中国科学技术大学　微电子学院

《神经网络及其应用》实验报告五

姓名　杨翊麟　　**学号**　BC24219010

时间　2025/06/24　　**指导教师**　徐奇

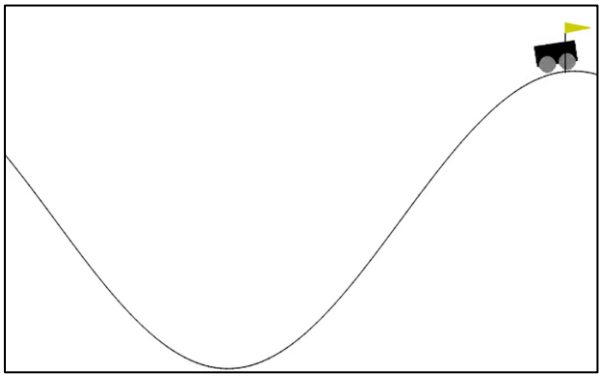
实验名称:基于 MountainCar-v0 环境的智能体训练强化学习算法对比研究

一、实验目的

1、了解强化学习的常用实验环境：Gym 库；
2、熟悉强化学习在连续控制任务中的应用意义；
3、基于 MountainCar-v0 环境依次比较 Q-learning、DQN、double DQN、double dueling DQN 算法的特性与差异。

二、实验准备

1、**Gym 环境：**MountainCar-v0，目标：将小车推上斜坡（水平位置达到≥0.5）、状态空间：位置与速度（范围：position [-1.2, 0.6]、velocity [-0.07, 0.07]）、动作空间：向左加速 0、不加速 1、向右加速 2



2、**实验环境：**Python：3.10 版本，gymnasium：1.1.1 版本，
torch：2.7.1+cu126
3、**评价指标：**平均奖励值（每 10 回合平均回报、训练回报）、稳定性（奖励标准差矩阵）

三、算法原理

1、**Q-Learning**，使用表格（Q-table）存储**状态-动作值函数 *Q(s, a)***，每次经历一个状态转移 (*s, a, r, s'*) 后，更新对应的 Q 值：

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

其中：α：学习率，γ：折扣因子

特点：简单易实现，但只能处理离散状态空间，高维度难以存储

2、**DQN（Deep Q-Network）**，使用**神经网络近似 Q 函数**，解决 Q-learning 无法处理高维/连续状态空间的问题。目标函数为：

$$\mathcal{L} = (y - Q(s,a;\theta))^2, \quad y = r + \gamma \max_{a'} Q(s',a';\theta^-)$$

其中：θ：当前网络参数；θ^-：目标网络参数（固定一段时间后更新）；采用**经验回放**打乱数据相关性，提升稳定性。

特点：能处理高维/连续状态，比 Q-Learning 泛化能力强，但 Q 值容易过估计，尤其在随机环境中，训练稳定性不高，对超参数较敏感。

3、**Double DQN（DDQN）**，解决 DQN 的 Q 值过估计问题，引入两组 Q 值：用一组选择动作，用另一组评估动作：

$$y = r + \gamma Q(s', \arg\max_{a'} Q(s',a';\theta);\theta^-)$$

使用当前网络 θ 选择动作，用目标网络 θ^-来计算 Q 值。

特点：有效缓解 Q 值过估计，提升稳定性和性能，通常收敛更快，性能优于 DQN，但算法更复杂

4、**Dueling Double DQN（Dueling DQN + Double DQN）**，在 Double DQN 基础上，引入 **Dueling 架构**：将 Q 值拆成两个子网络：

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right)$$

其中：*V(s)*：状态价值函数；*A(s, a)*：动作优势函数（相对于状态的偏好）。

特点：在某些状态下（如 MountainCar-v0 大部分时间无法控制方向），能更有效地学习**“状态值”**与**“动作优势”**，结合了 Double DQN 的优势，性能更稳、更强。但网络结构更复杂，在简单任务中优势不明显，适用于大型问题。

四、算法实现细节（double_dueling_dqn_mountaincar_optimized.py）

1）DuelingQNet 类

```
# 定义 Dueling DQN 的神经网络结构
class DuelingQNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim, action_dim):
        super().__init__()
        # 特征提取层
        self.feature = torch.nn.Sequential(
            torch.nn.Linear(state_dim, hidden_dim),  # 输入状态，输出隐藏特征
            torch.nn.ReLU()
        )
        # 优势函数分支 A(s,a)
        self.advantage = torch.nn.Sequential(
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, action_dim)  # 输出每个动作的优势值
        )
        # 状态值函数分支 V(s)
        self.value = torch.nn.Sequential(
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, 1)  # 只输出一个状态值
        )

    def forward(self, x):
        # 前向传播，融合 V(s) 和 A(s,a)
        x = self.feature(x)
        advantage = self.advantage(x)  # 计算优势
        value = self.value(x)  # 计算状态值
        return value + advantage - advantage.mean()  # Dueling DQN 合并公式
        # Q(s,a) = V(s) + A(s,a) - mean(A(s,a')) 用于去除冗余偏差
```

2）DQNAgent 类

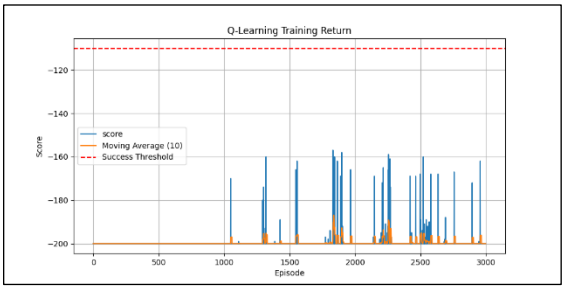
```
# 定义 DQN 智能体类
class DQNAgent:
    def __init__(self, state_dim, action_dim, hidden_dim=128, gamma=0.99,
lr=1e-3,
                 epsilon_start=1.0, epsilon_end=0.01, epsilon_decay=500,
                 target_update=100, buffer_size=10000, batch_size=64,
                 device='cpu', double_dqn=True):
```

```
        self.device = torch.device(device)  # 指定运行设备（CPU/GPU）
        self.action_dim = action_dim  # 动作空间维度

        # 初始化策略网络和目标网络
        self.policy_net = DuelingQNet(state_dim, hidden_dim,
action_dim).to(self.device)
        self.target_net = DuelingQNet(state_dim, hidden_dim,
action_dim).to(self.device)
        self.target_net.load_state_dict(self.policy_net.state_dict())  # 同步
目标网络参数

        self.optimizer = torch.optim.Adam(self.policy_net.parameters(),
lr=lr)  # 优化器

        self.replay_buffer = ReplayBuffer(buffer_size)  # 经验回放池
        self.gamma = gamma  # 折扣因子
        self.batch_size = batch_size  # 批次大小
        self.target_update = target_update  # 目标网络更新周期
        self.double_dqn = double_dqn  # 是否启用 Double DQN

        # ε-greedy 策略参数
        self.epsilon_start = epsilon_start
        self.epsilon_end = epsilon_end
        self.epsilon_decay = epsilon_decay
        self.total_steps = 0  # 用于记录步数以衰减 ε

        self.writer = SummaryWriter()  # TensorBoard 写入器

    def take_action(self, state):
        # 根据 ε-greedy 策略选择动作
        epsilon = self.epsilon_end + (self.epsilon_start - self.epsilon_end)
* \
                  math.exp(-1. * self.total_steps / self.epsilon_decay)
        self.writer.add_scalar("Epsilon", epsilon, self.total_steps)  # 记录
ε 的变化
        self.total_steps += 1
        if np.random.rand() < epsilon:
            return np.random.randint(self.action_dim)  # 随机探索
        state = torch.tensor([state], dtype=torch.float32).to(self.device)
        with torch.no_grad():
            return self.policy_net(state).argmax().item()  # 贪婪选择

    def update(self):
        if len(self.replay_buffer) < self.batch_size:
```

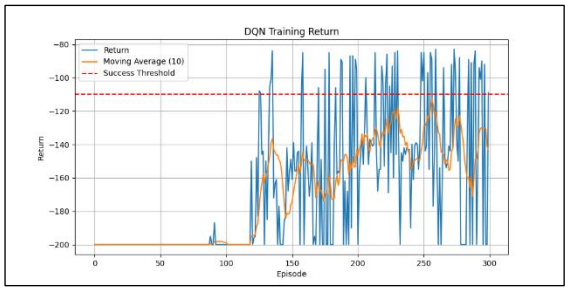
```
            return  # 样本不足，暂不更新

        # 从经验池采样一批数据
        transitions = self.replay_buffer.sample(self.batch_size)
        states = torch.tensor(transitions.state,
dtype=torch.float32).to(self.device)
        actions = torch.tensor(transitions.action,
dtype=torch.int64).unsqueeze(1).to(self.device)
        rewards = torch.tensor(transitions.reward,
dtype=torch.float32).unsqueeze(1).to(self.device)
        next_states = torch.tensor(transitions.next_state,
dtype=torch.float32).to(self.device)
        dones = torch.tensor(transitions.done,
dtype=torch.float32).unsqueeze(1).to(self.device)

        # 计算当前 Q(s,a)
        q_values = self.policy_net(states).gather(1, actions)  # 根据动作索引
取对应的 Q 值

        with torch.no_grad():
            if self.double_dqn:
                # Double DQN: 动作由 policy_net 决策，Q 值由 target_net 提供
                next_actions = self.policy_net(next_states).argmax(1,
keepdim=True)
                next_q_values = self.target_net(next_states).gather(1,
next_actions)
            else:
                # 普通 DQN: 直接使用 target_net 的最大 Q 值
                next_q_values = self.target_net(next_states).max(1,
keepdim=True)[0]

        # TD 目标值计算: r + γ * Q'
        targets = rewards + self.gamma * next_q_values * (1 - dones)

        # 计算均方误差损失
        loss = F.mse_loss(q_values, targets)

        # 反向传播更新网络
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # 记录损失到 TensorBoard
        self.writer.add_scalar("Loss", loss.item(), self.total_steps)
```

```
        # 每隔 target_update 步更新目标网络
        if self.total_steps % self.target_update == 0:
            self.target_net.load_state_dict(self.policy_net.state_dict())
```

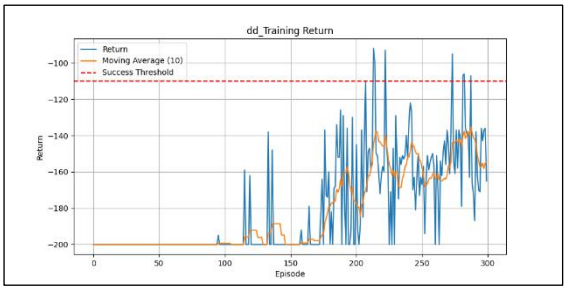
五、实验结果分析



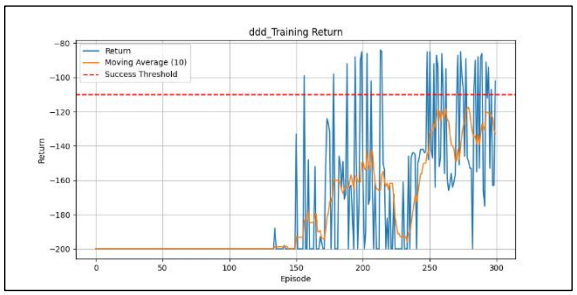
（a）Q-Learning 训练回报



（b）DQN 训练回报



（c）Double DQN 训练回报



（d）Dueling Double DQN 训练回报

从训练曲线可以看出，不同算法在 MountainCar-v0 环境中的表现存在显著差异。

Q-Learning 在连续状态空间中几乎无法学习有效策略，训练了超过 3000 个回合仍长期停留在最低得分 -200 附近，表明其不适用于此类问题。相比之下，

DQN 能够利用神经网络对状态进行有效建模，训练回报明显提升，但由于存在 Q 值过估计问题，回报曲线波动较大，策略稳定性不佳。

Double DQN 通过引入双网络机制，有效缓解了过估计问题，使训练回报更加稳定，平均得分高于 DQN，表现更加稳健。

进一步地，Dueling Double DQN 在架构上区分了状态价值和动作优势的估计，提升了在稀疏奖励场景下的学习效率，在学习速度、策略稳定性和最终表现上优于前三者。

综上，Q-Learning 表现最差，而 Dueling Double DQN 综合性能最佳，适合解决连续状态、稀疏奖励的问题。

六、组内成员分工

杨翊麟：主要负责项目代码的实现，参与项目的前期算法实验与论证，完成了 Double Dueling DQN 算法，额外开展了 CartPole-v1 和 Acrobot-v1 任务的实验。

易平川：主要负责报告的撰写，开展项目的前期算法实验与论证，开展了基础的 DQN 算法的比较实验。

杨礼睿：主要负责 PPT 制作与汇报，参与项目的前期算法实验与论证，补充实现了 Q-learning 算法，辅助报告的撰写。

七、个人总结

杨翊麟：

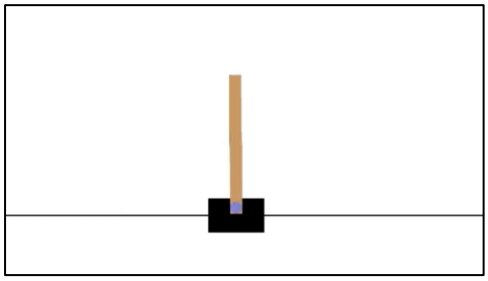
本次实验主要围绕 MountainCar-v0 环境，分别实现了 Q-Learning、DQN、Double DQN 和 Dueling Double DQN 四种强化学习算法，并对比了它们的训练效果。在实现过程中，我深入理解了各算法的原理，尤其是深度强化学习中神经网络如何近似 Q 函数，以及经验回放和目标网络等机制对训练稳定性的影响。

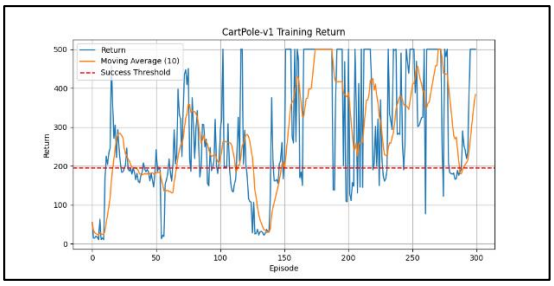
实验结果表明，传统的 Q-Learning 难以在连续状态空间中发挥作用，几乎无法收敛；而 DQN 利用神经网络较好地解决了这一问题，但存在较大的 Q 值过估计与训练波动。Double DQN 在此基础上引入了双网络结构，显著提升了训练稳定性和策略表现。进一步的 Dueling Double DQN 则在结构上引入了状态价值和动作优势的拆分，在面对稀疏奖励问题时具备更强的学习能力。

通过此次实验，我不仅加深了对各类 DQN 算法的理解，也掌握了强化学习模型的完整实现流程及调试方法，对训练稳定性、模型泛化能力等问题有了更加直观的认识。这为我今后深入研究更复杂的强化学习任务打下了坚实的基础。

*八、附加

（a）CartPole-v1 实验结果





（b）Acrobot-v1 实验结果