# 强化学习期末大作业

## 1 总体介绍

本次实验调研了PPO算法，介绍了从自然策略梯度算法、信赖域策略优化算法（TRPO）直到PPO算法的演进过程，以及算法迭代过程中的数学原理。并调研了PPO算法在多智能体情景下的应用IPPO。代码实现了：PPO Penalty以及PPO Clip算法，在Pendulum-v0环境中进行了实验；IPPO算法，在Combat 环境中进行了实验。

## 2.自然策略梯度算法

自然策略梯度算法揭露了传统策略梯度算法（如REINFORCE算法）的缺点以及补救的方法，尽管自然梯度已被TRPO和PPO等算法超越，但掌握它的基本原理对于理解这些当代RL算法至关重要。

### 2.1 传统策略梯度算法的缺陷

在传统的策略梯度算法中，我们根据目标函数梯度 $\nabla_\theta J(\theta)$ 和步长 $\alpha$ 更新策略权重 $\theta$ ，这样的更新过程可能会出现两个常见的问题:

- 过冲 (Overshooting) : 更新错过了奖励峰值并落入了次优策略区域
- 下冲 (Undershooting) : 在梯度方向上采取过小的更新步长会导致收敛缓慢

在监督学习问题中，overshooting并不是什么大问题，因为数据是固定的，我们可以在下一个epoch中重新纠正，但在强化学习问题中，如果因为overshooting陷入了一个较差的策略区域，则未来的样本批次可能不会提供太多有意义的信息，用较差的数据样本再去更新策略，从而陷入了糟糕的正反馈中无法恢复。较小的学习率可能会解决这个问题，但会导致收玫速度变慢的undershooting问题。

为了避免overshooting带来的严重后果，一种直觉方法是限制每次更新步长的上限:

$$\Delta\theta^* = \underset{\|\Delta\theta\| \leq \epsilon}{\operatorname{argmax}} J(\theta + \Delta\theta)$$

其中 $\|\triangle\theta\|$ 代表更新前后策略权重的欧式距离。
尽管听起来很合理，但实际效果并不是像我们预期的那样，原因是不同的分布对参数变化的敏感度是不同的。所以，只限制参数是不合理的，更应该考虑分布对参数变化的敏感度，传统的策略梯度算法无法考虑到这种曲率变化，我们需要引入二阶导数，这正是自然策略梯度相较于传统策略梯度算法的区别。

### 2.2 限制策略更新的差异

我们需要表示策略 (分布) 之间的差异，而不是参数本身的差异。计算两个概率分布之间的差异，最常见的是KL散度，也称为相对摘，描述了两个概率分布之间的距离：

$$\mathcal{D}_{\mathrm{KL}}\left(\pi_\theta \| \pi_{\theta+\Delta\theta}\right) = \sum_{x \in \mathcal{X}} \pi_\theta(x) \log\left(\frac{\pi_\theta(x)}{\pi_{\theta+}\Delta\theta(x)}\right)$$

调整后的策略更新限制为：

$$\Delta\theta^* = \underset{\mathcal{D}_{\mathrm{KL}}(\pi_\theta \| \pi_{\theta+\Delta\theta}) \leq \epsilon}{\operatorname{argmax}} J(\theta + \Delta\theta)$$

通过计算这个表达式，我们可以确保在参数空间中执行大更新的同时，保证策略本身的改变不超过间值。然而，计算KL散度需要遍历所有的状态-动作对，因此我们需要一些化简来处理现实的RL问题。首先，我们使用拉格朗日松驰将原表达式的发散约束转化为惩罚项，得到一个更容易求解的表达式:

$$\Delta\theta^* = \arg\max_{\Delta\theta} J(\theta+\Delta\theta) - \lambda\left(\mathcal{D}_{\mathrm{KL}}\left(\pi_\theta \| \pi_{\theta+\Delta\theta}\right) - \epsilon\right)$$

由于计算KL散度需要遍历所有的状态-动作对，我们必须用近似方法来化简。通过泰勒展开：

$$\Delta\theta^* \approx \arg\max_{\Delta\theta} J\left(\theta_{\text{old}}\right) + \nabla_\theta J(\theta)\big|_{\theta=\theta_{\text{old}}} \cdot \Delta\theta$$
$$-\frac{1}{2}\lambda\left(\Delta\theta^\top \nabla_\theta^2 \mathcal{D}_{\mathrm{KL}}\left(\pi_{\theta_{\text{old}}} \| \pi_\theta\right)\big|_{\theta=\theta_{\text{old}}} \Delta\theta\right) + \lambda\epsilon$$

目标函数近似于一阶泰勒展开（与散度相比，二阶展开可以忽略不计），KL散度近似于二阶泰勒展开 (零阶和一阶差分的计算结果为 0 )。我们可以进一步化简，通过

- 用费舍尔信息矩阵 (Fisher information matrix) 替换二阶KL导数
- 删除所有不依赖于 $\triangle\theta$ 的项

可以得到:

$$\Delta\theta^* \approx \arg\max_{\Delta\theta} \nabla_\theta J(\theta)\Big|_{\theta=\theta_{\text{old}}} \cdot \Delta\theta - \frac{1}{2}\lambda\left(\Delta\theta^\top F\left(\theta_{\text{old}}\right)\Delta\theta\right)$$

用费舍尔信息矩阵代替二阶导数的原因，除了符号紧凑性外，还可以大大减少计算开销。原先的Hessian矩阵是一个 $|\theta| \cdot |\theta|$ 的矩阵，每个元素都是二阶导数，完整的计算可能非常麻烦，而对于Fisher信息矩阵，有一个替代表达式：

$$F(\theta) = \mathbb{E}_\theta\left[\nabla_\theta \log \pi_\theta(x) \nabla_\theta \log \pi_\theta(x)^\top\right]$$

可以表示为策略梯度的外积，在局部等效于Hessian，同时计算效率更高。

## 2.3 解决KL约束问题

对于近似简化后的表达式，可以通过将关于 $\triangle\theta$ 的梯度设置为 0 ，来找到最优的权重更新 $\triangle\theta$ :

$$\begin{aligned}
0 &= \frac{\partial}{\partial\Delta\theta}\left(\nabla_\theta J(\theta)\Delta\theta - \frac{1}{2}\lambda\Delta\theta^\top F(\theta)\Delta\theta\right) \\
&= \nabla_\theta J(\theta) - \frac{1}{2}\lambda F(\theta)\Delta\theta \\
\therefore \lambda F(\theta)\Delta\theta &= -2\nabla_\theta J(\theta) \\
\Delta\theta &= -\frac{2}{\lambda} F(\theta)^{-1}\nabla_\theta J(\theta)
\end{aligned}$$

其中， $\lambda$ 是一个常数，可以吸收到学习率 $\alpha$ 中。根据 $\mathcal{D}_{\mathrm{KL}}\left(\pi_\theta \| \pi_{\theta+\Delta\theta}\right) \leq \epsilon$ ，我们可以推出动态学习率:

$$\alpha = \sqrt{\frac{2\epsilon}{\nabla J(\theta)^\top} F(\theta)^{-1}\nabla J(\theta)}$$

可以确保每次更新的KL散度（近似）等于 $\epsilon$ 。
最后，我们提取自然策略梯度，它是针对流形曲率校正的梯度：

$$\tilde{\nabla} J(\theta) = F(\theta)^{-1}\nabla J(\theta)$$

这种自然策略梯度在距离约束内给出了黎曼空间中最陡的下降方向，而不是传统上假设的欧几里德空间中的最陡下降方向。与传统的策略梯度相比，唯一的区别是与逆Fisher矩阵相乘。

最终的权重更新方案为：

$$\Delta\theta = \sqrt{\frac{2\epsilon}{\nabla J(\theta)^\top F(\theta)^{-1}\nabla J(\theta)}}\tilde{\nabla} J(\theta)$$

该方案的强大之处在于，无论分布的表示如何，它总是以相同的幅度改变策略。

## 2.4 算法实现

自然策略梯度算法的完整概述伪代码如下,其中策略梯度和Fisher矩阵在实际计算时都是使用样本估计的。

**Algorithm 1** Natural Policy Gradient

Input: initial policy parameters $\theta_0$
**for** $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
  Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
  Form sample estimates for
  - policy gradient $\hat{g}_k$ (using advantage estimates)
  - and KL-divergence Hessian / Fisher Information Matrix $\hat{H}_k$

  Compute Natural Policy Gradient update:

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{\hat{g}_k^T \hat{H}_k^{-1} \hat{g}_k}} \hat{H}_k^{-1} \hat{g}_k$$

**end for**

最终结果在两个方面不同于传统的策略梯度算法:

- 考虑到策略对局部变化的敏感性，策略梯度由逆Fisher矩阵校正，而传统的梯度方法假定更新为欧几里得距离。
- 更新步长 $\alpha$ 具有适应梯度和局部敏感性的动态表达式，确保无论参数化如何，策略变化幅度为 $\epsilon$。在传统方法中， $\alpha$ 通常设置为一些标准值，如0.1或0.01。

# 3. 信赖域策略优化算法 (TRPO)

Trust region policy optimization（TPRO）算法是现代强化学习的基础，它以自然策略梯度优化为基础，迅速获得普及，成为主流强化学习算法，因为它在经验上比自然策略梯度算法表现得更好、更稳定。尽管此后它已被近端策略优化 (PPO) 超越，但它的仍然具有重要的意义。

我们将讨论TRPO背后的单调改进定理（关注直觉）以及将其与自然策略梯度区分开的三个变化。

## 3.1 自然策略梯度算法的缺陷

- 近似值可能会违反KL约束，从而导致分析得出的步长过大，超出限制要求
- 矩阵 $F^{-1}$ 的计算时间太长，是 $O\left(N^3\right)$ 复杂度的运算

## 3.2 算法理论

针对自然策略梯度算法的问题，我们希望可以对策略的优化进行量化，从而保证每次的更新一定是优化作用的。为此，我们需要计算两种策略之间预期回报的差异。这里采用的是原策略预期回报添加新策略预期优势的方式。该表达式在原策略下计算优势函数，无需重新采样：

$$J\left(\pi_{\theta+\Delta\theta}\right) = J\left(\pi_\theta\right) + \mathbb{E}_{\tau\sim\pi_{\theta+}} \Delta\theta \sum_{t=0}^{\infty} \gamma^t A^{\pi_\theta}\left(s_t, a_t\right)$$

其中优势函数的定义为: $A^{\pi_\theta}(s,a) = \mathbb{E}\left(Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)\right)$
在给定的策略和状态下，计算特定动作 $a$ 的期望罗积奖励与总体期望值的差值，描述了该动作的相对吸引力.

由于时间范围是无限的，引入状态的折扣分布:

$$\rho_{\pi}(s)=\left(P\left(s_{0}=s\right)+\gamma P\left(s_{1}=s\right)+\gamma^{2} P\left(s_{2}=s\right)+\ldots\right)$$

原差异表达式可重新表示为:

$$J\left(\pi_{\theta+\Delta\theta}\right)=J\left(\pi_{\theta}\right)+\sum_{s\in\mathcal{S}}\rho_{\pi_{\theta+}\Delta\theta}(s)\sum_{a\in\mathcal{A}}\pi_{\theta+\Delta\theta}(a\mid s)A^{\pi_{\theta}}(s,a)$$

我们无法在不对更新策略进行采样的情况下知道对应于更新策略的状态分布，故引入近似误差，使用当前策略近似:

$$J\left(\pi_{\theta+\Delta\theta}\right)\approx J\left(\pi_{\theta}\right)+\sum_{s\in\mathcal{S}}\rho_{\pi_{\theta}}(s)\sum_{a\in\mathcal{A}}\pi_{\theta+\triangle\theta}(a\mid s)A^{\pi_{\theta}}(s,a)$$

接下来，我们将状态分布求和替换为期望，方便实际计算时使用蒙特卡洛模拟进行采样，同时将动作求和替换为重要性采样。通过重要性采样，可以有效利用当前策略的行动期望，并针对新策略下的概率进行了修正:

$$J\left(\pi_{\theta+\Delta\theta}\right)\approx J\left(\pi_{\theta}\right)+\mathbb{E}_{s\sim\rho_{\pi\theta}}\frac{\pi_{\theta+\Delta\theta}(a\mid s)}{\pi_{\theta}(a\mid s)}A^{\pi_{\theta}}(s,a)$$

描述更新策略相对于原策略的预期优势称为替代优势（surrogate advantage）：

$$J\left(\pi_{\theta+\Delta\theta}\right)-J\left(\pi_{\theta}\right)\approx\mathbb{E}_{s\sim\rho_{\pi\theta}}\frac{\pi_{\theta+\Delta\theta}(a\mid s)}{\pi_{\theta}(a\mid s)}A^{\pi_{\theta}}(s,a)=\mathcal{L}_{\pi_{\theta}}\left(\pi_{\theta+\Delta\theta}\right)$$

论文中推导出 $C$ 的值以及目标函数改进的下限。如果我们改进右侧，可以保证左侧也得到改进。本质上，如果替代优势 $\mathcal{L}_{\pi_{\theta}}\left(\pi_{\theta+\Delta\theta}\right)$ 超过最坏情况下的近似误差 $C\mathcal{D}_{KL}^{\max}\left(\pi_{\theta}\|\pi_{\theta+\Delta\theta}\right)$ ，我们一定会改进目标。

这就是单调改进定理。相应的过程是最小化最大化算法 (MM) 。即如果我们改进下限，我们也会将目标改进至少相同的量。

## 3.3 算法实现

在实际的算法实现方面，TRPO和自然策略梯度算法没有太大的区别。TRPO主要有三个改进，每个改进都解决了原始算法中的一个问题。TRPO的核心是利用单调改进定理，验证更新是否真正改进了我们的策略。

### 3.3.1 共轭梯度法 (conjugate gradient method)

在自然策略梯度算法中，计算逆Fisher矩阵是一个耗时且数值不稳定的过程，特别是对于神经网络，参数矩阵可以变得非常大，$O\left(\theta^{3}\right)$ 的时间复杂度将无法计算。

好消息是，我们对逆矩阵本身并不感兴趣。观察自然策略梯度的方程式，如果我们可以直接得到乘积 $F^{-1}\nabla\log\pi_{\theta}(x)$ ，就不再需要逆。

引入共轴梯度法，这是一个近似上式乘积的数值过程，这样我们就可以避免计算逆矩阵。共轭梯度通常在 $|\theta|$ 步内收敛，从而可以处理大矩阵。

共轭梯度法并不是TRPO中独有的，例如在Truncated Natural Policy Gradient中也部署了相同的方法，但它仍然是算法中的一个重要组成部分。

### 3.3.2 线搜索 (line search)

虽然自然梯度策略中提供了给定KL散度约束的最佳步长，但由于存在较多的近似值，实际上可能不满足该约束。

TRPO 通过执行线搜索来解决此问题，通过不断地迭代减小更新的大小，直到第一个不违反约束的更新。这个过程可以看作是不断缩小信任区域，即我们相信更新可以实际改进目标的区域。

**Algorithm 2** Line Search for TRPO

Compute proposed policy step $\Delta_k = \sqrt{\frac{2\delta}{\hat{g}_k^T \hat{H}_k^{-1} \hat{g}_k}} \hat{H}_k^{-1} \hat{g}_k$
**for** $j = 0, 1, 2, ..., L$ **do**
  Compute proposed update $\theta = \theta_k + \alpha^j \Delta_k$
  **if** $\mathcal{L}_{\theta_k}(\theta) \geq 0$ and $\bar{D}_{KL}(\theta || \theta_k) \leq \delta$ **then**
    accept the update and set $\theta_{k+1} = \theta_k + \alpha^j \Delta_k$
    break
  **end if**
**end for**

更新迭代减小的方式是使用指数衰减率 $\alpha^j$ ， $j$ 随迭代的进行而增加。同时，在约束条件的判断中，我们看到KL散度约束不是唯一满足的约束，它还保证了替代优势 $\mathcal{L}(\theta)$ 非负，这就是接下来介绍的改进检查。

### 3.3.3 改进检查

在TRPO中，我们并没有假设更新会提高替代优势 $\mathcal{L}(\theta)$ ，而是真正检查了它。尽管实际计算时需要根据旧策略计算优势，以及使用重要性抽样来调整概率，会花费一些时间，但验证更新是否真正改进了策略是有必要的。

### 3.3.4 算法完整框架

**Algorithm 1** Trust Region Policy Optimization

1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: Hyperparameters: KL-divergence limit $\delta$, backtracking coefficient $\alpha$, maximum number of backtracking steps $K$
3: **for** $k = 0, 1, 2, ...$ **do**
4: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
5: Compute rewards-to-go $\hat{R}_t$.
6: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
7: Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

8: Use the conjugate gradient algorithm to compute

$$\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$$

where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.
9: Update the policy by backtracking line search with

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$$

where $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.
10: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
11: **end for**

TRPO执行了共轭梯度算法、约束样本KL散度的线搜索和检查改进替代优势，相较于自然策略梯度算法有了较大的改进，代表了自然策略梯度发展的一个重要里程碑。

# 4. 近端策略优化算法（PPO）

## 4.1 TRPO算法的缺陷

TRPO算法解决了许多与自然策略梯度相关的问题，并获得了RL社区的广泛采用。但是，TRPO仍然存在一些缺点，特别是：

- 无法处理大参数矩阵：尽管使用了共轭梯度法，TRPO仍然难以处理大的 Fisher矩阵，即使它们不需要求逆
- 二阶优化很慢：TRPO的实际实现是基于约束的，需要计算上述Fisher矩阵，这大大减慢了更新过程。此外，我们不能利用一阶随机梯度优化器，例如ADAM
- TRPO 很复杂：TRPO很难解释、实现和调试。当训练没有产生预期的结果时，确定如何提高性能可能会很麻烦

在TRPO的基础上，Schulman等人引入了近端策略优化算法PPO (Proximal Policy Optimization)。有两种主要的PPO变体需要讨论（均在17年的论文中介绍）：PPO Penalty和PPO Clip。我们首先从PPO Penalty入手，它在概念上最接近TRPO。

## 4.2 PPO Penalty

TRPO在理论分析上推导出与KL散度相乘的惩罚项，但在实践中，这种惩罚往往过于严格，只产生非常小的更新。因此，问题是如何可靠地确定缩放参数 $\beta$，同时避免overshooting:

$$\Delta\theta^* = \underset{\Delta\theta}{\operatorname{argmax}}\, \mathcal{L}_{\theta+\Delta\theta}(\theta+\Delta\theta) - \beta\left(\mathcal{D}_{\mathrm{KL}}\left(\pi_\theta \| \pi_{\theta+\Delta\theta}\right)\right)$$

难点是很难确定适用于多个问题的某个 $\beta$ 值。事实上，即使是同一个问题，随着时间的推移，特征也可能发生变化。我们既不希望 $\beta$ 过小，与TRPO一样只产生较小的更新，也不希望 $\beta$ 过大，容易出现overshooting问题。

PPO通过设置目标散度 $\delta$ 的方式解决了这个问题，希望我们的每次更新都位于目标散度附近的某个地方。目标散度应该大到足以显著改变策略，但又应该小到足以使更新稳定。

每次更新后，PPO都会检查更新的大小。如果最终更新的散度超过目标散度的 1.5 倍，则下一次迭代我们将加倍 $\beta$ 来更加重惩罚。相反，如果更新太小，我们将 $\beta$ 减半，从而有效地扩大信任区域。迭代更新的思路与TRPO线搜索有一些相似之处，但PPO搜索是在两个方向上都有效的，而TRPO是单向减小的。

这里基于超过目标值的 1.5 倍将 $\beta$ 加倍或减半并不是数学证明的结果，而是基于启发式确定的。我们会不时违反约束条件，但通过调整 $\beta$ 可以很快地纠正它。根据经验，PPO对数值设置是非常不敏感的。总之，我们牺牲了一些数学上的严谨性来使实际的效果更好。

**Algorithm 4** PPO with Adaptive KL Penalty

Input: initial policy parameters $\theta_0$, initial KL penalty $\beta_0$, target KL-divergence $\delta$
**for** $k = 0, 1, 2, \ldots$ **do**
  Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
  Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
  Compute policy update

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}_{\theta_k}(\theta) - \beta_k \bar{D}_{KL}(\theta||\theta_k)$$

  by taking $K$ steps of minibatch SGD (via Adam)
  **if** $\bar{D}_{KL}(\theta_{k+1}||\theta_k) \geq 1.5\delta$ **then**
    $\beta_{k+1} = 2\beta_k$
  **else if** $\bar{D}_{KL}(\theta_{k+1}||\theta_k) \leq \delta/1.5$ **then**
    $\beta_{k+1} = \beta_k/2$
  **end if**
**end for**

与自然策略梯度和TRPO算法相比，PPO更容易实现。同时，我们可以使用流行的随机梯度下降算法（如ADAM等）执行更新，并在更新太大或太小时调整惩罚。总之，PPO比TRPO更容易使用，同时不失竞争力。PPO的第二种变体PPO Clip将会做的比PPO Penalty更好。

## 4.3 PPO Clip

Clipped PPO是目前最流行的PPO的变体，也是我们说PPO时默认的变体。PPO Clip相比于PPO Penalty效果更好，也更容易实现。

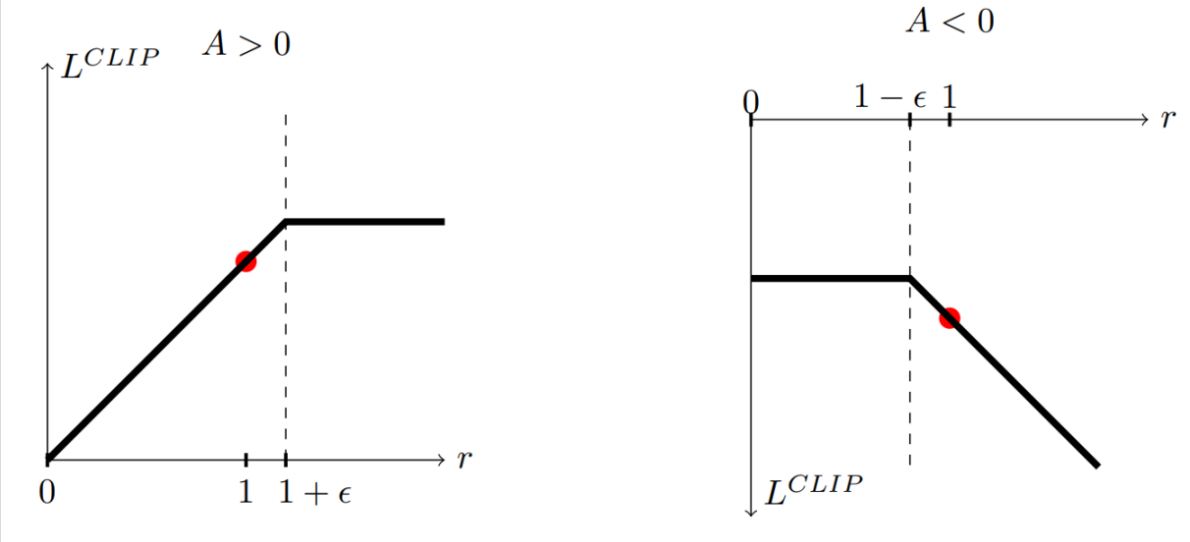
与PPO Penalty不同，与其费心随着时间的推移改变惩罚，PPO Clip直接限制策略可以改变的范围。我们重新定义了替代优势：

$$= \mathbb{E}_{\tau \sim \pi_\theta} \left[ \sum_{t=0}^{T} \left[ \min \left( \rho_t \left( \pi_\theta, \pi_{\theta_k} \right) A_t^{\pi_{\theta_k}}, \text{clip} \left( \rho_t \left( \pi_\theta, \pi_{\theta_k} \right), 1-\epsilon, 1+\epsilon \right) A_t^{\pi_{\theta_k}} \right) \right] \right]$$

其中， $\rho_t$ 为重要性采样:

$$\rho_t(\theta) = \frac{\pi_\theta \left( a_t \mid s_t \right)}{\pi_{\theta_k} \left( a_t \mid s_t \right)}$$

clip为截断函数，当重要性采样超出规定的上或下限后，函数会返回对应的上或下限。



项 $(1-\epsilon)A$ 和 $(1+\epsilon)A$ 不依赖于 $\theta$，产生的梯度为 0。因此，可信区域之外的样本被有效地抛出，避免过大的更新。我们没有明确地限制策略更新本身，只是简单地忽略了过度偏离策略所带来的优势。和 PPO Penalty一样，我们可以使用像ADAM等优化器来执行更新。PPO Clip中根据优势的正负性以及重要性采样的范围，可以分为六种可能的案例，在下图中对六种案例进行了总结。

| $p_t(\theta)\ > 0$ | $A_t$ | Return Value of $min$ | Objective is Clipped | Sign of Objective | Gradient |
|---|---|---|---|---|---|
| $p_t(\theta) \in [1-\epsilon, 1+\epsilon]$ | + | $p_t(\theta)A_t$ | no | + | ✓ |
| $p_t(\theta) \in [1-\epsilon, 1+\epsilon]$ | − | $p_t(\theta)A_t$ | no | − | ✓ |
| $p_t(\theta) < 1-\epsilon$ | + | $p_t(\theta)A_t$ | no | + | ✓ |
| $p_t(\theta) < 1-\epsilon$ | − | $(1-\epsilon)A_t$ | yes | − | **0** |
| $p_t(\theta) > 1+\epsilon$ | + | $(1+\epsilon)A_t$ | yes | + | **0** |
| $p_t(\theta) > 1+\epsilon$ | − | $p_t(\theta)A_t$ | no | − | ✓ |

为了实现想要达到的效果，我们应该调整 $\epsilon$，作为对KL散度的隐式限制。根据经验，$\epsilon = 0.1$ 和 $\epsilon = 0.2$ 是实际效果较好的值。尽管这些值与自然策略梯度的理论基础有些偏差（该理论建立在假设 $\pi_\theta = \pi_{\theta_k}$ 的局部近似值之上），但在实际运行时的效果较好。

**Algorithm 5** PPO with Clipped Objective

Input: initial policy parameters $\theta_0$, clipping threshold $\epsilon$
**for** $k = 0, 1, 2, \ldots$ **do**
Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
Compute policy update

$$\theta_{k+1} = \arg\max_\theta \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

by taking $K$ steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_k}\left[\sum_{t=0}^{T}\left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \mathrm{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k})\right]\right]$$

**end for**

# 5. IPPO（Independent PPO）

多智能体的情形相比于单智能体更加复杂，因为每个智能体在和环境交互的同时也在和其他智能体进行直接或者间接的交互。因此，多智能体强化学习要比单智能体更困难，其难点主要体现在以下几点:

- 由于多个智能体在环境中进行实时动态交互，并且每个智能体在不断学习并更新自身策略，因此在每个智能体的视角下，环境是非稳态的（non-stationary），即对于一个智能体而言，即使在相同的状态下采取相同的动作，得到的状态转移和奖励信号的分布可能在不断改变；
- 多个智能体的训练可能是多目标的，不同智能体需要最大化自己的利益；
- 训练评估的复杂度会增加，可能需要大规模分布式训练来提高效率。
- 本次调研只研究一种简单的情形，即完全去中心化方法：与完全中心化方法相反的范式便是假设每个智能体都在自身的环境中独立地进行学习，不考虑其他智能体的改变。完全去中心化方法直接对每个智能体用一个单智能体强化学习算法来学习。每个智能体看到的可能不是全局的状态。这样做的缺点是环境是非稳态的，训练的收玫性不能得到保证，但是这种方法的好处在于随着智能体数量的增加有比较好的扩展性，不会遇到维度灾难而导致训练不能进行下去。

IPPO就是一个经典完全去中心化算法。伪代码如下:

- 对于 N 个智能体，为每个智能体初始化各自的策略以及价值函数
- for 训练轮数 do

所有智能体在环境中交互分别获得各自的一条轨迹数据
对每个智能体，基于当前的价值函数计算优势函数A的估计
对每个智能体，通过最大化其 PPO的目标来更新其策略
对每个智能体，通过均方误差损失函数优化其价值函数

- end for

# 6. 算法代码实现

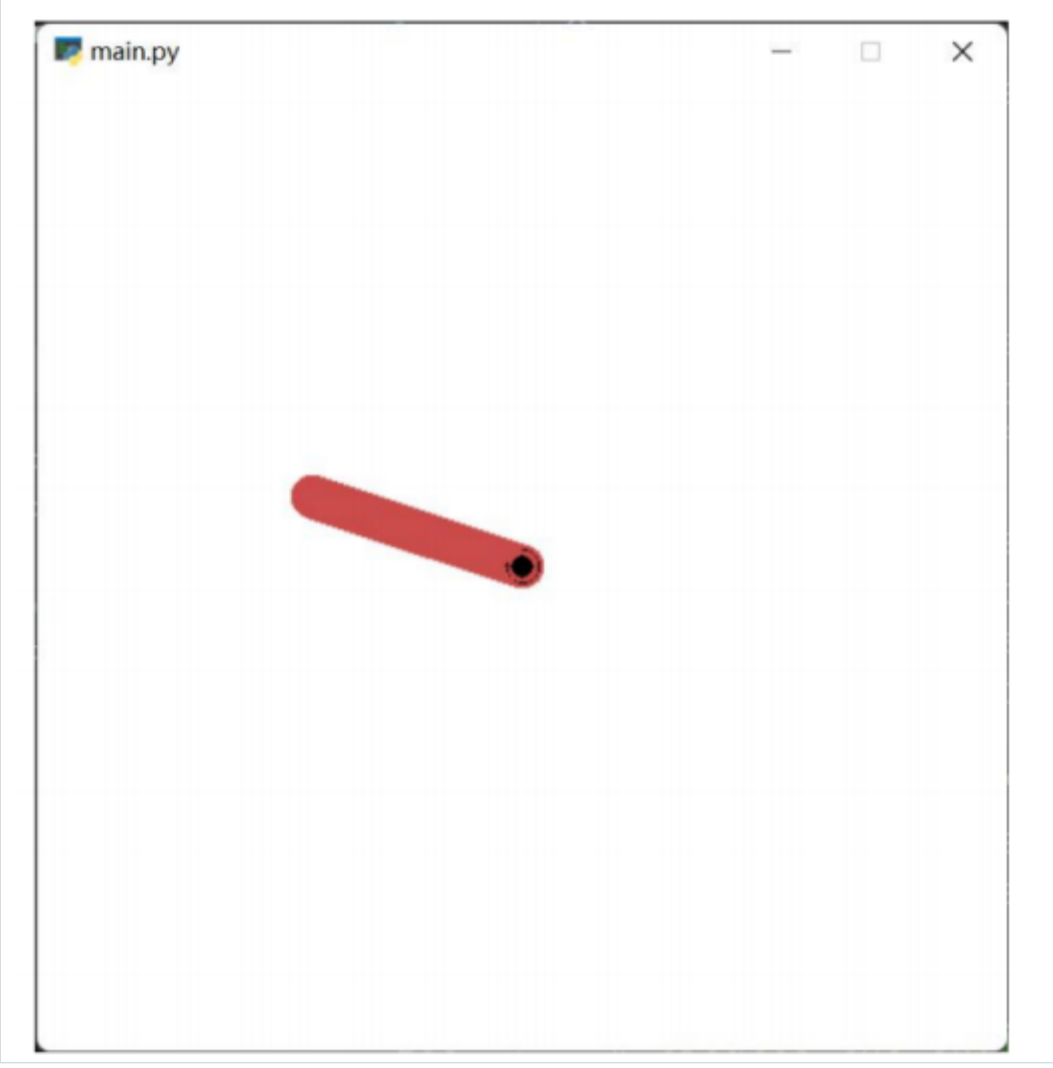
## 6.1 PPO

对于ppo，本次实验实现了PPO Penalty以及PPO Clip算法，在Pendulum-v0环境中进行了对比实验

## Pendulum-v0环境

即倒立摆问题。是控制文献中的经典问题。在这个版本的问题中，钟摆以随机位置开始，目标是将其向上摆动，使其保持直立。

类型：连续控制问题




Pendulum 三要素：

x-y：钟摆末端的笛卡尔坐标，单位是 m(米)。

theta：角度，单位是弧度。

tau：扭矩力，单位为 N/m。逆时针是正方向

## Observation Space

| Num | Observation | Min | Max |
|---|---|---|---|
| 0 | x = cos(theta) | -1.0 | 1.0 |
| 1 | y = sin(theta) | -1.0 | 1.0 |
| 2 | Angular Velocity | -8.0 | 8.0 |

摆锤自由端的 x-y 坐标和它的角速度。

## Action space

| Num | Action | Min | Max |
|---|---|---|---|
| 0 | Torque | -2.0 | 2.0 |

## Reward

奖励函数的定义如下：

$$r = -\left(\text{theta}^{2} + 0.1 \times \text{theta\_dt}^{2} + 0.001 * \text{torque}^{2}\right)$$

其中 theta 是摆锤的角度，初始值[-pi, pi]，之后经过 normalize 处理。(值为 0 表示在竖直状态)。

## 代码实现

训练参数设定：

```
import gym
import numpy as np
import tensorflow as tf

class PPO:
    def __init__(self, ep, batch, t='ppo2'):
        self.t = t
```

```
        self.ep = ep
        self.batch = batch
        self.log = 'model/{}_log'.format(t)
        self.env = gym.make('Pendulum-v0')
        self.bound = self.env.action_space.high[0]
        self.gamma = 0.9
        self.A_LR = 0.0001
        self.C_LR = 0.0002
        self.A_UPDATE_STEPS = 10
        self.C_UPDATE_STEPS = 10
        self.kl_target = 0.01
        self.lam = 0.5
        self.epsilon = 0.2
        self.sess = tf.compat.v1.Session()
        self.build_model()
```

模型的设定：

```
    def _build_critic(self):
        """Critic model."""
        with tf.variable_scope('critic'):
            x = tf.layers.dense(self.states, 100, tf.nn.relu)
            self.v = tf.layers.dense(x, 1)
            self.advantage = self.dr - self.v

    def _build_actor(self, name, trainable):
        """Actor model."""
        with tf.variable_scope(name):
            x = tf.layers.dense(self.states, 100, tf.nn.relu,
trainable=trainable)
            mu = self.bound * tf.layers.dense(x, 1, tf.nn.tanh,
trainable=trainable)
            sigma = tf.layers.dense(x, 1, tf.nn.softplus,
trainable=trainable)
            norm_dist = tf.distributions.Normal(loc=mu, scale=sigma)
            params = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
scope=name)
            return norm_dist, params

    def build_model(self):
        """Build model with PPO loss."""
        # Inputs
        self.states = tf.placeholder(tf.float32, [None, 3], 'states')
        self.action = tf.placeholder(tf.float32, [None, 1], 'action')
        self.adv = tf.placeholder(tf.float32, [None, 1], 'advantage')
        self.dr = tf.placeholder(tf.float32, [None, 1], 'discounted_r')

        # Build model
        self._build_critic()
        nd, pi_params = self._build_actor('actor', trainable=True)
```

```
        old_nd, oldpi_params = self._build_actor('old_actor',
trainable=False)

        # Define PPO loss
        with tf.variable_scope('loss'):
            # Critic loss
            self.closs = tf.reduce_mean(tf.square(self.advantage))
            # Actor loss
            with tf.variable_scope('surrogate'):
                ratio = tf.exp(nd.log_prob(self.action) -
old_nd.log_prob(self.action))
                surr = ratio * self.adv
                if self.t == 'ppo1':
                    self.tflam = tf.placeholder(tf.float32, None, 'lambda')
                    kl = tf.distributions.kl_divergence(old_nd, nd)
                    self.kl_mean = tf.reduce_mean(kl)
                    self.aloss = -(tf.reduce_mean(surr - self.tflam * kl))
                else:
                    self.aloss = -tf.reduce_mean(tf.minimum(
                        surr, tf.clip_by_value(ratio, 1. - self.epsilon, 1. +
self.epsilon) * self.adv))

        # Define optimizer
        with tf.variable_scope('optimize'):
            self.ctrain_op =
tf.train.AdamOptimizer(self.C_LR).minimize(self.closs)
            self.atrain_op =
tf.train.AdamOptimizer(self.A_LR).minimize(self.aloss)

        with tf.variable_scope('sample_action'):
            self.sample_op = tf.squeeze(nd.sample(1), axis=0)

        # Update old actor
        with tf.variable_scope('update_old_actor'):
            self.update_old_actor = [oldp.assign(p) for p, oldp in
zip(pi_params, oldpi_params)]

        self.sess.run(tf.global_variables_initializer())
```

模型的更新：

```
    def update(self, states, action, dr):
        """Update model."""
        self.sess.run(self.update_old_actor)
        adv = self.sess.run(self.advantage, {self.states: states, self.dr:
dr})

        # Update actor
        if self.t == 'ppo1':
            for _ in range(self.A_UPDATE_STEPS):
```

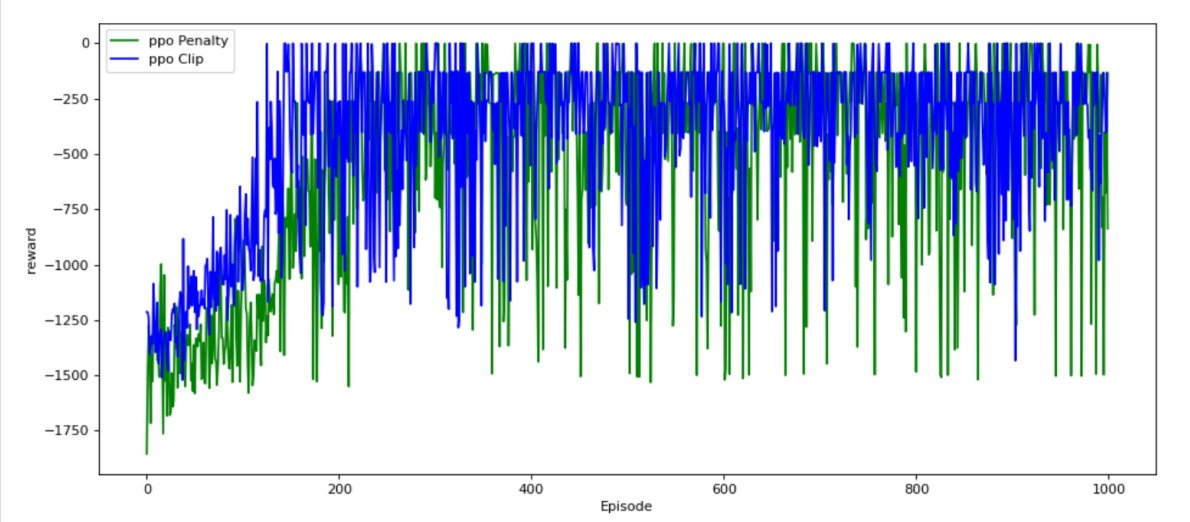
```
                _, kl = self.sess.run(
                    [self.atrain_op, self.kl_mean],
                    {self.states: states, self.action: action, self.adv: adv,
self.tflam: self.lam})
                if kl < self.kl_target / 1.5:
                    self.lam /= 2
                elif kl > self.kl_target * 1.5:
                    self.lam *= 2
        else:
            for _ in range(self.A_UPDATE_STEPS):
                self.sess.run(self.atrain_op, {self.states: states,
self.action: action, self.adv: adv})

        # Update critic
        for _ in range(self.C_UPDATE_STEPS):
            self.sess.run(self.ctrain_op, {self.states: states, self.dr: dr})
```

训练过程：

```
    def train(self):
        """Train method."""
        tf.reset_default_graph()
        history = {'episode': [], 'Episode_reward': []}
        for i in range(self.ep):
            observation = self.env.reset()
            states, actions, rewards = [], [], []
            episode_reward = 0
            j = 0
            while True:
                a = self.choose_action(observation)
                next_observation, reward, done, _ = self.env.step(a)
                states.append(observation)
                actions.append(a)
                episode_reward += reward
                rewards.append((reward + 8) / 8)
                observation = next_observation
                if (j + 1) % self.batch == 0:
                    states = np.array(states)
                    actions = np.array(actions)
                    rewards = np.array(rewards)
                    d_reward = self.discount_reward(states, rewards,
next_observation)
                    self.update(states, actions, d_reward)
                    states, actions, rewards = [], [], []
                if done:
                    break
                j += 1
            history['episode'].append(i)
            history['Episode_reward'].append(episode_reward)
```

```
            print('Episode: {} | Episode reward: {:.2f}'.format(i,
episode_reward))
        return history
```

## 结果展示



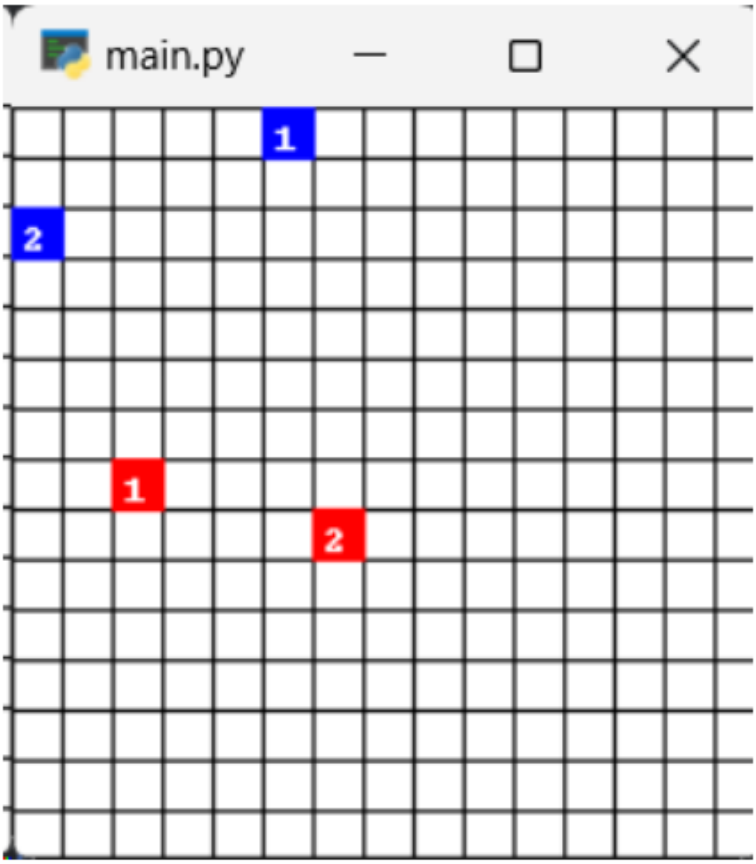

可以看到两种ppo算法都能收敛到最大的reward，但ppo Clip收敛的速度比较快。

## 6.2 IPPO

### 环境概述

本次实验是采用ma_gym库中的 Combat 环境。Combat 是一个在二维的格子世界上进行的两个队伍的对战模拟游戏，每个智能体的动作集合为：向四周移动格，攻击周围格3*3范围内其他敌对智能体，或者不采取任何行动。起初每个智能体有 3 点生命值，如果智能体在敌人的攻击范围内被攻击到了，则会扣 1 生命值，生命值掉为 0 则死亡，最后存活的队伍获胜。每个智能体的攻击有一轮的冷却时间。

在游戏中，我能够控制一个队伍的所有智能体与另一个队伍的智能体对战。另一个队伍的智能体使用固定的算法：攻击在范围内最近的敌人，如果攻击范围内没有敌人，则向敌人靠近。显然，每个智能体的目标函数相同，且可以共享策略网络以及参数，因为他们是同质的，即功能行动完全一样。




## 代码实现

### 模型定义

```
import torch
import torch.nn.functional as F

class PolicyNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim, action_dim):
        super(PolicyNet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = torch.nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc2(F.relu(self.fc1(x))))
        return F.softmax(self.fc3(x), dim=1)

class ValueNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim):
        super(ValueNet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = torch.nn.Linear(hidden_dim, 1)

    def forward(self, x):
        x = F.relu(self.fc2(F.relu(self.fc1(x))))
        return self.fc3(x)
```

ppo算法实现（clip版本）：

```
import torch
import torch.nn.functional as F

class PPO:
    ''' PPO算法,采用截断方式 '''
    def __init__(self, state_dim, hidden_dim, action_dim, actor_lr,
critic_lr, lmbda, eps, gamma, device):
        self.actor = PolicyNet(state_dim, hidden_dim, action_dim).to(device)
        self.critic = ValueNet(state_dim, hidden_dim).to(device)
        self.actor_optimizer = torch.optim.Adam(self.actor.parameters(),
lr=actor_lr)
        self.critic_optimizer = torch.optim.Adam(self.critic.parameters(),
lr=critic_lr)
        self.gamma = gamma
        self.lmbda = lmbda
        self.eps = eps  # PPO中截断范围的参数
        self.device = device

    def take_action(self, state):
        state = torch.tensor([state], dtype=torch.float).to(self.device)
        probs = self.actor(state)
        action_dist = torch.distributions.Categorical(probs)
        action = action_dist.sample()
        return action.item()

    def update(self, transition_dict):
        states = torch.tensor(transition_dict['states'],
dtype=torch.float).to(self.device)
        actions = torch.tensor(transition_dict['actions']).view(-1,
1).to(self.device)
        rewards = torch.tensor(transition_dict['rewards'],
dtype=torch.float).view(-1, 1).to(self.device)
        next_states = torch.tensor(transition_dict['next_states'],
dtype=torch.float).to(self.device)
        dones = torch.tensor(transition_dict['dones'],
dtype=torch.float).view(-1, 1).to(self.device)

        td_target = rewards + self.gamma * self.critic(next_states) * (1 -
dones)
        td_delta = td_target - self.critic(states)
        advantage = compute_advantage(self.gamma, self.lmbda,
td_delta.cpu()).to(self.device)

        old_log_probs = torch.log(self.actor(states).gather(1,
actions)).detach()
        log_probs = torch.log(self.actor(states).gather(1, actions))
        ratio = torch.exp(log_probs - old_log_probs)
        surr1 = ratio * advantage
        surr2 = torch.clamp(ratio, 1 - self.eps, 1 + self.eps) * advantage  #
截断
```

```
        actor_loss = torch.mean(-torch.min(surr1, surr2))  # PPO损失函数
        critic_loss = torch.mean(F.mse_loss(self.critic(states),
td_target.detach()))

        self.actor_optimizer.zero_grad()
        self.critic_optimizer.zero_grad()
        actor_loss.backward()
        critic_loss.backward()
        self.actor_optimizer.step()
        self.critic_optimizer.step()
```

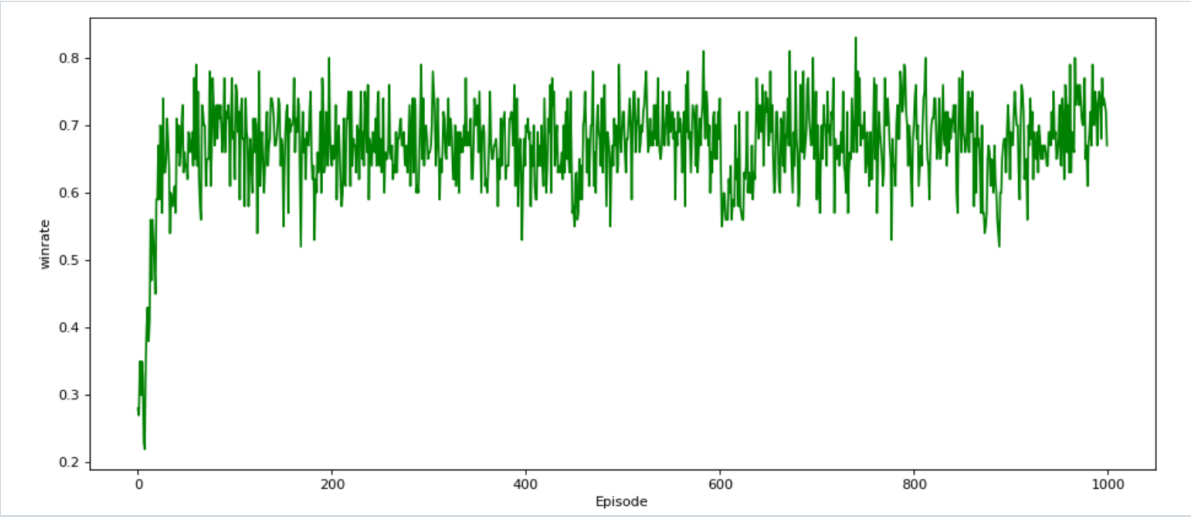
训练过程：

```
import torch
import time
from tqdm import tqdm

actor_lr = 3e-4
critic_lr = 1e-3
num_episodes = 100000
hidden_dim = 64
gamma = 0.99
lmbda = 0.97
eps = 0.2
device = torch.device("cuda") if torch.cuda.is_available() else
torch.device("cpu")
team_size = 2
grid_size = (15, 15)

# 创建Combat环境，格子世界的大小为15x15，己方智能体和敌方智能体数量都为2
env = Combat(grid_shape=grid_size, n_agents=team_size, n_opponents=team_size)
tt = time.time()
# writer = SummaryWriter(f'log/MARL_IPPO{tt}')
state_dim = env.observation_space[0].shape[0]
action_dim = env.action_space[0].n

# 两个智能体共享同一个策略
agent = PPO(state_dim, hidden_dim, action_dim, actor_lr, critic_lr, lmbda,
eps, gamma, device)
win_list = []
winrate_list = {'episode': [], 'win_rate': []}
global_step = 0

for i in range(10):
    with tqdm(total=int(num_episodes / 10), desc='Iteration %d' % i) as pbar:
        for i_episode in range(int(num_episodes / 10)):
            transition_dict_1 = {
                'states': [],
                'actions': [],
                'next_states': [],
```

```
                'rewards': [],
                'dones': []
            }
            transition_dict_2 = {
                'states': [],
                'actions': [],
                'next_states': [],
                'rewards': [],
                'dones': []
            }
            s = env.reset()
            terminal = False
            while not terminal:
                a_1 = agent.take_action(s[0])
                a_2 = agent.take_action(s[1])
                next_s, r, done, info = env.step([a_1, a_2])
                health = info['health']
                win = 1 if health[0] + health[1] > 0 else 0

                transition_dict_1['states'].append(s[0])
                transition_dict_1['actions'].append(a_1)
                transition_dict_1['next_states'].append(next_s[0])
                transition_dict_1['rewards'].append(r[0] + 100 if win else
r[0] - 0.1)
                transition_dict_1['dones'].append(False)

                transition_dict_2['states'].append(s[1])
                transition_dict_2['actions'].append(a_2)
                transition_dict_2['next_states'].append(next_s[1])
                transition_dict_2['rewards'].append(r[1] + 100 if win else
r[1] - 0.1)
                transition_dict_2['dones'].append(False)

                s = next_s
                terminal = all(done)
                win_list.append(1 if win else 0)
                print('epoch:', i_episode, 'win', win)
                # writer.add_scalar("win rate", win_list[-1],
global_step=global_step)
                global_step += 1

            agent.update(transition_dict_1)
            agent.update(transition_dict_2)

            if (i_episode + 1) % 100 == 0:
                win_rate = np.mean(win_list[-100:])
                print('win_rate', win_rate)
                winrate_list['episode'].append(i_episode + 1)
                winrate_list['win_rate'].append(win_rate)
```

## 实验结果

本次实验不是以reward作为评分标准，而是以游戏的胜利来进行评判，IPPO迭代结果如下：




可以看到对弈的结果一直在变好，我方的胜率一直在提升，说明处理这种简单的多智能体博弈问题IPPO表现还是非常不错的。