目录

1 算法解析

1.1 训练范式

这是一个博弈环境，因对手模型也成为了环境的一部分，整体动态性大幅提升。Baseline 采用自博弈（self-play）训练范式，智能体与自身对抗，并通过零和奖励机制抑制 reward hacking、防止不同智能体之间的“联手取巧”。

然而，在自博弈模式下，仅凭累积奖励难以判断训练收敛与否。为此，我们参照开发指南“训练中评估”章节，引入多种固定对手模型——在训练过程中定期让智能体与这些不同策略的对手对战，并以其对战胜率作为主要评估指标，从而更客观地衡量策略强度与收敛效果。

1.2 训练算法

本实验中，智能体采用 Proximal Policy Optimization（PPO）算法进行学习，其核心在于限制策略更新幅度，确保新策略不会过度偏离旧策略，从而提升训练稳定性并防止策略崩溃。PPO 基于 Actor–Critic 架构：

- **Actor 网络**：生成动作分布 $\pi_\theta(a \mid s)$，负责策略优化；
- **Critic 网络**：估计状态价值 $V_\phi(s)$，用于计算优势函数 $\hat{A}_t$，指导策略梯度方向。

PPO 通过重要性采样复用旧策略数据，定义概率比率

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}.$$

在 Dual-Clip 机制中，对正负优势分别进行不同方向的裁剪：当 $\hat{A}_t \geq 0$ 时抑制过度提升；当 $\hat{A}_t < 0$ 时抑制过度贬低。其目标函数定义为

$$L^{\text{DUALCLIP}}(\theta) = \mathbb{E}_t \begin{cases} \min\big(r_t(\theta)\,\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}_t\big), & \hat{A}_t \geq 0, \\ \max\Big(\min\big(r_t(\theta)\,\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}_t\big),\ c\,\hat{A}_t\Big), & \hat{A}_t < 0, \end{cases}$$

其中 $\epsilon > 0$ 为裁剪超参数，$c > 1$ 为负优势下界。

采用广义优势估计（GAE）平滑优势：

$$\delta_t = r_t + \gamma\, V_\phi(s_{t+1}) - V_\phi(s_t), \quad \hat{A}_t = \sum_{l=0}^{T-1} (\gamma\lambda)^l\, \delta_{t+l}.$$

完整损失结合价值函数和熵正则化：

$$L^V(\phi) = \mathbb{E}_t\big[\big(V_\phi(s_t) - R_t\big)^2\big], \quad R_t = \hat{A}_t + V_\phi(s_t), \tag{1}$$

$$S\big[\pi_\theta\big](s_t) = -\sum_a \pi_\theta(a \mid s_t)\, \log \pi_\theta(a \mid s_t), \tag{2}$$

$$L(\theta, \phi) = \mathbb{E}_t\Big[L^{\text{DUALCLIP}}(\theta) - c_1\, L^V(\phi) + c_2\, S\big[\pi_\theta\big](s_t)\Big], \tag{3}$$

其中 γ、λ、ϵ、c、c_1、c_2 均为超参数。

1.3 模型

模型的观测输入首先被拆分为多种信息子集——包括：

- **英雄特征**：己方与敌方英雄的生命值、法力值、坐标位置、朝向、技能冷却时间、当前状态（比如是否被眩晕、隐身等）；
- **防御塔状态**：各条防线塔的剩余生命值、等级，以及与英雄和小兵的相对距离；
- **小兵信息**：小兵类型（近战 / 远程）、生命值、当前位置与移动方向、正在攻击的目标等；
- **全局信息**：当前游戏时间、己方与敌方的经济差、各路推塔进度、当前大龙 / 小龙存活状态等高层次指标。

每一类子集先通过专用的编码器（通常是小型全连接网络或嵌入层）进行处理，编码器内部还包含了层归一化（LayerNorm）或批归一化（BatchNorm）以加速收敛。所有子集编码后的向量会在通道维度上拼接（concatenate）成一个长度为 `lstm_unit_size` 的高维特征向量。

为了满足 LSTM 对输入格式的要求——

$$[\,\texttt{batch\_size},\ \texttt{time\_steps},\ \texttt{lstm\_unit\_size}\,],$$

我们在送入网络前对样本执行以下操作：

1. 在 `definition.py` 中，`_reshape_lstm_batch_sample` 函数会：
   (a) 从缓冲区（buffer）中抽取连续的 `time_steps` 帧观测，每帧均已编码并拼接为向量；
   (b) 将这 `time_steps` 个向量按时间轴堆叠；
   (c) 将对应的第一帧的 LSTM 初始化状态（cell state 和 hidden state）一并打包；
   (d) 最终输出一个形状为 $[\texttt{batch\_size}, \texttt{time\_steps}, \texttt{lstm\_unit\_size}]$ 的张量以及初始状态对。
2. 在 `format_data` 中，将上述张量和状态写回 buffer，供后续训练或推理使用。

在**训练阶段**，以 $\texttt{batch\_size} \times \texttt{time\_steps}$ 的样本组为单位进行梯度更新；在**推理阶段**，仅使用当前帧（$\texttt{time\_steps} = 1$），以保证实时性。

LSTM 的输出会被拆分为两部分：

- **价值头**（Value Head）：用于估计当前状态价值；
- **动作头**（Policy Head）：输出多个离散动作的分布。

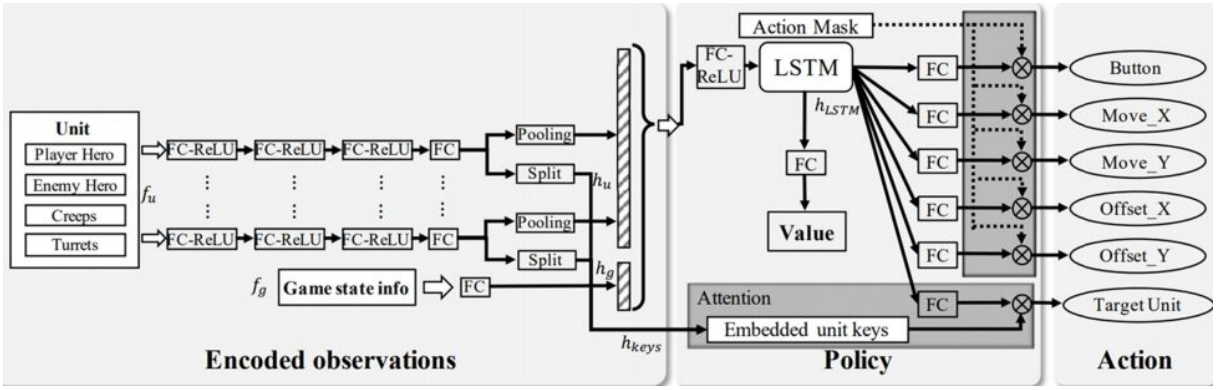

图 1: 网络架构图

两者共享底层特征提取网络，可显著降低计算冗余并提升推理效率。动作头总共包含 6 个离散维度，其中 **“目标单位”（target unit）**这一维度的概率分布是通过“朴素 Attention”实现的：将 LSTM 的输出向量与场上所有敌方候选目标向量做矩阵乘法，计算相似度得分后归一化，即可获得该维度的分布，如图 1 所示。

2 代码修改

2.1 奖励修改

Agent 在对局中，既要考虑打出有效的伤害，还要考虑到自身的存活率，命中率以及胜利条件，因此奖励设计在原有的基础（包括生命值、塔生命值、击杀数、经验等）上，再从以下几个方面入手：

2.1.1 承受伤害

Agent 在对局过程中，应该考虑其承受的伤害尽量小以保证存活，因此考虑加入对承受伤害的惩罚，从而引导 agent 减小其承受的伤害或者尽量对敌方攻击进行躲避。通过调用给出的接口，获取英雄所承受的伤害，并在 config 中将对应的 scale 设置为负数，以表示惩罚，奖励函数设计如下：

```
#config中的scale
"hero_hurt": −0.1,
#奖励函数设计
if reward_name == "hero_hurt":
    reward_struct.cur_frame_value = main_hero["totalBeHurtByHero"] / 2e4
```

2.1.2 打出伤害

Agent 在对局过程中的核心是打出伤害，对 agent 的伤害给予正奖励，从而鼓励 agent 提高自身伤害。奖励函数设计如下, 包括对敌方的伤害以及打出的总伤害：

```
#config中的scale
"total_damage": 0.1,
"hero_damage": 0.30,,
#奖励函数设计
if reward_name == "hero_damage":
    reward_struct.cur_frame_value = main_hero["totalHurtToHero"] / 2e4
if reward_name == "total_damage":
    reward_struct.cur_frame_value = main_hero["totalHurt"] / 6e4
```

2.1.3 暴击

在打出伤害的基础上，考虑 agent 打出暴击的伤害，以使其在一次伤害内造成的收益增大，引入暴击伤害，由暴击几率以及暴击伤害增益相乘，得到暴击伤害的期望，将其作为奖励，增强 agent 对暴击伤害的认识。奖励函数设计如下：

```
#config中的scale
" crit ": 0.01,
#奖励函数设计
if reward_name == "crit":
    reward_struct.cur_frame_value = main_hero["actor_state"][" values "][" crit_rate "] \
        * main_hero["actor_state "][" values "][" crit_effe "] / 1e4
```

2.1.4 技能命中

为了提升技能释放的命中率与战术价值，而非盲目使用技能，考虑智能体对局过程中技能的命中，让 agent 的命中率逐渐提升。获取 agent 在对局过程中技能槽中的所有技能的信息，以技能命中次数与释放技能次数进行比值，生成命中率，鼓励命中率上升。奖励函数设计如下：

```
#config中的scale
" skill_hit ": 0.01,
#奖励函数设计
if reward_name == "skill_hit":
    skill_state = main_hero[" skill_state "][" slot_states "]
    tmp = 0
    for skill_slot in skill_state :
        if skill_slot ["usedTimes"] != 0:
            tmp += skill_slot ["hitHeroTimes"] / ( skill_slot ["usedTimes"])
    reward_struct.cur_frame_value = tmp / len( skill_state )
```

2.1.5 无操作惩罚

为了惩罚 agent 在原地不动，从而被动挨打或者错失输出伤害的时机，判断其当前处于何种状态，如果为待在原地不动的状态，则给予惩罚。奖励函数设计如下：

```
#config中的scale
"no_ops": -0.001,
#奖励函数设计
if reward_name == "no_ops":
    if main_hero["actor_state "]["behav_mode"] == "State_Idle":
        reward_struct.cur_frame_value = True
    else :
        reward_struct.cur_frame_value = False
```

2.1.6 草丛隐身

除了正常进行伤害输出，可以利用草丛条件进行隐身，考虑到躲进草丛时 agent 不可见，且可利用草丛隐藏自身进行偷袭。因此对于在草丛中隐身给予奖励，先判断自身以及敌方是否不可见，若自身不可见敌方可见说明隐藏成功给予奖励，再根据攻击范围判断自身是否可以攻击到敌方，若可以再给予奖励。奖励函数设计如下：

```
#config中的scale
"in_grass": 0.001,
#奖励函数设计
if reward_name == "in_grass":
# 蹲草当“老六”
if not self.m_main_calc_frame_map["in_grass"].cur_frame_value:
    reward_struct.value = 0.0
else:
    # 基础奖励：在草丛中 0.25
    val = 0.25
    # 用 next(...) 快速找到主英雄和敌英雄
    main_hero = next(h for h in frame_data["hero_states"]
                     if h["player_id"] == self.main_hero_player_id)
    enemy_hero = next(h for h in frame_data["hero_states"]
                     if h["player_id"] != self.main_hero_player_id)
    # 1) “偷袭可见性” 条件：自己不可见且敌人可见 -> +0.5
    # 可见阵营，camp_visible[0]表示是否蓝方可见，camp_visible[1]表示是否红方可见
    main_vis_all = all(main_hero["actor_state"]["camp_visible"])
    enemy_vis_all = all(enemy_hero["actor_state"]["camp_visible"])
    if not main_vis_all and enemy_vis_all:
        val += 0.5
    # 2) 计算距离并判断是否在攻击范围内 -> +0.5
    mx, mz = main_hero["actor_state"]["location"]['x'],
    main_hero["actor_state"]["location"]['z']
    ex, ez = enemy_hero["actor_state"]["location"]['x'],
    enemy_hero["actor_state"]["location"]['z']
    hero_dist = math.dist((mx, mz), (ex, ez))
    attack_range = main_hero["actor_state"]['attack_range']
    if hero_dist <= attack_range:
        val += 0.5

    reward_struct.value = val
```

2.1.7 塔下行为奖惩

在王者荣耀此类 MOBA 游戏中，塔是一种高威胁的防御设施，合理地在敌方塔下进行战斗是英雄单挑取胜的重要组成部分。若敌方塔下无己方小兵，贸然进入塔下攻击将会吸引防御塔仇恨，带来重大风险，因此给予负奖励。而当敌方塔下有己方小兵时，优先攻击敌方塔或敌方小兵是合理行为，因为这有利于扩大经济优势以及消耗敌方塔的血条，应给予正向激励；若此时攻击敌方英雄，则会被防御塔集火，应给予惩罚。奖励函数设计如下

```
# config中的scale
"under_tower_behavior": 0.1,
# 奖励函数设计
elif reward_name == "under_tower_behavior":
    info = self.m_main_calc_frame_map[reward_name].cur_frame_value
    reward = 0.0
    if info["in_tower_range"]:
        has_ally_soldier = info["has_ally_soldier"]
        main_hero = next(h for h in frame_data["hero_states"]
                         if h["player_id"] == self.main_hero_player_id)
        runtime_id = main_hero["actor_state"]["runtime_id"]
        if not has_ally_soldier:
            reward -= 0.5 # 无兵塔下惩罚
        if "hurt_action" in frame_data.get("frame_action", {}):
            for hurt in frame_data["frame_action"]["hurt_action"]:
                if hurt["attacker"]["runtime_id"] != runtime_id:
                    continue
                target = hurt["hurt_target"]
                if target["camp"] != main_hero["actor_state"]["camp"]:
                    if target["sub_type"] == "ACTOR_SUB_TOWER"
                    and has_ally_soldier:
                        reward += 1.0 # 塔下攻击塔奖励
                    elif target["sub_type"] == "ACTOR_SUB_SOLDIER"
                    and has_ally_soldier:
                        reward += 0.2 # 塔下攻击敌兵奖励
                    elif target["sub_type"] == "ACTOR_SUB_HERO":
                        reward -= 0.3 # 塔下攻击英雄惩罚
    reward_struct.value = reward
```

2.1.8 被动技能奖励

在游戏对局中，利用英雄的被动技能同样是提高智能体战斗技巧的一种方式，游戏对局中狄仁杰的被动技能为普通攻击叠层，可获得额外攻速和移动速度提升，因此设置英雄触发被动技能的奖励，并且被动技能叠层越高，奖励越大，奖励函数设计如下：

```
# config中的scale
" passive_skills ": 0.01,
# 奖励函数设计
elif reward_name == "passive_skills":
    # 1) 初始化
    reward = 0.0
     passive_skills = main_hero.get(" passive_skill ", []) # 安全获取被动技能列表
    # 2) 计算总层数（跳过无效技能）
     total_level = sum( skill .get(" level ", 0) for skill in passive_skills )
    reward += total_level * 0.02 # 每层 +0.02
    # 3) 被动技能触发奖励
    for skill in passive_skills :
        if skill .get(" triggered ", False ): # 检查是否触发
            reward += 0.1 + skill .get(" level ", 0) * 0.02 # 基础触发 + 层数加成
    # 4) 高层数奖励
    if total_level >= 5:
        reward += 0.3
    elif total_level >= 3:
        reward += 0.1
    reward_struct .value = reward
```

2.2 算法修改

2.2.1 学习率衰减

在 PPO 算法中引入学习率衰减，主要通过动态调整优化步长来提升训练过程的稳定性和最终策略的收敛质量，可以提升训练稳定性、加速收敛使其逼近最优解。训练早期采用较高学习率，能快速引导策略向高回报区域移动，接近收敛时，衰减后的低学习率允许模型在最优解附近微调参数，避免因步幅过大而错过局部最优解，提升策略的精细度和泛化能力。相关实现代码如下：

```
# 更新学习率
```

```
self . scheduler . step( results [" total_loss "])
```

2.2.2 多 critic 网络引入

引入多 critic 网络是一种提升算法稳定性和性能的高级技术改进，其核心价值在于解决单一 critic 模型的估计偏差、训练波动及泛化局限问题，可减少价值估计偏差与方差、提升探索效率与策略鲁棒性。将奖励进行分组，每组奖励对应一个独立 critic（如伤害 critic、移动 critic、承伤 critic），每个 critic 仅优化其负责的子目标，使策略梯度更明确，多目标均衡提升，从而解决多目标冲突与优化不平衡问题。获取到对应的奖励后，将奖励进行分组，对每一个不同的 critic 网络使用不同组的奖励进行价值评估，再通过加权融合生成全局优势信号，实现多目标均衡优化。价值头原本只有一个。将其修改为多价值头，即将奖励分组，每组拟合一个值函数，改进估计效果降低方差。具体的实现是将奖励分组进行求和，用多个价值头分别估计，然后对每组的奖励和价值做 GAE，然后将多组的 GAE 加权求和得到优势值用于求 policy loss。然后组内的奖励和估计求 MSE，分组维求和得到 value loss，如下面代码所示：

```
"""output value"""
self .value_mlp = MLP([self.lstm_unit_size , 64, 1], "hero_value_mlp")
# 多头value，每组奖励一个head
self .value_group_mlps = nn.ModuleDict({
    group: MLP([self. lstm_unit_size , 64, 1], f"hero_value_{group}_mlp")
    for group in self .reward_groups
})
```

2.2.3 值函数裁剪

为了进一步提高值函数估计的稳定性，我们在传统的均方回归损失基础上引入了与策略 “Dual-Clip” 类似的裁剪机制。具体地，令旧参数下的值函数预测为 $V_{\theta_{t-1}}(s)$，新参数下的预测为 $V_{\theta_t}(s)$，以及目标回报 V_{targ}（可以由 $\hat{A}_t + V_{\theta_{t-1}}(s_t)$ 或蒙特卡洛回报得到）。则对单一值函数头的裁剪回归损失定义为：

$$L^V(\theta) = \mathbb{E}_t\Big[\max\Big\{\big(V_{\theta_t}(s_t) - V_{\text{targ}}\big)^2,\ \big(\text{clip}\big(V_{\theta_t}(s_t),\, V_{\theta_{t-1}}(s_t) - \varepsilon,\, V_{\theta_{t-1}}(s_t) + \varepsilon\big) - V_{\text{targ}}\big)^2\Big\}\Big].$$

其中，$\varepsilon > 0$ 控制裁剪区间宽度：如果新预测偏离过大，就退回到以旧预测为中心的保守范围内，从而避免值函数更新产生剧烈震荡。

2.2.4 局内奖励衰减以及分阶段

根据游戏经验，前期发育比较重要，且因为英雄能力的变化，获取同等的奖励，前期所需的时间会更长，而到了后期奖励会膨胀的太多，所以要对奖励加上一个时间的折扣。相关修改代码如下：

```
# 局内奖励随时间衰减
if self.time_scale_arg > 0:
    no_decay_keys = {"hp_point", "tower_hp_point", "kill", "death"}
    decay_factor = math.pow(0.6, frame_no / self.time_scale_arg)
    for key in self.m_reward_value:
        if key not in no_decay_keys:
            self.m_reward_value[key] *= decay_factor
```

另一方面，随着游戏进行，奖励的目标侧重点也应该变化：前期注重发育（打钱、升级），中后期慢慢过渡到 KDA 和推塔，最终的实现是生命值、推塔、击杀以及死亡的（零和）奖励不会随着时间衰减，并且在中后期还会提高相应的奖励权重。对应于多价值头中将奖励进行分组，奖励分组如下：

```
# 奖励分组配置
REWARD_GROUPS = {
    "decay" : ["money", "exp", "last_hit", "forward", "ep_rate", "total_damage",
               "hero_hurt", "hero_damage", "in_grass", "no_ops"],
    "no_decay" : ["death", "kill", "hp_point", "tower_hp_point"],
}
```

3 运行结果以及评估

由于受限于计算资源，我们难以进行严格的 benchmark 对比来全面验证各项设计的性能增益。为此，本实验参照第1.1节“训练中评估”方法，周期性地将当前策略与若干历史表现最优的基线模型进行对局，并以胜率变化作为初步判断依据。初始阶段，智能体胜率徘徊在约 80% 左右；随着 Dual-Clip PPO、多价值头及奖励设计的不断优化，胜率稳步攀升，最终超过 95%（如图 2 中黄线所示），这在一定程度上验证了方法的有效性。

与此同时，我们还对以下微观指标进行了多维评估：

- 平均推塔数量和推塔速度；
- 单局造成的总伤害与每分钟伤害率；
- 经济差距（金币/经验优势）随时间的变化趋势；
- 生存时长及死亡次数分布。

基于上述指标，我们动态调整奖励权重、时间衰减因子以及 Critic 网络结构，促进策略在不同阶段的均衡发展。经过约 100 小时的连续训练，最终获得了在胜率和各项微观指标上均表现优异的最优模型。

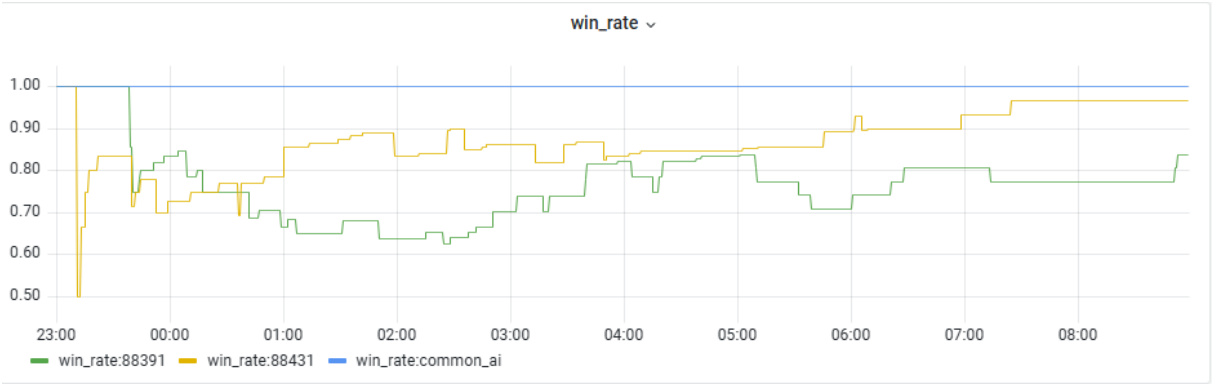

图 2: 智能体相对于历史基线模型的胜率随训练进度的变化曲线