开发指南

一、开发环境使用说明

当前开发机直接配置好了包括你需要的numpy torch scikit-learn等包，原则上不推荐安装其他包，但是若确有所需可以pip install或conda install，额外安装的包需要在提交的压缩包中readme.md中说明。

强化学习训练的reward、loss等数据除了print出来外，也可以尝试一些在线可视化方法，我们配置了swanlab等包。使用可以参考swanlab等官方repo查看同步训练信息和曲线等方法。

二、强化学习环境使用说明

1. 基础环境创建

```
from env_v3 import GorgeWalkEnv

# 创建单个环境实例
env = GorgeWalkEnv(render_mode=None, options={"usr_conf": usr_conf})
```

2. 自定义奖励塑形

虽然环境内置了基础奖励函数，但强烈建议根据任务需求自定义奖励配置：

```
custom_reward_shaping_conf = {
    "per_step_reward": 0.0,       # 鼓励快速完成
    "blocked_reward": 0.0,        # 惩罚无效动作
    "treasure_reward": 0.0,        # 可选择忽略宝箱
    "truncated_reward": 0.0,    # 强惩罚超时
    "terminated_reward": 0.0,    # 到达终点奖励
}
#具体数值需要你自定义
env = GorgeWalkEnv(
    render_mode=None,
    options={"usr_conf": custom_reward_shaping_conf}
)
```

3. 异步并行环境

为提高训练效率，推荐使用`gym.vector.AsyncVectorEnv`进行异步并行训练：

```python
from gym.vector import AsyncVectorEnv

def make_env():
    return GorgeWalkEnv(render_mode=None, options={"usr_conf": conf})

NUM_ENVS = 32  # 并行环境数量
env_fns = [make_env for _ in range(NUM_ENVS)]
vec_env = AsyncVectorEnv(env_fns)

# 批量交互
obs_batch = vec_env.reset()[^0]
actions = [agent.choose_action(obs) for obs in obs_batch]
next_obs, rewards, dones, truncated, infos = vec_env.step(actions)
```

三、算法开发与提交要求

1. 核心原则

- 允许使用任何强化学习算法（包括（深度）强化学习、表格方法等）
- 允许使用非学习方法（如有限状态机、启发式算法）
- **禁止**使用完全硬编码的记忆/遍历方法
- 按照下面的要求进行开发，最终提交整个项目的压缩包

2. 项目结构建议(可以参考示例Q-Learning)

```
project/
├── framework_diy/
|   ├── agent_diy.py        # 自定义Agent类
|   ├── trainer.py          # 训练器（如果使用强化学习）
|   └── buffer.py           # 经验回放缓冲区（如果使用强化学习）
├── config/
|   └── train_conf          # 训练配置参数（如果使用强化学习）
├── train_diy.py            # 训练主程序（如果使用强化学习）
├── eval_diy.py             # 评估脚本
├── models/                 # 保存的模型文件（如果使用强化学习）
└── readme.md               # 算法说明文档
```

3. 必需完成的代码

必须创建**自定义Agent类（/framework_diy/agent_diy.py）**，其中必须包含以下方法:
choose_action；除此之外需要**修改根目录下的eval_diy.py文件**，只需修改main函数中你的自定义智能体的声明部分（如果使用强化学习，需要加入导入模型部分）。其余部分不做要求。DIYAgent类也可以依据示例代码进行开发。

DIYAgent类讲解:

若使用强化学习训练，则需要定义load函数来导入你的模型文件、Q-table等；此外你可以定义eval_mode函数来切换训练和评估模式来提高性能

```python
class DIYAgent:
    def __init__(self, **kwargs):
```

```
        # 初始化参数
        pass

    #(required)
    def choose_action(self, raw_obs, info, use_mapping=False):
        """
        核心决策函数

        Args:
            raw_obs: 环境返回的原始观测（字典格式）
            info: 环境返回的原始信息（字典格式）
            #以上两个是必须的输入条件，下面use_mapping为可选
            use_mapping: 是否使用状态映射（如果使用强化学习，训练时推荐False，因为这时候obs和info内容较而且不一定都需要，建议预处理后获得处理后的state，再输入agent以及存入buffer，简化训练流程；评估时True）

        Returns:
            action: int, 选择的动作 (0-3)
        """
        if use_mapping:
            state = self.mapping(raw_obs)
        else:
            state = raw_obs
        # 决策逻辑
        return action

    #(optional)
    def eval_mode(self):
        """
        切换到评估模式
        - 关闭探索（如epsilon-greedy中设置epsilon=0）
        - 关闭dropout（如神经网络设置model.eval()）
        """
        self.epsilon = 0.0

    #(optional)
    def load(self, filepath):
        """
        加载已训练的模型参数

        Args:
            filepath: 模型文件路径
        """
        # 加载模型逻辑
        pass

    #(optional)
    def mapping(self, raw_obs):
        """
        (可选) 将原始观测映射为智能体使用的状态表示
        """
        return (raw_obs["agent_pos"][^0], raw_obs["agent_pos"][^1])
```

eval_diy.py文件修改方式见代码内注释，改动仅需2-3行代码。

4. 提交要求

你需要开发的整个文件夹压缩，下载后并提交压缩包；原则上不限制压缩包内文件数量。压缩包大小原则上不得超出100MB（包含训练的模型）。压缩包内至少需要包含以下内容：

1. **核心文件**：参考eval_q_learning.py构建，仅需要修改agent声明和load模型相关的内容：注意调用agent_diy并构建评估函数，确保`eval_diy.py`可以独立运行并成功加载模型
2. **模型文件**：包含训练好的模型参数文件
3. **说明文档**：提供`readme.md`，最好包含以下内容，以便于我们可以评估你的思路与创造性：
    - 算法原理和创新点
    - 超参数配置说明
    - 训练过程和性能分析
    - 运行环境依赖
4. **压缩包命名**：`姓名_算法名称_提交日期.zip`

四、示例：Q-Learning实现

一个简单的、未进行reward shaping的Q-Learning算法。其中agent、trainer、buffer等在framework_q_learning目录下，train和evaluation的主函数在根目录下。

1. Agent实现（agent_q_learning.py）

核心思路：使用Q-table存储状态-动作价值，状态简化为二维坐标(x, y)，动作空间4维。

关键设计：

- **状态表示**：仅使用`agent_pos`，忽略局部视野、宝箱等复杂信息
- **Q-table维度**：`(64, 64, 4)`，对应(x坐标, y坐标, 4个动作)
- **探索策略**：epsilon-greedy，epsilon从1.0衰减到0.01
- **更新公式**：标准Q-Learning，`Q(s,a) += lr * (r + gamma * max(Q(s',a')) - Q(s,a))`
- **mapping函数**：用于评估时从完整观测提取简化状态

```
import numpy as np

class QLearningAgent:
    def __init__(self, grid_size=64, lr=0.1, gamma=0.99,
                 epsilon=1.0, epsilon_min=0.01, epsilon_decay=0.995):
        self.grid_size = grid_size
        self.lr = lr
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        self.q_table = np.zeros((grid_size, grid_size, 4), dtype=np.float32)

    def choose_action(self, raw_obs, use_mapping=False):
        if use_mapping:
            state = self.mapping(raw_obs)
        else:
```

```
            state = raw_obs

        x, y = state
        if np.random.rand() < self.epsilon:
            return np.random.choice(4)
        return np.argmax(self.q_table[x, y])

    def learn(self, state, action, reward, next_state, done):
        x, y = state
        nx, ny = next_state
        q_predict = self.q_table[x, y, action]

        if done:
            q_target = reward
        else:
            q_target = reward + self.gamma * np.max(self.q_table[nx, ny])

        self.q_table[x, y, action] += self.lr * (q_target - q_predict)

    def update_epsilon(self):
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def save(self, filepath="q_table.npy"):
        np.save(filepath, self.q_table)

    def load(self, filepath="q_table.npy"):
        if os.path.exists(filepath):
            self.q_table = np.load(filepath)
```

2. 训练脚本(train_q_learning.py)

配置说明:

- `NUM_ENVS = 32`：并行32个环境
- `TOTAL_TIMESTEPS = 1,000,000`：总训练步数（分摊到32个环境，每个环境约31,250步）

```
from gym.vector import AsyncVectorEnv

NUM_ENVS = 32
TOTAL_TIMESTEPS = 1_000_000

# 创建并行环境
env_fns = [make_env for _ in range(NUM_ENVS)]
vec_env = AsyncVectorEnv(env_fns)

# 初始化agent和trainer
agent = QLearningAgent(grid_size=64)
trainer = Trainer(env=vec_env, agent=agent, buffer=None, num_envs=NUM_ENVS)

# 训练
trainer.train(total_timesteps=TOTAL_TIMESTEPS)
agent.save("q_table.npy")
```

3. 评估脚本(eval_q_learning.py)

功能：加载训练好的Q-table，单环境测试性能。

关键点：

- 使用`use_mapping=True`从完整观测提取状态
- 返回`info['game_info']['total_score']`作为评估指标

```
def evaluate_one_episode(agent, env):
    obs, info = env.reset()
    done = False

    while not done:
        action = agent.choose_action(obs, info, use_mapping=True)
        obs, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated

    return info['game_info']['total_score']

# 评估
env = make_env()
agent = QLearningAgent(grid_size=64)
agent.load("q_table.npy")
agent.eval_mode()

score = evaluate_one_episode(agent, env)
print(f"Episode总得分: {score}")
```

五、性能优化建议

仅供参考，优先级从上到下

1. **算法调整**：当前的Q-Learning示例仅使用极少量的观测即位置坐标，所以本身能力极其首先，可先扩展对observation、info的利用如包含局部视野、宝箱状态等。除此之外也推荐更换DQN、以及actor-critic的SOTA算法（TD3、SAC、PPO等）。也推荐尝试一些其他不依赖强化学习的方法。
2. **奖励工程**：调整奖励系数引导agent学习特定行为（如先不考虑宝箱，仅训练agent到达终点作为一个保底策略；后面再尝试迭代算法多收集宝箱）
3. **并行训练**：使用32个或更多并行环境可显著加速训练
4. **经验回放**：使用经验回放提高样本效率
5. **探索策略**：除epsilon-greedy外，可尝试Boltzmann探索、UCB等