中山大学计算机学院

人工智能

本科生实验报告

课程名称：Artificial Intelligence

学号	23336266	姓名	熊彦钧

一、 实验题目

实现 DQN 算法：在 CartPole-v0 环境中实现 DQN 算法。最终算法性能的评判标准：以算法收敛的 reward大小、收敛所需的样本数量给分。reward 越高（至少是 180 ，最大是 200 ）、收敛所需样本数量越少，分数越高。

Submission： 作业提交内容：需提交一个 zip 文件，包括代码以及实验报告 PDF。实验报告除了需要写 writing 部分的内容，还需要给出 reward 曲线图以及算法细节。
相关代码下载地址：https://github.com/ZYC9894/2024AI/tree/main/Homework/Experiment；
相关环境的说明文档：https://www.gymlibrary.dev/；
或 https://www.gymlibrary.dev/environments/classic_control/cart_pole/；

Supplement： 我们给出 DQN 在 cartpole 环境的训练曲线图作为参考。

二、 实验内容

1. 算法原理

DQN 的核心是通过神经网络近似 Q 函数（动作价值函数），结合经验回放（Experience Replay）和目标网络（Target Network）来稳定训练过程。其目标是学习一个策略，使得智能体在环境中获得的累计奖励最大化。

DQN 的三个关键组件的作用分别如下：

（1）Q-Network（在线网络）的作用：输入当前状态，输出每个动作的 Q 值（未来累计奖励的估计）。

Q-Network 的结构：3 层全连接神经网络（输入层→隐藏层→隐藏层→输出层），使用 ReLU 激活函数。

（2）Target Network（目标网络）的作用：计算目标 Q 值，用于稳定训练。其参数定期从在线网络复制而来。

（3）经验回放（Replay Buffer）的作用：存储转移样本（状态、动作、奖励、下一状态、终止标志），随机采样批次数据打破时序相关性。

DQN 的算法流程如下：

（1）动作选择（ε-贪心策略）

- **探索：**以概率 ε 随机选择动作（初始 ε=1.0）。

- **利用**：以概率 1-ε 选择当前 Q 值最大的动作。
- **ε 衰减**：ε 按 epsilon_decay（0.996）逐步衰减，最低至 epsilon_min（0.0001）。

（2）训练步骤

1. **采样**：从经验回放中随机抽取一个批次（batch_size=128）。
2. **计算目标 Q 值**：
   - 对于非终止状态：target = reward + γ * max(Q_target(next_state))
     （γ=0.95 为折扣因子）
   - 对于终止状态：target = reward
3. **计算损失**：均方误差（MSE）损失，比较在线网络的预测 Q 值和目标 Q 值。
4. **反向传播**：通过梯度下降（Adam 优化器，初始 lr=0.001）更新在线网络。

（3）学习率衰减

- 每 500 步按 lr_decay=0.9 衰减学习率，最低至 lr_min=0.0001。

2. 关键代码展示

（1）Q 网络结构：位于 agent_dqn.py 中

```python
class QNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(QNetwork, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.Linear(hidden_size, hidden_size)
        self.head = nn.Linear(hidden_size, output_size)

    def forward(self, inputs):
        hidden1 = torch.relu(self.layer1(inputs))
        hidden2 = torch.relu(self.layer2(hidden1))
        return self.head(hidden2)
```

原理：

- 三层全连接网络：输入层→隐藏层→隐藏层→输出层
- 使用 ReLU 激活函数增加非线性
- 输出层直接输出每个动作的 Q 值

（2）经验回放缓冲区：位于 agent_dqn.py 中

```python
class ReplayBuffer:
    def __init__(self, buffer_size):
        self.capacity = buffer_size
        self.storage = deque(maxlen=buffer_size)

    def __len__(self):
        return len(self.storage)

    def push(self, *transition):
        self.storage.append(transition)

    def sample(self, batch_size):
        return random.sample(self.storage, batch_size)

    def clean(self):
        self.storage.clear()
```

原理：

- 使用双端队列存储经验元组(s, a, r, s', done)
- 随机采样打破时间相关性，提高训练稳定性
- 固定容量，自动淘汰旧经验

（3）动作选择策略：位于 agent_dqn.py 中

```
    def make_action(self, observation, test=True):
        """
        Return predicted action of your agent
        Input:observation
        Return:action
        """
        # 将状态张量转移到正确的设备上
        state_tensor = torch.FloatTensor(observation).unsqueeze(0).to(self.device)

        # 如果是测试模式或符合贪婪策略条件，选择最佳动作
        if test or random.random() > self.epsilon:
            with torch.no_grad():
                action = self.q_net(state_tensor).max(1)[1].item()
        else:
            # 随机探索
            action = self.env.action_space.sample()

        return action
```

原理：

- **ε-贪心策略**: 以概率 ε 随机探索，以概率(1-ε)选择最优动作
- **探索衰减**: ε 值随训练进行逐渐减小
- **测试模式**: 测试时完全贪心选择

（4）网络更新核心算法：位于 agent_dqn.py 中

```
def _update_network(self):
    """更新Q网络"""
    # 从回放缓冲区中采样一批经验
    transitions = self.replay_buffer.sample(self.batch_size)
    batch = list(zip(*transitions))

    # 提取批次数据并确保它们在正确的设备上
    states = torch.FloatTensor(np.array(batch[0])).to(self.device)
    actions = torch.LongTensor(np.array(batch[1])).to(self.device)
    rewards = torch.FloatTensor(np.array(batch[2])).to(self.device)
    next_states = torch.FloatTensor(np.array(batch[3])).to(self.device)
    dones = torch.FloatTensor(np.array(batch[4])).to(self.device)

    # 计算当前Q值和目标Q值
    current_q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)

    with torch.no_grad():
        next_q_values = self.target_q_net(next_states).max(1)[0]
        target_q_values = rewards + (1 - dones) * self.gamma * next_q_values

    # 计算损失并更新网络
    loss = nn.MSELoss()(current_q_values, target_q_values)

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

    self.train_step += 1
```

原理：

1. **Bellman 方程**: Q(s,a) = r + γ × max Q(s',a')
2. **TD 误差**: 使用 MSE 损失函数最小化预测 Q 值与目标 Q 值差异
3. **目标网络**: 使用固定的 target_q_net 计算目标值，避免训练不稳定
4. **批量更新**: 同时处理多个样本，提高效率

（5）训练主循环：位于 agent_dqn.py 中

```python
def run(self):
    """
    Implement the interaction between agent and environment here
    """
    current_state, _ = self.env.reset()
    time_step = 0

    for episode_idx in range(1, self.args.n_frames + 1):
        terminated_flag = False
        episode_reward = 0

        while not terminated_flag:
            time_step += 1

            # 选择动作
            chosen_action = self.make_action(current_state, test=False)

            # 执行动作
            next_state, reward, terminated, truncated, _ = self.env.step(chosen_action)
            terminated_flag = terminated or truncated

            # 存储经验
            self.replay_buffer.push(current_state, chosen_action, reward, next_state, terminated_flag)

            # 如果经验缓冲区足够大，进行训练
            if len(self.replay_buffer) > self.batch_size:
                self._update_network()

                # 根据频率更新目标网络
                if time_step % self.update_target_freq == 0:
                    self.target_q_net.load_state_dict(self.q_net.state_dict())

                # 更新探索率
                self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)

            current_state = next_state
            episode_reward += reward
```

训练流程：

1. **环境交互**: 智能体与环境交互获取经验
2. **经验存储**: 将(s,a,r,s',done)存入回放缓冲区
3. **网络训练**: 从缓冲区采样批次数据更新主网络
4. **目标网络更新**: 定期将主网络参数复制到目标网络
5. **参数调整**: 动态调整探索率等超参数

（6）可视化结果：位于 main.py 中

由于篇幅问题，此处不再展示代码；

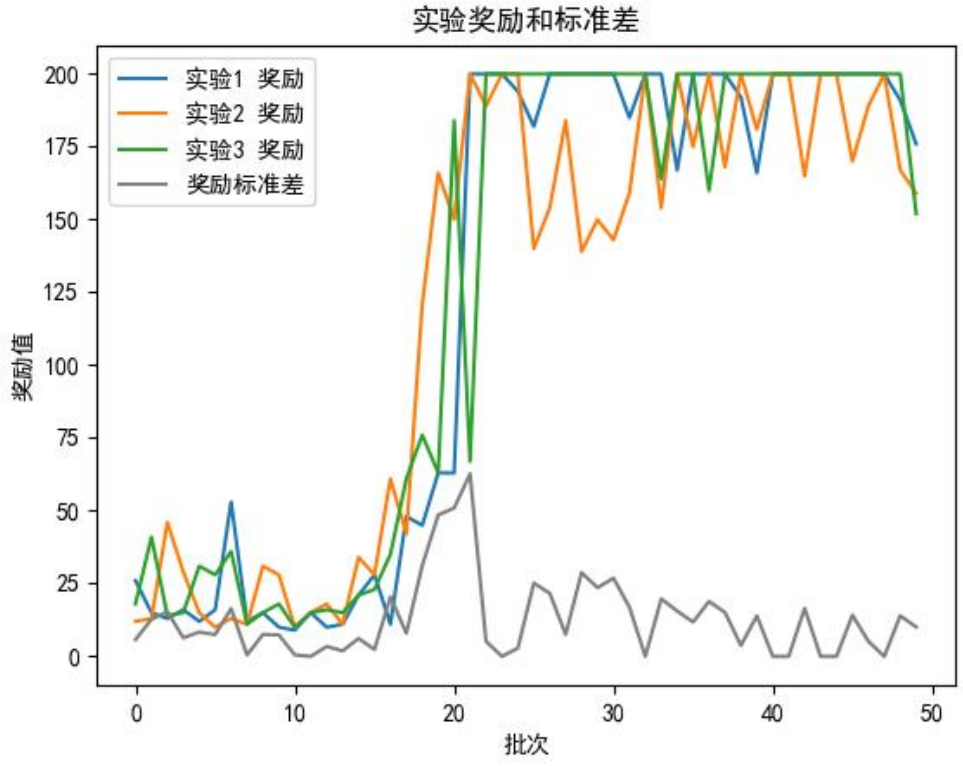
三、 实验结果及分析

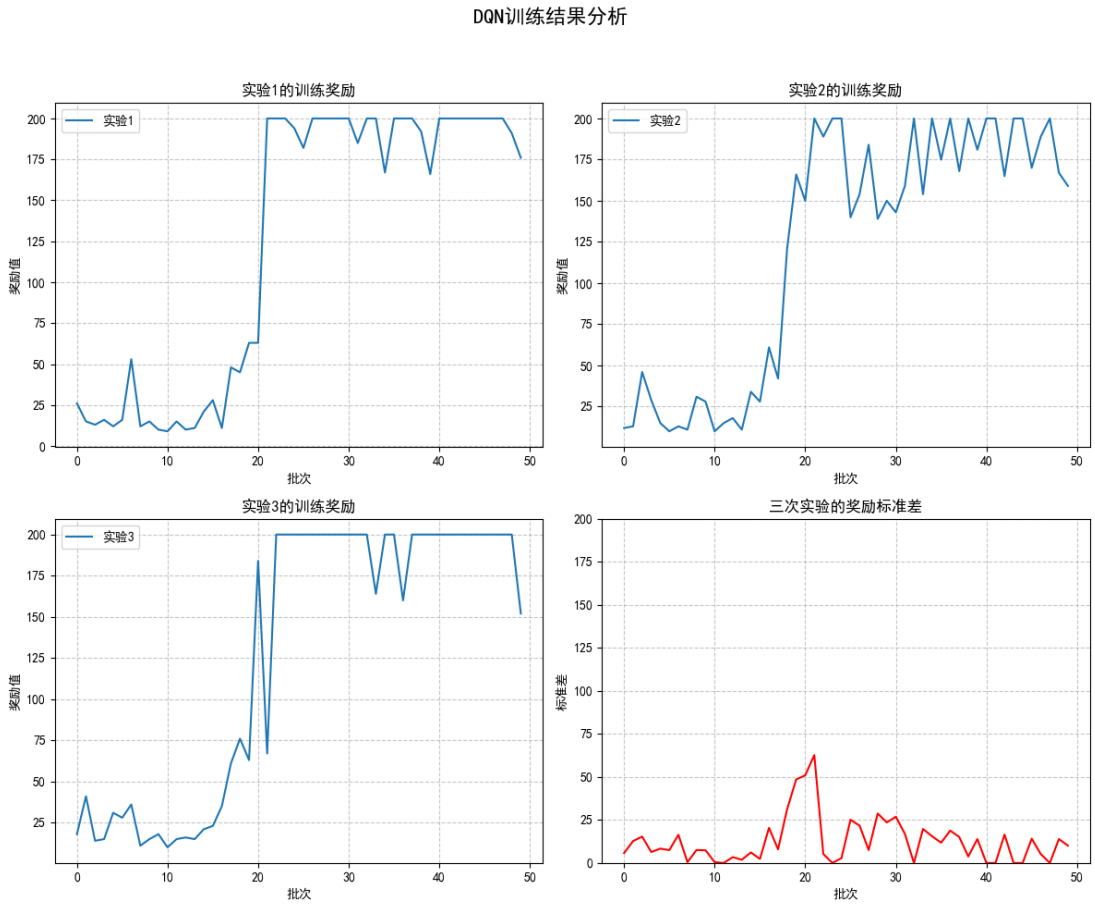
1. 实验结果展示示例

程序运行后生成的图表和数据如下：

```
Rewards: [[ 26.  15.  13.  16.  12.  16.  53.  12.  15.  10.   9.  15.  10.  11.
   21.  28.  11.  48.  45.  63.  63. 200. 200. 200. 194. 182. 200. 200.
  200. 200. 200. 185. 200. 200. 167. 200. 200. 200. 192. 166. 200. 200.
  200. 200. 200. 200. 200. 200. 191. 176.]
 [ 12.  13.  46.  29.  15.  10.  13.  11.  31.  28.  10.  15.  18.  11.
   34.  28.  61.  42. 121. 166. 150. 200. 189. 200. 200. 140. 154. 184.
  139. 150. 143. 159. 200. 154. 200. 175. 200. 168. 200. 181. 200. 200.
  165. 200. 200. 170. 189. 200. 167. 159.]
 [ 18.  41.  14.  15.  31.  28.  36.  11.  15.  18.  10.  15.  16.  15.
   21.  23.  35.  61.  76.  63. 184.  67. 200. 200. 200. 200. 200. 200.
  200. 200. 200. 200. 200. 164. 200. 200. 160. 200. 200. 200. 200. 200.
  200. 200. 200. 200. 200. 200. 200. 152.]]
```

```
奖励标准差: [ 5.73488351 12.75408431 15.32608524  6.37704216  8.33999734  7.48331477
 16.39105447  0.47140452  7.54247233  7.36357401  0.47140452  0.
  3.39934634  1.88561808  6.12825877  2.3570226  20.41785711  7.9302515
 31.2018518  48.55466564 50.95313751 62.69680127  5.18544973  0.
  2.82842712 25.13961018 21.68460796  7.54247233 28.75567577 23.57022604
 26.87005769 16.93779469  0.         19.75404319 15.55634919 11.78511302
 18.85618083 15.08494467  3.77123617 13.9124245   0.          0.
 16.49915823  0.          0.         14.14213562  5.18544973  0.
 13.92838828 10.07747764]
```





由上图我们可以看到实验 1-3 结果较为一致 。

**收敛的 reward 大小：实验的奖励值最终稳定在 200。

**收敛所需的样本数量：实验在大约第 20 个批次后开始显著提高，并在第 30 个批次左右达到稳定状态。

**三次实验的奖励标准差：从标准差图表可以看出，实验 2 在中期有较大的波动，这可能表明算法在该阶段的稳定性稍差。实验 1 和实验 3 的标准差相对较小，表明算法在这些实验中的表现更为稳定。

总体评估

**算法性能：所有实验的算法性能都很好，奖励值都达到了 180 以上。且达到了最大奖励 200。

**收敛速度：所有实验都在大约 30 个批次后收敛，收敛速度较快。

**稳定性：实验 1 和实验 3 的表现更为稳定，实验 2 在中期虽然有较大的波动，但是从三次实验的奖励标准差来看，实验较为稳定。

|----------如有优化，请重复 1，2，分析优化后的算法结果----------|

四、 参考资料