电 子 科 技 大 学

实 验 报 告

学 生 姓 名： 卢 晓 雅　　学　号： 2020080904026

指导教师：李晶晶

实验地点：主楼 A2-412　　实验时间：2022 年 12 月 10 日星期六

一、实验室名称：计算机学院实验中心

二、实验项目名称：强化学习实验

三、实验学时：16 学时

四、实验原理：

1、腾讯开悟平台简介

开悟是腾讯牵头构建的 AI 多智能体与复杂决策开放研究平台，依托腾讯 AI Lab 和「王者荣耀」在算法、算力、实验场景方面的核心优势，为学术研究人员和算法开发者开放的国内领先、国际一流研究与应用探索平台。开悟平台使用《王者荣耀》的游戏环境进行 AI 模型的训练。

“王者荣耀”是一款在手机上游玩的多人在线战斗竞技场(MOBA)游戏。在“王者荣耀”中，玩家使用移动按钮来控制英雄的移动，并使用其他按钮来控制英雄的普攻和技能。 除了通过操作主界面观察周围环境， 玩家还可以通过左上角小地图了解全地图情况，其中可观察的炮塔、小兵和英雄以缩略图的形式显示。游戏存在战争迷雾机制，只有目前单位属于友方阵营或者处于友方阵营观测范围内才能被观测到。

游戏开始时，每个玩家控制英雄，从基地出发，通过杀死或摧毁其他游戏单位（例如敌方英雄、小兵、炮塔）获得金币和经验，购买装备和升级技能，以此

提升英雄的能力。 获胜目标是摧毁对手的炮塔和基地水晶，同时保护自己的炮塔和基地水晶。

2、强化学习算法原理

- 深度 Q 网络(Deep Q-network, DQN)

算法 14.5: 带经验回放的深度 Q 网络

输入: 状态空间 $\mathcal{S}$，动作空间 $\mathcal{A}$，折扣率 γ，学习率 α

初始化经验池 $\mathcal{D}$，容量为 N;
随机初始化 Q 网络的参数 ϕ;
随机初始化目标 Q 网络的参数 $\hat{\phi} = \phi$;
repeat
  初始化起始状态 s;
  repeat
    在状态 s，选择动作 $a = \pi^{\epsilon}$;
    执行动作 a，观测环境，得到即时奖励 r 和新的状态 s';
    将 s, a, r, s' 放入 $\mathcal{D}$ 中;
    从 $\mathcal{D}$ 中采样 ss, aa, rr, ss';
    $y = \begin{cases} rr, & ss' \text{为终止状态}, \\ rr + \gamma \max_{a'} Q_{\hat{\phi}}(\mathrm{ss}', \mathrm{a}'), & \text{否则} \end{cases}$;
    以 $\left(y - Q_{\phi}(\mathrm{ss}, \mathrm{aa})\right)^2$ 为损失函数来训练 Q 网络;
    $s \leftarrow s'$;
    每隔 C 步，$\hat{\phi} \leftarrow \phi$;
  until s 为终止状态;
until $\forall s, a$，$Q_{\phi}(\mathrm{s}, \mathrm{a})$ 收敛;
输出: Q 网络 $Q_{\phi}(\mathrm{s}, \mathrm{a})$

- 近端策略优化(Proximal Policy Optimization, PPO)

Algorithm 1 PPO, Actor-Critic Style

for iteration=1, 2, ... **do**
  for actor=1, 2, ..., N **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Algorithm 1 Proximal Policy Optimization (adapted from [8])

for $i \in \{1, \cdots, N\}$ **do**
  Run policy π_θ for T timesteps, collecting $\{s_t, a_t, r_t\}$
  Estimate advantages $\hat{A}_t = \sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t)$
  $\pi_{\text{old}} \leftarrow \pi_\theta$
  for $j \in \{1, \cdots, M\}$ **do**
    $J_{PPO}(\theta) = \sum_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \hat{A}_t - \lambda \text{KL}[\pi_{old}|\pi_\theta]$
    Update θ by a gradient method w.r.t. $J_{PPO}(\theta)$
  end for
  for $j \in \{1, \cdots, B\}$ **do**
    $L_{BL}(\phi) = -\sum_{t=1}^{T} (\sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t))^2$
    Update ϕ by a gradient method w.r.t. $L_{BL}(\phi)$
  end for
  if $\text{KL}[\pi_{old}|\pi_\theta] > \beta_{high} \text{KL}_{target}$ **then**
    $\lambda \leftarrow \alpha\lambda$
  else if $\text{KL}[\pi_{old}|\pi_\theta] < \beta_{low} \text{KL}_{target}$ **then**
    $\lambda \leftarrow \lambda/\alpha$
  end if
end for

- 深度确定性策略梯度(Deep Deterministic Policy Gradient, DDPG)

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

五、实验目的：

实验要求理解强化学习基本原理并学习强化学习经典算法，尝试使用深度 Q 网络(Deep Q-network, DQN)、近端策略优化(Proximal Policy Optimization, PPO)、深度确定性策略梯度(Deep Deterministic Policy Gradient, DDPG)等方法基于腾讯开悟平台训练强化学习智能体解决王者荣耀中墨家机关道 1v1 对战问题，理解强

化学习算法原理并通过实战部署掌握算法的应用。

六、实验内容：

以小组的形式组建战队，在给定的资源下训练并提交一个模型，控制**鲁班七号英雄**进行墨家机关道 1v1 对战。主要考查单智能体解决方案，模型结构设计，强化学习算法设计和训练方式探索。

要求：

1、**以战队为单位**提交在开悟平台挑战赛常规赛中的得分及排名截图。

2、**以个人为单位**提交实验报告。

七、实验器材（设备、元器件）：

PC 微机一台

八、实验步骤：

1. 配置环境，运行测试。

2. 调节超参数

1）调节 slow time

2）根据 config.json 中每个子奖励的含义，调节权重观察不同权重策略下智能体的行动特点。

2）理解 _calculate_loss()函数，调整三个子损失之间的权重。

3. 理解 _inference() 函数，给出函数的逻辑、流程、实现的功能；给出自己对网络结构的修改，对比代码修改前后的结果：（1）修改后 vs 修改前（2）修改后 vs baseline0（3） 修改前 vs baseline0。

4. 解释 PPO 算法及平台相关代码。

九、实验数据及结果分析：

(一) 代码分析

强化学习框架

1v1 开悟平台的训练框架 SAIL 基本类似 Asynchronous Advantage Actor-Critic（A3C）框架。在 A3C 中，使用神经网络输入现在的状态，输出价值函数 V(s) 的估计值（评价某个状态的好坏）和策略 π(s)（一组行动概率分布），使用 $\frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{Tn}(r_t^n + V^\pi(S_{t+1}^n) - V^\pi(S_t^n))$作为损失函数，其中 r 表示 reward，N 为采样的数量，T_n 为采样的步长。

开悟的框架在 A3C 的基础上做了改进，增加了储存样本的 Mempool 和储存模型的 Modelpool，训练时最多使用 8 个 Actor，训练过程包含以下三个步骤，如图 1 所示：

1. 将 Learner 中的网络参数复制给 Actor，完成模型同步；
2. Actor 根据策略 π(s)与环境进行交互，采集样本，并将样本存储在 Mempool 中；
3. Learner 使用 Mempool 中的样本计算损失（开悟的框架对 A3C 使用的损失函数做了一些改进，详见损失函数部分），并进行梯度下降，更新网络参数。

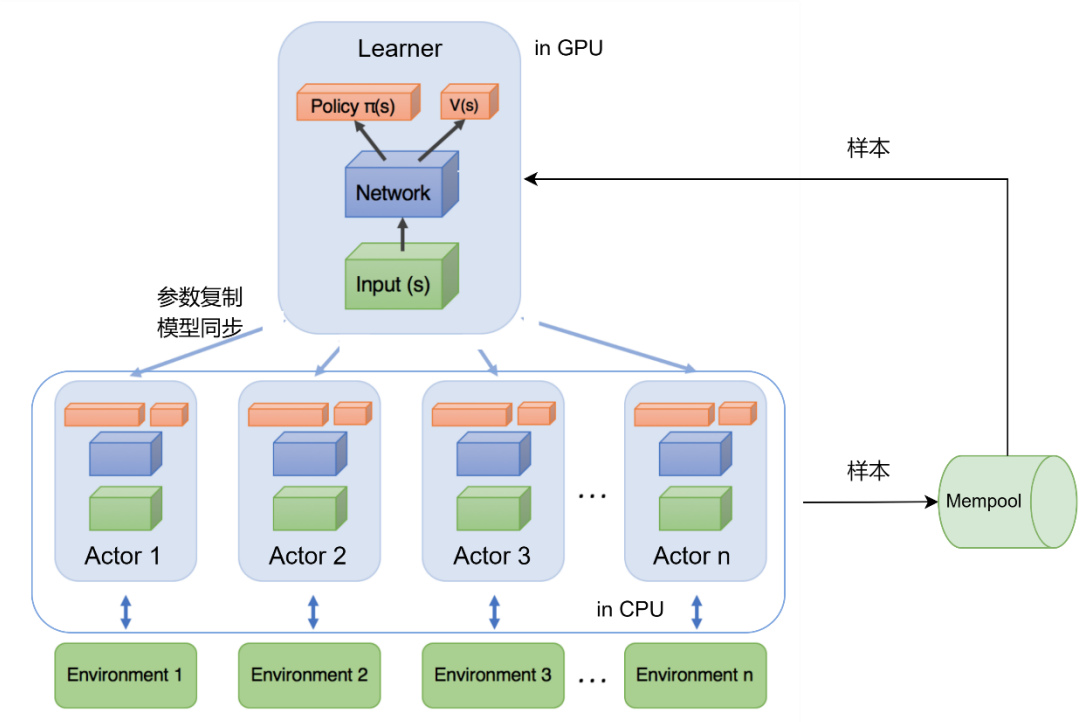

图 1（该图参考了博客[1]）

神经网络部分（_inference 函数）：

该部分没有使用 tensorflow 封装好的 dense，而是直接设置权重和偏置变量，并写出前向计算的过程，如代码 1，全连接层权重的 shape=[输入维度，输出维度]，偏置的 shape=[输出维度]。

```
fc3_hero_weight = self._fc_weight_variable(
    shape=[256, 128], name="fc3_hero_emy_weight"
)
fc3_hero_bias = self._bias_variable(
    shape=[128], name="fc3_hero_emy_bias"
)
fc3_hero_result = tf.add(
    tf.matmul(fc2_hero_result, fc3_hero_weight),
    fc3_hero_bias,
    name="fc3_hero_emy_result_%d" % index,
)
```

代码 1

根据代码梳理出的网络架构如图 2，网络可分为三个部分，第一部分使用三层全连接层分别从不同对象的信息中提取高位特征，产生的结果做两种并行的处理，一种是做 max pooling 后把不同对象的特征和全局特征合并到一起，另一种

是切分出全连接层输出的最后 32 维添加到 embed 列表中，第二部分由一层全连接层和一层长短期记忆网络（LSTM）组成，合并后的特征被输入第二部分，进行综合分析，学习技能组合，第三部分使用全连接网络产生 policy 和 value，产生 policy 的时使用了注意力机制。

LSTM 的代码如下，代码中调用 tensorflow 中的 BasicLSTMcell，使用 for 循环连接多个 LSTM 单元，即上一个单元生成的 h 和 c 又输入到下一单元，设置 step=16，即考虑前 16 步的游戏情况来产生下一步的决策。LSTM 单元的内部结构、串联方式以及 LSTM 网络中各变量的维度如图 3 所示，LSTM 单元的输入输出由遗忘门、输入门和输出门控制，遗忘门的输出 f_t 与上一时刻的细胞状态 c_t 相乘，乘数为 0 的信息被全部丢弃，为 1 的被全部保留，决定了上一细胞状态有多少能进入当前状态；输入门 i_t 决定输入信息有哪些被保留，输入信息包含当前时刻输入 x_t 和上一时刻隐层输出 h_{t-1} 两部分；输出门 o_t 决定 h_{t-1} 和 x_t 中哪些信息将被输出。

```
lstm_cell = tf.contrib.rnn.BasicLSTMCell(
    num_units=self.lstm_unit_size, forget_bias=1.0
)
with tf.variable_scope("rnn"):
    state = lstm_initial_state
    lstm_output_list = []
    for step in range(self.lstm_time_steps):
        lstm_output, state = lstm_cell(
            reshape_fc_public_result[:, step, :], state
        )
        lstm_output_list.append(lstm_output)
    lstm_outputs = tf.concat(lstm_output_list, axis=1, name="lstm_outputs")
    self.lstm_cell_output = state.c
    self.lstm_hidden_output = state.h
reshape_lstm_outputs_result = tf.reshape(
    lstm_outputs,
    [-1, self.lstm_unit_size],
    name="reshape_lstm_outputs_result",
)
```

代码 2

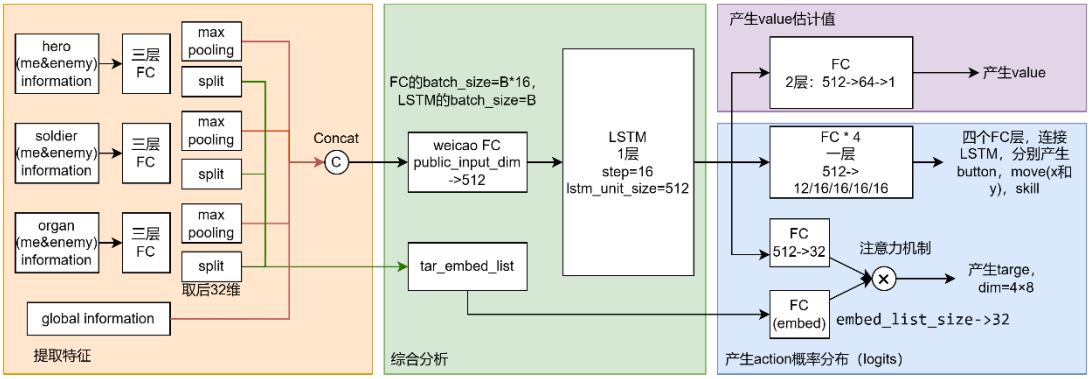

图 2

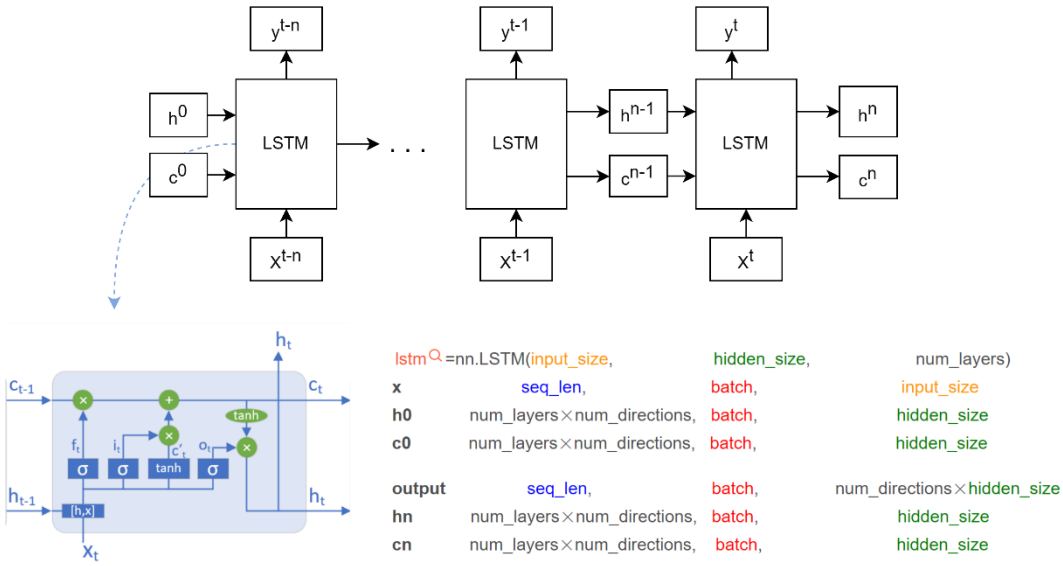

图 3

损失函数部分（_calculate_loss 函数）：

函数中计算的 loss 包含三部分，value loss，policy loss 和 entropy loss。

Value loss 就是 advantage 的估计值的平方，即$(r-v(s))^2$，v (s)即为前文分析过的神经网络的输出。

Policy loss 实现的是 PPO 算法[2]中提出的 loss function，由于 advantage 估计是不完全准确的，存在偏差，如果一次更新中 advantage 的估计偏差太大，之后更新的 policy 可能会完全偏离正确的方向，从而进入恶性循环，为解决这个问

题，TRPO[3]使用新旧 policy 之间的 KL 距离作为正则项，来保证每一次的 policy 更新在一个 trust Region 里，从而保证 policy 的单调提升，PPO 则是对 ratio，即新旧 policy 的偏差做修剪（clip），如果 ratio 偏差超过ε就做修剪，修剪后梯度也直接变为 0，神经网络也就不再更新参数了，不会偏离正确方向了，如图 4。policy loss 的计算分为两部分，第一部分是计算 action 的概率分布，计算方式类似计算监督学习分类任务中样本属于某类的概率，先对 action 使用 one hot 编码，再使用 softmax 函数对输出进行处理，对应代码中的"label_probability = 1.0 * label_exp_logits / label_sum_exp_logits"，第二部分就是代入 PPO 算法中计算 loss 的公式，代码注释如下：

```
policy_p = tf.reduce_sum(one_hot_actions * label_probability, axis=1)
# 计算log前+0.00001是为了防止出现log0=∞，导致梯度爆炸的情况
# 计算ratio，先log后exp，ratio=new_policy/old_policy
policy_log_p = tf.log(policy_p + 0.00001)
old_policy_p = tf.reduce_sum(
    one_hot_actions * old_label_probability_list[task_index] + 0.00001,
    axis=1,
)
old_policy_log_p = tf.log(old_policy_p)
final_log_p = final_log_p + policy_log_p - old_policy_log_p
ratio = tf.exp(final_log_p)
# 比标准的PPO增加了一次clip
clip_ratio = tf.clip_by_value(ratio, 0.0, 3.0)
# 计算r*A
surr1 = clip_ratio * advantage
# 计算clip(r, 1-ε, 1+ε)*A
surr2 = (
    tf.clip_by_value(
        ratio, 1.0 - self.clip_param, 1.0 + self.clip_param
    )
    * advantage
)
# Loss_clip=min(r*A, clip(r, 1-ε, 1+ε)*A)，并对不同类别的action加权
temp_policy_loss = -tf.reduce_sum(
    tf.to_float(weight_list[task_index]) * tf.minimum(surr1, surr2)
) / tf.maximum(tf.reduce_sum(tf.to_float(weight_list[task_index])), 1.0)
self.policy_cost = self.policy_cost + temp_policy_loss
```

代码 3

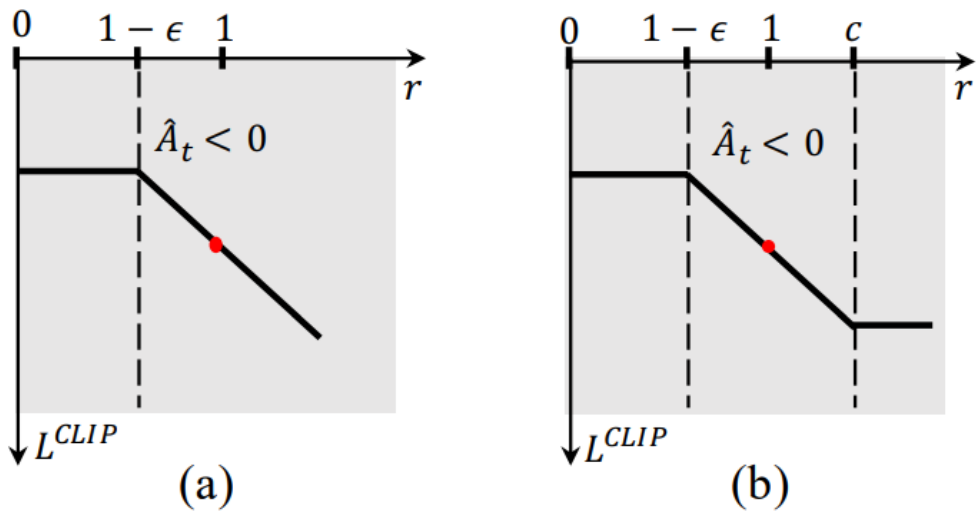

图 4 图(a)是标准的 PPO，图(b)是王者 AI 使用的双向 PPO（图片引自参考论文[4]）

```
policy_p = tf.reduce_sum(one_hot_actions * label_probability, axis=1)
policy_log_p = tf.log(policy_p + 0.00001)
old_policy_p = tf.reduce_sum(
    one_hot_actions * old_label_probability_list[task_index] + 0.00001,
    axis=1,
)
old_policy_log_p = tf.log(old_policy_p)
final_log_p = final_log_p + policy_log_p - old_policy_log_p
ratio = tf.exp(final_log_p)
clip_ratio = tf.clip_by_value(ratio, 0.0, 3.0)
surr1 = clip_ratio * advantage
surr2 = (
    tf.clip_by_value(
        ratio, 1.0 - self.clip_param, 1.0 + self.clip_param
    )
    * advantage
)
temp_policy_loss = -tf.reduce_sum(
    tf.to_float(weight_list[task_index]) * tf.minimum(surr1, surr2)
) / tf.maximum(tf.reduce_sum(tf.to_float(weight_list[task_index])), 1.0)
self.policy_cost = self.policy_cost + temp_policy_loss
```

代码 4

随着训练进行神经网络输出的 action 分布的 variance 会越来越小，反映到统计指标上就是 entropy=$\sum_{i=1}^{n} plog(p)$(p 是执行某个 action 的概率)的越来越小。使用负的 entropy 作为 loss，就能用于强迫神经网络输出的 action 分布不过于集中，避免模型在训练早期受到各种局部最优解的干扰而过早迷失方向，从而起到加强探索的效果。Entropy 系数（代码中的 var_beta）就类似于ε-greedy 策略中

的ε，entropy 系数越大，则模型探索新的 action 的倾向越强，合理的系数既能确保训练早期充分探索从而使模型向正确方向前进，又能使模型在训练中后期充分利用学到的技能从而获得高性能。对于不同任务，最优 entropy 系数往往各不相同，需要若干次试错才能找到。比如在训练开始后 policy entropy 快速下降说明模型陷入了局部最优，没学到有用技能，这时就应该提升 entropy 系数；如果训练很长时间 policy entropy 仍然未下降或者下降缓慢，说明模型探索范围过于广泛，学到的知识被随机性淹没，无法进一步用来提升性能，此时应该适当降低 entropy 系数。由于计算 policy loss 时已经算出了 action 分布，entropy loss 直接代入$\sum_{i=1}^{n} plog(p)$公式计算即可，但计算中增加了对不合法的 action 进行 mask 操作和对不同类的 action（button，move，skill，target）进行加权的操作。

```
temp_entropy_loss = -tf.reduce_sum(
    label_probability_list[current_entropy_loss_index]
    * legal_action_flag_list[task_index]
    * tf.log(
        label_probability_list[current_entropy_loss_index] + 0.00001
    ),
    axis=1,
)
temp_entropy_loss = -tf.reduce_sum(
    (temp_entropy_loss * tf.to_float(weight_list[task_index]))
) / tf.maximum(
    tf.reduce_sum(tf.to_float(weight_list[task_index])), 1.0
)  # add - because need to minize
```

代码 5

由此可见，value loss 起到了指导模型向正确的方向优化的作用，policy loss 起到过滤偏离当前太远的 policy，使 action 分布不会过于分散的作用，entropy loss 起到强迫模型加强探索，避免陷入局部最优解的作用，policy loss 和 entropy loss 在拮抗关系下共同辅助模型的优化。

(二) 运行测试

测试和运行的截图如下:

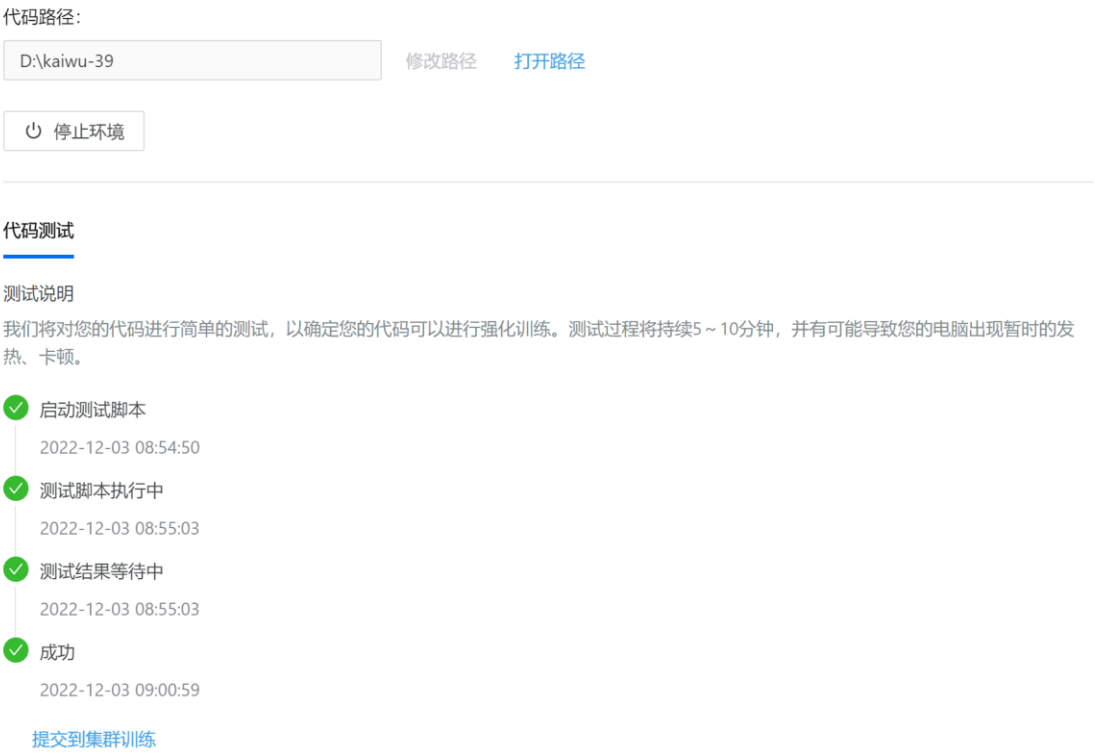

图 4

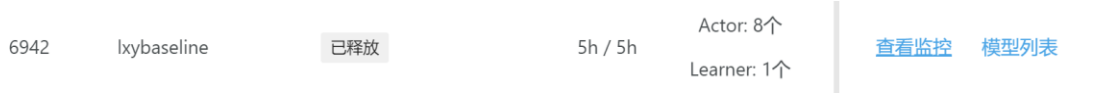

图 5

(三) 超参数设置分析

1) Slow_time

该参数用来控制样本的消耗速度，slow_time=0.1 则代表每训练一个 step 便会休眠 0.1s。slow_time 设置越大，休眠时间越长，样本消耗:样本生成比例越小，同一个样本被重复用于训练的次数越少，过拟合的风险也就会降低，所以应该尽量控制样本消耗:样本生成比例在 5:1 以下。

Slow_time=0 时样本被重复利用的次数太多，过拟合严重，训练 16h 的模型无法战胜训练 8h 的模型，slow_time=0.2 时，如图 6，训练过程中的 reward 趋

于平稳，6h 之后 hurt 下降，训练 14h 的模型略优于训练 30h 的模型，表现出轻微过拟合，slow_time=0.4 时训练 30h 后仍未出现过拟合。经过实验最终设置 slow_time=0.5，样本消耗：样本生成约等于 3.6。

图 6

2）loss

根据上文的代码分析可知，loss 是 entropy loss，value loss，policy loss 三部分的加和，我们试图通过类似消融的方法检测三部分 loss 的重要性，我们将这三部分分别乘 0.01 系数再进行训练。

Policy loss 或者 entropy loss 的系数乘 0.01 之后训练效果会变差很多，训练出的人机几乎不会离开泉水，被对方轻松击败。这验证了我们前文对代码的分析，value loss 是指引学习方向的核心 loss，entropy loss 和 policy loss 辅助 value loss，同时它们之间是拮抗的关系，当把它们其中的一个乘以 0.01 的系数时，它们的拮抗关系被完全破坏了，产生的 action 过于集中或过于发散，模型训练效果极差，因此 entropy loss 和 policy loss 的比例只能在 1：1 附近微调。

Value loss 被乘了 0.01 之后，训练效果反而比原模型好了很多，这可能是因

为 policy 和 entropy 发挥的辅助作用被放大。对战情况如下图:

图 7

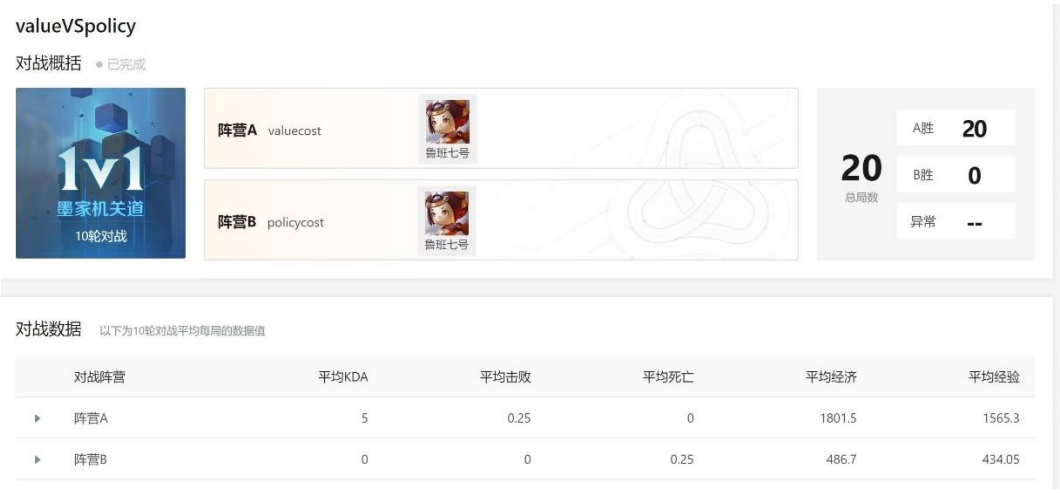

图 8

图 9

虽然也是使用梯度下降策略进行的优化，但强化学习的 loss 不会像监督学习的 loss 一样呈现明显的下降趋势，这可能是因为训练过程中的 reward 和 value

会同步上升，advantage 的估计值也维持平稳，如图 10，value loss 和 policy loss 都比较平稳，entropy loss 缓慢上升，这是因为模型训练到后期已经逐渐找到了正确的方向，所以模型探索（exploration）的倾向减弱，沿原方向寻找最优解（exploitation）的倾向增强，产生的 action 分布更集中，entropy 减小，-entropy 上升。

图 10

3）reward

通过观看该模型的对战视频我们发现按初始 reward 设置训练出的模型太过保守，血量掉到 1/4 以下就会回泉水加血，不管当前是否有对战或者推塔的优势，回泉水过于频繁浪费了大量的时间，推塔也不够积极。同时我们参考了王者荣耀 AI 的相关论文[4]中的 reward 设置，论文中的 reward 修改方案是将塔血量的 reward 从 5 提升至 10，鼓励人机推塔，将经济 reward 从 0.06 提高到 0.08，将 ep_rate reward 从 0.75 提升至 0.8，将被击杀的惩罚从-0.6 改成-0.5，减小回泉水加血的概率。

我们小组认为最佳的方案应该是根据战争的不同阶段使用不同的 reward 设置，比如开局应该清理兵线发育，后期应该全力推塔，但由于后端代码无法修改，且手册中并未说明表示游戏时期的特征位于 global_feature_list 的第几维，无法提取特征，所以很遗憾未能实现该方案。

Reward	Weight	Type	Description
hp_point	2.0	dense	the health point of hero
tower_hp_point	10.0	sparse	the health point of turrets and base
money (gold)	0.008	dense	the gold gained
ep_rate	0.8	dense	the rate of mana
death	-1.0	sparse	being killed
kill	-0.5	sparse	kill an enemy hero
exp	0.008	dense	the experience gained
last_hit	0.5	sparse	last hitting to enemy units

表 1（引自参考论文[4]）

(四) 网络结构调整

1) 所有全连接网络的层数都加一，将原本的输入输出维度为（input_dim, hidden_dim）的第一层 FC，改成输入输出维度为(input_dim, hidden_dim*2)和(hidden_dim*2, hidden_dim)两层 FC。

2) 增加一层 LSTM，多层 LSTM 一般使用相等的隐藏神经元个数，将第一层的输出直接 h 作为第二层对应 LSTM 单元的输入，代码如下:

```python
# 修改第一层的输出结果的维度，(batch_size, seq_len, hidden_unit_size)
    # reshape_lstm_outputs_result = tf.reshape(
    #     lstm_outputs,
    #     [-1, self.lstm_unit_size],
    #     name="reshape_lstm_outputs_result")
    reshape_fc_public_result = tf.reshape(
        lstm_outputs,
        [-1, self.lstm_time_steps, 512],
        name="reshape_lstm_outputs_result",)

# 增加第二层
with tf.variable_scope("public_lstm_2"):
    lstm_cell = tf.contrib.rnn.BasicLSTMCell(
        num_units=self.lstm_unit_size, forget_bias=1.0
    )
    with tf.variable_scope("rnn_2"):
        state = lstm_initial_state
        lstm_output_list = []
        for step in range(self.lstm_time_steps):
            lstm_output, state = lstm_cell(
                reshape_fc_public_result[:, step, :], state
            )
            lstm_output_list.append(lstm_output)
        lstm_outputs = tf.concat(lstm_output_list, axis=1, name="lstm_outputs_2")
        self.lstm_cell_output = state.c
        self.lstm_hidden_output = state.h
    reshape_lstm_outputs_result = tf.reshape(
        lstm_outputs,
        [-1, self.lstm_unit_size],
        name="reshape_lstm_outputs_result_2",
    )
```

代码 6

学习率相当于梯度下降的搜索步长，网络越深就应该使用越小的学习率，学习率过大可能会错过最优解，无法深入。我们加深网络后初次尝试时没有调整学习率，导致训练结果无法收敛，各项指标波动很大，对战效果也极差，之后将学习率调小整至 1×10^{-5}，训练效果有明显的提升。图 11 是加深网络后未修改学习率训练十分钟的监控图，图 12 是修改学习率后的监控图，可以看出明显的差异，图 12 在 13 分钟后胜率稳定在 1。

图 11

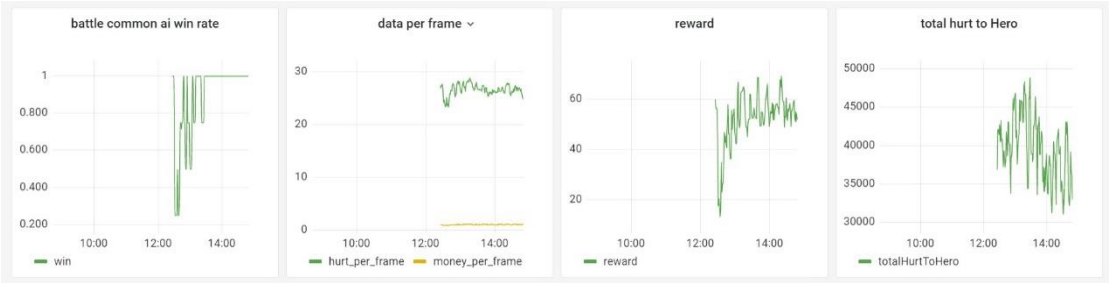

图 12

3） NoisyNet[5]

在强化学习中，避免过拟合的方法除了我们已经采用的 entropy loss 以外，还可以使用ε-greedy 策略在选择动作时采取一部分随机的动作。ε-greedy 策略运用到神经网络中就是在连接权重和偏置上加入参数化的噪音，使得输出的 Q function 有一定的随机性。具体操作是将参数 ω 看做是一个分布而不是一个数值，即$\omega \sim N(\mu^{\omega}, \sigma^{\omega})$。使用重参数化方法，先采样一个噪声$\varepsilon^{\omega} \sim N(0,1)$，即$\omega = \mu^{\omega} + \sigma^{\omega} \odot \varepsilon^{\omega}$，其中 ⊙表示矩阵中的元素逐一相乘，此时需要学习的参数是μ^{ω}和σ^{ω}, 需要学习的参数个数变为原来的 2 倍。同理, 偏置变为$b = \mu^{b} + \sigma^{b} \odot \varepsilon^{b}$, $\varepsilon^{b} \sim N(0,1)$。

将上述过程用图模型的形式表达如图 12。可以看到，虽然将确定的参数ω和 b 变为了高斯分布的形式，但通过重参数化的技巧，将随机采样ε的过程移动到输入端，变化后的 NoisyNet 还是可微的，能够通过梯度下降来训练。

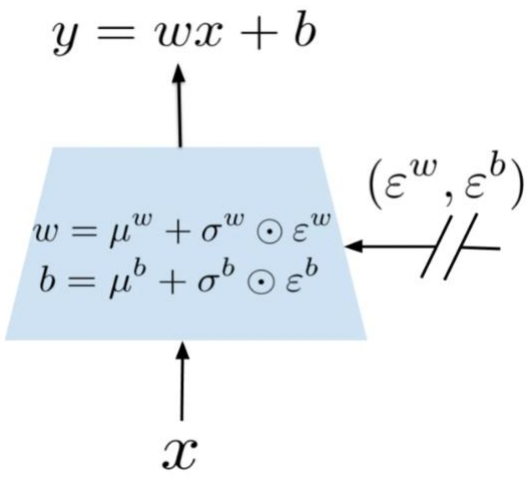

图 12

```python
# 从正态分布中采样高斯噪声
def _get_noise(self, shape):
        if len(np.shape(shape)==2):
            sample = np.random.multivariate_normal(np.zeros(shape[1]),
                np.eye(shape[1]), shape[0], 'raise')
        else:
            sample = np.random.multivariate_normal(np.zeros(shape[0]),
                np.eye(shape[0]), 1, 'raise')
            sample = sample.reshape(shape[0])
        return tf.Variable(sample)
```

```python
# w=w_mu+w_sigma*w_epsilon
fc2_label_weight_mu = self._fc_weight_variable(
    shape=[self.lstm_unit_size, self.label_size_list[index]],
    name="fc2_label_%d_weight_mu" % (index),
)
fc2_label_weight_sigma = self._fc_weight_variable(
    shape=[self.lstm_unit_size, self.label_size_list[index]],
    name="fc2_label_%d_weight_sigma" % (index),
)
fc2_label_weight_epsilon = self._get_noise(
    shape=[self.lstm_unit_size, self.label_size_list[index]]
)
fc2_label_weight = tf.add(
    fc2_label_weight_mu,
    tf.multiply(fc2_label_weight_sigma, fc2_label_weight_epsilon)
)
```

```
# b=b_mu+b_sigma*b_epsilon
fc2_label_bias_mu = self._bias_variable(
    shape=[self.label_size_list[index]],
    name="fc2_label_%d_bias_mu" % (index),
)
fc2_label_bias_sigma = self._bias_variable(
    shape=[self.label_size_list[index]],
    name="fc2_label_%d_bias_sigma" % (index),
)
fc2_label_bias_epsilon = self._get_noise(
    shape=[self.label_size_list[index]]
)
fc2_label_bias = tf.add(
    fc2_label_bias_mu,
    tf.multiply(fc2_label_bias_sigma, fc2_label_bias_epsilon)
)

# linear layer: output=W*X+b
fc2_label_result = tf.add(
    tf.matmul(reshape_lstm_outputs_result, fc2_label_weight),
    fc2_label_bias,
    name="fc2_label_%d_result" % (index),
)
```

代码 7

十、实验结论：

经过上述探索，我们总结出的最佳参数设置和训练方法如下：

网络结构：由于增加 LSTM 层或者向 FC 层中增加噪声后模型要训练更长时间才能达到较好的效果，我们最终没有采用这两种修改方案，只增加了 FC 层的层数。

Slow time：0.5

Learning rate：先用 learning rate = 0.0001 训练一段时间，当模型对 common ai 胜率在 0-1 摇摆时，暂停训练，调整 learning rate = 0.00001，继续训练直至对 common ai 胜率保持为 1。

Loss：先用原始 loss 训练一段时间，当模型对 common ai 胜率在 0-1 摇摆时，暂停训练，给 value_cost 乘上系数 0.01，继续训练直至对 common ai 胜率保持为 1。

Reward：模型训练起来之后，reward 是致胜的关键，reward 不能一次调整太多，要每隔一段时间逐步调整，比如：先调高 kill 的 reward，让模型学会击杀，再调高 money 的 reward，让模型学会提高经济，最后提高减少对方塔血量的 reward，让模型学会推塔。具体改动如下：

首先使用如下 reward

{

"reward_money": "0.008"（初始值 0.006），

"reward_exp": "0.008"（初始值 0.006），

"reward_hp_point": "2.0",

"reward_ep_rate": "0.8"（初始值 0.75），

"reward_kill": "-0.2"（初始值-0.6），

"reward_dead": "-1.0",

"reward_tower_hp_point": "8.0"（初始值 5.0），

"reward_last_hit": "0.5",

"log_level": "8"

}

之后，增加"reward_tower_hp_point"至"10.0"，再之后更改如下：

{

"reward_money": "0.014",

```
"reward_exp": "0.014" ,
"reward_hp_point": "2.1",
"reward_ep_rate": "0.8",
"reward_kill": "3.5",
"reward_dead": "-3.0",
"reward_tower_hp_point": "12.0",
"reward_last_hit": "0.75",
"log_level": "8"
}
```


之后再次提高"reward_tower_hp_point"的权重至 15，此时可以和 baseline0 打成平手。最后再次提高"reward_tower_hp_point"的权重至 18。

最终的对战结果如下：

每次测试与 baseline0 对战 20 局，13 次测试中我方的获胜局数分别为：15，15，14，18，16，12，14，17，15，19，14，14，13，测试 20 局的平均获胜局数为 15.08，最高获胜局数 19 局，与 baseline1 对战 20 局的最高获胜局数为 8 局，战况图如下：

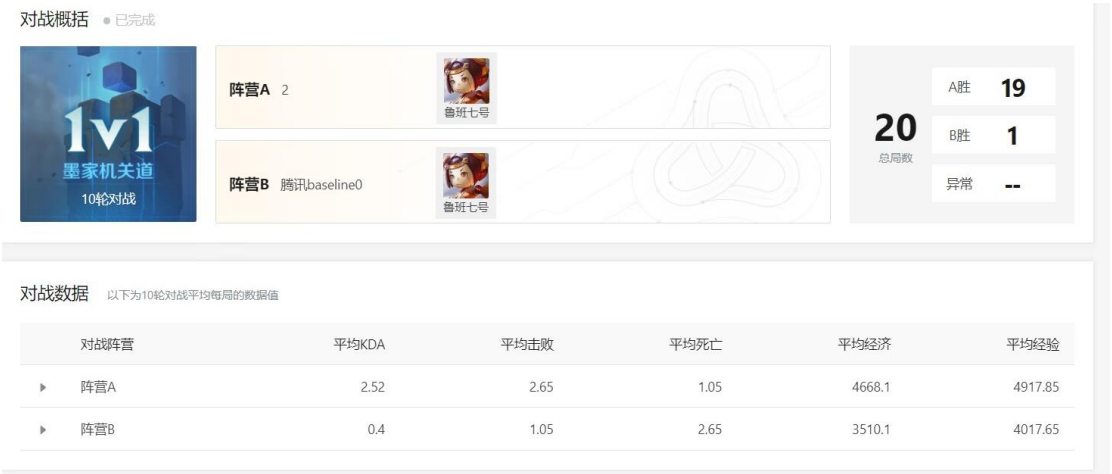

图 13

图 14

天梯赛结果（12.26）

排名	战队	成员	胜局 (总局数)
	猫猫队睡大觉-2	梁家萱	54 (60)
	aoer-3	王宇亭	35 (60)
	逻辑写不队-2	康智宏、纪明蔚	31 (60)
4	Apomorphism-2	吉志学	0 (60)

图 15

挑战赛结果（12.26）

战队	成员	挑战模型	胜局 (总局数)
aoer-3	王宇亭	腾讯baseline0	14 (20)

图 16

十一、总结及心得体会：

实验分工:

我们小组定的目标是在做实验探索的同时，尽可能提升模型的能力，即：在优先保证做的相关实验充足的前提下，使用我们在实验中得到的相关结论，使得模型训练的结果达到不错的效果。在实验过程中，小组同学一起讨论阅读提供的文档，并通过阅读相关的论文，解读了开悟平台提供的源代码。大家一起参与了调整超参数、修改网络结构的方案制定和测试实验（我们小组的成员共同制定好方案后，每人测试一种实验方案，并行进行）。此外，我还完成了 noisy net 部分的代码修改和讲解视频的录制。小组所有同学对于团队的贡献占比相同。

心得体会:

模型需要经过长时间的训练，消耗大量数据才会达到比较好的效果，刚开始的很多问题都是训练不充分导致的，比如回泉水的时机和停留的时长不恰当，鲁班会浪费比较多的时间空放攻击来积攒大招，不会放技能和躲避敌方的技能，这些问题都可以通过延长训练时间（在保证不过拟合的前提下）来解决。

当这些基本操作训练无误后，影响比赛结果的主要是双方的策略，这就需要每训练一段时间（我们组是 30h）后观看录像，根据录像调整 reward 后再进行下一阶段的训练。1V1 对战比较简单，一局的时间也很短，因此推塔在对战中起到了非常重要的决定胜负的作用，通过观看与 baseline1 的对战视频我们发现，我们训练出来的模型太过保守，前期清理兵线和回泉水加血的时间太长，被 baseline1 迅速地推了塔，后面我们增加了推塔的奖励，与 baseline1 的对战效果有了明显的改善。由此可知，在训练的前期可以不把某一动作的奖励提得太高，训练模型学会所有基本操作，之后再提高推塔等关键动作的奖励，促进模型学会最适宜的对战策略。

十二、对本实验过程及方法、手段的改进建议：

在开悟平台的框架中，样本是通过当前模型与自己或者与 common AI 模型对战产生的，当模型训练到比较强时，模型与 common AI 对战的胜率将保持在 1，模型就很难学到新的策略，此时可以将 common AI 替换成其他更强的模型来促进模型学到新知识。

Algorithm.py 中将四种 cost 都加入了列表中，但监视器上只显示出三种 cost，查看后端代码（路径为“D:\kaiwu-39\build\code\common\monitor\dashboard\GPU Monitor New2.json”）发现有命名和实际展示的内容不符的情况，但由于开

悟平台进行远程训练时后端部分并没有采用我们所提交的这份代码，我们无法修复该漏洞，希望助教学长能帮助我们向开悟的相关人员反馈并修复漏洞。

```
self.all_loss_list = [
    self.cost_all,
    self.value_cost,
    self.policy_cost,
    self.entropy_cost,
]
```

代码 8

```
{
  "alias": "total loss",
  "groupBy": [
    {
      "params": [
        "5m"
      ],
      "type": "time"
    },
    {
      "params": [
        "null"
      ],
      "type": "fill"
    }
  ],
  "measurement": "gpu_ip_info",
  "orderByTime": "ASC",
  "policy": "autogen",
  "query": "SELECT mean(\"entropy_cost\") FROM \"au
  "rawQuery": false,
  "refId": "B",
  "resultFormat": "time_series",
  "select": [
    [
      {
        "params": [
          "total_loss"
```

代码 9

参考文献：

[1] A. Juliani, “Simple Reinforcement Learning with Tensorflow Part 8: Asynchronous Actor-Critic Agents (A3C),” *Emergent // Future*, Jun. 30, 2017. https://medium.com/emergent-future/simple-reinforcement-learning-with-

tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2 (accessed Dec. 23, 2022).

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms.” arXiv, Aug. 28, 2017. Accessed: Dec. 25, 2022. [Online]. Available: http://arxiv.org/abs/1707.06347

[3] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, “Trust Region Policy Optimization.” arXiv, Apr. 20, 2017. Accessed: Dec. 25, 2022. [Online]. Available: http://arxiv.org/abs/1502.05477

[4] D. Ye *et al.*, “Mastering Complex Control in MOBA Games with Deep Reinforcement Learning.” arXiv, Dec. 15, 2020. Accessed: Dec. 23, 2022. [Online]. Available: http://arxiv.org/abs/1912.09729

[5] M. Fortunato *et al.*, “Noisy Networks for Exploration.” arXiv, Jul. 09, 2019. Accessed: Dec. 23, 2022. [Online]. Available: http://arxiv.org/abs/1706.10295

报告评分：

指导教师签字：