智能空中博弈算法挑战赛

智能体开发指南

南开大学

二〇二四年七月

目 录

1 智能体开发流程指导说明

1.1 概述

人工智能开发接口提供使用 Python 脚本编写的推演交互功能模块，Python 程序主要包含两个部分：环境模块（custom_env 文件夹）、训练主体类模块（trainable_class）与智能体模块（agent 文件夹）。param.py 包含了智能体训练所需用到的参数，main_train.py 包含了训练的函数调用示例。

如图 1 所示，agent 文件夹下包含基类智能体（base_agent.py）和红蓝双方规则智能体(red_agent.py 与 blue_agent.py)的示例程序；custom_env 文件夹下包含基类环境类（env_zhikong.py）和 1v1 示例环境类（DogFight.py）；训练主体类模块包含自定义模型文件夹（custom_model）与自定义策略文件夹（custom_policy）。

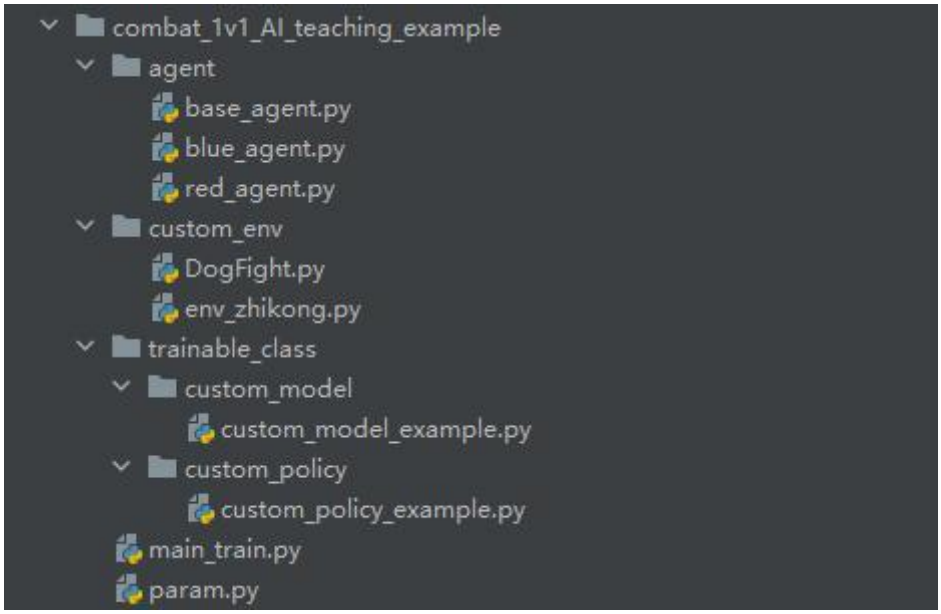

图 1　Python 程序架构

1.2 博弈推演平台构建与配置

仿真后台（仿真引擎）可以通过命令行直接启动，算法定义在 custom_env/env_zhikong.py 文件中，具体启动代码如图 2 所示。

```
def create_entity(self):
    is_success = False
    while not is_success:
        try:
            self.excute_cmd = f'{self.excute_path} Ip={self.IP} Port={self.PORT} ' \
                              f'PlayMode={self.RENDER} ' \
                              f'RedNum={self.red_num} BlueNum={self.blue_num} ' \
                              f'Scenes={self.scenes}'
            print('Creating Env', self.excute_cmd)
            self.unity = os.popen(self.excute_cmd)
            time.sleep(20)
            self._connect()
            is_success = True
            print('Env Created')
        except Exception as e:
            print('Create failed and the reason is ', e)
            time.sleep(5)
```

图 2　仿真引擎启动

启动参数可在环境中自行定义，具体参数与含义简介如下：

Ip=ip 地址（127.0.0.1）

Port=端口号（8888）

RedNum=红方数量

BlueNum=蓝方数量

PlayMode=0/1（训练模式/演示模式）

Scenes=1/2/3/4/5(阿尔卑斯山/阿塔卡马沙漠/空场景/黄石公园/科罗拉多大峡谷)

若不输入可为默认值 Ip 127.0.0.1 Port 8888 RedNum 6 BlueNum 6 PlayMode 0 Scenes=1。

2 智能体训练流程

为了便于开发使用，本部分以 1v1 为例，简要阐述开发流程。

RLlib 是开源的工业级强化学习库，可高效分配训练资源（CPU、GPU 与内存等），并提供优秀的异常处理等功能，可为算法的训练效率提供保障；近端裁剪策略梯度算法（PPO）是当前常用的强化学习基线算法，并以训练稳定著称，因此本实验基于 RLlib 实现工程开发，以 PPO 为底层更新算法，训练得到 1v1 场景下的固定翼无人机博弈智能体。

算法主体由环境设计、模型架构、算法更新与训练主体四部分组成。具体而言，环境设计部分实现了状态编码、动作处理与回报设计，对博弈问题进行建模并编码；模型架构部分设计了算法训练所需的深度神经网络，本实验包括值函数网络与策略函数网络；算法更新部分定义了模型优化器与损失函数，用于智能体的更新；训练主体则负责环境设计、模型架构与算法更新三部分的调配，实现多线程间的协调与资源分配，为智能体采样与更新的有序进行提供保障。

2.1 环境设计

环境设计部分由状态编码、动作处理与回报设计三部分组成，本小节详细阐述了其具体含义并提供具体代码样例，以方便开发者的后续开发，具体可见 custom_env/DogFight.py。

2.1.1 状态编码

状态s是智能体对自身与环境状态的数学编码，本实验假设环境为完全可观，为此可获取敌方任意时刻的位姿与武器信息，为我方智能体决策提供辅助。

本实验设计的状态编码包括红蓝双方

（1）基础信息：生命值与高度共 2 维；

（2）位姿信息：经纬度与姿态共 6 维；

（3）速度信息：机体坐标系下三个方向的速度信息共 3 维；

（4）角速度信息：机体坐标系下三个方向的角速度信息共 3 维；

（5）武器信息：两种武器的剩余数量、是否加载完毕与是否被地方武器锁定等信息共 9 维。

红蓝双方各包含 23 维单体状态，加入 2 维死亡信息，共计 48 维。为阐述更加清晰，单体状态编码部分的代码表示如下：

```
1.  单体无人机的状态编码
2.  def _state_process(self, flag):
3.      """
4.      TODO：本函数是状态编码信息，此处只是编码了生命值和高度信息
5.      TODO：编程开发者可以参考智空文档，自行定义智能体训练需要的观测信息
6.      """
7.      if self.is_done[flag]:
8.          return [0 for i in range(23)]
9.      else:
10.         post_process_obs = []
11.         obs = self.obs_tot[flag][f'{flag}_0']
12.         # 生命值和高度
13.         post_process_obs.append(obs['LifeCurrent'])
14.         post_process_obs.append(obs['position/h-sl-ft'])
15.         # 经纬度
16.         post_process_obs.append(obs['position/long-gc-deg'])
17.         post_process_obs.append(obs['position/lat-geod-deg'])
18.         # 姿态
19.         post_process_obs.append(obs['attitude/pitch-rad'])
20.         post_process_obs.append(obs['attitude/roll-rad'])
21.         post_process_obs.append(obs['attitude/psi-deg'])
```

```
22.         post_process_obs.append(obs['aero/beta-deg'])
23.         # 速度
24.         post_process_obs.append(obs['velocities/u-fps'])
25.         post_process_obs.append(obs['velocities/v-fps'])
26.         post_process_obs.append(obs['velocities/w-fps'])
27.         # 角速度
28.         post_process_obs.append(obs['velocities/p-rad_sec'])
29.         post_process_obs.append(obs['velocities/q-rad_sec'])
30.         post_process_obs.append(obs['velocities/r-rad_sec'])
31.         # 武器
32.         post_process_obs.append(obs['SRAAMCurrentNum'])
33.         post_process_obs.append(obs['AMRAAMCurrentNum'])
34.         post_process_obs.append(obs['SRAAM1_CanReload'])
35.         post_process_obs.append(obs['SRAAM2_CanReload'])
36.         post_process_obs.append(obs['AMRAAM1_CanReload'])
37.         post_process_obs.append(obs['AMRAAM2_CanReload'])
38.         post_process_obs.append(obs['AMRAAM3_CanReload'])
39.         post_process_obs.append(obs['AMRAAM4_CanReload'])
40.         post_process_obs.append(obs['MissileAlert'])
41.         return post_process_obs
```

2.1.2 动作处理

动作a是智能体与环境交互时执行的指令，本实验中，智能体需在平稳控制无人机的同时，合理分配武器切换与攻击指令，因此，本实验的动作可表示为：

$$a = [a_m, a_w],$$

其中a_m, a_w分别表示机动控制指令与武器控制指令。

本实验通过油门、副翼、升降舵与方向舵实现对无人机的控制，即

$$a_m = [c_e, c_a, c_r, c_T],$$

其中c_e, c_a, c_r, c_T分别表示无人机的升降舵、副翼、方向舵与油门。

武器控制指令a_w包括武器切换与武器发射，可表示如下：

$$a_w = [l_w, s_w].$$

值得注意的是，机动控制指令为连续控制量，武器控制指令则为离散控制量，为保持动作类型的一致性，对机动控制量进行离散化处理，结合武器控制指令，动作空间最终可表示如下：

$$\begin{cases} c_e \in \{-1,0,1\} \\ c_a \in \{-1,0,1\} \\ c_r \in \{-1,0,1\} \\ c_T \in \{0,0.5,1\} \\ l_w \in \{0,1\} \\ s_w \in \{0,1\} \end{cases}.$$

综上，连续-离散混合动作空间维度为 324 维，动作解码部分代码表示如下：

```
动作解码
def input_index_action(self, action):
    action = int(action)
    switch_missile_action = int(action // 162)
    action %= 162
    launch_missile_action = int(action // 81)
    action %= 81
    action_ce = action // 27
    action %= 27
    action_ca = action // 9
    action %= 9
    action_cr = action
    action %= 3
    action_cT = action
```

2.1.3 回报设计

回报r是智能体与环境进行交互后，环境对智能体行为好坏的反馈，其设计方式直接影响着智能体的训练方向与训练速度。

在 1v1 博弈问题中，智能体旨在击杀对手的同时保证自身存活，一时的优势或劣势地位无法改变博弈的根本目标，基于威胁指数的稠密回报设计可能使智能体过度在意一时的得失，忽略博弈的根本目的，导致训练偏离博弈目标；然而仅使用胜负作为回报则会引起稀疏回报问题，这是由于 1v1 博弈问题具有交互序列长等特点，常常需要成百上千步才能得到最终的胜负信息。因此，如何权衡稠密回报与稀疏回报的设计，是留给开发者解决的一大难题。

为便于阐述，本实验使用基于胜负信息的稀疏回报，具体瞬时回报设计如下：

$$r = \begin{cases} 1, \ if \ win \\ 0, \ if \ tie \\ -1, \ if \ loss \end{cases}.$$

回报设计的核心代码表示如下：

```
瞬时回报设计
def get_win_loss_reward(self):
    """
    Returns:   win_loss reward
    """
    control_side = self.control_side
    opponent_side = self.opponent_side
    if self.is_done['__all__']:
        control_side_death_event = self.obs_tot[control_side][f'{control_side}_0']['DeathEvent']
        opponent_side_death_event = self.obs_tot[opponent_side][f'{opponent_side}_0']['DeathEvent']
        # 1. 没打死对方给予负向回报
        if control_side_death_event == 99 and opponent_side_death_event == 99:
```

```
            return 0
        # 2. 红蓝都死了，激进型目的达到部分，武器给予正向回报
        if control_side_death_event != 99 and opponent_side_death_event != 99:
            return 0
        # 3. 红方死了，蓝方没死，一定给予惩罚
        if control_side_death_event != 99:
            return -1
        # 4. 红方没死，蓝方死了，给于奖励
        else:
            return 1
    return 0
```

2.2 模型架构

模型架构部分设计了算法训练所需的深度神经网络，具体可见 trainable_class/custom_model/custom_model_example.py 文件，本实验选用 PPO 算法，共包括值函数网络与策略函数网络，网络的示意图如图所示。

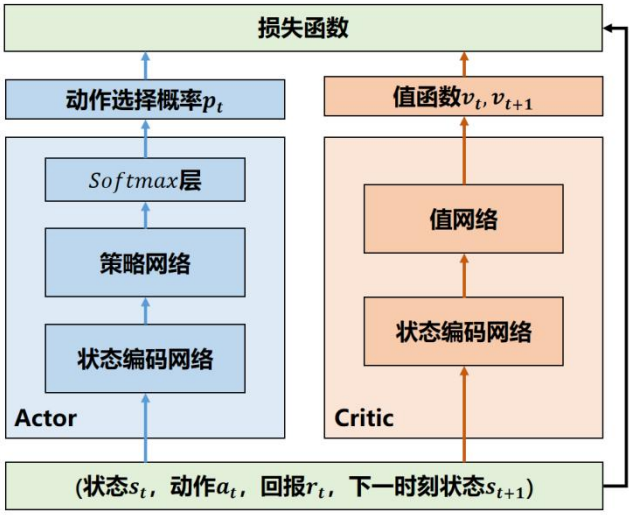

图 3　网络设计示意图

为便于开发者学习，我们将网络定义的核心部分代码如下，开发者可根据项目需求自行更改网络设置：

```
模型架构核心代码
class Custom_Model_Example(TorchModelV2, nn.Module):
    def __init__(
        self, obs_space: gym.spaces.Space,
        action_space: gym.spaces.Space, num_outputs: int,
        model_config: ModelConfigDict, name: str,
    ):
        TorchModelV2.__init__(
            self, obs_space, action_space, num_outputs, model_config, name
        )
        nn.Module.__init__(self)
        hiddens = list(model_config.get("fcnet_hiddens", [])) + list(
            model_config.get("post_fcnet_hiddens", [])
        )
        activation = model_config.get("fcnet_activation")
        if not model_config.get("fcnet_hiddens", []):
            activation = model_config.get("post_fcnet_activation")
        ###################################################
        # 策略网络的隐藏层
        ###################################################
        layers = []
        prev_layer_size = int(np.product(obs_space.shape))
        self._logits = None
        for size in hiddens[:-1]:
            layers.append(
                SlimFC(
                    in_size=prev_layer_size, out_size=size,
                    initializer=normc_initializer(1.0), activation_fn=activation
                ))
            prev_layer_size = size
        if len(hiddens) > 0:
            layers.append(
                SlimFC(
                    in_size=prev_layer_size, out_size=hiddens[-1],
                    initializer=normc_initializer(1.0), activation_fn=activation,
```

```
            ))
        prev_layer_size = hiddens[-1]
    self._logits = SlimFC(
        in_size=prev_layer_size, out_size=num_outputs,
        initializer=normc_initializer(0.01), activation_fn=None,
    )
    self._hidden_layers = nn.Sequential(*layers)
    ##################################################
    # 值函数的网络部分，包括了值函数的编码网络与最后的输出层
    ##################################################
    prev_vf_layer_size = int(np.product(obs_space.shape))
    vf_layers = []
    for size in hiddens:
        vf_layers.append(
            SlimFC(
                in_size=prev_vf_layer_size, out_size=size,
                activation_fn=activation, initializer=normc_initializer(1.0),
            ))
        prev_vf_layer_size = size
    self._value_branch_separate = nn.Sequential(*vf_layers)
    self._value_branch = SlimFC(
        in_size=prev_layer_size, out_size=1,
        initializer=normc_initializer(0.01), activation_fn=None,
    )
    self._features = None
    self._last_flat_in = None
```

2.3 算法更新

算法更新部分定义了模型优化器与损失函数，用于智能体的更新，具体可见 trainable_class/custom_policy/custom_policy_example.py 文件。模型优化器采用 Adam 优化器，同时优化值网络与策略网络。损失函数由值网络损失与策略网络损失组成，值网络损失与策略网络

损失的具体计算方法已在 2.2 中进行了详细阐述。

2.3.1 值网络损失

为便于开发者学习，我们将值网络损失核心代码展示如下：

```
值网络损失计算
value_fn_out = model.value_function()
vf_loss = torch.pow(
    value_fn_out - train_batch[Postprocessing.VALUE_TARGETS], 2.0
)
vf_loss_clipped = torch.clamp(vf_loss, 0, self.config["vf_clip_param"])
mean_vf_loss = reduce_mean_valid(vf_loss_clipped)
```

2.3.2 策略网络损失

策略网络损失核心代码展示如下：

```
策略网络损失计算
logp_ratio = torch.exp(
    curr_action_dist.logp(train_batch[SampleBatch.ACTIONS])
    - train_batch[SampleBatch.ACTION_LOGP]
)
surrogate_loss = torch.min(
    train_batch[Postprocessing.ADVANTAGES] * logp_ratio,
    train_batch[Postprocessing.ADVANTAGES]
    * torch.clamp(
        logp_ratio, 1 - self.config["clip_param"], 1 + self.config["clip_param"]
    ),
)
mean_policy_loss = reduce_mean_valid(-surrogate_loss)
```

2.4 训练主体

训练主体类负责环境设计、模型架构与算法更新三部分的调配，实现多线程间的协调与资源分配，为智能体采样与更新的有序进行提

供保障。本实验选用的 PPO 算法是典型 on-policy 算法，因此遵循“采样-更新”迭代更新架构。具体而言：在采样阶段，训练主体类接收环境返回的状态，根据预设探索策略与模型计算当前时刻动作，与环境进行交互并获得反馈信息；待采样足够数据后，进入更新阶段，此时训练主体类从缓存中采集训练样本，根据损失更新采样模型。整个过程中，训练主体类负责保证采样与更新阶段的模型参数的实时性，为智能体采样与更新的有序进行提供保障。

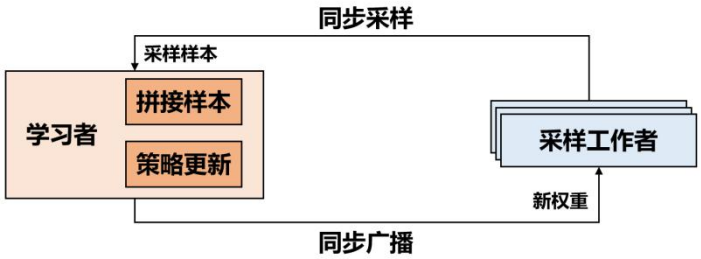

图 4　训练流程

为便于开发者学习，我们将训练主体类的组成框架部分代码展示如下，开发者可在 training_step(self)函数中定义采样与训练的时序关系（同步/异步等），具体方法可参考 RLlib 教学文档[1]：

```
class Custom_Trainable_Policy(PPO):
    @override(PPO)
    def get_default_policy_class(self, config):

    @classmethod
    @override(Algorithm)
    def get_default_config(cls) -> AlgorithmConfigDict:

    @override(Algorithm)
    def validate_config(self, config: AlgorithmConfigDict) -> None:

    @ExperimentalAPI
```

```
13.    def training_step(self) -> ResultDict:
```

3 智能体提交说明

3.1 基类智能体

为了方便统一对战调用形式，在文件夹 agent/base_agent.py 中定义了一个基类智能体：Base_agent。该基类包含 get_action 函数，用于接受平台态势数据，返回智能控制指令。平台返回的态势数据与返回平台的控制指令的数据格式详见文档“****”。

Base_Agent 类的具体属性与具体方法如表 1 与表 2 所示。

Base_Agent 类的属性

属性	类型	说明
side	string	智能体对应的推演方
control_num	int	智能体包含的智能体数量
agent_config	dict	智能体运行所需的额外参数

Base_Agent 类的方法功能

方法	功能说明
__init__	智能体初始化
agent_step	智能体决策
reset	智能体重置

Base_Agent 的具体代码定义如图 5 所示。

```python
class Base_agent:
    def __init__(self, control_num, side, agent_config):
        """
        :param control_num: 我方需要控制的智能体数量
        :param side: 我方控制的阵营，包含'red'与'blue'
        """
        self.side = side
        self.control_num = control_num
        self.agent_config = agent_config
        self.obs_tot = None

    def reset(self):
        pass

    def agent_step(self, obs):
        """
        :param obs: 平台提供的返回信息
        :return: 需要向平台发送的操控信息
        """
        pass
```

图 5 Base_Agent 基类智能体定义

3.2 自定义智能体

选手自定义的智能体**必须继承基类智能体**，并在初始化时**调用父类的初始化函数**。

可按下述顺序完成智能体类：**态势信息构建和预处理—>搭建神经网络结构—>调用并加载已训练模型—>根据编码的态势生成动作—>按照平台接收数据格式规范化动作。**

需要注意的是，某一方智能体只能获取本推演方的态势信息（战损信息除外），也只能对本推演方的平台进行控制，目前的发布版本并没有进行相关限制，但比赛进行时仿真引擎会检查 python 的命令有效性并过滤非法指令（对仿真逻辑没有影响）。