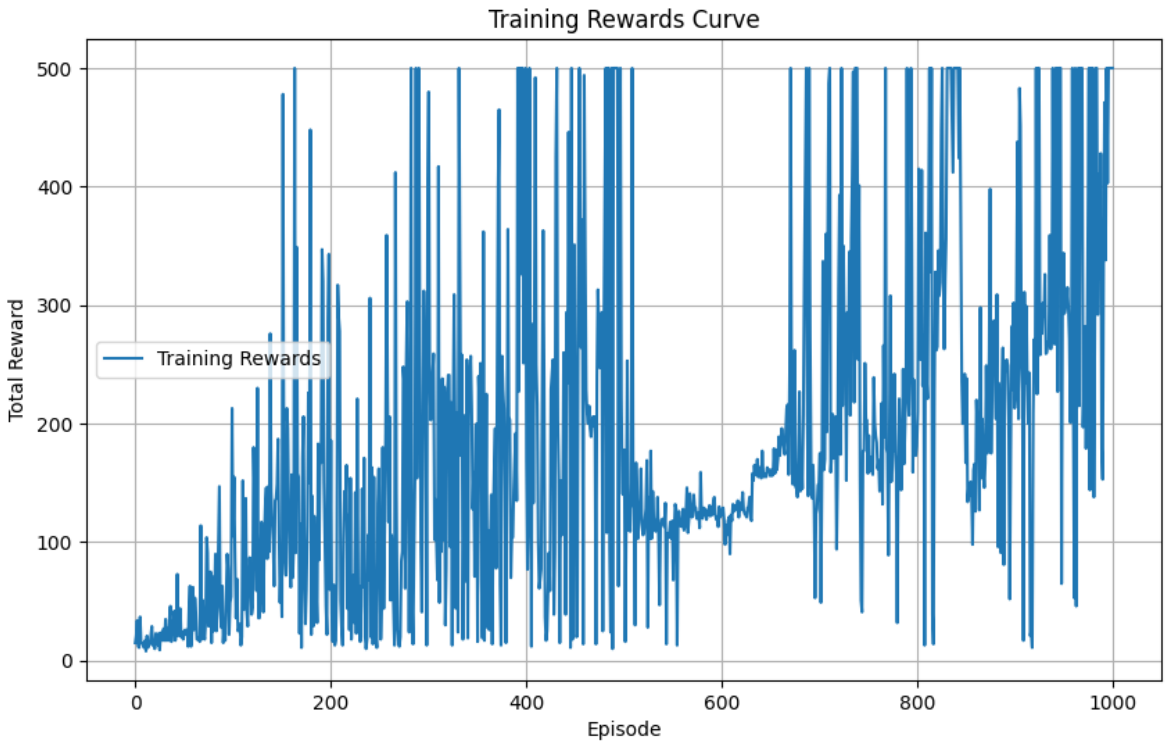
```python
class DqnNetwork(nn.Module):
    def __init__(self, stateDim, actionDim):
        super(DqnNetwork, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(stateDim, 128),  # 增加网络容量
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, actionDim)
        )

    def forward(self, x):
        return self.fc(x)
```

3 实验结果与分析

3.1 训练曲线

训练过程中智能体的表现逐渐改善，奖励值的提升反映了策略的进步。初始阶段由于高探索率，智能体的表现不稳定，但随着epsilon值的逐渐减小，智能体开始利用已学到的策略，奖励值增长更加显著。



3.2 测试结果

经过充分训练，智能体在CartPole环境中的表现显著提高，最终测试的平均奖励接近理论最高值。这验证了DQN算法在处理此类动态平衡任务时的有效性和可靠性。

4 结论

本案例展示了深度强化学习技术在复杂控制任务中的应用潜力。通过合理设计的DQN算法，不仅实现了高效的学习过程，还成功解决了CartPole游戏中的平衡挑战，体现了该技术的广泛适用性和强大能力。

5 GitHub

https://github.com/Myocardial-infarction-Jerry/Reinforcement-Learning

核心代码展示：

```python
def train(epsilon):  # Training and checkpoint saving
    for episode in tqdm(range(episodes), desc="TrainingProgress"):
        state, _ = env.reset()
        state = np.array(state)
        totalReward = 0

        for t in range(500):
            action = selectAction(state, epsilon)
            nextState, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            # Check if the cart position exceeds bounds
            cartPosition = nextState[0]
            if abs(cartPosition) > env.unwrapped.x_threshold:
                done = True
                reward = -1.0  # Penalize for going out of bounds

            nextState = np.array(nextState)
            replayBuffer.push(state, action, reward, nextState, done)
            state = nextState
            totalReward += reward

            if done:
                break

            if len(replayBuffer) > batchSize:
                states, actions, rewards_, nextStates, dones = replayBuffer.sample(
                    batchSize)

                statesTensor = torch.FloatTensor(np.array(states)).to(device)
                actionsTensor = torch.LongTensor(
                    actions).unsqueeze(1).to(device)
                rewardsTensor = torch.FloatTensor(
                    rewards_).unsqueeze(1).to(device)
                nextStatesTensor = torch.FloatTensor(
                    np.array(nextStates)).to(device)
                donesTensor = torch.FloatTensor(dones).unsqueeze(1).to(device)

```

```
            qValues = policyNetwork(statesTensor).gather(1, actionsTensor)
            nextQValues = targetNetwork(
                nextStatesTensor).max(1, keepdim=True)[0]
            targetQValues = rewardsTensor + gamma * \
                nextQValues * (1 - donesTensor)

            loss = lossFunction(qValues, targetQValues)

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

    epsilon = max(epsilon * epsilonDecay, epsilonMin)
    if episode % targetUpdateFrequency == 0:
        targetNetwork.load_state_dict(policyNetwork.state_dict())

    rewardList.append(totalReward)
```