

# 作业4实验报告

231240002余孟凡 231240002@smail.nju.edu.cn

南京大学计算机科学与技术系, 南京 210093

## 摘要

关键词: 强化学习, DQN, DDQN, 贝尔曼方程。

## 1 理解框架代码

### 1.1 强化学习的基本概念

在强化学习中, 系统通常由以下几个核心组件构成:

#### 1. 智能体 (Agent):

智能体是学习和决策的主体, 负责在每个时间步与环境交互并学习。智能体的目标是找到最佳策略 (Policy), 选择能带来最大长期回报的动作。

在框架代码中, Agent主要通过 DQNAgent 和 DDQNAgent 两个类来实现。

#### 2. 环境 (Environment):

环境是智能体交互的外部世界, 智能体通过动作影响环境, 环境向智能体反馈状态和奖励信息。

在框架代码中, Enviroment是CartPole-v1。

#### 3. 状态 (State, S):

状态表示环境在某一时间点的特定信息, 是智能体决策的依据。

在框架代码中的CartPole环境中, 状态包含平衡杆角度、手推车位置等信息。

#### 4. 动作 (Action, A):

智能体在给定状态下可以执行的行为集合。在CartPole环境中, 动作可能是向左或向右施加一个力。

### 5. 奖励 (Reward, R):

环境对智能体选择的动作进行反馈，用数值奖励的形式衡量动作的好坏。

在CartPole中：每次成功保持杆子平衡：奖励+1，杆子倒下或小车超出范围：奖励-1，结束回合。

### 6. 策略 (Policy, $\pi$ ):

策略是智能体选择动作的规则。在深度强化学习中，策略通常由神经网络参数化。

在框架代码中,策略包含两种：选择价值最高的动作:`return action_values.argmax(dim=1).item()`和随机选择动作:`np.random.randint(self.action_dim)`

### 7. 价值函数 (Value Function):

用于评估一个状态或状态-动作对的好坏程度，代表从当前开始能获得的总回报的期望值。有两种：

- 状态价值函数  $V(s)$ ：对于状态  $s$ ，智能体的累计收益期望值。
- 动作价值函数  $Q(s, a)$ ：对于状态  $s$  选择动作  $a$  时，智能体的累计收益期望值。

在框架代码中，使用QNetwork类中的forward方法返回 $Q(s, a)$ 的值。

## 1.2 强化学习的目标

**长期回报最大化：**强化学习的目标是找到一个最优的策略  $\pi^*$ ，使得策略下的每个动作都能够引导最大化累积回报。

累积回报 (Return,  $G_t$ ) 定义为：

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \quad (1)$$

其中：

- $R_{t+1}$ ：时间  $t + 1$  时获得的奖励；
- $\gamma$ ：折扣因子（介于0和1之间），用于衡量越远期的奖励权重逐渐减小。例如，短期收益比长期不确定的收益更有价值。

智能体的学习过程就是优化这个累积回报的过程。

## 1.3 深度Q学习 (DQN) 的框架

代码实现的智能体基于深度Q学习 (Deep Q-Learning, DQN)。我们先了解Q-Learning方法，再扩展到DQN。

### 1.3.1 Q-Learning的核心思想

Q-Learning是一种基于动作价值函数  $Q(s, a)$  的强化学习算法。通过学习  $Q(s, a)$ ，智能体可以选择最优动作。

**Q值更新公式（Bellman方程）：**

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R + \gamma \max_{a'} Q(s', a') - Q(s, a)) \quad (2)$$

其中：

- $s, s'$ ：当前状态和下一状态；
- $a, a'$ ：当前动作和下一动作；
- $R$ ：从环境中得到的即时奖励；
- $\alpha$ ：学习率（Learning Rate）；
- $\gamma$ ：折扣因子。

Q-Learning核心思想是不断逼近动作价值函数  $Q(s, a)$ ，最终收敛到最优Q值。

### 1.3.2 深度Q学习（DQN）

在传统Q-Learning中，通常用一个表格来表示  $Q(s, a)$ 。但当状态和动作空间很大（甚至连续）时，Q表将变得不可行。因此，DQN用神经网络来近似这个动作价值函数。

#### 1.3.3 问题一：如何用神经网络估计Q值？

- 在代码中，`QNetwork` 类定义了一个神经网络，它接收输入状态  $s$ ，并输出对应于所有可选动作的Q值  $Q(s, a; \theta)$ 。
- 通过调整网络参数  $\theta$ ，网络学会最大化回报。

#### 1.3.4 问题二：如何训练这个神经网络？

- 在每一步中存储经验（状态、动作、奖励、下一状态、是否终止）到经验回放缓冲区（`buffer`）。
- 定期从缓冲区随机采样小批量数据，计算目标Q值：

$$Q_{\text{target}} = R + \gamma \max_{a'} Q(s', a'; \theta^-) \quad (3)$$

- 使用均方误差（MSE）作为损失函数：

$$\mathcal{L} = (Q(s, a; \theta) - Q_{\text{target}})^2 \quad (4)$$

- 通过梯度下降算法（Adam）更新网络参数  $\theta$ 。

### 1.3.5 代码中体现的DQN具体方法

- 本地网络 (qnetwork\_local): 用于当前估计  $Q(s, a; \theta)$ 。
- 目标网络 (qnetwork\_target):
  - 专门用于计算目标值  $Q_{\text{target}}$ 。
  - 为了稳定训练, 目标网络参数  $\theta^-$  只在一段时间后才更新, 而非每一时间步都更新。

### 1.3.6 双重DQN (Double DQN)

双重DQN是为了解决DQN中的 **Q值过估计问题** 提出的算法。过估计可能在某些场景中妨碍智能体的学习, 比如某些动作的奖励被系统高估, 导致智能体选择次优策略。

区别:

- DQN直接用目标网络找出最大值:

$$Q_{\text{target}} = R + \gamma \max_{a'} Q(s', a'; \theta^-) \quad (5)$$

- Double DQN则分两步:

1. 先用 **本地网络** 选择动作  $a^* = \arg \max_{a'} Q(s', a'; \theta)$ 。
2. 再用 **目标网络** 估计该动作的Q值:

$$Q_{\text{target}} = R + \gamma Q(s', a^*; \theta^-) \quad (6)$$

这样的分工显著减轻了Q值的过估计, 提高了性能。

## 1.4 强化学习的过程总结 (结合代码)

以下是整个训练流程:

```
if __name__ == '__main__':
    args = get_args()
    # 设置环境
    env = gym.make('CartPole-v1')
    buffer = deque(maxlen=args.buffer_size)

    # 初始化智能体
    input_dim = env.observation_space.shape[0]
    output_dim = env.action_space.n
    if args.agent_name.lower() == "dqn":
        agent = DQNAgent(state_dim=input_dim, action_dim=output_dim, buffer_size=args.buffer_size, seed=1234,
                           lr=args.lr)
    elif args.agent_name.lower() == "ddqn":
        agent = DDQNAgent(state_dim=input_dim, action_dim=output_dim, seed=1234, lr=args.lr, device="cpu")
    else:
        raise ValueError("不支持的智能体类型! 请选择 'dqn' 或 'ddqn'.")

    # 开始训练
    train(args, agent, buffer, env)
```

图 1: 初始化代码

### 1.4.1 环境与智能体初始化

- 创建强化学习环境（CartPole-v1）。
- 初始化智能体，包括DQN或DDQN模型、经验缓冲区。

```
def train(args, agent, buffer, env):  # 训练函数
    for episode in range(args.num_episodes):
        state = env.reset()
        epsilon = max(args.epsilon_end, args.epsilon_start * (args.epsilon_decay_rate ** episode))

        losses = []
        total_return = 0
        for step in range(args.max_steps_per_episode):
            # 选择并执行动作
            action = agent.act(state, epsilon)
            next_state, reward, done, _ = env.step(action)

            # 存储数据到缓冲区
            buffer.append((state, action, reward, next_state, done))

            # 如果缓冲区中经验足够，则进行学习
            if len(buffer) >= args.batch_size:
                batch = random.sample(buffer, args.batch_size)
                loss = agent.learn(batch, args.gamma)
                losses.append(loss.item())

            total_return += reward
            state = next_state

        if done:
            break

        # 计算平均损失
        average_loss = np.mean(losses) if losses else 0.0
        # 评估策略的回报
        eval_return = eval_policy(agent, env)

    print(
        f"训练 {args.num_episodes} 回合 (step + 1)，训练损失 {average_loss:.4f}，评估回报 {eval_return:.2f}"
    )
```

图 2: 训练过程

### 1.4.2 训练主循环

对于每一回合（Episode）：

1. 从环境中重置新的初始状态 `state`。
2. 初始化当前探索率（`epsilon`），根据定义的探索-利用平衡策略进行动作选择。
3. 智能体与环境交互：
  - 选择动作 `action`；
  - 执行动作，得到下一个状态和奖励；
  - 将 `state`, `action`, `reward`, `next_state`, `done` 加入缓冲区。
4. 如果缓冲区已有足够经验，开始采样，并调用智能体的 `learn` 方法更新网络。

```
def eval_policy(agent, env): 1 个用法 新 *
    """
    评估智能体的策略
    """
    state = env.reset()
    done = False
    total_return = 0
    while not done:
        action = agent.act_no_explore(state)
        next_state, reward, done, _ = env.step(action)
        state = next_state
        total_return += reward
    return total_return
```

图 3: 评估方法

### 1.4.3 评估

- 在固定时间间隔评估智能体策略，记录性能数据。

### 1.4.4 结果输出

- 每回合打印训练损失、评估回报；如果达到目标，可以提前终止。

## 1.5 DQN代码解读

```
# 定义DQN智能体
class DQNAgent: 2 用法 新 *
    def __init__(self, state_dim, action_dim, buffer_size, seed, lr, device="cpu"):...
    def act(self, state, eps=0.):...
    def act_no_explore(self, state):...
    def learn(self, experiences, gamma):...
    def soft_update(self, local_model, target_model, tau):...
```

图 4: DQN代码简要

### 1. 类的初始化

state\_dim: 状态空间的维度，在CartPole环境中，状态包括小车的位置、速度、杆的角度和角速度，总共四个维度。

action\_dim: 动作空间的维度。对CartPole环境，表示env.action\_space.n。

buffer\_size: 经验回放缓冲区的大小，用于存储智能体与环境交互的经验。

lr: 学习率，用于优化器更新网络参数的步长。

本地网络 (qnetwork\_local): 用于估计当前策略的Q值。

目标网络 (qnetwork.target): 用于稳定训练, 提供目标Q值。目标网络的参数定期从本地网络复制过来。

## 2. 动作选择 (act 方法)

$\epsilon$ -贪心策略: 平衡探索 (选择随机动作) 与利用 (选择当前最优动作)。

## 3. 无探索动作选择 (act\_no\_explore 方法)

总是选择当前估计的最佳动作, 适用于策略评估或部署阶段, 不适合训练过程中使用。

## 4. 学习更新 (learn 方法)

计算目标Q值:

下一状态Q值 (Q\_targets\_next): 通过目标网络预测下一个状态的Q值, 选择最大值作为未来奖励的估计。

总体目标Q值 (Q\_targets): 根据贝尔曼方程计算, 总奖励等于当前奖励加上折扣后的未来奖励, 若为终止状态则未来奖励为0。

计算当前Q值 (Q\_expected): 通过本地网络预测当前状态下选择动作的Q值。

损失计算与反向传播:

损失函数: 均方误差损失, 用于衡量预测Q值与目标Q值之间的差距。

优化步骤: 清零梯度, 反向传播损失, 更新网络参数。

软更新目标网络。

## 5. 软更新目标网络 (soft\_update 方法)

对于目标网络和本地网络的每一对对应参数, 执行以下更新:

$$\theta_{\text{target}} = \tau \cdot \theta_{\text{local}} + (1 - \tau) \cdot \theta_{\text{target}}$$

默认的 $\tau$ 是1e-3, 更新比较缓慢。

# 1.6 DDQN

由于DDQN继承自DQN, 二者的唯一区别在于learn中计算最大Q值的方法:

```
def learn(self, experiences, gamma): 2个用法(2个动态) 新*
    """
    从经验中学习, 更新网络参数
    """
    device = self.device
    states, actions, rewards, next_states, dones = zip(*experiences)
    states = torch.from_numpy(np.vstack(states)).float().to(device)
    actions = torch.from_numpy(np.vstack(actions)).long().to(device)
    rewards = torch.from_numpy(np.vstack(rewards)).float().to(device)
    next_states = torch.from_numpy(np.vstack(next_states)).float().to(device)
    dones = torch.from_numpy(np.vstack(dones).astype(np.uint8)).float().to(device)

    # 计算下一个状态的最大Q值
    Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)
    Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

    # 计算当前状态的Q值
    Q_expected = self.qnetwork_local(states).gather(1, actions)

    # 计算损失并反向传播
    loss = F.mse_loss(Q_expected, Q_targets)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    # 软更新目标网络参数
    self.soft_update(self.qnetwork_local, self.qnetwork_target, tau=1e-3)
    return loss
```

图 5: DQNlearn

```
def learn(self, experiences, gamma): 2个用法(2个动态) 新*
    """
    双DQN的学习方法, 减少过拟合现象
    """
    device = self.device
    states, actions, rewards, next_states, dones = zip(*experiences)
    states = torch.from_numpy(np.vstack(states)).float().to(device)
    actions = torch.from_numpy(np.vstack(actions)).long().to(device)
    rewards = torch.from_numpy(np.vstack(rewards)).float().to(device)
    next_states = torch.from_numpy(np.vstack(next_states)).float().to(device)
    dones = torch.from_numpy(np.vstack(dones).astype(np.uint8)).float().to(device)

    # 使用本地网络选择动作
    action_max = torch.argmax(self.qnetwork_local(next_states), dim=1).unsqueeze(1)
    Q_targets = rewards + (
        gamma * torch.gather(self.qnetwork_target(next_states).detach(), dim=1, action_max) * (1 - dones))

    # 计算当前状态的Q值
    Q_expected = self.qnetwork_local(states).gather(1, actions)

    # 计算损失并反向传播
    loss = F.mse_loss(Q_expected, Q_targets)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    # 软更新目标网络参数
    self.soft_update(self.qnetwork_local, self.qnetwork_target, tau=1e-3)
    return loss
```

图 6: DDQNleran

### 1.6.1 DQN的计算方式

# 计算下一个状态的最大Q值

```
Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)
Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
```

1. 从目标网络 (qnetwork\_target) 中直接计算最大Q值 (max Q):

- `self.qnetwork_target(next_states).detach()`: 用目标网络预测下一状态的所有动作的Q值, 并断开梯度 (`detach()`) 以避免更新目标网络的参数。
- `.max(1)[0]`: 对每个下一状态, 选择所有动作中最大的Q值, 它作为下一状态的“最大回报”。



## 2. 目标Q值计算（Bellman更新）:

$$Q_{\text{target}} = r + \gamma \cdot \max_{a'} Q_{\text{target}}(s', a') \cdot (1 - \text{done}) \quad (7)$$

其中:

- **rewards**: 当前状态的即时奖励  $r$ 。
- **gamma**: 折扣因子, 用于平衡当前奖励和未来奖励的权重。
- $\max_{a'} Q_{\text{target}}(s', a')$ : 从目标网络计算出的未来最大Q值。
- $(1 - \text{done})$ : 确保如果是终止状态 ( $\text{done}=\text{True}$ ), 未来奖励为0。
- **目标网络负责预测下一状态的最大Q值**: 直接使用同一个目标网络既决定动作 (选择Q值最大的位置) 又评估该动作的值。
- **容易出现Q值过估计问题**:
  - 因为在DQN中, 同一个目标网络既负责预测动作, 也负责评估该动作的Q值。当网络预测有偏差时, 这种方式可能会错误地高估某些动作的Q值。

### 1.6.2 DDQN的计算方式

# 使用本地网络选择动作

```
action_max = torch.argmax(self.qnetwork_local(next_states), dim=1).unsqueeze(1)
Q_targets = rewards + (
gamma * torch.gather(self.qnetwork_target(next_states).detach(), 1, action_max) * (1 - done))
```

#### 1. 动作选择与动作评估分离（Double DQN）:

- `torch.argmax(self.qnetwork_local(next_states), dim=1).unsqueeze(1)`:
  - 使用**本地网络** (`qnetwork_local`) 对下一状态的所有动作Q值进行评估, 选择Q值最大的动作  $a^* = \arg \max Q_{\text{local}}(s', a)$ 。
  - 这里的本地网络用于确定动作索引, 而不直接参与Q值的计算。
  - 本地网络通常训练中能得到快速更新, 能更准确地反映动作选择的策略。
- `self.qnetwork_target(next_states).detach()`:
  - 使用目标网络对下一状态真正进行评估, 获取上一步由本地网络选择的动作对应的Q值:

$$Q_{\text{target}}(s', a^*) \quad (8)$$

- 通过 `torch.gather()` 使用前一步选出的动作索引提取Q值。

## 2. 目标Q值计算（Bellman更新）：

$$Q_{\text{target}} = r + \gamma \cdot Q_{\text{target}}(s', a^*) \cdot (1 - \text{done}) \quad (9)$$

这种分离使得目标网络不会同时负责选择动作和评估该动作的值，从而减少过高估值的可能性。

### 3. 特点：

- **动作选择由本地网络负责：**本地网络更新更频繁，能动态反映当前策略情况。
- **动作评估由目标网络负责：**目标网络的更新较慢，提供更加稳定的评估。
- **有效缓解了Q值过估计问题：**
  - 通过这种分离选择和评估的方式，减少了因为网络预测误差而导致的高估行为。
- 比DQN稍微复杂，但训练效果通常更好，稳定性更高。

## 2 三个变量分别有何作用？

### 2.1 学习率 ( $lr$ )

学习率（Learning Rate）是一个用于控制模型在优化过程中参数更新步长的超参数。在代码中， $lr$  被传递给优化器 `optim.Adam`，用于更新神经网络的权重参数。

作用：

- **控制更新步长：**学习率决定了每次参数更新时迈出的步伐大小。较高的学习率意味着每次更新参数时变化较大，可能加速收敛，但也可能导致震荡或发散。较低的学习率则使得参数更新更为细致，但可能导致收敛速度过慢。
- **优化过程的稳定性：**合适的学习率能够在保持优化过程稳定的同时，加快收敛速度。过高或过低的学习率都可能影响优化效果，导致训练失败或效率低下。

在代码中的实现：

```
self.optimizer = optim.Adam(self.qnetwork  
                              _local.parameters(), lr)
```

这里使用了 Adam 优化器，并将学习率  $lr$  作为其参数传入。Adam 优化器结合了动量（Momentum）和自适应学习率调整的优点，通常能比简单的 SGD 优化器更快收敛。

## 2.2 折扣因子 ( $\gamma$ )

**定义：**折扣因子（Discount Factor）是一个介于0和1之间的超参数，用于衡量未来奖励的现值。在强化学习中， $\gamma$  决定了智能体在决策时对未来奖励的重视程度。

**作用：**

- **权衡即时奖励与未来奖励：** $\gamma$  越接近1，智能体越重视未来的奖励，愿意为了长期回报而做出牺牲当前的奖励。相反， $\gamma$  越接近0，智能体更倾向于获取当前的奖励，较少考虑未来的回报。
- **控制收益的累计和稳定性：**适当的折扣因子可以防止未来奖励的累加导致收益爆炸，同时确保智能体关注于达到长期目标而不是短期的局部最优。

具体理论在上文贝尔曼公式中提到了。

**在代码中的实现：**

$$Q\_targets = rewards + (\gamma * Q\_targets\_next * (1 - dones))$$

这里， $\gamma$  用于将未来的 Q 值折现后加到当前的奖励上，从而计算目标 Q 值。

## 2.3 经验回放缓冲区大小 (*buffer\_size*)

**定义：**经验回放缓冲区（Experience Replay Buffer）用于存储智能体在与环境交互过程中积累的经验（状态、动作、奖励、下一个状态、是否终止）。*buffer\_size* 指的是缓冲区能够存储的最大经验数量。

**作用：**

- **打破数据相关性：**通过随机采样存储在缓冲区中的经验，经验回放有效地打破了时间序列中的数据相关性，使得训练过程更加稳定和高效。
- **提高数据利用率：**将经验存储在缓冲区中，允许智能体多次使用相同的经验进行学习，从而提高样本的利用率，减少样本浪费。
- **平衡新旧经验：**缓冲区的大小决定了智能体能记住多少过去的经验，较大的缓冲区能够保存更多的新经验，同时仍保留一部分旧经验，帮助智能体在不同的状态下进行更全面的学习。

**在代码中的实现：**

$$buffer = deque(maxlen=args.buffer\_size)$$

这里使用了 *deque*（双端队列）来实现固定大小的缓冲区，当超出 *buffer\_size* 时，最早的经验将被自动移除以腾出空间。

## 3 act 和 act\_no\_explore

### 3.1 act 函数

**定义：** act 函数根据当前状态选择动作，采用的是  $\epsilon$ -贪心策略 ( $\epsilon$ -Greedy Policy)。这种策略在选择动作时，以  $\epsilon$  的概率进行随机探索 (Exploration)，以  $1 - \epsilon$  的概率选择当前估计最优的动作 (Exploitation)。

**代码实现：**

```
def act(self, state, eps=0.):

    state_tensor = torch.from_numpy(state).float().unsqueeze(0).to(self.device)

    self.qnetwork_local.eval()
    with torch.no_grad():
        action_values = self.qnetwork_local(state_tensor)
    self.qnetwork_local.train()

    if np.random.random() > eps:
        return action_values.argmax(dim=1).item()
    else:
        return np.random.randint(self.action_dim)
```

**作用：**

- **平衡探索与利用：** 通过引入  $\epsilon$  参数，智能体在训练过程中既能探索新的动作以发现潜在的高回报动作，也能利用已知的最优动作以最大化即时奖励。
- **促进策略优化：**  $\epsilon$ -贪心策略有助于避免智能体陷入局部最优，通过随机探索增加了策略优化的多样性和广度。

### 3.2 act\_no\_explore 函数

**定义：** act\_no\_explore 函数在选择动作时不进行任何探索，始终选择当前估计的最优动作。这意味着智能体完全依赖于已学到的策略，不进行随机动作选择。

**代码实现：**

```
def act_no_explore(self, state):

    state_tensor = torch.from_numpy(state).float().unsqueeze(0).to(self.device)
```

```

self.qnetwork_local.eval()
with torch.no_grad():
    action_values = self.qnetwork_local(state_tensor)
self.qnetwork_local.train()

return action_values.argmax(dim=1).item()

```

作用：

- **确定性动作选择：**智能体始终选择当前估计的最佳动作，适用于评估或部署阶段，以确保行为的稳定性和可预测性。
- **策略评估：**通过消除随机性，`act_no_explore` 函数有助于准确评估智能体当前策略的性能。

### 3.3 `act` 与 `act_no_explore` 的不同

因素	<code>act</code> 函数	<code>act_no_explore</code> 函数
策略类型	$\epsilon$ -贪心策略	确定性贪心策略
探索概率	$\epsilon$	0
动作选择	随机探索或选择最佳动作	始终选择最佳动作

### 3.4 在训练过程中采用 `act_no_explore` 函数可能带来的问题

在训练过程中使用 `act_no_explore` 函数意味着智能体完全依赖于当前的估计策略进行动作选择，而不进行任何形式的探索。这种做法可能导致以下问题：

#### 3.4.1 陷入局部最优

- **缺乏探索能力：**智能体无法主动探索未尝试过的动作，从而可能错过发现更高回报的策略。
- **早期收敛于次优策略：**由于缺乏随机探索，智能体可能过早地收敛于一个局部最优策略，无法进一步优化。

#### 3.4.2 数据多样性不足

- **经验回放数据单一：**使用确定性策略会导致经验回放缓冲区中的数据缺乏多样性，降低训练的有效性和泛化能力。
- **梯度更新方向受限：**训练过程中，梯度更新主要基于有限的状态-动作对，可能导致网络参数更新受限，影响模型的学习效果。

### 3.4.3 收敛速度和稳定性问题

- **震荡或发散风险增加：**缺乏探索可能导致智能体在某些状态下反复选择同一动作，梯度更新方向不明确，增加收敛过程中的不稳定性。
- **学习过程不充分：**智能体未能充分体验环境中不同的状态转移，导致对环境动力学的理解不全面，影响最终策略的质量。

### 3.4.4 探索-利用平衡破坏

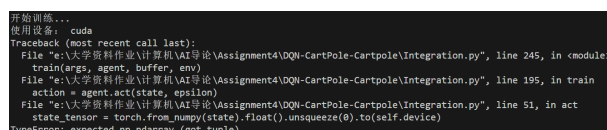
- **探索不足：**训练过程中无法充分探索状态空间，智能体难以发现全局最优策略。
- **策略改进受限：**智能体无法通过探索获取新的高回报动作，限制了策略的持续改进能力。

## 4 更新框架代码

### 4.1 CUDA加速

由于我在上一作业中安装的CUDA为12.1版本pytorch为2.3.0版本，并且CUDA没有向后兼容性，于是我决定提升框架代码的pytorch版本。

在pytorch2.3.0版本中直接运行发生如下报错：



```
开始训练...
使用设备: cuda
Traceback (most recent call last):
  File "e:\大学资料作业\计算机\AI导论\Assignment4\DQN-CartPole-Cartpole\Integration.py", line 245, in <module>
    train(args, agent, buffer, env)
  File "e:\大学资料作业\计算机\AI导论\Assignment4\DQN-CartPole-Cartpole\Integration.py", line 195, in train
    action = agent.act(state, epsilon)
  File "e:\大学资料作业\计算机\AI导论\Assignment4\DQN-CartPole-Cartpole\Integration.py", line 51, in act
    state_tensor = torch.from_numpy(state).float().unsqueeze(0).to(self.device)
TypeError: expected np.ndarray (got tuple)
```

图 7: 报错

这表明在调用 `torch.from_numpy(state)` 时，`state` 的类型是 `tuple`，而不是预期的 `np.ndarray` 类型。

经过STFW，这是因为这里的 `state` 来源于 `env.reset()`。

我在这个环境下安装的Gym是0.26.1的，而老环境是0.16.0。

在较新的 Gym 版本中，`env.reset()` 的返回值类型发生了变化。

所以我们应该修改关于这个API的调用：

#### 1. 修改 `env.reset()` 的调用：

将原本的：

```
state = env.reset()
```

修改为：

```

state = env.reset()
if isinstance(state, tuple):
    state = state[0]

```

这样可以确保无论 Gym 版本如何，都能正确获取到 `state`。

## 2. 修改 `env.step(action)` 的调用：

将原本的：

```
next_state, reward, done, _ = env.step(action)
```

修改为：

```

next_state, reward, terminated, truncated, _ = env.step(action)
done = terminated or truncated

```

同时，确保 `next_state` 是 `np.ndarray` 类型：

```

if isinstance(next_state, tuple):
    next_state = next_state[0]

```

改进之后，相比之前半小时才能训练完成的代码，7-8分钟就能够训练完成，效率提升4倍。

## 4.2 数据可视化

目前修改后的代码能跑通，不过数据使用控制台输出不够直观，我们尝试对数据进行可视化等处理：

首先使用scv格式记录数据：

```

import csv
from datetime import datetime

# 在train函数前添加
Codeium: Refactor | Explain | Generate Docstring | X
def create_csv_file(filename):
    with open(filename, 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['Episode', 'Steps', 'Training_Loss', 'Evaluation_Return'])

Codeium: Refactor | Explain | Generate Docstring | X
def append_to_csv(filename, episode, steps, loss, eval_return):
    with open(filename, 'a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow([episode, steps, loss, eval_return])

Codeium: Refactor | Explain | X
def generate_filename(args):
    """
    根据超参数生成唯一的文件名
    """
    filename = f"{args.agent_name}_ " \
               f"eps{args.epsilon_start:.2f}_ " \
               f"decay{args.epsilon_decay_rate:.2f}_ " \
               f"gamma{args.gamma:.2f}_ " \
               f"lr{args.lr:.0e}_ " \
               f"{datetime.now().strftime('%Y%m%d_%H%M%S')}.csv"
    return filename

```

图 8: 数据记录为CSV

对于每个文件的题目设置超参数，便于调参时对比。

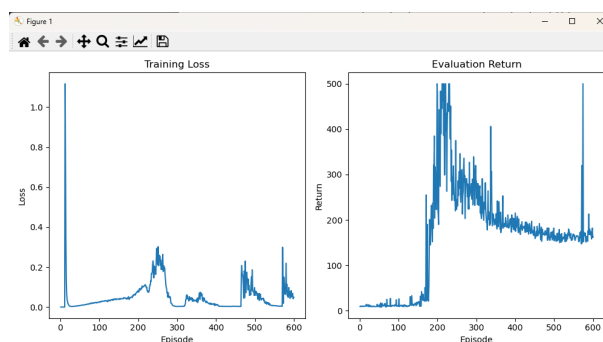


图 9: 示例

### 4.3 模型评估

我们将模型保存下来，然后编写一个测试函数，每次只调用`act_no_explore`，玩100次计算回报的平均值和标准差，就有了一个指标可以直观衡量模型的能力了。

```
def evaluate_model(model_path, env_name='CartPole-v1', num_episodes=100):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f"使用设备: {device}")
    # 创建环境
    env = gym.make(env_name)
    # 初始化智能体
    input_dim = env.observation_space.shape[0]
    output_dim = env.action_space.n
    agent = DQNAgent(...)
    # 加载模型
    agent.load_model(model_path)

    # 评估指标
    total_rewards = []

    # 进行多轮评估
    for episode in range(num_episodes):
        reset_result = env.reset()
        if isinstance(reset_result, tuple):
            state = reset_result[0]
        else:
            state = reset_result
        done = False
        episode_reward = 0
        while not done:
            # 选择动作时不探索
            action = agent.act_no_explore(state)
            step_result = env.step(action)

            if len(step_result) == 5: ...
            else: ...

            episode_reward += reward

        total_rewards.append(episode_reward)

    # 计算评估结果
    avg_reward = np.mean(total_rewards)
    std_reward = np.std(total_rewards)
    print(f"平均奖励: {avg_reward:.2f} ± {std_reward:.2f}")
    return avg_reward, std_reward
```

图 10: 模型评估器

我们也可以渲染一下环境，让他直观展示一下运动过程，方便我们对模型之间的差别有一个直观的印象。



```
def visualize_model_performance(model_path, env_name='CartPole-v1'):
    # 设备和环境初始化
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    env = gym.make(env_name, render_mode='human')
    # 初始化智能体
    input_dim = env.observation_space.shape[0]
    output_dim = env.action_space.n
    agent = DDQNAgent(
        state_dim=input_dim,
        action_dim=output_dim,
        seed=1234,
        lr=1e-3,
        device=device
    )
    # 加载模型
    agent.load_model(model_path)

    # 可视化一个回合
    reset_result = env.reset()
    if isinstance(reset_result, tuple):
        state = reset_result[0]
    else:
        state = reset_result

    done = False

    while not done: ...

    env.close()
```

图 11: 可视化

最后写一个批量导入的脚本方便对比观察：

```
if __name__ == "__main__":
    # 评估所有模型
    model_paths = find_model_paths("models/dqn_datalog")

    dataset = []

    for model_path in model_paths:
        warnings.filterwarnings("ignore", category=DeprecationWarning)
        print(f"评估模型: {model_path}")
        avg_reward, std_reward = evaluate_model(model_path)
        dataset.append((model_path, avg_reward, std_reward))
        # visualize_model_performance(model_path)

    # 评估结果排序
    dataset.sort(key=lambda x: x[1], reverse=True)

    # 评估结果可视化(柱状图)
    model_names = [os.path.basename(model_path) for model_path, _, _ in dataset]
    avg_rewards = [avg_reward for _, avg_reward, _ in dataset]
    std_rewards = [std_reward for _, _, std_reward in dataset]
    x = np.arange(len(model_names))
    fig, ax = plt.subplots()
    ax.bar(x, avg_rewards, yerr=std_rewards, align='center', alpha=0.5, ecolor='black', capsize=10)
    ax.set_ylabel('Average Reward')
    ax.set_xticks(x)
    ax.set_xticklabels(model_names, rotation=45)
    ax.set_title('Model Performance Evaluation')
    plt.tight_layout()
    plt.show()
    print("评估完成")

    # 评估结果可视化(散点图)
    avg_rewards = [avg_reward for _, avg_reward, _ in dataset]
    std_rewards = [std_reward for _, _, std_reward in dataset]
    plt.scatter(avg_rewards, std_rewards)
    plt.xlabel('Average Reward')
    plt.ylabel('Standard Deviation')
    plt.title('Model Performance Evaluation')
    plt.tight_layout()
    plt.show()
```

图 12: 可视化脚本

## 5 调参

使用默认参数，我们得到了如下结果：

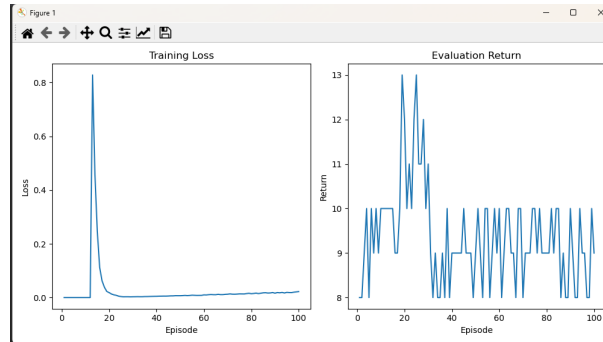


图 13: 默认结果

**训练损失：**训练损失在前几个回合中迅速下降，说明模型在学习过程中逐渐掌握了环境规则。不过损失在达到零附近后趋于平稳，表明模型在此阶段未再显著改进。

**评估回报：**评估回报波动较大，没有显著趋势向上。这可能是由于模型过度拟合训练数据，或探索策略设置不够合理。

## 5.1 调整学习率

修改学习率至 $1e-5$ ，这么做是为了拉长损失降低的过程，防止出现最后损失没有收敛的情况。

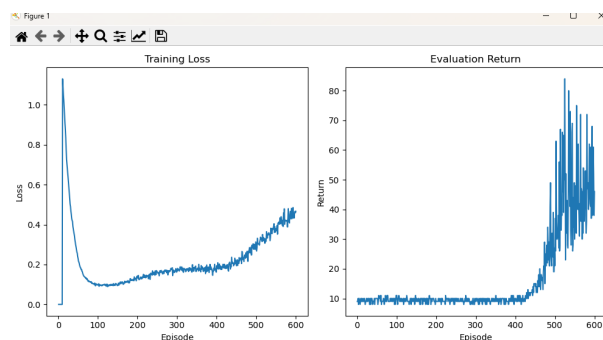


图 14: 修改学习率至 $1e-5$

可以看到损失更大且仍然没有收敛迹象，回报仍然不稳定。

修改学习率至 $1e-3$ ，这么做是试图使其快速收敛，虽然默认模型已经足够快速，这一工作基本可以确定无望。

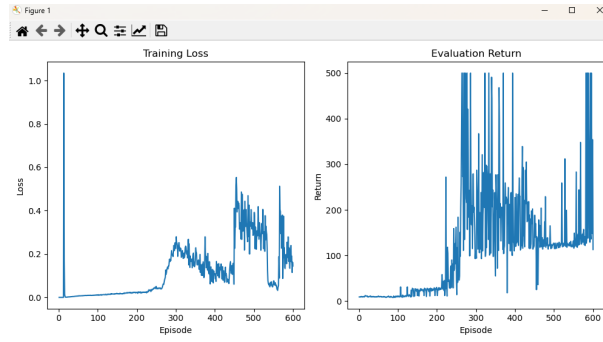


图 15: 修改学习率至 $1e-3$

进行性能对比:

```
评估模型: models/dqn_data/dqn_eps0.99_decay0.99_gamma0.99_lr1e-03_20241125_142503.csv/20241125_143157.pth
使用设备: cuda
平均奖励: 111.23 ± 6.54
评估模型: models/dqn_data/dqn_eps0.99_decay0.99_gamma0.99_lr2e-04_20241125_145050.csv/20241125_145931.pth
使用设备: cuda
平均奖励: 145.83 ± 53.37
评估模型: models/dqn_data/dqn_eps0.99_decay0.99_gamma0.99_lr2e-05_bs256_wf10_buffer10000_max500_20241125_150756.csv/20241125_150841.pth
使用设备: cuda
平均奖励: 46.03 ± 14.46
```

图 16: 性能对比

## 5.2 自动化调参

上述对学习率的调整过程挺低效的, 花了10分钟才调了三个参数, 于是我决定使用网格调参法, 我接下来简要介绍参数设计思路, 然后开始坐等运行结果。

### 5.2.1 调整gamma

对于CartPole这种短期任务, 过高的gamma可能导致训练困难。可以尝试将gamma从0.99调整到0.95或0.9。

### 5.2.2 batch\_size

增大批大小可以更稳定地估计梯度. 可以将batch\_size从256增加到512或更高。

### 5.2.3 buffer\_size

更大的缓冲区可以存储更多的经验, 增加数据多样性, 防止过拟合。

## 5.3 开始训练

```
# 定义超参数搜索空间
hyperparameter_space = {
    "num_episodes": [500, 1000, 1500],
    "lr": [1e-3, 1e-4, 1e-5],
    "gamma": [0.95, 0.99, 0.9],
    "update_frequency": [4, 8, 16],
    "buffer_size": [5000, 10000, 20000],
    "max_steps_per_episode": [500, 1000, 2000],
    "batch_size": [64, 128, 256],
    "epsilon_decay_rate": [0.995, 0.99, 0.98]
}
```

图 17: 参数网格

这是我第一版设计的参数网格，我很快意识到了问题：这样要训练6561组！就算一组参数3分钟，这得训练到猴年马月去。

于是我决定分析一下参数之间的相关程度，把那些相互影响比较小的参数分别训练，影响比较大的合并训练，这样不仅能节省时间，还能对独立参数更详细的进行微调。

### 5.3.1 调整学习率

```
# 学习率超参数搜索空间
hyperparameter_space = {
    "lr": [3e-3, 9e-4, 8e-4, 7e-4, 6e-4, 5e-4, 4e-4, 3e-4, 2e-4, 1e-4, 9e-5, 8e-5, 7e-5, 6e-5, 5e-5, 4e-5, 3e-5, 2e-5, 1e-5, 1e-6],
    "num_episodes": [500, 1000, 1500],
    "gamma": [0.95, 0.99, 0.9],
    "update_frequency": [4, 8, 16],
    "buffer_size": [5000, 10000, 20000],
    "max_steps_per_episode": [500, 1000, 2000],
    "batch_size": [64, 128, 256],
    "epsilon_decay_rate": [0.995, 0.99, 0.98]
}
```

图 18: 学习率

```

超参数组合: {'lr': 1e-06}, 评估回报: 9.0
最佳超参数组合: {'lr': 6e-05}, 最佳评估回报: 428.0
超参数: {'lr': 0.001}, 评估回报: 94.0
超参数: {'lr': 0.0009}, 评估回报: 18.0
超参数: {'lr': 0.0008}, 评估回报: 406.0
超参数: {'lr': 0.0007}, 评估回报: 98.0
超参数: {'lr': 0.0006}, 评估回报: 118.0
超参数: {'lr': 0.0005}, 评估回报: 169.0
超参数: {'lr': 0.0004}, 评估回报: 250.0
超参数: {'lr': 0.0003}, 评估回报: 259.0
超参数: {'lr': 0.0002}, 评估回报: 167.0
超参数: {'lr': 0.0001}, 评估回报: 274.0
超参数: {'lr': 9e-05}, 评估回报: 338.0
超参数: {'lr': 8e-05}, 评估回报: 330.0
超参数: {'lr': 7e-05}, 评估回报: 193.0
超参数: {'lr': 6e-05}, 评估回报: 428.0
超参数: {'lr': 5e-05}, 评估回报: 286.0
超参数: {'lr': 4e-05}, 评估回报: 339.0
超参数: {'lr': 3e-05}, 评估回报: 176.0
超参数: {'lr': 2e-05}, 评估回报: 34.0
超参数: {'lr': 1e-05}, 评估回报: 9.0
超参数: {'lr': 1e-06}, 评估回报: 9.0
最佳超参数: {'lr': 6e-05}

```

图 19: 网格输出

可以看到，网格给出的最优学习率是6e-5，我们可以使用自己编写的测试来验证一下：

```

评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr1e-06
41125_182853.csv20241125_182928.pth
平均奖励: 9.29 ± 0.75
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr2e-04
41125_170527.csv20241125_171552.pth
平均奖励: 176.45 ± 14.41
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr2e-05
41125_182730.csv20241125_182819.pth
平均奖励: 31.30 ± 4.39
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr3e-04
41125_165441.csv20241125_170527.pth
平均奖励: 286.76 ± 60.44
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr3e-05
41125_182416.csv20241125_182730.pth
平均奖励: 146.14 ± 17.96
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr4e-04
41125_164716.csv20241125_165440.pth
平均奖励: 184.81 ± 27.76
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr4e-05
41125_181635.csv20241125_182415.pth
平均奖励: 337.87 ± 72.42
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr5e-04
41125_163742.csv20241125_164715.pth
平均奖励: 171.26 ± 29.39
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr5e-05
41125_180748.csv20241125_181635.pth
平均奖励: 269.13 ± 13.98
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr6e-04
41125_163109.csv20241125_163742.pth
平均奖励: 115.46 ± 2.79
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr6e-05
41125_175631.csv20241125_180748.pth
平均奖励: 408.96 ± 20.38
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr7e-04
41125_162204.csv20241125_163109.pth
平均奖励: 104.28 ± 19.22
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr7e-05
41125_174557.csv20241125_175631.pth
平均奖励: 214.60 ± 23.34
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr8e-04
41125_161510.csv20241125_162204.pth
平均奖励: 255.17 ± 30.83
评估模型: models/dqn_datalog\dqn_eps0.90_decay0.99_gamma0.99_lr8e-05
41125_173855.csv20241125_174556.pth
平均奖励: 213.09 ± 28.61

```

图 20: 验证结果

可以发现确实是 $6e-5$ 表现最好，平均奖励为： $408.96 \pm 20.38$ 。修改学习率后，我们尝试修改num\_episodes。

### 5.3.2 调整num\_episodes

理论上讲，越大的num\_episodes肯定效果越好，因为在训练的早期阶段，智能体可能会更多地进行探索，而随着训练的进行，它会逐渐利用已学到的知识。增加num\_episodes 可以让这种探索与利用的平衡更加明显。在一些情况下，过少的训练回合可能导致智能体在特定的状态下过拟合，而增加训练回合可以帮助智能体更好地泛化到未见过的状态。

但是，我的NVIDIA GeForce RTX 3090 Laptop GPU毕竟还是算力有限，我需要找到一个能够满意完成任务的最少num\_episodes。

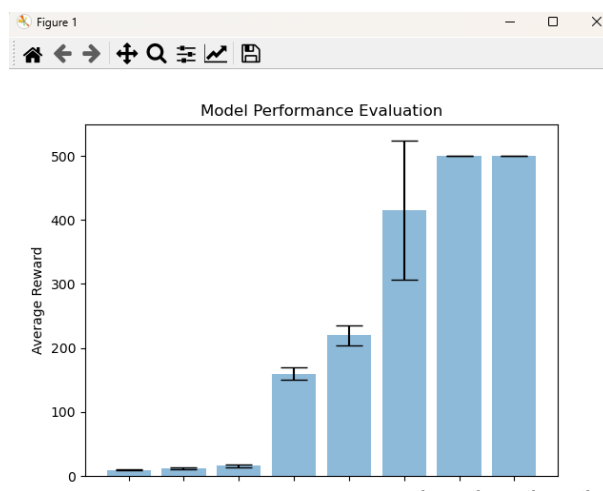


图 21: 调整结果

从左往右每次增多100，可以看到，在700时已经完美完成任务了，但是这并不是我们想要的。

**为什么？** 因为我们所要做的事情是调参，是用最小的算力完成任务，所以我们应该选取斜率最大的区间也就是500-600之间，这样才能更明显的反映出我们调参的变化。

最终决定调参时设置为550，最终模型设置为700.

### 5.3.3 调整gamma因子

在这里我意识到我可以用类似二分查找或者先训练几个测试一下的方法，不必一次性完成所有训练，所以可以少点训练，多一点观察。

”gamma”: [0.99, 0.96, 0.93, 0.90, 0.87],

```

最佳超参数组合: {'gamma': 0.93}, 最佳评估回报: 419.0
超参数: {'gamma': 0.99}, 评估回报: 327.0
超参数: {'gamma': 0.96}, 评估回报: 303.0
超参数: {'gamma': 0.93}, 评估回报: 419.0
超参数: {'gamma': 0.9}, 评估回报: 202.0
超参数: {'gamma': 0.87}, 评估回报: 283.0
最佳超参数: {'gamma': 0.93}

```

图 22: 网格输出

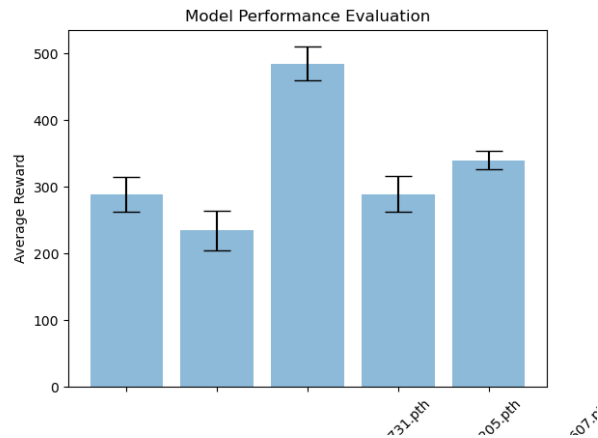


图 23: 验证结果

很奇怪的形状，不是凸型的局部最优。图中在0.93处局部最优，但是在降低更多后又有所回升，为了更相信的探究原因，我们细化参数，从0.03的间隔调整至0.01，同时到最后继续降低，等待结果。

我以0.005为间隔，从0.99设置到0.50，观察图像，却发现此时奖励函数变得非常奇怪：

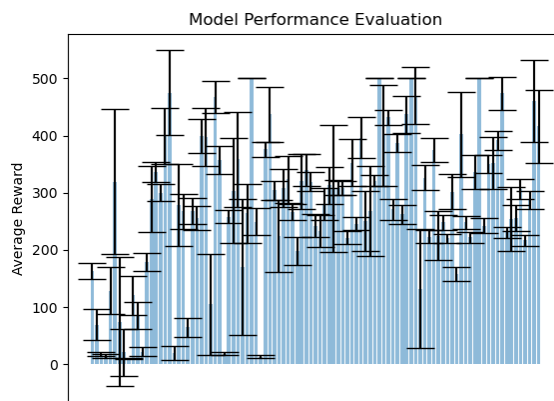


图 24: 左侧为0.5，右侧为0.99

此时出现了几个峰值，0.68，0.81，0.85，0.93。那么我认为选择周围梯度最小的0.85是合适的选择。

### 5.3.4 调整更新频率

降低迭代次数至450，更利于营造差异。

较低的更新频率可能导致学习不稳定，因为智能体会在较长的时间内使用相同的网络权重进行决策，可能会导致过拟合于某些状态或动作。相反，较高的更新频率可以使智能体更快地适应环境的变化，但也可能导致学习过程的噪声增加。

更新频率高可能会加快收敛速度，因为智能体可以更频繁地根据新的经验更新其策略。然而，过于频繁的更新可能导致策略的震荡，反而减慢收敛。

我们决定按2的指数进行尝试，从1到128，可以看到，模型在每一轮训练就更新时表现最好，而这样带来的额外开销在这个任务重并不明显，因此可以使用1。

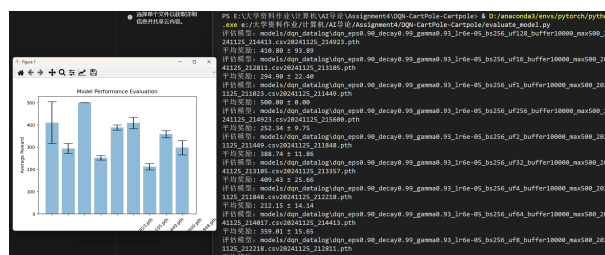


图 25: 验证结果

### 5.3.5 调整缓存区大小

降低迭代次数至350，更利于营造差异。

缓冲区大小我认为是和迭代次数一样的东西：多多益善。较大的缓冲区可以存储更多的经验，从而提高经验的多样性，有助于智能体更全面地学习不同状态下的最佳策略。同时，可以通过引入更丰富的历史经验来增强学习的稳定性，减少模型对某些特定经验的依赖。

```
最佳超参数组合: {'buffer_size': 2000}, 最佳评估回报: 500.0
超参数: {'buffer_size': 1000}, 评估回报: 92.0
超参数: {'buffer_size': 2000}, 评估回报: 500.0
超参数: {'buffer_size': 4000}, 评估回报: 395.0
超参数: {'buffer_size': 8000}, 评估回报: 330.0
超参数: {'buffer_size': 16000}, 评估回报: 16.0
超参数: {'buffer_size': 32000}, 评估回报: 439.0
超参数: {'buffer_size': 64000}, 评估回报: 500.0
最佳超参数: {'buffer_size': 2000}
```

图 26: 训练结果



```

评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs256_uf1_buffer1000_ma
平均奖励: 79.85 ± 17.75
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs256_uf1_buffer16000_ma
平均奖励: 15.41 ± 1.87
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs256_uf1_buffer2000_ma
平均奖励: 482.57 ± 24.54
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs256_uf1_buffer32000_ma
平均奖励: 399.58 ± 62.29
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs256_uf1_buffer4000_ma
平均奖励: 377.81 ± 46.86
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs256_uf1_buffer64000_ma
平均奖励: 500.00 ± 0.00
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs256_uf1_buffer8000_ma
平均奖励: 385.01 ± 15.68

```

图 27: 验证结果

从结果来看，2000和64000都达到了500的上限，这说明我们现在模型的能力已经能够很轻松的到达500了，考虑细化网格并提高最大限制次数。

至于之后训练可以使用2000的大小，进一步降低开销，最后训练最优模型使用64000。

### 5.3.6 调整每批次样本大小

回调训练轮次至450。

将训练轮次设置为2的指数，进行网格对比：

```

最佳超参数组合: {'batch_size': 64}, 最佳评估回报: 477.0
超参数: {'batch_size': 4}, 评估回报: 48.0
超参数: {'batch_size': 8}, 评估回报: 19.0
超参数: {'batch_size': 16}, 评估回报: 8.0
超参数: {'batch_size': 32}, 评估回报: 155.0
超参数: {'batch_size': 64}, 评估回报: 477.0
超参数: {'batch_size': 128}, 评估回报: 124.0
超参数: {'batch_size': 256}, 评估回报: 402.0
超参数: {'batch_size': 512}, 评估回报: 215.0
超参数: {'batch_size': 1024}, 评估回报: 10.0
最佳超参数: {'batch_size': 64}

```

图 28: 训练结果

```

评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs1024_uf1_buffer2000_ma
平均奖励: 9.32 ± 0.77
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs128_uf1_buffer2000_ma
平均奖励: 122.15 ± 3.63
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs16_uf1_buffer2000_ma
平均奖励: 9.34 ± 0.76
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs256_uf1_buffer2000_ma
平均奖励: 364.39 ± 31.74
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs32_uf1_buffer2000_ma
平均奖励: 165.02 ± 9.31
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs4_uf1_buffer2000_ma
平均奖励: 39.60 ± 11.41
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs512_uf1_buffer2000_ma
平均奖励: 223.72 ± 14.83
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs64_uf1_buffer2000_ma
平均奖励: 470.44 ± 10.86
评估模型: models/dqn_dataLog\dqn_eps0.90_decay0.99_gamma0.85_lr6e-05_bs8_uf1_buffer2000_ma
平均奖励: 22.22 ± 14.54

```

图 29: 验证结果

做到这里，我意识到一个问题，每批次样本大小和缓存区大小是相关的，应该尽量保证比例的统一，否则做出来的调参是无意义的。

### 5.3.7 调整探索率衰减率

较高的衰减率（如0.99或0.999）可以让智能体在较长时间内保持较高的探索率，适合于复杂的环境和长时间的训练。较低的衰减率（如0.95）则会让智能体较快地收敛到较低的探索率，适合简单或稳定的环境。

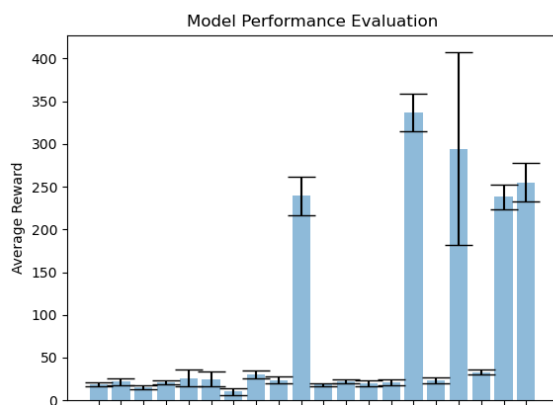


图 30: 衰减率，左侧为0.9，右侧为0.99

再来观察损失的衰减：

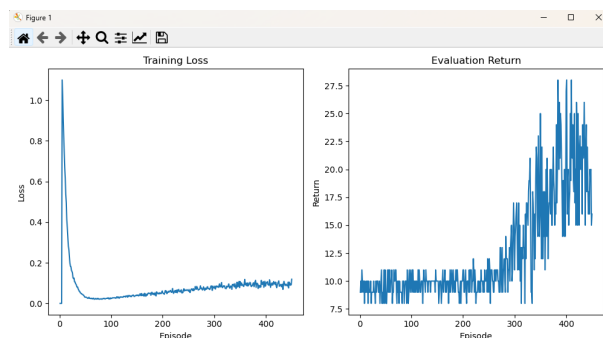


图 31: 0.90时

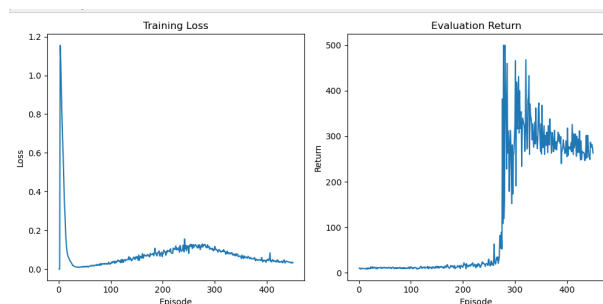


图 32: 0.99时

我们注意到当开始进行no\_explore时，良好的训练样本应该收敛而不是发散。当然应该增大训练轮次也会收敛，不过我们的目的是最小学习代价，要取代价最小的参数。

### 5.3.8 调整最低探索率

最低探索率决定了后期还有多少探索的空间，因此可以适当增大训练轮次，来观察效果。

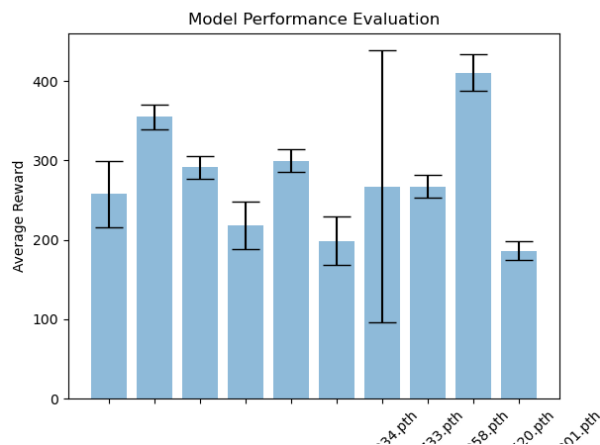


图 33: 验证结果

可以看出只要把最低探索率来控制在一个合理的范围内不会特别影响训练结果，不过相对来说还是0.07比较好。

### 5.3.9 调整最初探索率

相对来说就更无关紧要了，随便跑几个数吧。

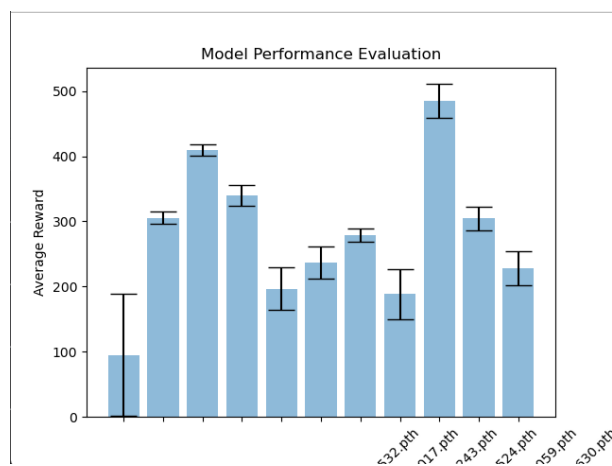


图 34: 左侧为0.9,右侧为1.0

找出最佳参数为0.98.

## 5.4 最终调参结果

那么我最终的调参结果为：

```

Codeium: Refactor | Explain | X
def get_args():
    """
    解析命令行参数
    """
    parser = argparse.ArgumentParser()
    parser.add_argument("--agent_name", type=str, default="dqn", help="选择智能体类型: dqn 或 ddqn")
    parser.add_argument("--num_episodes", type=int, default=550, help="训练的总回合数")
    parser.add_argument("--max_steps_per_episode", type=int, default=1000, help="每个回合的最大步数")
    parser.add_argument("--epsilon_start", type=float, default=0.98, help="初始探索率")
    parser.add_argument("--epsilon_end", type=float, default=0.05, help="最低探索率")
    parser.add_argument("--epsilon_decay_rate", type=float, default=0.985, help="探索率衰减率")
    parser.add_argument("--gamma", type=float, default=0.85, help="折扣因子")
    parser.add_argument("--lr", type=float, default=6e-5, help="学习率")
    parser.add_argument("--buffer_size", type=int, default=64000, help="经验回放缓冲区大小")
    parser.add_argument("--batch_size", type=int, default=64, help="每批次的样本大小")
    parser.add_argument("--update_frequency", type=int, default=1, help="网络更新频率")
    args = parser.parse_args()
    return args

```

图 35: 调参结果

## 6 换用DDQN

任务要求我们分析区别，理论上的区别前文已经讲过了，现在来看看一些特征：

### 6.1 训练时间

对比文件的时间戳，我发现，DDQN(8分钟)要比DQN(4分钟)的训练时间长一倍左右，但是训练效果却非常好。

### 6.2 训练效果

对两个模型进行可视化，观看他们的动作视频：

可以发现DDQN明显有一个来回摇摆保持平衡位置在中部的动作，而DQN却是尽量让他保持直立，而不考虑底座已经越来越靠边直到出界。

结合理论中所说“旨在解决DQN中存在的过高估计Q值的问题”，我认为这是DQN对于已经学习状态的奖励过高估计导致的偏向性。

### 6.3 探索DDQN性能极限

如果我们降低缓冲区大小或者训练轮次，使得DDQN与DQN训练时间一样，也就是消耗算力大致相同，那结果又如何呢？

先改为350轮进行尝试：

```

回合 320/350 步数 362: 训练损失 0.0739, 评估回报 452.0
回合 321/350 步数 331: 训练损失 0.0836, 评估回报 333.0
回合 322/350 步数 435: 训练损失 0.0770, 评估回报 477.0
回合 323/350 步数 249: 训练损失 0.0758, 评估回报 478.0
回合 324/350 步数 349: 训练损失 0.0799, 评估回报 500.0
回合 325/350 步数 400: 训练损失 0.0773, 评估回报 500.0
回合 326/350 步数 500: 训练损失 0.0687, 评估回报 500.0
回合 327/350 步数 500: 训练损失 0.0709, 评估回报 500.0
回合 328/350 步数 434: 训练损失 0.0718, 评估回报 500.0
回合 329/350 步数 500: 训练损失 0.0697, 评估回报 500.0
回合 330/350 步数 500: 训练损失 0.0612, 评估回报 500.0
回合 331/350 步数 500: 训练损失 0.0714, 评估回报 500.0
回合 332/350 步数 500: 训练损失 0.0763, 评估回报 500.0
回合 333/350 步数 500: 训练损失 0.0668, 评估回报 500.0
回合 334/350 步数 500: 训练损失 0.0629, 评估回报 500.0
回合 335/350 步数 500: 训练损失 0.0768, 评估回报 500.0
回合 336/350 步数 500: 训练损失 0.0637, 评估回报 500.0
回合 337/350 步数 500: 训练损失 0.0674, 评估回报 500.0
回合 338/350 步数 500: 训练损失 0.0641, 评估回报 500.0
回合 339/350 步数 500: 训练损失 0.0686, 评估回报 500.0
回合 340/350 步数 500: 训练损失 0.0656, 评估回报 500.0
回合 341/350 步数 500: 训练损失 0.0679, 评估回报 500.0
回合 342/350 步数 500: 训练损失 0.0624, 评估回报 500.0
回合 343/350 步数 500: 训练损失 0.0590, 评估回报 500.0
回合 344/350 步数 500: 训练损失 0.0748, 评估回报 500.0
回合 345/350 步数 500: 训练损失 0.0786, 评估回报 500.0
回合 346/350 步数 500: 训练损失 0.0664, 评估回报 500.0
回合 347/350 步数 500: 训练损失 0.0710, 评估回报 500.0
回合 348/350 步数 500: 训练损失 0.0631, 评估回报 500.0
回合 349/350 步数 500: 训练损失 0.0713, 评估回报 500.0
回合 350/350 步数 500: 训练损失 0.0679, 评估回报 500.0
训练完成, 最终模型已保存到 models/ddqn_datalog/ddqn_eps0.9
r64000_max1000_20241126_151944.csv20241126_152210.pth
超参数组合: {}, 评估回报: 500.0
最佳超参数组合: {}, 最佳评估回报: 500.0
超参数: {}, 评估回报: 500.0
最佳超参数: {}

```

图 36: 令人震惊的碾压级表现!

令人震惊的碾压级表现! 仅用时2分44秒完成完美训练!

再次降低为300轮: 此次训练时间已经降低到了44秒, 训练结果为平均奖励: 498.54  
± 6.65

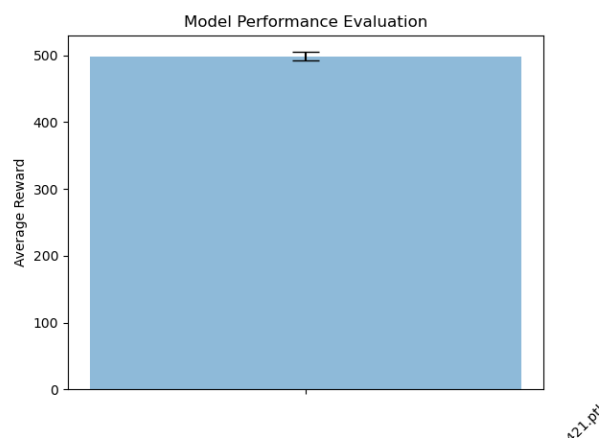


图 37: 训练结果

说明这里就是模型保持性能的最小时间了, 4倍短于DQN, 改进力度非常大。

## 7 结束语

强化学习确实是一个很新很有挑战性的领域，在一开始理解代码时遇到了许多困难，不过在查阅资料的过程中，发现所有的一切都可以在13年的那篇万恶之源(Playing Atari with Deep Reinforcement Learning)和15年DeepMind的(Human-level control through deep reinforcement learning)中找到。

不过我觉得最适合让初学者理解的应该是这篇：(Reinforcement Learning Explained Visually (Part 5): Deep Q Networks, step-by-step)，这篇里面的流程图画的真的很好（贴一张）

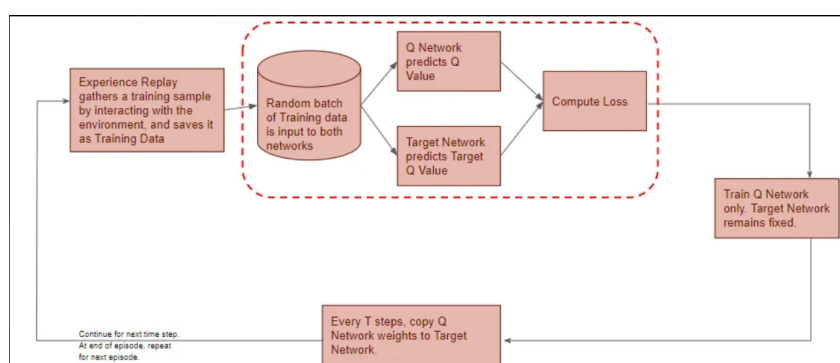


图 38: 流程图

适合给人一个直观的理解。

还有我的NVIDIA GeForce RTX 3090 Laptop GPU至少满负荷运行了100个小时，算折旧不少钱呢。

## 参考文献

(Playing Atari with Deep Reinforcement Learning, <https://arxiv.org/abs/1312.5602>)

(Human-level control through deep reinforcement learning, <https://www.nature.com/articles/nature14236>)

(Reinforcement Learning Explained Visually (Part 5): Deep Q Networks, step-by-step, <https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b>)