

# dqn的实现

## Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do**

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

        With probability  $\varepsilon$  select a random action  $a_t$

        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$

        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$

        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$

        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

        Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

        Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

**End For**

DQN = Q-Learning + 深度神经网络 (Deep Learning)。

这张图片展示的是强化学习领域非常经典的一个算法：**带有经验回放的深度 Q 网络 (Deep Q-learning with experience replay, 通常简称为 DQN)** 的伪代码。

这个算法由 DeepMind 提出，标志着深度学习与强化学习结合的重大突破（最初用于让 AI 玩雅达利游戏）。它解决了使用神经网络逼近 Q 函数时容易出现的不稳定和发散问题。

为了让你更清晰地理解，我们把这端伪代码拆解成几个核心部分来看看：

## 1. 初始化阶段 (Initialization)

在正式开始训练之前，算法需要准备三个关键组件：

- **Initialize replay memory  $D$  to capacity  $N$** : 初始化一个容量为  $N$  的“经验回放池”(Replay Buffer)。它就像是智能体的记忆库，用来存放过去的经历。
- **Initialize action-value function  $Q$  with random weights  $\theta$** : 初始化**主 Q 网络** (Main Network)，参数为  $\theta$ 。这个网络负责在每一步选择动作，并在训练中不断更新。
- **Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$** : 初始化**目标 Q 网络** (Target Network)，初始参数  $\theta^-$  和主网络完全一样。这是 DQN 保持稳定的一个关键技

巧（双网络结构）。

## 2. 交互与数据收集 (Interaction & Data Collection)

代码进入了两个循环：外层是 Episode（回合），内层是 Time step（时间步）。

- **$\epsilon$ -greedy 策略选择动作**: 在每个时间步  $t$ ，智能体以  $\epsilon$  的概率随机选择一个动作（探索环境），否则选择当前 Q 网络认为价值最大的动作  $\operatorname{argmax}_a Q(\phi(s_t), a; \theta)$ （利用现有知识）。
- **执行动作与观察**: 智能体在环境（仿真器）中执行动作  $a_t$ ，得到奖励  $r_t$  和下一个画面的观测值  $x_{t+1}$ 。
- **状态处理**: 将信息组合并预处理成状态  $\phi_{t+1}$ （在原始论文中，通常是把连续几帧图像叠在一起作为输入，以捕捉动态信息）。
- **存储经验 (Store transition)**: 将这一步的完整经历  $(\phi_t, a_t, r_t, \phi_{t+1})$  打包成一个元组，存入刚才初始化的经验回放池  $D$  中。

## 3. 学习与网络更新 (Learning & Optimization)

这是算法最核心的训练过程，发生在每一步数据存入回放池之后：

- **经验回放 (Sample random minibatch)**: 从回放池  $D$  中随机抽取一小批（minibatch）历史经验。这一步打破了连续数据之间的时间相关性，让神经网络训练更加稳定。
- **计算目标值 (Set  $y_j$ )**: 这里计算的是 TD Target（时序差分目标），它是神经网络试图逼近的“正确答案”：
  - 如果这一步游戏结束了（episode terminates），目标值就是当前的奖励  $r_j$ 。
  - 如果游戏还没结束，目标值就是： $r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-)$ 。注意这里使用的是目标网络  $\hat{Q}$ ，而不是主网络。它评估的是在下一个状态下能拿到的最大未来预期奖励（ $\gamma$  是折扣因子）。
- **梯度下降 (Gradient descent)**: 计算目标值  $y_j$  与主网络当前预测值  $Q(\phi_j, a_j; \theta)$  之间的均方误差（MSE Loss），然后通过梯度下降来更新主网络的参数  $\theta$ 。
- **更新目标网络 (Reset  $\hat{Q} = Q$ )**: 每经过  $C$  个步长，就把主网络的参数  $\theta$  复制给目标网络的参数  $\theta^-$ 。这种“延迟更新”避免了目标值在训练过程中剧烈波动，相当于在追逐一个相对静止的靶子。

总结来说，这个算法的两大灵魂机制是：

1. **Experience Replay（经验回放）**: 随机打乱过去的的数据来训练，提高数据利用率并稳定训练。
2. **Target Network（目标网络）**: 用一个更新频率较慢的独立网络来计算 Loss 的目标值，防止训练过程出现震荡（即自己追着自己的尾巴跑）。

## 3. 代码实现

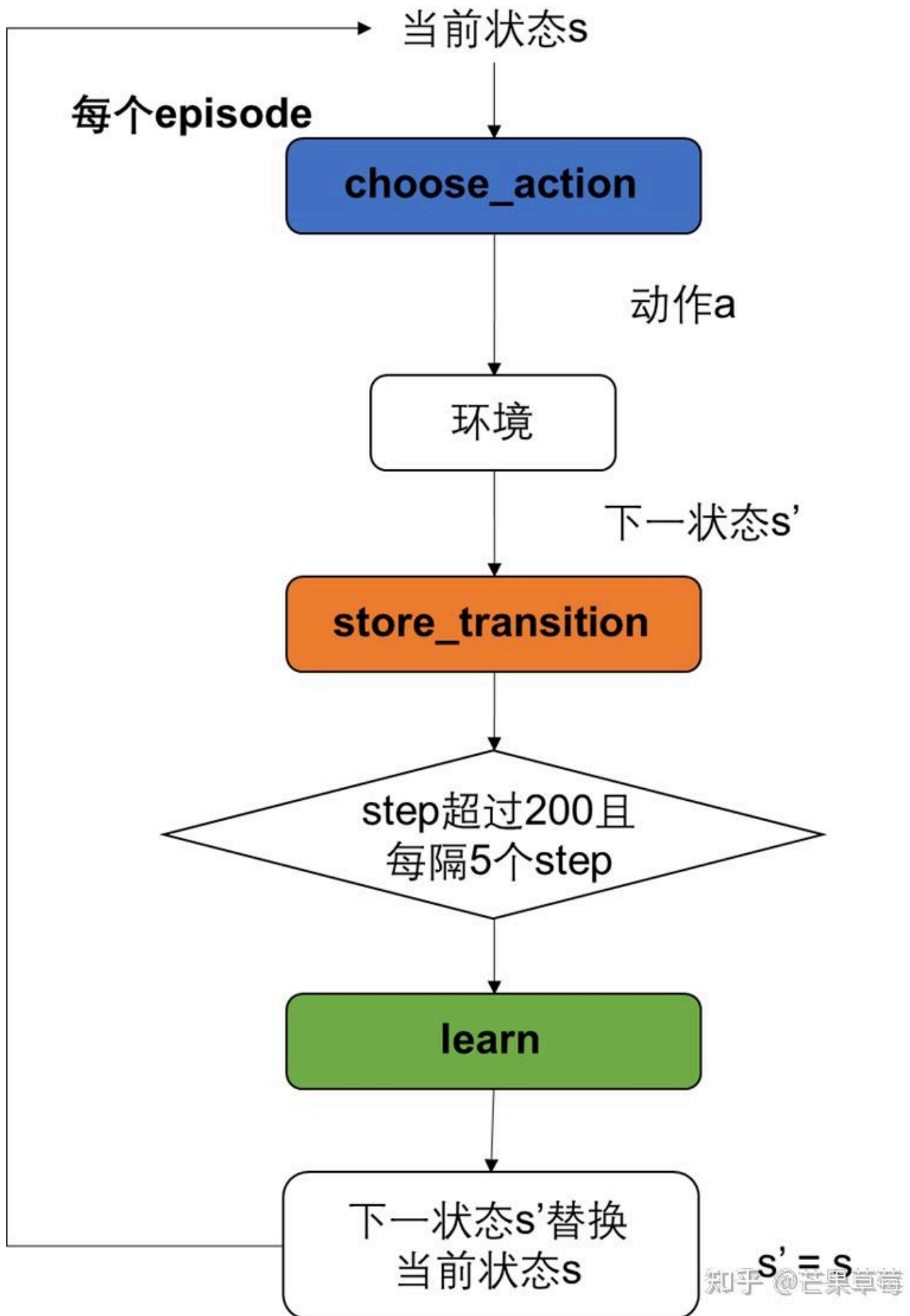
由于想要快速验证这个dqn的实现，我选择了使用flybird作为训练的对象，以下是flybird的游戏简介：

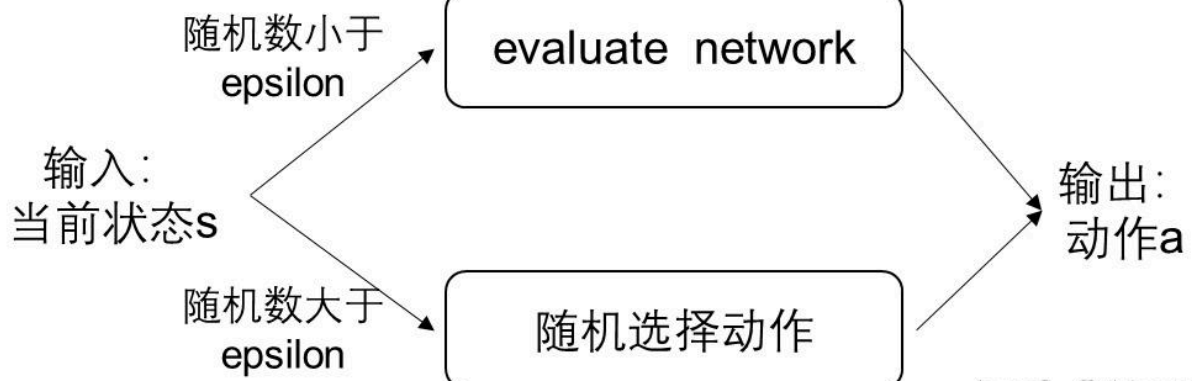
## 游戏介绍

《Flappy Bird》（像素鸟）是一款采用复古像素画风的无尽生存类街机游戏，其**玩法**极其极简，玩家只需通过不断点击按键来控制小鸟振翅对抗重力，使其在不断向左移动、高低不一的绿色水管缝隙中精准穿梭；游戏的**得分机制**简单直白，小鸟每成功安全穿过一对水管即可获得1分，并且没有最终的通关终点；而该游戏的**核心特性**则在于其硬核的“零容错率”与“高频试错”机制——小鸟一旦触碰水管或掉落地面就会瞬间死亡并立刻重置，这种操作极简但极难精通的设定，配合着无尽模式的心理压力，造就了它极高的挑战性和令人极度上瘾的属性。

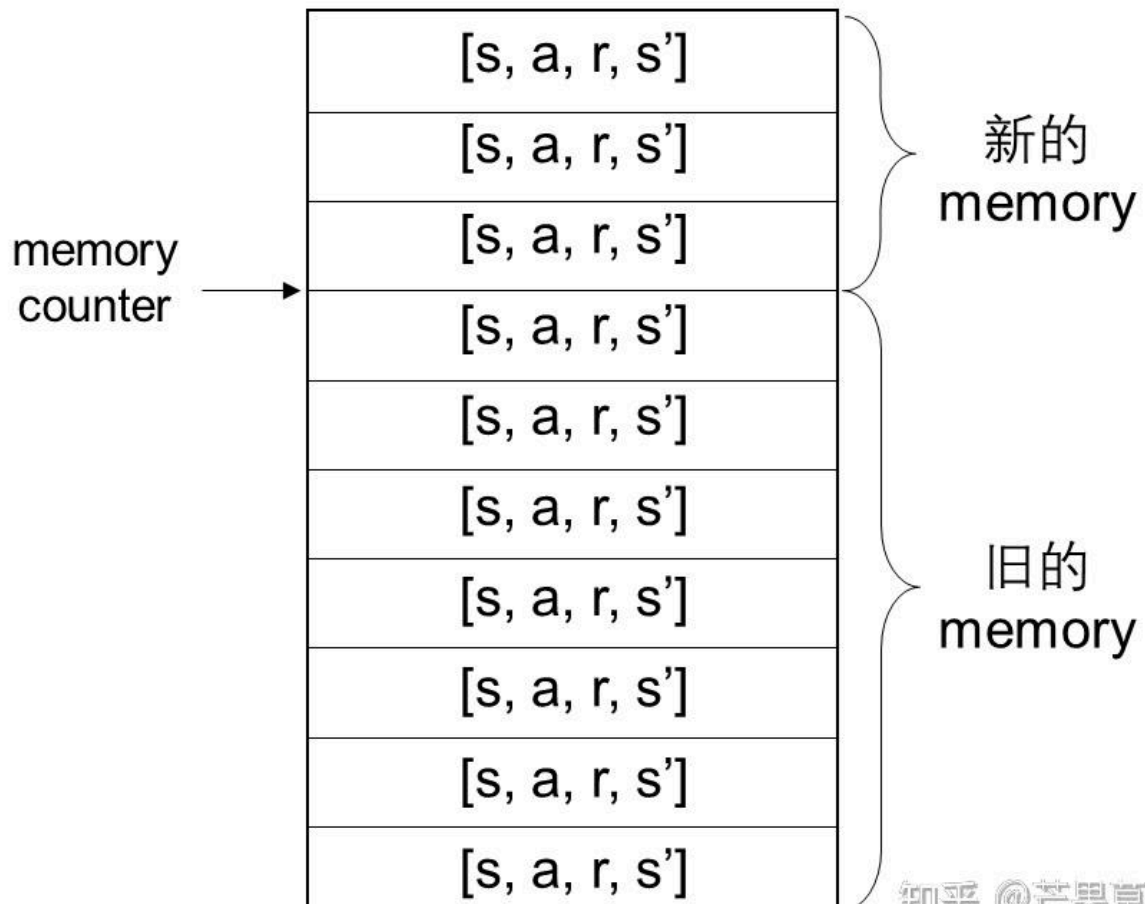
## 代码流程图

## 算法一般流程图



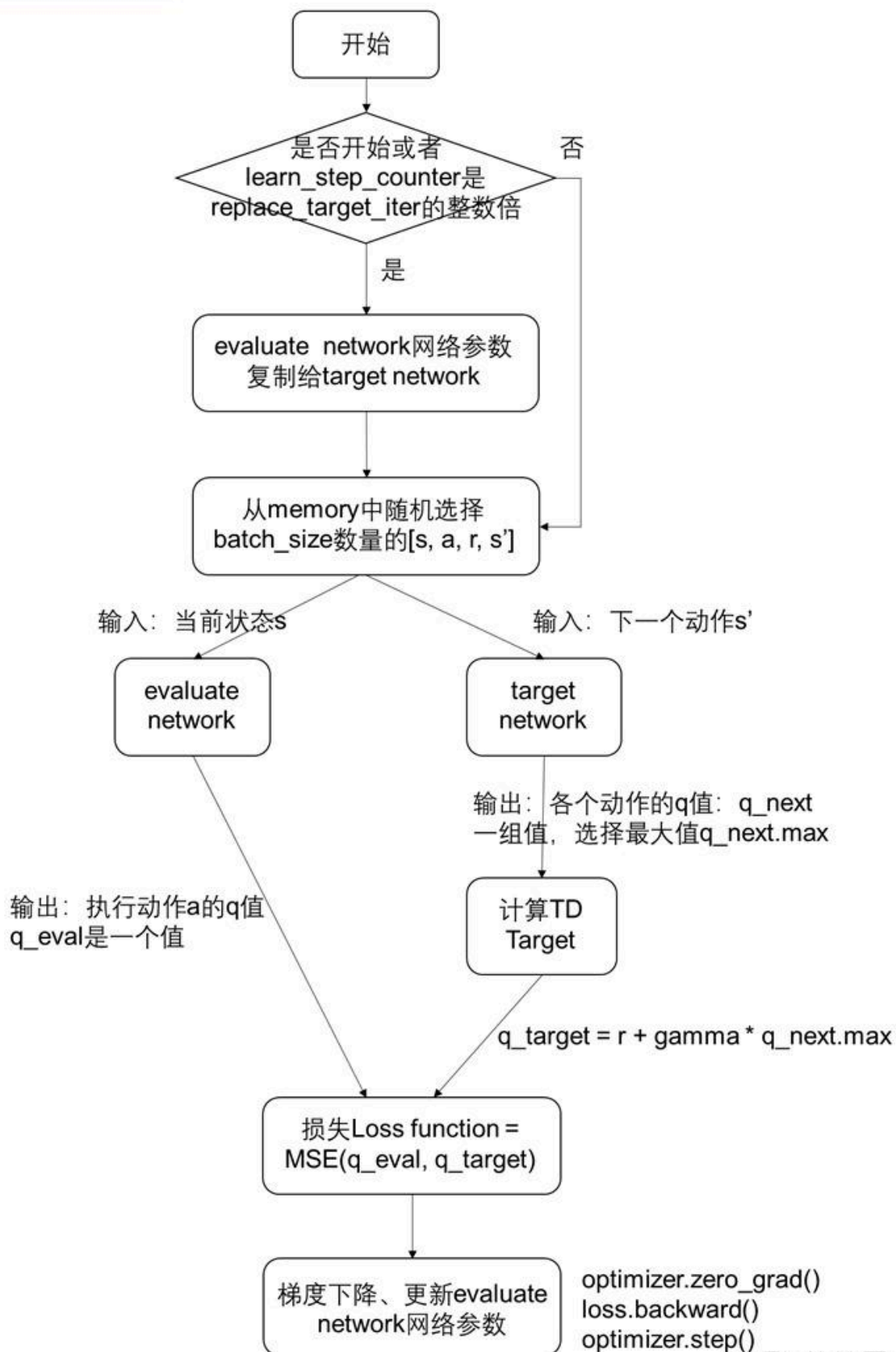
**choose\_action**

知乎 @芒果草莓

**store\_transition**

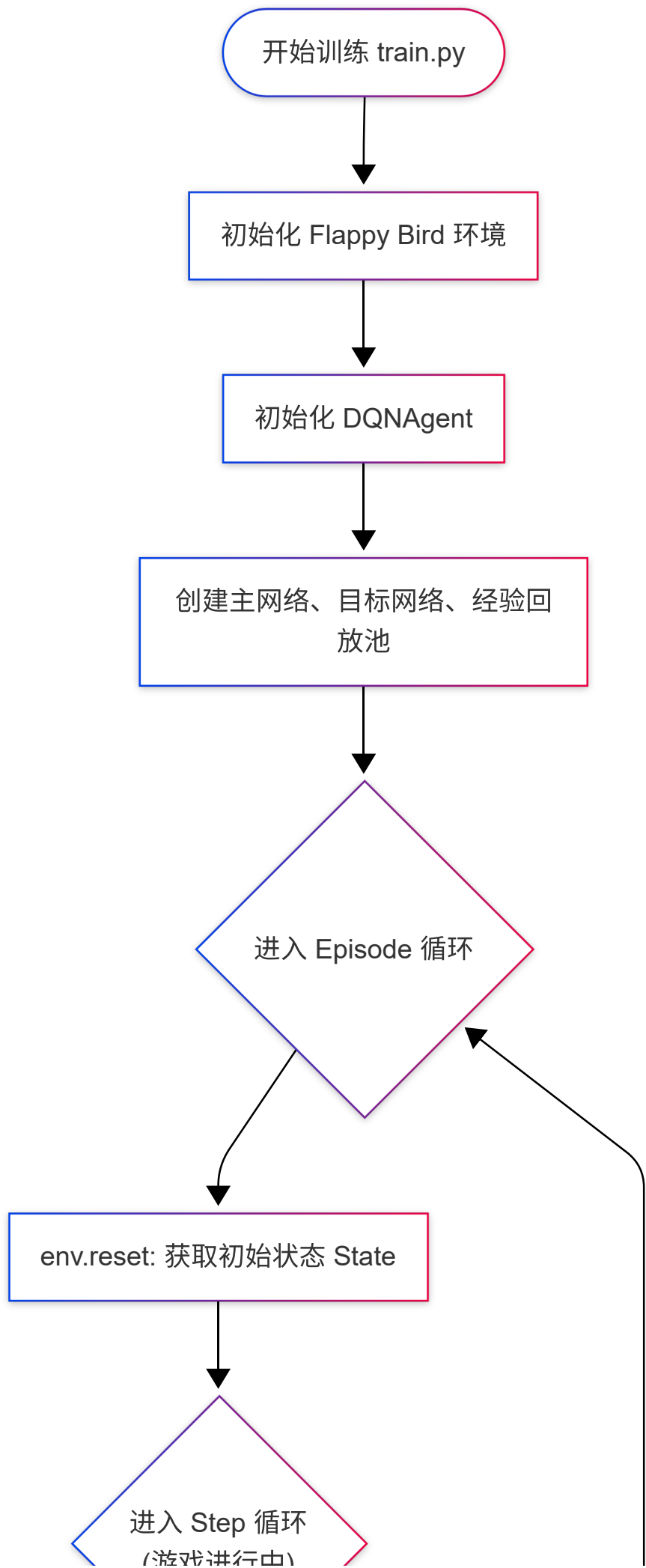
知乎 @芒果草莓

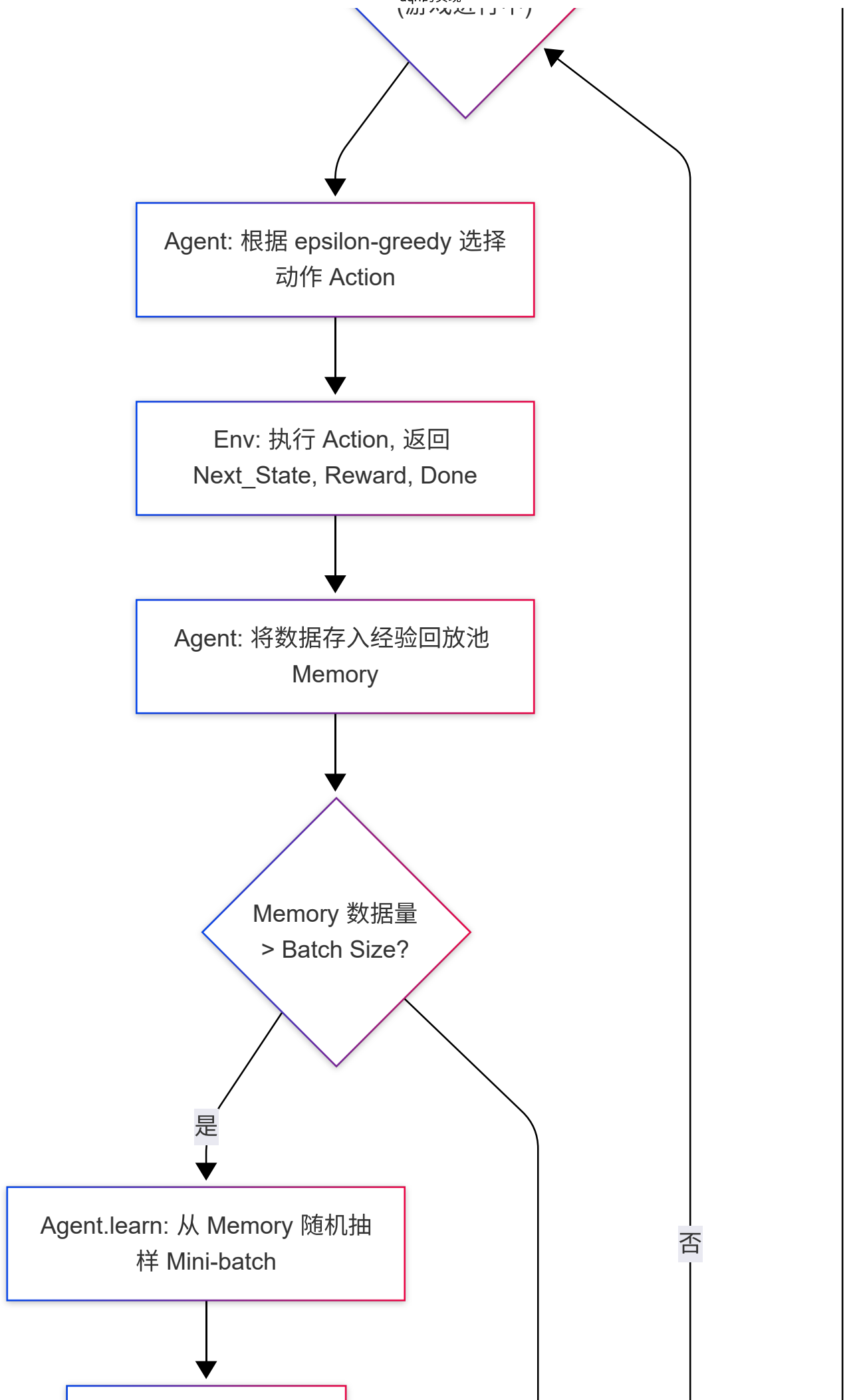
## learn



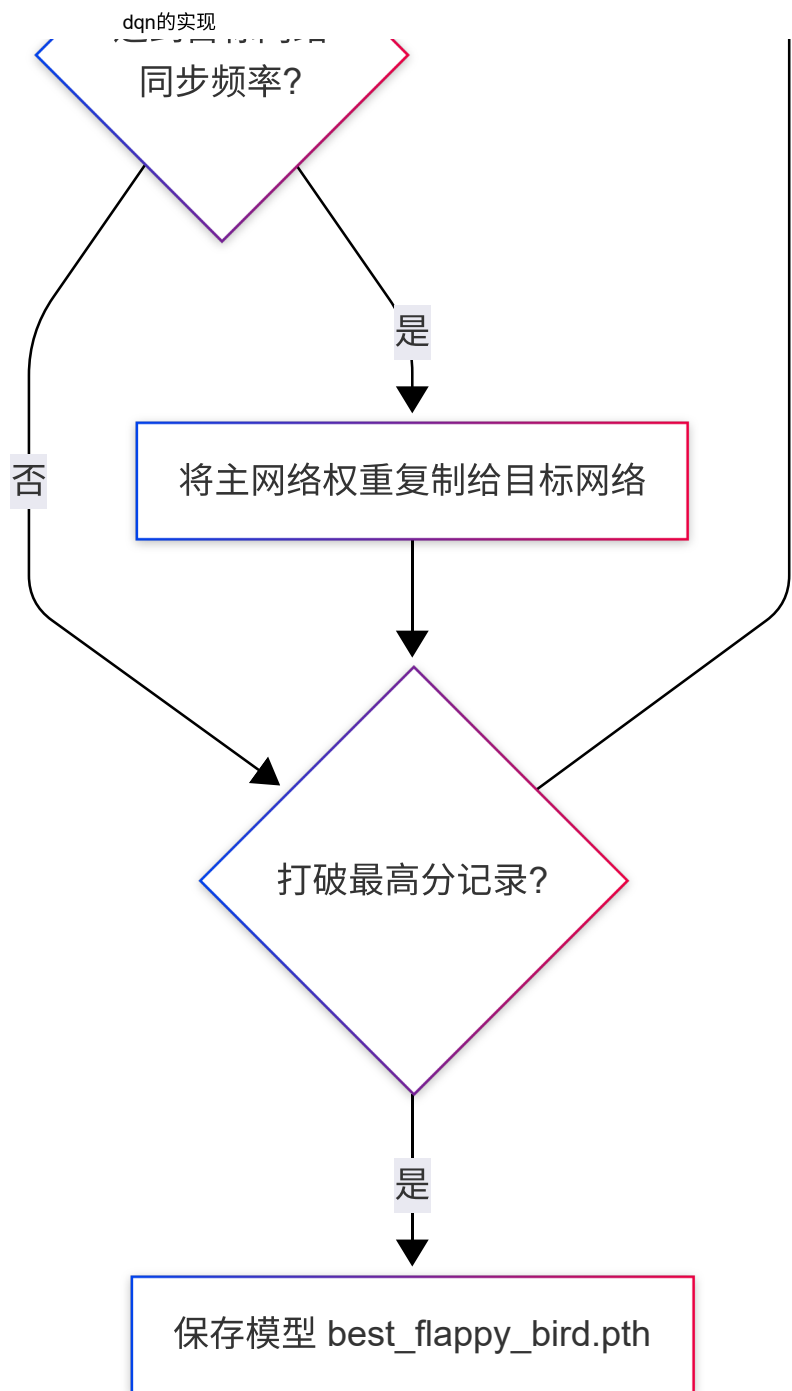
知乎 @芒果草莓

本项目流程图









## 环境选择

本实验采用 Gymnasium (原 OpenAI Gym) 作为强化学习的环境框架。选择该框架是因为它提供了一套高度标准化的环境交互 API ( `step`, `reset` ), 实现了智能体算法与底层环境物理逻辑的完美解耦。这不仅极大地提高了算法代码的复用性, 也使得本实验的结果具备了与国际主流强化学习基准 (Benchmark) 进行公平对比的学术价值。

最吸引我的一点就是, 我不用在耗费时间在环境的搭建上, 只需要来注意agent的设计与算法的设计就可以了, 并且假如明天你的导师说: “别搞像素鸟了, 去搞个倒立摆 (CartPole) 或者走迷宫吧!” 你**完全不需要修改** `dqn_agent.py` 里的**任何一行核心算法代码**, 你只需要在 `train.py` 里把一行代码改掉:

- 从: `env = gym.make("FlappyBird-v0")`

- 改成： `env = gym.make("CartPole-v1")`

你的 AI 就能立刻无缝衔接，开始学习新游戏。这就叫“一次编写，到处运行”。

## 代码理解

### 串行代码理解

因为是第一次写这样的代码，所以是先让 **ai** 帮我写出来然后我自己根据理解来进行修改，也是因为第一次写这个代码所以可能会注释很多的基础知识。

首先，文件是由三部分构成，`train.py`, `dqn_agent.py`, `test.py`，在训练的过程之中我们不使用渲染的方法，可以加速我们的训练的过程，然后再 `test.py` 里面就是把我们的训练的结果拿过来使用，这时候，为了更加直观地展示这个动画，我们就会把他渲染出来。`dqn_agent.py` 就是我们的大脑负责决策的代理。

```
env = gymnasium.make("FlappyBird-v0", use_lidar=False)
```

`use_lidar=False`（这个库特有的参数）

- **含义：** `lidar` 是“激光雷达”的意思。 `False` 表示关闭它。
- **作用：** 如果设为 `True`，环境会模拟出几条激光射线从鸟的身上发射出去，用来探测管子的距离（返回的数据会非常庞大和复杂，通常是一大串距离数值）。
  - 设为 `False` 后，环境就会变得非常清爽，它只会返回给你最核心的 **12 个数字**（比如鸟的高度、速度、下一根管子的 X 和 Y 坐标等）。
  - **结论：** 对于初学者和我们写的简易 DQN 来说，关闭雷达（`use_lidar=False`）能让 AI 更容易看懂数据，训练速度会快得多。

`FlappyBird-v0`

- **含义：** 环境的唯一识别码（ID）。
- **作用：** 告诉工厂你要玩哪款游戏。`FlappyBird` 是游戏名，`-v0` 代表这是该游戏的第 0 个版本（如果以后作者修复了 bug 或改了规则，可能会推出 `-v1`、`-v2`）。

```
state_dim = env.observation_space.shape[0] # 12
```

```
action_dim = env.action_space.n # 2
```

1. `state_dim = env.observation_space.shape[0]`（状态维度）

- **字面意思**：获取环境（env）观测空间（observation\_space）的第一个形状数值（shape[0]）。
- **通俗理解**：“AI 的输入有多少个数字？”
- **针对 Flappy Bird**：这个库规定了它会返回 12 个关键数字给 AI（包括鸟的高度、速度、离下一根管子的水平距离、管子上下沿的高度等）。
  - 所以这里的 state\_dim 结果就是 12。
- **为什么这么写**：为了**通用性**。如果你明天把环境换成了“倒立摆”，那里的状态只有 4 个数字，代码会自动把 state\_dim 设为 4，而不需要你手动去改。

## 2. action\_dim = env.action\_space.n (动作维度)

- **字面意思**：获取环境（env）动作空间（action\_space）的数量（n）。
- **通俗理解**：“AI 有多少种选择？”
- **针对 Flappy Bird**：这个游戏里你只有两个选择：
  1. 0：什么都不做（任由重力下坠）。
  2. 1：点击屏幕（向上扇动翅膀）。
  - 所以这里的 action\_dim 结果就是 2。
- **为什么要这么写**：同样是为了**通用性**。如果换成一个有“上下左右”四个键的游戏，action\_dim 就会自动变成 4。

# 训练超参数

EPISODES = 100000

BATCH\_SIZE = 64

TARGET\_UPDATE\_FREQ = 10 # 每过 10 局同步一次目标网络

EPISODES代表的是训练的次数你可以理解为有多少条命

BATCH\_SIZE = 64（批量大小）

- **物理意义**：每次“大脑更新”时，从记忆库中抽取的样本数量。
- **串行逻辑**：
  1. 小鸟每飞一步，就把（当前状态，动作，奖励，下个状态）存进书包（经验回放池）。
  2. 当书包里的数据够多时，AI 会停下来，随机从书包里抓出 64 条历史记录。
  3. 神经网络根据这 64 条记录算出一个平均的误差，然后调整自己的参数。
- **为什么要 64?**：如果你只拿 1 条记录更新，AI 容易被某一次偶然的幸运或倒霉带偏（随机性太大）；拿 64 条一起看，它能学到更稳健的规律。
- TARGET\_UPDATE\_FREQ = 10（目标网络同步频率）

- **这是最关键的稳定性设计！**
- **背景知识：**DQN 内部有两个一模一样的网络：
  1. **Policy Net (策略网络)：**它是“学生”，时刻在学习，参数一直在变。
  2. **Target Net (目标网络)：**它是“参考书”，用来计算目标分数值。

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

    total_reward = 0

    score = 0 # 记录穿过的管子数

    done = False
```

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

- **动作：**环境彻底重启。
- **物理意义：**小鸟回到出发点（通常是屏幕左侧中间），水管重新随机生成。
- **返回值：**
  - `state`：初始的 **12 个数字**（坐标、速度等）。这是 AI 睁开眼看到的第一幕。
  - `info`：一个字典，包含一些辅助信息（比如当前得分、游戏模式等），通常训练时用不到，但必须接收它。

```
done = False
```

- **物理意义：**设置一个“死亡标志位”。
- **逻辑作用：**只要小鸟没撞墙，`done` 就一直是 `False`。一旦撞墙，环境会返回 `done = True`，从而跳出当前的 `while` 循环，结束这一局。

```
while not done:

    # 1. 选动作

    action = agent.select_action(state)

    # 2. 与环境交互

    next_state, reward, terminated, truncated, info =
env.step(action)

    done = terminated or truncated

    # 3. 存入经验池
```

```

agent.memory.push(state, action, reward, next_state, done)

# 4. 学习更新

agent.learn(BATCH_SIZE)

state = next_state

total_reward += reward

score = info.get('score', 0)

```

`action = agent.select_action(state)`: AI 查看当前 12 个数字（状态），决定是点击屏幕还是什么都不做。

这里面运行着 **Epsilon-Greedy ( $\epsilon$ -贪婪)** 策略。

- **前期**: AI 大多在“瞎猜”(探索)，寻找通往管子的路。
- **后期**: AI 大多在“查表”(利用)，执行它认为得分最高的动作。  
`terminated/truncated`: 撞墙了吗？或者游戏是不是强制结束了？

## dqn\_agent

```

# 设置设备（如果有 GPU 则使用 GPU，否则使用 CPU）

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

```

# 1. 定义 Q 神经网络（因为输入是12个数字，用全连接层即可）

```

class QNetwork(nn.Module):

    def __init__(self, state_dim, action_dim):

        super(QNetwork, self).__init__()

        self.fc1 = nn.Linear(state_dim, 128)

        self.fc2 = nn.Linear(128, 128)

        self.fc3 = nn.Linear(128, action_dim)

```

```
def forward(self, x):

    x = F.relu(self.fc1(x))

    x = F.relu(self.fc2(x))

    return self.fc3(x)
```

self.fc1 = nn.Linear(state\_dim, 128)输入层

self.fc2 = nn.Linear(128, 128)隐藏层

self.fc3 = nn.Linear(128, action\_dim)输出层

## 为什么 Flappy Bird 常用 128 或 256?

对于 12 个输入特征来说，128 是一个“性能甜点”：

1. **特征组合足够多**：128 个神经元足以穷举出“高度”、“速度”和“障碍物距离”之间的各种逻辑关系。
2. **冗余度适中**：强化学习的信号（Reward）非常嘈杂，128 能够提供一定的容错空间，让网络在受到某些错误经验冲击时，不至于全盘崩溃。

### # 2. 定义经验回放池

```
class ReplayBuffer:
```

```
    def __init__(self, capacity):
```

```
        self.buffer = deque(maxlen=capacity)
```

这是 `deque` 的精髓。当存入的经验超过容量（比如你设定的 10 万条）时，它会自动“撕掉”最旧的一页，填入新的一页。这保证了 AI 永远在学习最近、最相关的经验。

```
    def push(self, state, action, reward, next_state, done):
```

```
        self.buffer.append((state, action, reward, next_state, done))
```

记录刚才的数据，它完整记录了\*\*“处境(s) -> 决策(a) -> 结果(r) -> 新处境(s') -> 死了没(d)”\*\*。这五个数字构成了一个完整的因果逻辑链。

```
    def sample(self, batch_size):
```

```
batch = random.sample(self.buffer, batch_size)
```

**\*\*`batch`\*\***: 是一个列表，里面塞了 64 个小元组，像这样：`[(s,a,r,s,d), (s,a,r,s,d), ...]`。

```
state, action, reward, next_state, done = map(np.stack, zip(*batch))
```

把这些分散的数字“啪”地一声压成整齐的**\*\*矩阵（Tensor/Array）\*\***。

- **\*\*结果\*\***: 原来的 64 个独立数字，变成了形状为 `(64, 12)` 的矩阵。这样你的神经网络（PyTorch）才能一次性处理它们，发挥显卡的并行计算能力。

```
return state, action, reward, next_state, done
```

```
def __len__(self):
```

```
    return len(self.buffer)
```

```
# 超参数
```

```
self.lr = 1e-3
```

```
self.gamma = 0.99
```

```
self.epsilon = 1.0          # 初始探索率
```

```
self.epsilon_min = 0.01     # 最低探索率
```

```
self.epsilon_decay = 0.995 # 探索率衰减
```

## 1. self.lr = 1e-3 (学习率 - Learning Rate)

- **物理意义**: 神经网络权重更新的步长。
- **形象理解**: “AI 的谦虚程度”。
  - 如果 lr 太大 (如 0.1): AI 发现一次失误就会全盘否定之前的经验, 导致表现剧烈震荡。
  - 如果 lr 太小 (如 1e-6): AI 极其固执, 学得非常慢, 可能跑几天几夜都没反应。
- **针对 Flappy Bird**: 1e-3 (即 0.001) 是一个很稳的标准值。

## 2. self.gamma = 0.99 (折扣因子 - Discount Factor)

- **物理意义**: 未来奖励的价值在今天的折现率。

- **形象理解：“AI 的远见程度”。**
  - 它衡量的是：“**现在的安全**”和“**未来的得分**”哪个更重要。
  - $\gamma = 0$ ：近视眼。只管这一帧活没活着，完全不管前面有没有管子。
  - $\gamma = 0.99$ ：深谋远虑。它意识到，为了钻过 5 秒后的管子，现在必须提前调整高度。
- **计算公式：**目标  $Q$  值计算的核心公式是  $R + \gamma \times \max Q(s')$ 。

### 3. Epsilon 三人组 (探索与利用)

这是你刚才看到“最高分从 0 变到 2”的关键逻辑。

- `self.epsilon = 1.0`：初始状态。100% 的时间在乱飞，纯粹是为了撞大运收集第一批数据。
- `self.epsilon_decay = 0.995`：
  - **串行逻辑：**每局比赛结束后，`epsilon = epsilon * 0.995`。
  - 这意味着 AI 随着“投胎”次数增加，会越来越相信自己的大脑。
- `self.epsilon_min = 0.01`：“**保留一点点好奇心**”。即便 AI 已经是世界冠军了，我们也保留 1% 的概率让它乱跳一下。万一能发现更优的飞行路线呢？

```
def select_action(self, state):

    if random.random() < self.epsilon:

        return random.randrange(self.action_dim) # 随机探索

    else:

        with torch.no_grad():

            state_tensor =
torch.FloatTensor(state).unsqueeze(0).to(device)

            q_values = self.policy_net(state_tensor)

            return q_values.argmax().item()      # 利用已知最优动作
```

逻辑拆解

#### 第一部分：随机探索 (Exploration)

```
if random.random() < self.epsilon:
    return random.randrange(self.action_dim)
```

- **动作：** 掷一个骰子（0到1之间的随机数）。如果这个数小于目前的 `epsilon`，AI 就直接闭着眼瞎选一个动作（0 或 1）。
- **目的：寻找可能性。** 即便目前大脑认为“不跳”更好，但万一“跳一下”能发现新大陆呢？这就是为什么你刚开始训练时，小鸟一直在乱飞。

## 第二部分：经验利用 (Exploitation)

Python

```
else:
    with torch.no_grad():
        state_tensor = torch.FloatTensor(state).unsqueeze(0).to(device)
        q_values = self.policy_net(state_tensor)
        return q_values.argmax().item()
```

- **动作：** 如果随机数没落在探索区，AI 就会**认真思考**。
- `torch.no_grad()`：告诉 PyTorch，“我现在只是在做决定，不需要计算梯度，别浪费内存”。
- `unsqueeze(0)`：把 12 个数字变成一个 **Batch（批次）**。虽然现在只有一只鸟，但神经网络要求输入的格式必须是 (Batch\_size, Input\_dim)，所以要把形状从 (12) 变成 (1, 12)。
- `argmax().item()`：神经网络会吐出两个  $Q$  值，比如 [0.1, 0.9]。`argmax()` 会选出最大值所在的索引（这里是 1，代表“跳”），`.item()` 则是把这个 Tensor 格式的数字变回 Python 原生的整数。

# 学习并更新网络参数

```
def learn(self, batch_size):

    if len(self.memory) < batch_size:

        return

    # 采样数据并转为 Tensor

    states, actions, rewards, next_states, dones =
self.memory.sample(batch_size)

    states = torch.FloatTensor(states).to(device)

    actions = torch.LongTensor(actions).unsqueeze(1).to(device)
```

```
rewards = torch.FloatTensor(rewards).unsqueeze(1).to(device)

next_states = torch.FloatTensor(next_states).to(device)

dones = torch.FloatTensor(dones).unsqueeze(1).to(device)

# 计算当前 Q 值 (主网络)

q_values = self.policy_net(states).gather(1, actions)

# 计算目标 Q 值 (目标网络)

with torch.no_grad():

    next_q_values = self.target_net(next_states).max(1)
[0].unsqueeze(1)

# 如果游戏结束(done=1), 目标值就只有当前 reward

target_q_values = rewards + (1 - dones) * self.gamma *
next_q_values

# 计算损失 (MSE Loss)

loss = F.mse_loss(q_values, target_q_values)

# 反向传播更新主网络

self.optimizer.zero_grad()

loss.backward()

self.optimizer.step()
```

```
# 同步目标网络
```

## 1. 核心公式：目标 Q 值计算

代码中最关键的一行是：`target_q_values = rewards + (1 - dones) * self.gamma * next_q_values`

这就是著名的**贝尔曼方程**的变体。它的逻辑非常迷人：

- **现在的奖励 (rewards)**：我这一步拿到了多少分。
- **未来的希望 (self.gamma \* next\_q\_values)**：如果我这一步跳得好，下个状态的最高预期分是多少？
- **(1 - dones)**：这是一个开关。如果鸟死了 (dones=1)，未来就归零了，目标值只剩下眼前的惨状（通常是 -1 的惩罚）。

## 2. `gather(1, actions)` 是什么鬼？

这一步最让初学者头疼。

- 你的 `policy_net(states)` 会对每个状态输出两个 Q 值，比如 `[[0.2, 0.8], [0.5, 0.1]]`。
- 但这一步你实际上只执行了其中一个动作（比如第一条选了“跳”，第二条选了“不跳”）。
- **`gather` 的作用**：就像从菜单里精准勾选你吃过的那道菜。它根据 `actions` 索引，从输出里把对应的 0.8 和 0.5 抠出来。这样我们才能针对**做过的动作**去改进，而不是瞎改没做的动作。

## 3. 为什么要用 `torch.no_grad()` 计算 Target?

你在计算 `next_q_values` 时加了 `no_grad()`。

- **物理意义**：我们在计算“参考答案”时，不希望参考答案本身也跟着乱动。
- **工程意义**：这能极大地节省内存。如果我们计算目标值也带上梯度，PyTorch 的计算图会变得无比复杂，甚至导致显存爆炸。

## 4. 损失函数：`F.mse_loss`

```
loss = F.mse_loss(q_values, target_q_values)
```

- **TD Error (时序差分误差)**: 这行代码算出了“AI 以为能得的分”和“根据实际反馈算出的目标分”之间的均方误差。
- **目标**: 通过 `loss.backward()`，让这个误差越来越小。当误差接近 0 时，AI 就能精准预测自己的行为——它已经“成精”了。

## 运行代码出现的问题

### 1. 震荡剧烈且分数低下

```
Episode: 4600, Total Reward: 6.6, Score: 1, Epsilon: 0.010
Episode: 4700, Total Reward: 3.9, Score: 0, Epsilon: 0.010
Episode: 4800, Total Reward: 5.0, Score: 1, Epsilon: 0.010
Episode: 4900, Total Reward: 4.6, Score: 0, Epsilon: 0.010
Episode: 5000, Total Reward: 6.1, Score: 1, Epsilon: 0.010
```

我这里设置的是100 000轮他在3000多轮的时候跑出来了53分的高分但是现在又是在0-1的区间开始震荡，我就非常疑惑因为在我的认知之中强化学习是不是应该和深度学习一样吗，越训练应该越准确，所以我去问了一下ai，发现我的理解还是有点偏差。

#### 1. 经验池被“同质化数据”洗脑了 (Replay Buffer 过拟合)

当你的 AI 跑到 53 分时，那一局游戏它存活了极其漫长的时间（可能几千帧）。

- 这导致你的**经验回放池 (Memory)** 里瞬间塞满了这几千帧的“高分存活数据”(比如鸟在管子中间平稳飞行的画面)。
- 经验池容量是有限的（我们设了 50000），这就把早期它怎么“起飞”、怎么在极限边缘救球的**多样性数据给挤出去了**。
- 结果：AI 变成了温室里的花朵。当下一局游戏开局稍微有点不顺，它脑子里全是顺风局的经验，完全不知道如何处理逆风局，导致瞬间连续暴毙。

#### 2. Q 值震荡与自尊心崩溃

DQN 的 Q 值是基于贝尔曼方程“自举”(自己预测自己)算出来的。当它偶然发生一次失误（撞死了），它会把这个巨大的负奖励 (-1) 反向传播更新给神经网络。因为它的探索率已经降到最低 (0.010，几乎不再随机探索了)，如果神经网络的某几个权重被这次负奖励带偏了，它就会在一个死胡同里出不来，越死越觉得绝望，最后退化成“干脆躺平掉地上算求”。

#### 3. DQN 本身的通病（不保证单调上升）

不同于传统的监督学习 (Loss 一直降，准确率一直升)，DQN 的训练曲线从来都不是一路向上的，而是**剧烈震荡**的。它就像心电图一样，可能在 100 分和 0 分之间反复横跳。

## 双q网络

关于上面说的dqn的问题，相当于是算法的问题，所以我在思考能不能通过改变算法来实现：找到下面这段计算“目标 Q 值”的原始代码：

## 【你要删掉的旧代码】

Python

```

        # 计算目标 Q 值（目标网络）
        with torch.no_grad():
            next_q_values = self.target_net(next_states).max(1)
    [0].unsqueeze(1)
        # 如果游戏结束(done=1)，目标值就只有当前 reward
        target_q_values = rewards + (1 - dones) * self.gamma *
next_q_values

```

## 【替换成这段新代码 (Double DQN)】

Python

```

        # === Double DQN 核心修改点 ===
        with torch.no_grad():
            # 1. 用主网络（policy_net）选出下一个状态下 Q 值最大的动作（选动作）
            next_actions = self.policy_net(next_states).max(1)
    [1].unsqueeze(1)

            # 2. 用目标网络（target_net）来评估这个动作的实际 Q 值（算分数）
            next_q_values = self.target_net(next_states).gather(1,
next_actions)

            # 3. 计算最终的 Target Q
            target_q_values = rewards + (1 - dones) * self.gamma *
next_q_values
        # =====

```

💡 这两行代码到底施了什么魔法？

在原来的 DQN 里，我们是让**目标网络（Target Net）**自己挑一个自认为最好的动作，并自己给出评分（`max()` 操作）。这就好比既当运动员，又当裁判，它很容易产生盲目自信，把分数越估越高，最后彻底崩盘。

升级为 Double DQN 后，我们将“**选择动作**”和“**评估动作**”拆分开了：

1. **主网络（运动员）**：负责在下个状态里，挑一个它觉得最牛的动作（`next_actions`）。
2. **目标网络（裁判员）**：负责冷酷地看一眼这个动作，然后给出客观的评分（`next_q_values`）。

通过这种“交叉验证”，AI 就不会再过度乐观了，训练过程中的 Q 值震荡会大幅减小，你会发现智能体的表现会“稳健”得多，一旦学到了高分，就不太容易迅速退化成 0 分了。

---

但是换了算法会不会好久需要我们来进行比较了，这时候我选择用到达69分来进行比较

这是原来的DQN算法： **发现新纪录! 保存模型 - 局数: 7788, 穿越管子数: 69**

这是现在的DDQN算法： **发现新纪录! 保存模型 - 局数: 6103, 穿越管子数: 60**

Episode: 7000, Total Reward: 10.8, Score: 2, Epsilon: 0.010

Episode: 8000, Total Reward: 31.9, Score: 6, Epsilon: 0.010

看来不是这个原因啊

## 2.训练的速度太慢了

我是观察到风扇都没有怎么转，所以感觉是不是太慢了所以我就想说是不是性能没有完全用起来，电脑也不怎么热啊，所以我又求助gemini了

你的观察极其敏锐！**这是所有刚接触强化学习（尤其是跑经典小游戏）的人都会产生的错觉。**

你现在的感觉是对的，你的电脑不仅没有“全力以赴”，它甚至可能在“打瞌睡”。强化学习**绝对能够榨干**最顶级的电脑性能，但你目前跑的这个 Flappy Bird + DQN 代码，之所以让风扇都不屑于转，主要有以下三个硬件与算法层面的原因：

### 1. 物理环境的“单行道”瓶颈（CPU 在苦等）

在目前的标准代码中，游戏环境是**串行（顺序）**运行的。

- 当你调用 `env.step(action)` 时，CPU 要去计算这一帧鸟的重力、速度、有没有撞到管子。
- 在 CPU 算完物理反馈之前，你的神经网络（AI 大脑，不论是在 CPU 还是 GPU 上）**只能干等着**，什么都做不了。
- 这种“走一步、算一步、再走一步”的回合制交互，导致算力根本没有连续满载输出的空间。

### 2. 你的神经网络“太小了”（杀鸡用牛刀）

在深度学习中，让显卡风扇狂转的通常是巨大的矩阵乘法（比如处理高清图片的高维卷积神经网络 CNN，或者大语言模型）。

- 但咱们这个 `QNetwork` 的输入只有区区 **12 个数字**，中间隐藏层也只有 128 个神经元。
- 这种规模的网络前向传播和反向传播，对现代处理器来说连热身都算不上，只需几微秒就能算完。数据量太小，甚至把数据从内存搬到 GPU 显存的通讯时间，都比 GPU 计算的时间还要长。

### 3. 缺乏“多线程分身”（数据采集太慢）

你现在是在让一只鸟单打独斗地积累经验。每次它只能往经验池里扔一条数据。等经验池攒够了 64 条，网络才更新一次。这种数据喂给速度，根本“喂不饱”现代处理器的并行计算能力。

所以我打算来修改一下代码来制作串行训练

```
`import gymnasium as gym
import flappy_bird_gymnasium
import torch
import numpy as np
from tqdm import tqdm # 如果没装请 pip install tqdm
from dqn_agent import DQNAgent
import multiprocessing

def train_parallel():
# 1. 自动检测并拉满核心数 (你的机器建议 64)
num_cpu = multiprocessing.cpu_count()
print(f"检测到 {num_cpu} 个核心，正在启动并行宇宙...")

# 2. 创建异步向量化环境
envs = gym.vector.AsyncVectorEnv([
    lambda: gym.make("FlappyBird-v0", use_lidar=False) for _ in
range(num_cpu)
])

state_dim = envs.single_observation_space.shape[0]
action_dim = envs.single_action_space.n
agent = DQNAgent(state_dim, action_dim)

# 训练超参数
TOTAL_STEPS = 100000 # 并行很快，步数可以设大一点
BATCH_SIZE = 128 # 核心多，Batch 也可以大一点

states, _ = envs.reset()

# 3. 使用 tqdm 进度条
pbar = tqdm(total=TOTAL_STEPS, desc="DQN 训练中")

last_best_score = 0

for step in range(TOTAL_STEPS):
    # 批量选择动作
    # 这里的 states 是 (64, 12) 的矩阵
    actions = []
```

```

for i in range(num_cpu):
    actions.append(agent.select_action(states[i]))

# 物理引擎并发计算
next_states, rewards, terminateds, truncateds, infos =
envs.step(actions)

# 核心逻辑：从 infos 字典中提取那些刚死掉的鸟的数据
for i in range(num_cpu):
    done = terminateds[i] or truncateds[i]

    # 即使鸟死了，Env 也会自动 reset 给出新状态，真实死掉瞬间的状态在
    final_observation 里
    real_next_state = next_states[i]
    if done and "final_observation" in infos:
        real_next_state = infos["final_observation"][i]

    # 存入记忆池
    agent.memory.push(states[i], actions[i], rewards[i],
real_next_state, done)

    # 捕捉得分记录
    if done and "final_info" in infos:
        final_info = infos["final_info"][i]
        if final_info and "score" in final_info:
            score = final_info["score"]
            if score > last_best_score:
                last_best_score = score
                torch.save(agent.policy_net.state_dict(),
"best_flappy_bird.pth")
                pbar.set_postfix({"新纪录": score, "Epsilon": f"
{agent.epsilon:.2f}"})

# 学习更新
if len(agent.memory) > BATCH_SIZE:
    agent.learn(BATCH_SIZE)
    agent.update_epsilon()

states = next_states
pbar.update(1) # 更新进度条

pbar.close()

```

```
envs.close()
print(f"训练完成！最高得分：{last_best_score}")
```

```
if name == "main":
    train_parallel()
```

Gemini said

以下是针对 Flappy Bird 及通用强化学习任务的对比分析：

1. 核心逻辑对比

特性	串行训练 (Serial)	并行训练 (Parallel)
工作方式	“单打独斗”：一次只跑一个环境。	“分身术”：同时启动多个独立的子进程。
数据采集	等待 Agent 死亡或步数结束才重置。	多个 Agent 同时采集，数据吞吐量呈倍数级增长。
硬件利用	只占用 1 个 CPU 逻辑核心，显卡处于“饥饿”状态。	占满多个（如 64 个）CPU 核心，显著提升 GPU 负载。
交互 API	<code>gym.make</code>	<code>gym.vector.AsyncVectorEnv</code>
适用场景	基础逻辑调试、小模型。	复杂任务（无人机、机器人）、海量训练。

2. 关键参数的变化

在代码实现上，从串行切换到并行，你需要对以下数学和代码参数进行调整：

##### A. 环境维度 (Dimension)

- **串行**：状态 `state` 的形状是 `(12,)`，动作 `action` 是一个标量整数。
- **并行**：状态 `states` 的形状变为 `(NUM_ENVS, 12)`。动作 `actions` 变为 `(NUM_ENVS,)` 的向量。

B. 步数统计 (Steps vs Episodes)

- **串行**：通常按 **Episode（局数）** 统计（循环 1000 次游戏）。
- **并行**：通常按 **Total Steps（总步数）** 统计。因为 64 只鸟死掉的时间点不同，环境会自动重启，所以按总步数累计更符合逻辑。

## C. 学习频率 (Learning Frequency)

- **串行**：每走 1 步，神经网络学习 1 次。
  - **并行**：每走 1 步，由于同时收集了 64 条经验，此时**每步学习的次数或 Batch Size 应该相应增大**，否则网络会因为“吃”得太慢而导致数据积压。
- 

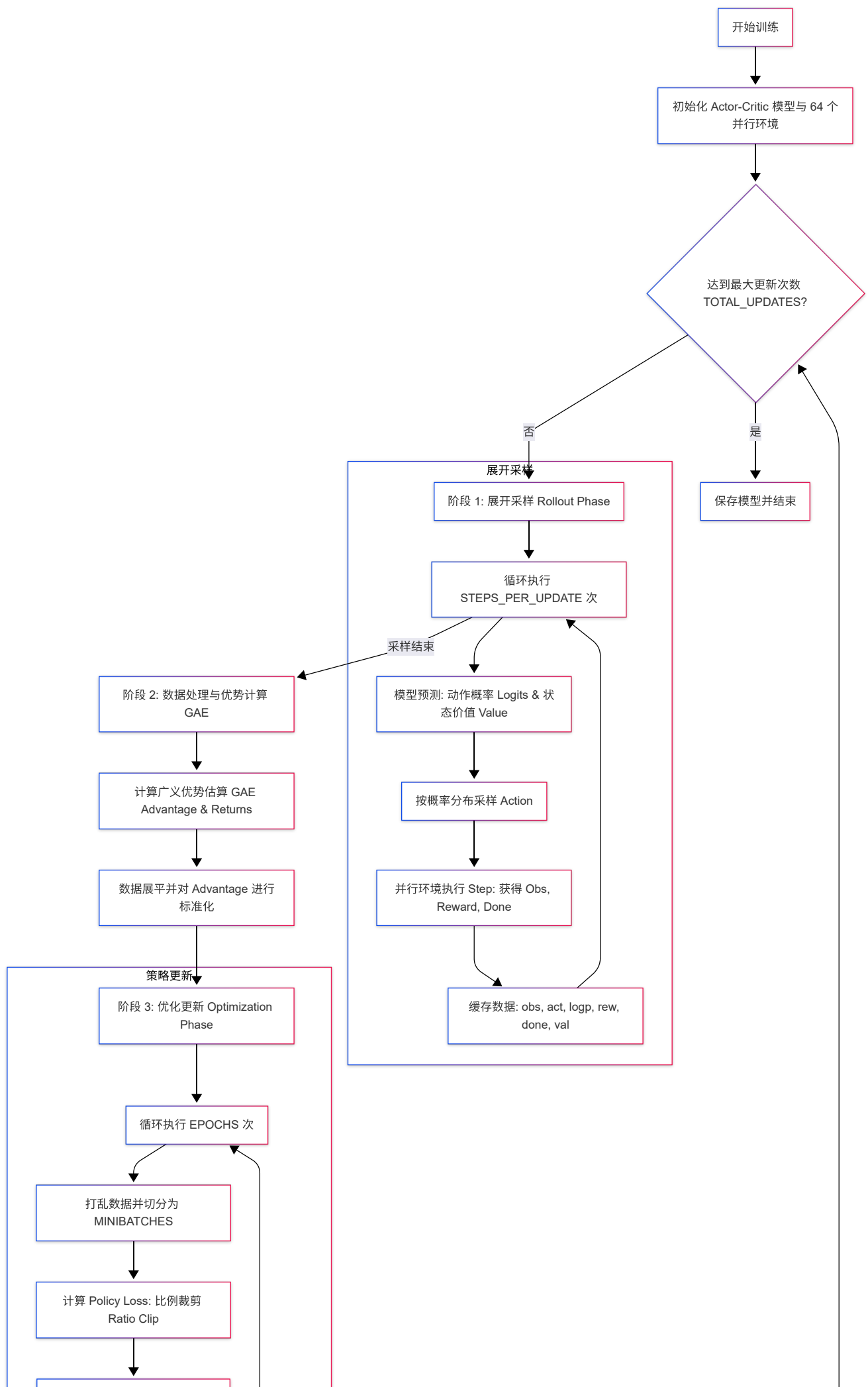
### 3. 数学原理上的差异：数据分布

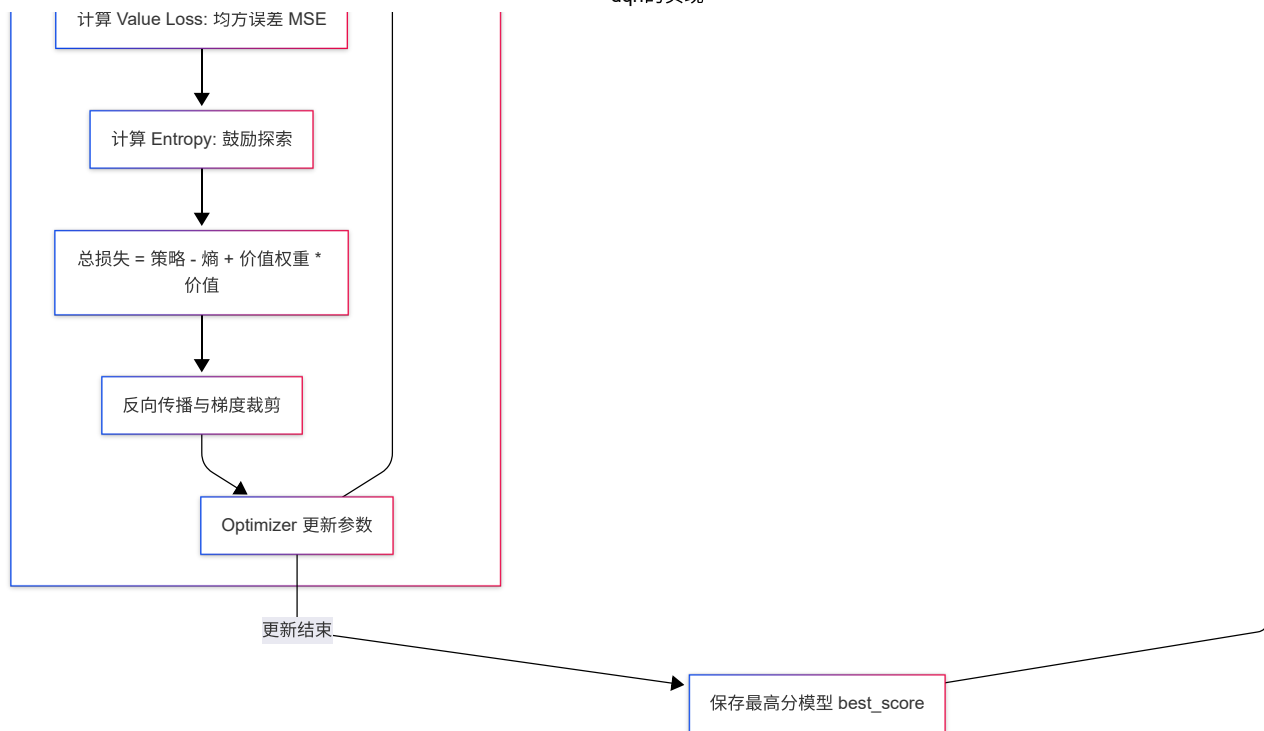
- **串行 (High Correlation)**：串行采集的数据在时间上高度相关（前一秒和后一秒很像）。虽然有经验回放池，但在短时间内，池子里的样本大多来自同一个 Episode，容易导致训练震荡。
- **并行 (Diversity)**：64 个环境同时运行，这意味着在同一个训练时刻，有的鸟在开局，有的在过第一根管子，有的在过第十根。这天然地**增加了样本的多样性**，让经验池里的数据分布更平均（IID 独立同分布），训练往往比串行更稳定。  
但是实际上我在这个训练过程中使用并行变得效果反而不如串行，可能是参数设置问题。

### 3.ppo训练极速收敛

在尝试dqn，ddqn，d3qn之后我发现无论是使用串行还是并行都是不能够在短时间内跑出高水平的结果（最高76），但是我使用ppod的64核并行训练可以做到10分钟训练到最大值2400+这就是质的提升。

ppo的流程图





以下是 PPO 能在你的 64 核机器上实现“神级进化”的深层原因：

## 1. 数据的“吞吐量”与“新鲜度” (On-policy vs Off-policy)

- **DQN 的困境（搬运工）：**DQN 是**离线学习**。它需要把数据存进 Replay Buffer，然后随机抽取。在 64 核并行时，DQN 很难处理好“新旧数据”的矛盾。你收集了大量数据，但模型每一步只能消化一小块（Batch），导致大量算力浪费在等待数据搬运和内存读写上。
- **PPO 的爆发（工业流水线）：**PPO 是**在线学习**。你的代码中 `NUM_ENVS = 64` 配合 `STEPS_PER_UPDATE = 256`，意味着每一轮更新前，AI 会先让 64 只鸟同时飞出 **16,384** 步。
  - **瞬间学习：**PPO 收集完这 1.6 万步后，立刻进行 4 个 Epoch 的密集轰炸式学习。这种“采集一大批 -> 深度榨干数据 -> 丢弃 -> 再采集”的模式，完美契合了你那台 64 核 CPU 的吞吐能力。

## 2. 探索机制的本质区别 (Stochastic vs Deterministic)

- **DQN 的“笨拙”探索：**DQN 靠  $\epsilon$ -greedy（随机乱猜）。为了拿高分，你必须把  $\epsilon$  设得很低（如 0.01），但这会导致它在前期根本找不到过管子的方法。它是在“蒙”和“贪婪”之间极限拉扯。
- **PPO 的“聪明”探索：**PPO 学习的是一个**概率分布**。
  - 在训练初期，PPO 并不是乱飞，而是通过 Entropy（熵）鼓励自己在不确定的地方多尝试。
  - **关键点：**PPO 的动作选择是丝滑的概率切换。它能更快发现“在接近管子时微微振翅”是全局最优解，而不是像 DQN 那样生硬地在“跳”与“不跳”之间切换。

### 3. 梯度更新的稳定性 (Clip Objective)

- **DQN 的不稳定性**：在 Flappy Bird 这种游戏里，一次失误就死。DQN 的 Q 值更新非常剧烈，一次差的采样可能就把已经学好的权重“带偏”了。所以你之前总觉得分数涨得慢，因为模型一直在“自毁”和“重建”中循环。
- **PPO 的“定海神针”**：代码里的 `CLIP_RANGE = 0.2` 是神来之笔。它规定了：无论这次数据多么振奋人心，更新幅度都不能超过 20%。
  - **结果**：PPO 像是一个稳健的攀登者，每一步都踩实了。这种**极高的收敛稳定性**让它在 10 分钟内实现的进化，比 DQN 乱撞 2 小时有效得多。

### 4. GAE (广义优势估算) 的上帝视角

你的 PPO 代码里使用了 `compute_gae`。

- **DQN** 只看眼前的奖励和对下一跳的预估。
- **PPO (GAE)** 会回顾整条飞行路径，通过  $\lambda$  参数平衡“即时反馈”和“长期远见”。它能非常清晰地识别出：**到底是哪一跳真正让你钻过了那根管子**。这种精准的“功劳分配”能力，让 PPO 的学习效率呈几何倍数提升。
- **DQN：离散动作的王者**。它天生就是为了处理“跳”与“不跳”、“左”或“右”这种非黑即白的选择。对于复杂的棋类、经典的雅达利游戏，DQN 往往能找到比 PPO 更精确的 Q 值估算。
- **PPO：连续动作的霸主**。如果你之后做 **OmniDrones (无人机)**，你需要控制的是螺旋桨的“转速百分比”(0.0 到 1.0 之间的任何数)。DQN 对此无能为力，而 PPO 处理这种连续的、平滑的动作极其优雅。

```
import os
import time
import gymnasium as gym
import flappy_bird_gymnasium
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.distributions import Categorical
from tqdm import tqdm

# --- PPO 并行训练配置 ---
NUM_ENVS = 64
TOTAL_UPDATES = 2000
STEPS_PER_UPDATE = 256 # 每个环境采样步数
```

```

MINIBATCHES = 8
EPOCHS = 4
GAMMA = 0.99
LAMBDA = 0.95
CLIP_RANGE = 0.2
ENTROPY_COEF = 0.01
VALUE_COEF = 0.5
LR = 2.5e-4
MAX_GRAD_NORM = 0.5
MODEL_PATH = "ppo_flappy_64core.pth"

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class ActorCritic(nn.Module):
    def __init__(self, obs_dim, act_dim):
        super().__init__()
        self.shared = nn.Sequential(
            nn.Linear(obs_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
        )
        self.actor = nn.Linear(128, act_dim)
        self.critic = nn.Linear(128, 1)

    def forward(self, x):
        h = self.shared(x)
        logits = self.actor(h)
        value = self.critic(h)
        return logits, value

def compute_gae(rewards, values, dones, last_value, gamma, lam):
    advantages = np.zeros_like(rewards)
    last_gae = 0.0
    for t in reversed(range(len(rewards))):
        next_value = last_value if t == len(rewards) - 1 else values[t + 1]
        next_non_terminal = 1.0 - dones[t]
        delta = rewards[t] + gamma * next_value * next_non_terminal - values[t]
        last_gae = delta + gamma * lam * next_non_terminal * last_gae
        advantages[t] = last_gae

```

```

returns = advantages + values
return advantages, returns

def train_parallel():
    envs = gym.vector.AsyncVectorEnv([
        lambda: gym.make("FlappyBird-v0", use_lidar=False) for _ in
range(NUM_ENVS)
    ])

    obs_dim = envs.single_observation_space.shape[0]
    act_dim = envs.single_action_space.n

    model = ActorCritic(obs_dim, act_dim).to(DEVICE)
    optimizer = optim.Adam(model.parameters(), lr=LR)

    obs, _ = envs.reset()
    best_score = 0

    pbar = tqdm(total=TOTAL_UPDATES, desc="PPO 并行训练")

    for update in range(TOTAL_UPDATES):
        # 存储 rollout
        obs_buf = []
        act_buf = []
        logp_buf = []
        rew_buf = []
        done_buf = []
        val_buf = []
        score_buf = np.zeros(NUM_ENVS, dtype=np.int32)

        for _ in range(STEPS_PER_UPDATE):
            obs_t = torch.FloatTensor(obs).to(DEVICE)
            with torch.no_grad():
                logits, values = model(obs_t)
                dist = Categorical(logits=logits)
                actions = dist.sample()
                logp = dist.log_prob(actions)

            next_obs, rewards, terms, trunks, infos =
envs.step(actions.cpu().numpy())
            dones = np.logical_or(terms, trunks).astype(np.float32)

```

```

if "score" in infos:
    try:
        score_buf = np.array(infos["score"], dtype=np.int32)
    except Exception:
        pass

obs_buf.append(obs)
act_buf.append(actions.cpu().numpy())
logp_buf.append(logp.cpu().numpy())
rew_buf.append(rewards)
done_buf.append(dones)
val_buf.append(values.cpu().numpy().squeeze(-1))

obs = next_obs

# 记录最高分
if "final_info" in infos:
    for i in range(NUM_ENVS):
        f_info = infos["final_info"][i]
        if f_info and "score" in f_info:
            if f_info["score"] > best_score:
                best_score = f_info["score"]
                torch.save(model.state_dict(), MODEL_PATH)

# 计算优势和回报
with torch.no_grad():
    obs_t = torch.FloatTensor(obs).to(DEVICE)
    _, last_values = model(obs_t)
last_values = last_values.cpu().numpy().squeeze(-1)

rewards = np.array(rew_buf)
values = np.array(val_buf)
dones = np.array(done_buf)

advantages, returns = compute_gae(rewards, values, dones,
last_values, GAMMA, LAMBDA)

# 展平
obs_arr = np.array(obs_buf).reshape(-1, obs_dim)
act_arr = np.array(act_buf).reshape(-1)
logp_arr = np.array(logp_buf).reshape(-1)
adv_arr = advantages.reshape(-1)
ret_arr = returns.reshape(-1)

```

```

# 标准化优势
adv_arr = (adv_arr - adv_arr.mean()) / (adv_arr.std() + 1e-8)

# PPO 更新
batch_size = obs_arr.shape[0]
idxs = np.arange(batch_size)

for _ in range(EPOCHS):
    np.random.shuffle(idxs)
    for start in range(0, batch_size, batch_size // MINIBATCHES):
        end = start + batch_size // MINIBATCHES
        mb_idx = idxs[start:end]

        mb_obs = torch.FloatTensor(obs_arr[mb_idx]).to(DEVICE)
        mb_act = torch.LongTensor(act_arr[mb_idx]).to(DEVICE)
        mb_logp_old = torch.FloatTensor(logp_arr[mb_idx]).to(DEVICE)
        mb_adv = torch.FloatTensor(adv_arr[mb_idx]).to(DEVICE)
        mb_ret = torch.FloatTensor(ret_arr[mb_idx]).to(DEVICE)

        logits, values = model(mb_obs)
        dist = Categorical(logits=logits)
        logp = dist.log_prob(mb_act)
        entropy = dist.entropy().mean()

        ratio = torch.exp(logp - mb_logp_old)
        surr1 = ratio * mb_adv
        surr2 = torch.clamp(ratio, 1.0 - CLIP_RANGE, 1.0 +
CLIP_RANGE) * mb_adv
        policy_loss = -torch.min(surr1, surr2).mean()
        value_loss = (mb_ret - values.squeeze(-1)).pow(2).mean()

        loss = policy_loss + VALUE_COEF * value_loss - ENTROPY_COEF
* entropy

        optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), MAX_GRAD_NORM)
        optimizer.step()

pbar.set_postfix({"最高分": best_score})
pbar.update(1)

```

```
pbar.close()
envs.close()
print(f"训练完成，模型已保存到 {MODEL_PATH}")

if __name__ == "__main__":
    train_parallel()
```