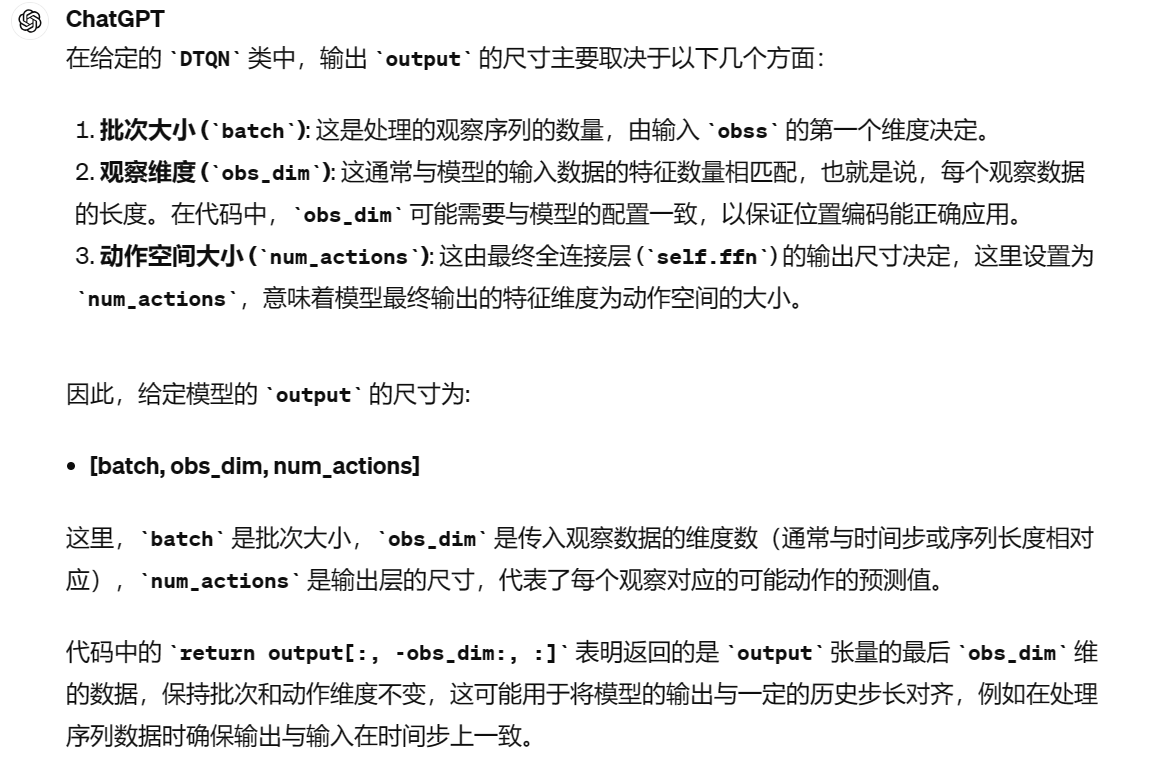
* **obs\_dim**: 观察向量的长度。int
* **num\_actions**: 环境中可能的动作数量，即动作空间的大小 int
* **action\_dim**: 嵌入向量的维度，即你希望每个动作映射到高维空间中的点的维度。int
* **outer\_embed\_size**: 结果嵌入向量的长度。这个参数指定了将观察向量映射到的目标空间的维度。int
* **embed\_size**: 层的维度，即输入和输出张量的特征维数。int
* **context\_len**: 序列的长度，即序列中元素的数量. int
* **inner\_embed\_size**: 网络的维度，原始Transformer中称为d\_k。int
* **num\_heads**: 在多头注意力中使用的头数。int
* **num\_layers**: 使用的Transformer块的数量。int
* **history\_len**: 可以接收的观察序列的最大长度。int
* **dropout**: 丢弃率，默认为**0.0**，防止过拟合。float
* **gate**: 在注意力和前馈子模块后使用哪一层（选项：**res**或**gru**），默认为**res**。**attn\_gate**：注意力子模块之后的组合层。**mlp\_gate：**前馈子模块之后的组合层
* **identity**: 是否使用恒等映射重排，默认为**False**。
* **pos**: 使用哪种类型的位置编码。**0**表示不使用位置编码，**1**表示使用学习到的位置编码，**sin**表示使用正弦位置编码，默认为**1**。
* **discrete**: 环境是否有离散观察，默认为**False**。
* **vocab\_sizes**: 仅限离散环境。表示环境中的观察数量。如果环境有多个观察维度，每个维度有不同数量的观察，可以提供一个向量，默认为**None**。





|  |  |  |  |
| --- | --- | --- | --- |
| **参数名** | **类型** | **默认值** | **描述** |
| **project-name** | 字符串 | **"DTQN-test"** | 用于 **wandb** 的项目名称或本地日志的目录名，存储结果。 |
| **disable-wandb** | 标志 | **FALSE** | 使用此标志来进行本地日志记录，而不是使用 **wandb**。 |
| **time-limit** | 浮点数 | **None** | 允许的作业时间限制，对于某些集群作业（如slurm）有用。 |
| **model** | 字符串 | **"DTQN"** | 要使用的网络模型。 |
| **envs** | 字符串列表 | **"DiscreteCarFlag-v0"** | 使用的环境域。可以提供多个环境，但它们必须有相同的观察和动作空间。 |
| **num-steps** | 整数 | **2\_000\_000** | 训练代理的步数。 |
| **tuf** | 整数 | **10\_000** | 每多少步进行一次（硬）目标网络更新。 |
| **lr** | 浮点数 | **3.00E-04** | 优化器的学习率。 |
| **batch** | 整数 | **32** | 批次大小。 |
| **buf-size** | 整数 | **500\_000** | 回放缓冲区中存储的时间步数。 |
| **eval-frequency** | 整数 | **5\_000** | 代理评估之间的训练时间步数。 |
| **eval-episodes** | 整数 | **10** | 每个评估期的剧集数。 |
| **device** | 字符串 | **"cuda"** | 使用的 PyTorch 设备。 |
| **context** | 整数 | **50** | 训练网络时使用的上下文长度。 |
| **obs-embed** | 整数 | **8** | 仅对于离散观察域。给每个观察分配的特征数。 |
| **a-embed** | 整数 | **0** | 给每个动作分配的特征数。0 值将阻止策略使用前一个动作。 |
| **in-embed** | 整数 | **128** | 网络的维度，在 transformer 中被称为 **d\_model**。 |
| **max-episode-steps** | 整数 | **-1** | 环境中允许的最大步数。 |
| **seed** | 整数 | **1** | 使用的随机种子。 |
| **save-policy** | 标志 | **FALSE** | 使用此标志保存策略，以便以后加载用于渲染。 |
| **verbose** | 标志 | **FALSE** | 将评估结果打印到控制台。 |
| **render** | 标志 | **FALSE** | 享受模式（注意：必须有已训练的策略保存）。 |
| **history** | 整数 | **50** | 用于训练每个上下文的（中间）Q值数量。 |
| **heads** | 整数 | **8** | transformer 的头数。 |
| **layers** | 整数 | **2** | transformer 的块数。 |
| **dropout** | 浮点数 | **0** | Dropout 概率。 |
| **discount** | 浮点数 | **0.99** | 折扣因子。 |
| **gate** | 字符串 | **"res"** | 使用的合并步骤。 |
| **slurm-job-id** | str | 0 |  |
| **identity** | flag | FALSE | 是否使用身份映射重排序。 |
| **pos** | str | "learned" | 要使用的位置编码的类型。 |
| **bag-size** | int | 0 | 持久记忆袋的大小。 |
| **network\_factory** | Callable[[], Module] | 无 | 用于创建神经网络模型的函数或可调用对象。 |
| **buffer\_size** | int | 无 | 经验回放缓冲区的大小。 |
| **device** | torch.device | 无 | 指定模型和数据应在哪个设备上运行（如CPU或GPU）。 |
| **env\_obs\_length** | int | 12 | 环境观察的维度或长度。 |
| **max\_env\_steps** | int | 50 | 每个episode的最大环境步数。 |
| **obs\_mask** | Union[int, float] | 无 | 观察遮罩值，用于处理部分观察问题。 |
| **num\_actions** | int | 6 | 可行动作的数量。 |
| **grad\_norm\_clip** | float | 1 | 梯度裁剪的阈值。 |
| **target\_update\_frequency** | int | 10\_000 | 目标网络更新频率。 |

Number of steps to train the agent：num-steps（int）（2000000）

num-steps 整数 2\_000\_000 训练代理的步数。

tuf 整数 10\_000 #每多少步进行一次（硬）目标网络更新。

lr 浮点数 3.00E-04 #优化器的学习率。

batch 整数 32 批次大小。

buf-size 整数 500\_000 回放缓冲区中存储的时间步数。

context 整数 50 训练网络时使用的上下文长度。

in-embed 整数 128 网络的维度，在 transformer 中被称为 d\_model。

history 整数 50 用于训练每个上下文的（中间）Q值数量。

heads 整数 8 transformer 的头数。

layers 整数 2 transformer 的块数。

dropout 浮点数 0 Dropout 概率。

discount 浮点数 0.99 折扣因子。

gate 字符串 "res" 使用的合并步骤。

bag-size int 0 持久记忆袋的大小。

buffer\_size int 无 经验回放缓冲区的大小。

env\_obs\_length int 12 环境观察的维度或长度。

max\_env\_steps int 50 每个episode的最大环境步数。

obs\_mask Union[int, float] 无 观察遮罩值，用于处理部分观察问题。

num\_actions int 6 可行动作的数量。

grad\_norm\_clip float 1 梯度裁剪的阈值。

target\_update\_frequency int 10\_000 目标网络更新频率。

让我们通过一个简化的例子来解释DTQN中bag的概念和作用：

假设有一个强化学习任务，代理（Agent）需要在一个迷宫中找到出口。这个迷宫有一个特殊的规则：某些门只在代理已经获取了特定的钥匙后才能打开。这些钥匙和门分布在迷宫的不同位置，代理需要记住哪些钥匙已经拾取过，并且在遇到相应的门时使用它们。

在这个任务中，代理的观察可能包括它当前所在位置的周围环境（例如，前、后、左、右的墙壁和门的存在）。然而，这些即时的观察信息并不包含关于已经拾取过的钥匙的信息，这对于成功完成任务是至关重要的。

这就是bag发挥作用的地方：

* **记忆重要信息**：每当代理拾取一个钥匙时，这个钥匙的信息（比如钥匙的类型或拾取的位置）可以被存储到bag中。这样，即使在多步之后，当代理面对一个需要特定钥匙才能打开的门时，它仍然能够“记得”自己已经拥有了这把钥匙。
* **提供决策支持**：在做出决策时（例如，决定下一步行动），代理不仅考虑当前的观察，还能够访问bag中存储的信息。通过结合这些信息，代理可能认识到当前面对的门可以用它之前拾取的某把钥匙打开，即使在当前观察中没有直接的提示。
* **动态信息管理**：如果迷宫很大，代理可能会遇到许多不同的钥匙和门。bag允许代理动态地管理这些信息，例如，当一个钥匙已经被使用后，相关信息可能不再重要，可以从bag中移除，为新信息腾出空间。

通过这个例子，我们可以看到，bag作为一种记忆机制，使得代理能够在面对需要长期记忆和信息管理的复杂任务时，表现得更加有效。这对于那些简单的基于当前观察的策略难以解决的问题尤为重要，因为它提供了一种机制来维持和利用关于环境的关键历史信息。

network\_factory = make\_dtqn(MODEL\_MAP[model\_str])

def make\_dtqn(network\_cls):

"""Creates DTQN"""

return lambda: network\_cls(

env\_obs\_length,

num\_actions,

embed\_per\_obs\_dim,

action\_dim,

inner\_embed,

num\_heads,

num\_layers,

context\_len,

dropout=dropout,

gate=gate,

identity=identity,

pos=pos,

discrete=is\_discrete\_env,

vocab\_sizes=obs\_vocab\_size,

target\_update\_frequency=target\_update\_frequency,

bag\_size=bag\_size,

).to(device)

MODEL\_MAP = {

"DTQN": DTQN,

"DTQN-bag": DTQN,

"ADRQN": ADRQN,

"DRQN": DRQN,

"DARQN": DARQN,

"DQN": DQN,

}

self.policy\_network = network\_factory()

要存储观察之外的额外信息，如“钥匙能否开门”这类状态信息，你需要扩展背包（Bag）机制以支持这种类型的数据。具体来说，你可以在背包数据结构中添加新的字段来存储这些额外的信息，同时修改相关的方法以处理这些新数据。以下是一种可能的实现方式：

**步骤1：扩展背包数据结构**

首先，假设“钥匙能否开门”的信息可以被表示为一个布尔值，你需要在背包（Bag）类中添加一个新的数组来存储这个信息。例如：

pythonCopy code

解释

class Bag:

def \_\_init\_\_(self, size, obs\_length, obs\_mask):

self.size = size

self.obs\_length = obs\_length

self.obs\_mask = obs\_mask

self.obss = np.full((size, obs\_length), obs\_mask)

self.actions = np.zeros((size, 1), dtype=np.uint8)

# 新增：存储钥匙信息的数组

self.key\_info = np.zeros((size, 1), dtype=np.bool\_)

**步骤2：修改存储方法以包括新信息**

接下来，你需要修改或添加方法以存储和处理新的“钥匙信息”。如果有一个特定的方法来存储背包中的数据，确保它也处理了**key\_info**。例如，如果你有一个方法来添加一个观察和动作到背包，你也应该让它接收钥匙信息作为参数：

pythonCopy code

解释

def add\_to\_bag(self, obs, action, key\_can\_open\_door):

# 这里是添加到背包的逻辑，以下为伪代码

idx = find\_storage\_index() # 找到存储位置的逻辑

self.obss[idx] = obs

self.actions[idx] = action

self.key\_info[idx] = key\_can\_open\_door

**步骤3：在数据抽样时考虑新信息**

当你从背包中抽样数据供训练使用时，也需要抽样新添加的“钥匙信息”。修改**sample\_with\_bag**方法或类似的抽样方法，以返回这个额外的信息：

pythonCopy code

解释

def sample\_with\_bag(self, batch\_size):

# 原有的抽样逻辑

...

# 新增：抽样钥匙信息

bag\_key\_info = self.key\_info[some\_indices\_based\_on\_your\_logic]

...

return (obss, actions, ..., bag\_key\_info)

**步骤4：在训练和决策中使用新信息**

最后，确保在模型的训练和决策逻辑中使用这些额外的信息。这可能意味着你需要调整网络架构来处理额外的输入，或者在决策时考虑这些额外的状态信息。

通过上述步骤，你可以在背包机制中存储和使用除观察和动作以外的额外信息，如钥匙能否开门的状态，从而增强模型的决策能力。这种方法提供了灵活性，允许你根据特定任务的需要存储和利用各种类型的信息。

**ReplayBuffer**

**ReplayBuffer** 类是用于强化学习中的经验回放机制的数据结构，它允许存储和随机访问代理与环境交互的历史数据。这有助于在训练过程中从先前的经验中学习，并提高算法的稳定性和性能。

初始化：replay buffer大小：buffer\_size: int,

观测维度：obs\_dim: Union[int, Tuple],

填充观测数据的掩码值：obs\_mask: int,

Episode的最大长度：max\_episode\_length: int,

每个episode的长度：**episode\_lengths**

每次抽取的连续序列长度：context\_len: Optional[int] = 1,

总episode数：buffer\_size/ max\_episode\_length

存储的数据：

观测数据 (obss)：

存储在一个形状为 [max\_size, max\_episode\_steps + 1, env\_obs\_length] 的数组中。这里 max\_episode\_steps + 1 是为了容纳每个episode的起始和最终观测，+1 是为了确保有足够的空间来存储每个episode的第一个和最后一个观测。

动作 (actions)、奖励 (rewards)、完成标志 (dones)：

actions ， rewards 和dones分别存储在形状为 [max\_size, max\_episode\_steps + 1, 1] 和 [max\_size, max\_episode\_steps, 1] 的数组中。

主要函数

**1.store\_obs()**：

* + 在每个episode开始时调用，用于存储第一个观测。
  + 调用**cleanse\_episode()**以清理当前episode的数据，确保不会有来自上一个episode的数据干扰。

**2.store()**：

* + 存储给定时间步的观测、动作、奖励和结束状态。
  + 更新相应的存储数组，并递增位置索引，准备下一个时间步的数据存储。
  + 动态管理当前episode的长度和在buffer中的位置。

1. **can\_sample()**：
   * 判断是否有足够的数据进行批量采样。确保存储的episode数量大于请求的batch\_size。
2. **flush()**：
   * 当一个episode完全存储后调用，准备开始新的episode的数据存储。
   * 重置在episode中的观测索引，以便从头开始存储新的episode。
3. **cleanse\_episode()**：
   * 清理指定episode的数据，包括观测、动作、奖励和结束标志，以避免旧数据的影响。
4. **sample()** 和 **sample\_with\_bag()**：
   * 从buffer中随机抽取bstch\_size个episode的context\_len长度的数据，用于训练。
   * **sample\_with\_bag()**还考虑了从一个额外的“bag”中采样数据，用于特定的训练场景，如持久记忆或特定策略的实施。

**报告：Agent 类的概述和工作流程**

概述

**Agent** 类负责管理一个强化学习代理的整个生命周期，包括与环境的交互、决策制定、学习过程和内存管理。这个类封装了用于训练和评估智能体的逻辑，同时处理经验回放缓冲区和一个可选的额外信息存储“背包”。

输入与输出

* **输入**:
  + 环境提供的观测值(**obs**)
  + 代理选择的动作(**action**)
  + 从环境获得的奖励(**reward**)
  + 表示当前状态是否为终端状态的标志(**done**)
* **输出**:
  + 动作选择(**action**)
  + 训练更新后的网络参数
  + 可选的调试或状态信息

主要组件与工作流程

1. **初始化 (\_\_init\_\_)**:
   * 设置经验回放缓冲区大小、设备类型、观测维度等基本配置。
   * 初始化训练和评估模式下的上下文和背包。
   * 配置深度学习网络、优化器和损失计算策略。
2. **重置上下文 (context\_reset)**:
   * 用于在每个新的环境交互开始时重置代理的上下文和背包状态。
   * 存储初始观测到经验回放缓冲区。
   * 重置代理的内部状态，如历史记录和内部计数器。
3. **观测处理 (observe)**:
   * 处理来自环境的新观测、动作、奖励和完成状态。
   * 更新上下文和背包，如果必要，将被驱逐的观测和动作尝试存储到背包中。
   * 在训练模式下，将新的转换存储到经验回放缓冲区。
4. **动作决策 (get\_action)**:
   * 基于当前策略网络，结合背包和当前上下文，决定下一步的动作。
   * 实现ε-贪心策略，以一定概率随机选择动作，以促进探索。
5. **训练 (train)**:
   * 从经验回放缓冲区中抽取一批数据。
   * 使用深度学习模型对当前策略进行评估和更新。
   * 根据损失函数更新网络参数，并定期同步目标网络。
6. **网络更新 (target\_update)**:
   * 定期将学习到的网络参数复制到目标网络，以稳定学习过程。

概述

**evaluate** 函数负责评估一个强化学习代理在一个指定环境中表现的效率。该函数使用贪心策略（greedy policy），在指定的测试环境中运行一定数量的独立测试轮次（episodes），以评估代理在控制环境中的性能。

输入与输出

* **输入**:
  + **agent**: 需要评估的代理对象。
  + **eval\_env**: 用于评估的环境，应兼容 OpenAI Gym 接口。
  + **eval\_episodes**: 指定需要运行多少个完整的测试轮次。
  + **render**: 可选的布尔值，指定是否在评估期间渲染环境的视觉输出。
* **输出**:
  + **mean\_success**: 轮次中成功的平均比例。
  + **mean\_return**: 每轮次平均获得的总回报。
  + **mean\_episode\_length**: 每轮次平均持续的步数。

工作流程

1. **初始化评估环境**:
   * 将代理的网络设置为评估模式，这通常会禁用如 dropout 等对训练有利的特性。
   * 初始化统计指标：总回报、成功次数和总步数。
2. **执行多轮评估**:
   * 对于每个评估轮次，重置代理的上下文和环境的状态。
   * 根据贪心策略（ε=0）选择动作，即完全依赖当前策略网络的推荐而不进行探索。
   * 若启用渲染，则在每个时间步展示环境的状态。
3. **收集和记录结果**:
   * 累计每轮的回报，并在轮次结束时更新总回报和总步数。
   * 检查每轮的成功标志（如果有的话），以统计成功轮次的数量。
4. **计算和返回统计结果**:
   * 计算在所有评估轮次中的平均成功率、平均回报和平均轮次长度。
   * 将代理的网络配置恢复到训练模式，准备进行进一步的训练或其他操作。

在这段代码中，`test` 函数的目的是评估给定智能体的性能，通过计算一系列预设数量的情节（episodes）中累积的奖励来衡量。这个函数旨在提供一种方式来评估训练好的智能体或加载已有的模型文件，并通过实际与环境的交互来测试其性能。

### 输入参数:

- `model\_file`: 可选参数，指定一个预训练模型的文件路径。如果提供，函数会从这个文件中加载模型。

- `no`: 整数，指定要运行的情节数量，默认为20。

- `stat`: 布尔值，如果为True，则函数会返回每个情节的奖励列表，以及总累积奖励。

### 输出:

- 如果 `stat` 为 True，返回一个元组 `(cum\_reward, reward\_list)`，其中 `cum\_reward` 是所有情节的总累积奖励，`reward\_list` 是每个情节的奖励列表。

- 如果 `stat` 为 False，仅返回 `cum\_reward`。

### 工作流程:

1. \*\*模型加载\*\*：如果提供了 `model\_file`，函数会从这个文件加载模型。

2. \*\*环境交互\*\*：函数初始化一系列变量来追踪奖励，并开始对指定数量的情节进行迭代。

3. \*\*情节迭代\*\*：在每个情节中，智能体从环境的初始状态开始，不断进行以下步骤直到情节结束：

- 获取当前状态的观察值。

- 使用贪婪策略选择动作，该策略基于当前状态的观察值通过智能体的Q网络计算得出。

- 执行选定的动作，接收环境的下一个状态、奖励和是否达到终止状态的标志。

- 累积奖励到总奖励和情节奖励中。

4. \*\*累积奖励\*\*：记录每个情节的奖励，并添加到总奖励中。

5. \*\*返回结果\*\*：根据 `stat` 参数的值返回累积奖励和/或每个情节的奖励列表。

### 函数的作用：

- 评估已训练智能体的性能，通过与环境的实际交互来检验其决策能力。

- 提供了一种方式来查看智能体在多个独立情节中的行为表现，帮助理解其学习效果和稳定性。

- 允许通过加载预训练的模型来快速验证不同训练策略的效果。

这个函数对于实际部署和测试RL模型是非常有用的，特别是在调试和模型选择阶段。

概述

**train** 函数主要负责训练强化学习代理以在特定环境下进行决策优化。该函数结合了训练和定期评估，以追踪学习过程中的性能变化。通过迭代更新，代理学习最大化其从环境中获取的累积奖励。

输入与输出

* **输入**:
  + **agent**: 要训练的代理对象。
  + **envs**: 训练用的环境列表。
  + **eval\_envs**: 评估用的环境列表。
  + **env\_strs**: 环境名称列表。
  + **total\_steps**: 训练的总时间步数。
  + **eps**: 用于动态调整探索和利用的epsilon策略。
  + **eval\_frequency**: 每多少步进行一次评估。
  + **eval\_episodes**: 每次评估中运行的轮次数。
  + **policy\_path**: 存储策略和检查点的路径。
  + **save\_policy**: 是否保存策略。
  + **logger**: 用于记录训练过程的日志工具。
  + **mean\_success\_rate**, **mean\_episode\_length**, **mean\_reward**: 用于追踪平均成功率、平均轮次长度和平均奖励的工具。
  + **time\_remaining**: 设定的时间限制。
  + **verbose**: 是否在标准输出中打印更新。
* **输出**:
  + 函数没有返回值，但会在指定路径保存策略模型，同时通过日志记录器输出训练过程中的详细信息。

工作流程

1. **初始化**:
   * 设置代理网络为训练模式，随机选择一个环境进行训练，重置代理的上下文和环境状态。
2. **训练迭代**:
   * 在指定的总时间步数内循环执行，每个步骤中选择动作，执行环境交互，记录结果，并适时进行梯度更新。
   * 在每个步骤中，如果达到预定的评估频率，会对所有评估环境执行性能评估，并记录结果。
3. **评估和日志记录**:
   * 定期评估代理在各个环境上的表现，记录成功率、返回值和轮次长度等统计数据，并通过日志工具记录。
   * 如果启用了详细输出，将在控制台打印训练进度和评估结果。
4. **策略保存和时间检查**:
   * 根据设置定期保存训练好的策略网络。
   * 如果定义了时间限制并达到该限制，将保存当前状态并终止训练。

结论

**train** 函数通过结合系统的训练和评估机制，提供了一个全面的方法来优化强化学习代理的决策能力。此外，该函数还考虑了操作的实用性，如定时保存和时间限制管理，确保了训练过程的高效和安全。