

1. 强化学习基础

1.1 有限马尔可夫决策过程

1.2 价值函数与优势函数

状态价值函数，在Sutton的著作中，将状态价值函数的定义如下：

$$v_{\pi}(s) \doteq \mathbb{E}_{\pi} \left[G_t | S_t = s \right] = \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \right], \forall s \in \mathcal{S}$$

状态价值函数反映了从状态 $S_t = s$ 开始，按照策略 π 执行动作后，智能体预计将获得的总奖励的期望值。

而价值函数满足某种递归关系，对于任何策略 π 和状态 s ， s 的价值与其后继状态的价值关系如下：

$$\begin{aligned} v_{\pi}(s) &\doteq \mathbb{E}_{\pi} \left[G_t | S_t = s \right] \\ &= \mathbb{E}_{\pi} \left[R_t + \gamma G_{t+1} | S_t = s \right] \\ &= \sum_a \pi(a|s) \sum_{r,s'} p(r, s'|s, a) (r + \gamma v_{\pi}(s')) \end{aligned}$$

| **动作价值函数** 则是根据策略 π ，从状态 s 开始，执行动作 a 之后所有可能的决策序列的期望回报，我们用符号 $q_{\pi}(s, a)$ 表示：

$$\begin{aligned} q_{\pi}(s, a) &\doteq \mathbb{E}_{\pi} \left[G_t | S_t = s, A_t = a \right] \\ &= \sum_{r,s'} p(r, s'|s, a) (r + \gamma v_{\pi}(s')) \end{aligned}$$

值得注意的是回报 R_t 是一个简写，完整的表示应该是 $R(S_t, A_t, S_{t+1})$ 。我们将智能体在策略 π 控制下的预期折扣收益用符号 $J(\pi)$ 表示：

$$J(\pi) = \mathbb{E}_{\pi} \left[G_t \right] = \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t, S_{t+1}) \right]$$

公式变体，在某些强化学习论文中，作者会为了简化公式的表达或者让某个特定的推导更清晰，而对标准符号做一些变动。如省略某些下角标或者是加上下角标变量，亦或者是对公式进行拓展，使得读者在阅读不同论文或材料时感到困惑，在本文接下来的内容中会涉及到不同的论文公式推导，为方便读者理解，笔者先列举一些公式的变体：

$$\begin{aligned}
J(\pi) &= \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t, S_{t+1}) \right] \\
&= \mathbb{E}_{s_0, a_0, \dots} \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t, S_{t+1}) \right] \\
&= \mathbb{E}_{\tau \sim (\pi, E)} \left[G(\tau) \right] \\
&= \sum_{\tau \sim (\pi, E)} P(\tau | \pi) G(\tau)
\end{aligned}$$

公式的改写本质上是对同一个问题的不同表示方式，如上式，第一二行通常用于标准的动态规划方法或分析中，直接表示在时间步 t 处从状态 S_t 开始采取动作 A_t 所获得的累积奖励的期望。第三四行则是从策略 π 和环境 E 生成的轨迹序 τ 的角度出发。同样，状态价值函数与动作价值函数也可以进行适当改写：

$$\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi \left[G_t | S_t = s \right] \\
&= \mathbb{E}_\pi \left[R_t + \gamma G_{t+1} | S_t = s \right] \\
&= \mathbb{E}_{S_t=s, A_t, \dots} \left[\sum_{t'=t}^{\infty} \gamma^{t'-t} R(S_{t'} = s, A_{t'}, S_{t'+1}) \right], \\
q_\pi(s, a) &\doteq \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \\
&= \mathbb{E}_{S_t=s, A_t=a, \dots} \left[\sum_{t'=t}^{\infty} \gamma^{t'-t} R(S_{t'} = s, A_{t'} = a, S_{t'+1}) \right]
\end{aligned}$$

优势函数，

$$A_\pi(s, a) = \underbrace{q_\pi(s, a)}_{\text{On-Policy}} - \underbrace{v_\pi(s)}_{\text{On-Policy}}$$

2. Reward Modeling

3. 策略梯度算法

我们现在来看一下什么是策略梯度算法，以及其背后的动机与直觉。我们希望我们的策略能够使得回报的期望最大，动作的轨迹是在策略 π_θ 的控制下产生的，因此我们的目标函数可以表达如下：

$$\begin{aligned}
J(\pi_\theta) &= \mathbb{E}_{\tau \sim (\pi_\theta, E)} \left[G(\tau) \right] \\
&= \sum_{\tau \sim (\pi_\theta, E)} P(\tau | \theta) G(\tau)
\end{aligned} \tag{2.1}$$

其中 E 是环境，这个目标函数衡量了在指定环境下从我们的策略中采样的轨迹的理论收益。如果我们想找到最大化这个目标函数的参数 θ ，我们可以通过梯度上升的方式不断迭代更新 θ 。更新过程表示如下：

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_t} \tag{2.2}$$

其中 $\nabla_\theta J(\pi_\theta)|_{\theta_t} = \nabla_{\theta_t} J(\pi_{\theta_t})$ ，也就是所谓的**策略梯度**。现在的问题在于，我们如何计算 $\nabla_\theta J(\pi_\theta)|_{\theta_t}$ ？利用策略梯度则需要一个明确的可计算的表达式。接下来我们进一步探索如何找到这个明确的表达式并加以运用。

1.轨迹的概率, 给定策略 π_θ 下轨迹 $\tau = (s_0, s_1, \dots, s_{T+1})$ 出现的概率如下:

$$P(\tau|\theta) = \rho(s_0) \prod_{t=0}^T P(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t) \quad (2.3)$$

2.Log-Derivative Trick, 利用log求导的技巧 $\partial_x \log(f(x)) = \frac{\partial_x f(x)}{f(x)}$, 对轨迹概率关于 θ 求导, 我们有:

$$\nabla_\theta P(\tau|\theta) = \nabla_\theta \log P(\tau|\theta) \cdot P(\tau|\theta) \quad (2.4)$$

3.轨迹的对数概率, $P(\tau|\theta)$ 的对数概率如下:

$$\log P(\tau|\theta) = \underbrace{\log(\rho(s_0))}_{\text{与}\theta\text{无关}} + \sum_{t=0}^T \underbrace{\log P(s_{t+1}|s_t, a_t)}_{\text{与}\theta\text{无关}} + \log(\pi_\theta(a_t|s_t)) \quad (2.5)$$

4.轨迹对数概率的梯度, 由于部分项与 θ 无关, 所以 $\nabla_\theta \log P(\tau|\theta)$ 表达如下:

$$\begin{aligned} \nabla_\theta \log P(\tau|\theta) &= \nabla_\theta \sum_{t=0}^T \log(\pi_\theta(a_t|s_t)) \\ &= \sum_{t=0}^T \nabla_\theta \log(\pi_\theta(a_t|s_t)) \end{aligned} \quad (2.6)$$

将上述式子结合, 求出 $\nabla_\theta J(\pi_\theta)$ 则有:

$$\begin{aligned} \nabla_\theta J(\pi_\theta) &= \nabla_\theta \sum_{\tau \sim (\pi_\theta, E)} P(\tau|\theta) G(\tau) \\ &= \sum_{\tau \sim (\pi_\theta, E)} \nabla_\theta P(\tau|\theta) G(\tau) \\ &= \sum_{\tau \sim (\pi_\theta, E)} \nabla_\theta \log P(\tau|\theta) P(\tau|\theta) R(\tau) \\ &= \sum_{\tau \sim (\pi_\theta, E)} \sum_{t=0}^T \nabla_\theta \log(\pi_\theta(a_t|s_t)) P(\tau|\theta) G(\tau) \\ &= \mathbb{E}_{\tau \sim (\pi_\theta, E)} \left[\sum_{t=0}^T \nabla_\theta \log(\pi_\theta(a_t|s_t)) G(\tau) \right] \end{aligned} \quad (2.7)$$

也就是说, 我们可以根据采样到的轨迹来计算策略梯度, 即如果收集到了一系列由 π_θ 产生的轨迹 $D = \{\tau_i\}_{i=1}^N$, 我们可以利用如下公式估计策略梯度:

$$\hat{\nabla}_\theta J(\pi_\theta) = \frac{1}{|D|} \sum_{i=1}^N \sum_{t=0}^T \nabla_\theta \log(\pi_\theta(a_t|s_t)) G(\tau) \quad (2.8)$$

如上便是策略梯度最简单的形式。如果我们能够在环境中运行策略来收集轨迹数据集, 我们可以计算策略梯度并采取更新步骤。

Don't Let the Past Distract You.策略梯度的公式告诉我们对于一个轨迹 $\tau = (s_0, s_1, \dots, s_T)$, 每一个时刻的动作执行后对应的策略梯度都会乘以一个因子, 即回报 $R(\tau)$, 但这并没有什么意义, 因为是否强化智能体当前的决策动作只因和当前动作执行后产生的影响有关, 如果当前动作执行后的收益低则不应当强化当前动作, 反之亦然, 而不能受先前因素的影响。因此, 对于回报 $R(\tau)$ 我们可以做出适当改进, 不再考虑当前时刻 t' 之前的收益, 则策略梯度可以重写为:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim (\pi_\theta, E)} \left[\sum_{t=0}^T (\nabla_\theta \log(\pi_\theta(a_t|s_t)) \sum_{t=t'}^T R(s_{t'}, a_{t'}, s_{t'+1})) \right] \quad (3.10)$$

在这个形式下，动作只基于在采取行动后获得的奖励而得到强化。我们将这种形式称为“**Reward-to-go**”，因为回报为在轨迹的一个点之后奖励的总和。

策略梯度算法有一些变种形式，这些变种都与我们先前所学习的内容有所关联，接下来我们学习一些变种算法从而对策略梯度算法有更深入的理解。

Baseline in Policy Gradients. 使用策略梯度面临着一个问题——准确地估算出梯度需要大量的样本轨迹，使用 EGLP 引理，我们可以证明**Reward -to-go**——尽管没有改变政策梯度的期望值——减少了我们估计的方差，因此减少了估计策略梯度所需的轨迹总数。而 EGLP 的直接结果是对于任何直接依赖于状态的函数 b ，我们有：

$$\mathbb{E}_{a_t \sim \pi_\theta} [\nabla_\theta \log(\pi_\theta(a_t | s_t)) b(s_t)] = 0 \quad (3.11)$$

这使得我们可以在不改变期望值的情况下在策略梯度表达式上加上或减去任意项：

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim (\pi_\theta, E)} \left[\sum_{t=0}^T \left(\nabla_\theta \log(\pi_\theta(a_t | s_t)) \left(\sum_{t'=t}^T R(s_{t'}, a_{t'}, s_{t'+1}) - \underbrace{b(s_t)}_{\text{Baseline}} \right) \right) \right] \quad (3.12)$$

在这个表达式中，任何函数 b 都称作 baseline。最常见的基线选择是状态价值函数 $V^\pi(s_t)$ 。回想一下，这是一个智能体从状态开始，然后在其剩余生命周期内按照策略行事时获得的平均回报。从经验上讲，这种选择具有减少策略梯度样本估计方差的效果，使得策略梯度学习更快更稳定。从直观上讲，假设在某个状态 s_t 下，智能体采取了一个动作 a_t ，并得到了一个回报。如果这个回报远高于该状态的预期回报（即 $V^\pi(s_t)$ ），我们认为这个动作“更好”；如果低于预期回报，我们认为该动作“不好”，通过减去 $V^\pi(s_t)$ ，我们将焦点集中在“超出预期的部分”（即优势）上。这就像在与基准相比时，我们只关注某个动作相对于平均策略的改进或削弱。

① Note

值得注意的是， $V^\pi(s_t)$ 并不能准确的计算，所以需要近似计算，通常我们用一个神经网络 $V_\phi^\pi(s_t)$ 估计价值函数，它与策略网络同时更新，而 V_ϕ^θ 的优化目标通常是最小均方误差（包括 VPG, TRPO, PPO 等），即：

$$\phi_k = \arg \min_{\phi} \mathbb{E}_{s_t, \hat{R}_t \sim \pi_k} [(V_\phi(s_t) - \hat{R}_t)^2] \quad (3.13)$$

我们可以以一种更一般地形式写出策略梯度：

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim (\pi_\theta, E)} \left[\sum_{t=0}^T \left(\nabla_\theta \log(\pi_\theta(a_t | s_t)) \Psi(t) \right) \right] \quad (3.14)$$

$\Psi(t) = R(\tau)$ 时 $\nabla_\theta J(\pi_\theta)$ 是基础， $\Psi(t) = \sum_{t=t'}^T R(s_{t'}, a_{t'}, s_{t'+1})$ 时是 **Reward-to-go**， $\Psi(t) = \sum_{t=t'}^T R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t)$ 是 **Reward-to-go with baseline**。此外， $\Psi(t)$ 的选择还可以是动作价值函数 $Q^\pi(s_t, a_t)$ ，优势函数 $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$ ，利用优势函数的策略梯度的公式化极为常见，并且不同算法使用的优势函数有许多不同的估计方法。

3.1 TRPO

为保证上下文符号一致，笔者在本章节推导上的符号并未遵循原论文，进行了一定改动。

一个策略 $\tilde{\pi}$ 关于另一个策略 π 的预期收益在累计时间步上的优势为：

$$J(\tilde{\pi}) - J(\pi) = \mathbb{E}_{\tau \sim \tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t A_\pi(s_t, a_t) \right] \quad (3.15)$$

证明如下：

$$\begin{aligned}
& \mathbb{E}_{\tau \sim \tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi}(s_t, a_t) \right] \\
&= \mathbb{E}_{\tau \sim \tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t (q_{\pi}(s_t, a_t) - v_{\pi}(s_t)) \right] \\
&= \mathbb{E}_{\tau \sim \tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1}) + \gamma v_{\pi}(s_{t+1}) - v_{\pi}(s_t)) \right] \\
&= \mathbb{E}_{\tau \sim \tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) + \gamma^{t+1} v_{\pi}(s_{t+1}) - \gamma^t v_{\pi}(s_t) \right] \\
&= \mathbb{E}_{\tau \sim \tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \right] + \mathbb{E}_{\tau \sim \tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^{t+1} v_{\pi}(s_{t+1}) - \gamma^t v_{\pi}(s_t) \right] \\
&= J(\tilde{\pi}) + \mathbb{E}_{\tau \sim \tilde{\pi}} \left[\sum_{t=1}^{\infty} \gamma^t v_{\pi}(s_t) - \sum_{t=0}^{\infty} \gamma^t v_{\pi}(s_t) \right] \\
&= J(\tilde{\pi}) - \mathbb{E}_{\tau \sim \tilde{\pi}} \left[v_{\pi}(s_0) \right] \\
&= J(\tilde{\pi}) - J(\pi)
\end{aligned} \tag{3.16}$$

如果能保证 $J(\tilde{\pi}) - J(\pi)$ 大于 0，则能说明更新后的策略一直在进步，而优势函数这一项又可以改写成：

$$\begin{aligned}
& \mathbb{E}_{\tau \sim \tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi}(s_t, a_t) \right] \\
&= \sum_{t=0}^{\infty} \sum_s p(s_t = s | \tilde{\pi}) \sum_a \tilde{\pi}(a_t = a | s) \gamma^t A_{\pi}(s, a) \\
&= \sum_{t=0}^{\infty} \sum_s \gamma^t p(s_t = s | \tilde{\pi}) \sum_a \tilde{\pi}(a_t = a | s) A_{\pi}(s, a) \\
&= \sum_s \sum_{t=0}^{\infty} \gamma^t p(s_t = s | \tilde{\pi}) \sum_a \tilde{\pi}(a_t = a | s) A_{\pi}(s, a) \\
&= \sum_s \rho_{\tilde{\pi}}(s) \sum_a \tilde{\pi}(a_t = a | s) A_{\pi}(s, a)
\end{aligned} \tag{3.17}$$

其中， $\rho_{\tilde{\pi}}(s) = p(s_0 = s | \tilde{\pi}) + \gamma p(s_1 = s | \tilde{\pi}) + \dots$ ， $\tilde{\pi}$ 是之前的策略 π 更新后的新策略，上述式子中涉及到 $p(s_t = s | \tilde{\pi})$ 与 $\tilde{\pi}(a_t = a | s)$ ，即我们要按照新的策略与环境交互才能得到轨迹，先确定新的策略 $\tilde{\pi}$ 并得到一定量的样本才能求解，并计算是否满足 $\mathbb{E}_{\tau \sim \tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi}(s_t, a_t) \right] > 0$ 。TRPO 利用函数 $\mathcal{L}_{\pi}(\tilde{\pi})$ 代替原始目标函数：

$$\mathcal{L}_{\pi}(\tilde{\pi}) = J(\pi) + \sum_s \rho_{\pi}(s) \sum_a \tilde{\pi}(a | s) A_{\pi}(s, a). \tag{3.18}$$

只要策略更新的幅度不大，就可以用 $\mathcal{L}_{\pi}(\tilde{\pi})$ 近似原本的 $J(\tilde{\pi})$ ，所以那我们怎么来保证其更新幅度不要太大呢？为了解决这个求解信任区域的问题，文中引入了 Kakade & Langford (2002) 的结论——Conservative policy iteration：

$$\begin{aligned}
\pi_{\text{new}}(a | s) &= (1 - \alpha)\pi_{\text{old}}(a | s) + \alpha\pi'(a | s) \\
\eta(\pi_{\text{new}}) &\geq L_{\pi_{\text{old}}}(\pi_{\text{new}}) - \frac{2\epsilon\gamma}{(1 - \gamma)^2}\alpha^2
\end{aligned} \tag{3.19}$$

where $\epsilon = \max_s |\mathbb{E}_{a \sim \pi'(a|s)} [A_{\pi}(s, a)]|$

有了这个下界表达式，我们可以利用**minorization-maximization**算法通过 $\mathcal{L}_{\pi_{old}}(\pi_{new})$ 迭代 $J(\pi_{new})$ 。该算法具体细节不在本文涉及范围内，值得注意的是，该原始结论只适合混合策略，但实际应用中的混合策略很少使用，因此作者将该结论拓展到了一般随机策略[x]。最终的优化目标变成：

$$\underset{\pi_\theta}{\text{maximize}} \left[\mathcal{L}_{\pi_{\theta_{old}}}(\pi_\theta) - CD_{\text{KL}}^{\max}(\pi_{\theta_{old}}, \pi_\theta) \right]. \quad (3.20)$$

其中：

$$\mathcal{L}_{\pi_{old}}(\pi_\theta) = J(\pi) + \sum_s \rho_{\pi_{\theta_{old}}}(s) \sum_a \pi_\theta(a | s) A_{\pi_{\theta_{old}}}(s, a).$$

由于 $\pi_\theta(a | s)$ 与新策略有关，无法对其直接采样，因此我们通过重要性采样的方式进行采样，我们将式子右边项进行变体：

$$\begin{aligned} & \sum_s \rho_{\pi_{\theta_{old}}}(s) \sum_a \pi_\theta(a | s) A_{\pi_{\theta_{old}}}(s, a) \\ &= \sum_s \sum_t \gamma^t p(s_t = s | \pi_{\theta_{old}}) \sum_a \pi_\theta(a | s) A_{\pi_{\theta_{old}}}(s, a) \\ &= \sum_t \gamma^t \sum_s p(s_t = s | \pi_{\theta_{old}}) \sum_a \pi_\theta(a | s) A_{\pi_{\theta_{old}}}(s, a) \\ &= \sum_t \gamma^t \mathbb{E}_{s \sim \rho_{old}} \left[\sum_a \pi_\theta(a | s) A_{\pi_{\theta_{old}}}(s, a) \right] \\ &= \frac{1}{1-\gamma} \mathbb{E}_{s \sim \rho_{old}} \left[\sum_a \pi_\theta(a | s) A_{\pi_{\theta_{old}}}(s, a) \right] \end{aligned} \quad (3.21)$$

$\sum_a \pi_\theta(a | s) A_{\pi_{\theta_{old}}}(s, a)$ 可以通过重要性采样的方式重新表述成：

$$\begin{aligned} \sum_a \pi_\theta(a | s) A_{\pi_{\theta_{old}}}(s, a) &= \mathbb{E}_{a \sim q} \left[\frac{\pi_\theta(a | s)}{q(a | s)} A_{\pi_{\theta_{old}}}(s, a) \right] \\ &\mapsto \mathbb{E}_{a \sim \pi_{\theta_{old}}} \left[\frac{\pi_\theta(a | s)}{\pi_{\theta_{old}}(a | s)} A_{\pi_{\theta_{old}}}(s, a) \right] \end{aligned}$$

故最终的优化目标为：

$$\begin{aligned} & \arg \max_{\theta} \mathbb{E}_{s \sim \rho_{old}, a \sim \pi_{\theta_{old}}} \left[\frac{\pi_\theta(a | s)}{\pi_{\theta_{old}}(a | s)} A_{\pi_{\theta_{old}}}(s, a) \right] \\ & \text{subject to } \mathbb{E}_{s \sim \rho_{old}} \left[D_{KL}(\pi_{\theta_{old}}(\cdot | s) || \pi_\theta(\cdot | s)) \right] \leq \delta \end{aligned} \quad (3.22)$$

3.2 PPO

我们现在将深入理解为语言模型对齐奠定基础的算法——PPO(近端策略优化算法)，PPO是一种基于策略梯度的强化学习算法，PPO的核心思想是通过在每次更新时保持策略的“平稳性”或“稳定性”，避免过度优化，从而减少策略更新过程中的波动性，PPO算法的优化目标如下：

$$J(\theta) = \frac{1}{G} \sum_{i=1}^G \min \left(\frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)} A_t, \text{clip} \left(\frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)}, 1 - \varepsilon, 1 + \varepsilon \right) A_t \right)$$

注意到min函数会使优化目标有不同取值的选择，我们来解释不同情况下的取值情况，首先定义策略比率为：

$$R(\theta) = \frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)}$$

当优势是正数且策略比率超过 $1 + \varepsilon$ 时，意味着新的策略更可能采取 a_t 行动， $\text{clip}(\cdot)$ 通过裁剪比率防止策略更新过大，并限制 $R(\theta)$ 的变化范围，此时优化目标：

$$J(\theta) = \min(R(\theta), 1 + \varepsilon)A_t = (1 + \epsilon)A_t$$

意味着优化目标会增加策略比率，使得动作更可能发生，当优势是正数而策略比率小于 $1 - \varepsilon$ 时，优化目标变成：

$$J(\theta) = \min(R(\theta), 1 - \varepsilon)A_t = R(\theta)A_t$$

意味着优化目标会减小策略比率，使得动作没那么可能发生。同理，如果策略比率本身介于 $(1 - \varepsilon, 1 + \varepsilon)$ 之间，则有：

$$J(\theta) = \min(R(\theta), R(\theta)) = R(\theta)$$

没有任何影响，当优势是负数时且策略比率 $R(\theta) < 1 - \varepsilon$ 时，有：

$$J(\theta) = \min(R(\theta)A_t, (1 - \varepsilon)A_t) = (1 - \varepsilon)A_t$$

其他情况同理，在信任区域内的优化目标与策略梯度是一致的。

接下来我们将结合部分的代码深入理解PPO算法如何在RLHF中大展身手，如何完成大语言模型对齐的任务。

3.2.1 DeepSpeed-Chat 源码解析

在training/step3_rlhf_finetuning/main.py下的for循环内有两个比较重要的函数，第一个是第537行的generate_experience，另一个是第553行的train_rlhf。分别用于生成轨迹以及计算损失函数。

```
DeepSpeed-Chat/dschat/rllhf/ppo_trainer.py
```

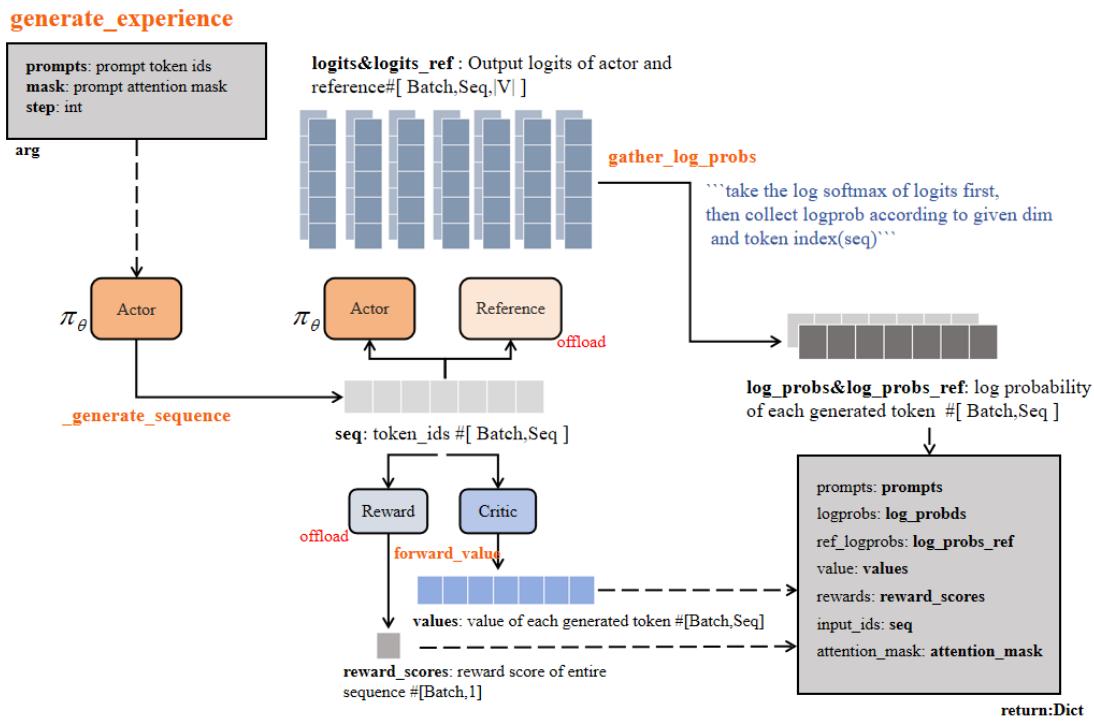
```
521
522     for epoch in range(args.num_train_epochs):
523         print_rank_0(
524             f"Beginning of Epoch {epoch+1}/{args.num_train_epochs}, Total G"
525             f" args.global_rank")
526         for step, (batch_prompt, batch_unsupervised) in enumerate(
527             zip(prompt_train_dataloader, unsupervised_train_dataloader)):
528
529             batch_prompt = to_device(batch_prompt, device)
530
531             # prompts = batch_prompt['prompt']
532             # length = prompts.size(-1)
533             # if length > args.max_prompt_seq_len:
534             #     prompts = prompts[:, length - args.max_prompt_seq_len:]
535             #     raise ValueError("Prompt length is too long")
536
537             out = trainer.generate_experience(batch_prompt[0]['prompt'],
538                                             batch_prompt[0]['prompt_att_mask'],
539                                             step)
540
541             training_start = time.time()
542             if batch_unsupervised is not None:
543                 ...
544             else:
```

```
DeepSpeed-Chat/training/step3_rllhf_finetuning/main.py
```

```
def generate_experience(self, prompts, mask, step):
    self.eval()
    generate_start = time.time()
    seq = self._generate_sequence(prompts, mask, step)
    generate_end = time.time()
    if seq is None:
        ...
    else:
        self.train()
        pad_token_id = self.tokenizer.pad_token_id
        attention_mask = seq.not_equal(pad_token_id).long()
        with torch.no_grad():
            output = self.actor_model(seq, attention_mask=attention_mask)
            output_ref = self.ref_model(seq, attention_mask=attention_mask)
            reward_score = self.reward_model.forward_value(
                seq, attention_mask,
                prompt_length=self.prompt_length)['chosen_end_scores'].detach() #[B,1]
            values = self.critic_model.forward_value(
                seq, attention_mask, return_value_only=True).detach()[:, :-1] #{"value": [B, S, 1]}
            ...
            values = values.gather(1, seq)

        logsitc = output.logits #[B,S,H]
        logsitc_ref = output_ref.logits #[B,S,H]
        if self.compute_fp32_loss:
            logsitc = logsitc.to(torch.float)
            logsitc_ref = logsitc_ref.to(torch.float)
        self.generate_time = generate_end - generate_start
        return {
            'prompts': prompts,
            'logprobs': gather_log_probs(logsitc[:, :-1, :, seq[:, 1:]]),
            'ref_logprobs': gather_log_probs(logsitc_ref[:, :-1, :, seq[:, 1:]]),
            'value': values,
            'rewards': reward_score,
            'Input_ids': seq,
            "attention_mask": attention_mask,
```

先看第一个函数generate_experience，返回的字典中包括了`actor`和`reference`的对数几率，并且还有`actor`生成的序列的奖励以及每一个`token`对应的价值。价值计算由`critic_model.forward_value()`产生，对应实现在`dschat/utils/model/reward_model.py`。函数中实现的功能如下图所示：



该函数中又涉及到了三个重要的函数，由图中的橙色粗体表示，分别是`_generate_sequence`，作用是`actor`根据给定的`prompt`生成完整的序列，以及`forward_value`，`Reward`和`Critic`分别生成奖励和`token`序列的价值，最后是根据给定的维度和索引收集指定的对数概率，由函数`gather_loh_probs`完成。

```

143 >     transformer_outputs = self.rwtransformer(...
150     hidden_states = transformer_outputs[0]
151     values = self.v_head(hidden_states).squeeze(-1)
152     if return_value_only:
153         return values
154     else:
155         # [0 0 0 0 prompt, answer, 0 0 0 0] for step 3, we have padding at the beginning
156         # [prompt, answer, 0, 0, 0, 0] this is normal
157         assert prompt_length > 1, "prompt_length must be greater than 1 to help select the end score"
158         bs = values.size(0)
159         seq_len = input_ids.shape[1]
160         chosen_end_scores = [
161             ] # we use this name for consistency with the original forward function
162         for i in range(bs):
163             input_id = input_ids[i]
164             value = values[i]
165
166             c_inds = (input_id[prompt_length:] == self.PAD_ID).nonzero()
167             # here we only use the answer part of the sequence so we do not need to care about the padding at the beginning
168             c_ind = c_inds[0].item() + prompt_length if len(
169                 c_inds) > 0 else seq_len
170             chosen_end_scores.append(value[c_ind - 1])
171
172         return {
173             "values": values,
174             "chosen_end_scores": torch.stack(chosen_end_scores),
175         }
    
```

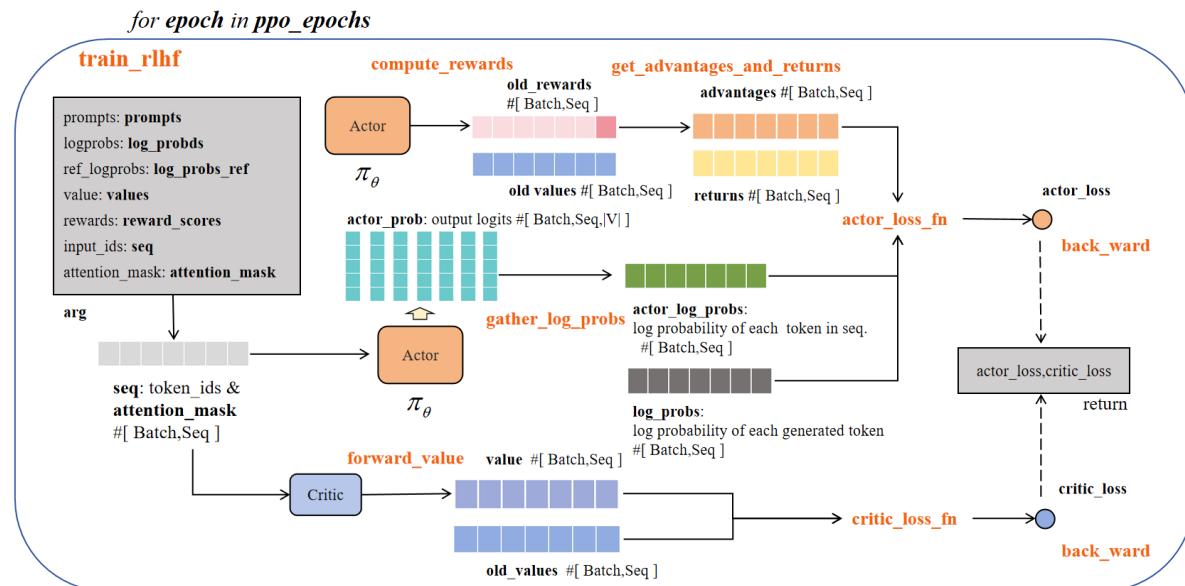
在`forward_value`函数中，目的是为了拿到模型输出的奖励，因此需要找到整个序列中只属于`answer`的这部分`token`，第166行得到的`c_inds`是模型输出的回答部分结束的位置索引，第168 – 169行是拿到最后一个位置前的输出的`value`，并返回`value`序列以及最后一个位置获得的奖励`chosen_end_scores`。在拿到了生成的轨迹和奖励与价值后，继续往下看到`for`循环内的第564行代码，在`train_rlhf`中完成了损失函数计算和反向传播，该函数输入变量是`generate_experience`函数的返回结果，即。

```

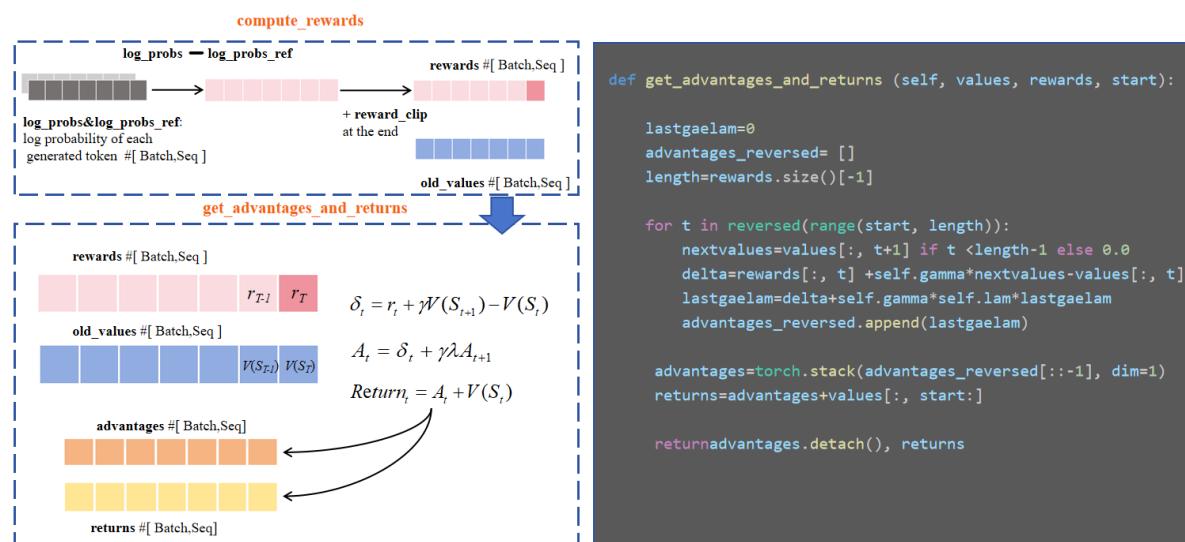
543     training_start = time.time()
544     > if batch_unsupervised is not None:
545     else:
546         unsup_dataset = unsup_mini_dataset.add(
547             [None] * args.per_device_generation_batch_size)
548
549     exp_dataset = exp_mini_dataset.add(out)
550
551     if exp_dataset is not None:
552         inner_iter = 0
553         actor_loss_sum, critic_loss_sum, unsup_loss_sum = 0, 0, 0
554         average_reward = 0
555
556     if args.actor_gradient_checkpointing:
557         rlhf_engine.actor.gradient_checkpointing_enable()
558
559     for ppo_ep in range(args_ppo_epochs):
560         for i, (exp_data, unsup_data) in enumerate(
561             zip(exp_dataset, unsup_dataset)):
562             actor_loss, critic_loss = trainer.train_rlhf(exp_data)
563             actor_loss_sum += actor_loss.item()
564             critic_loss_sum += critic_loss.item()
565             average_reward += exp_data["rewards"].mean()
566
567             if unsupervised_training_enabled:
568                 inner_iter += 1
569                 if args.enable_ema:
570                     moving_average(rlhf_engine.actor,
571                                     rlhf_engine.actor_ema,
572                                     zero_stage=args.actor_zero_stage)
573
574             random.shuffle(exp_dataset)
575             random.shuffle(unsup_dataset)
576
577     with torch.no_grad():
578         old_rewards = self.compute_rewards(prompts, log_probs,
579                                         ref_log_probs, reward_score,
580                                         action_mask)
581         ends = start + action_mask[:, start:].sum(1) + 1
582         # we need to zero out the reward and value after the end of the conversation
583         # otherwise the advantage/return will be wrong
584         for i in range(old_rewards.shape[0]):
585             old_rewards[i, ends[i]:] = 0
586             old_values[i, ends[i]:] = 0
587         advantages, returns = self.get_advantages_and_returns(
588             old_values, old_rewards, start)
589
590         ## process the new outputs
591         batch = {'input_ids': seq, 'attention_mask': attention_mask}
592         actor_prob = self.actor_model(**batch, use_cache=False).logits
593         actor_log_prob = gather_log_probs(actor_prob[:, :-1, :, seq:, 1:])
594         actor_loss = self.actor_loss_fn(actor_log_prob[:, start:], log_probs[:, start:], advantages, action_mask[:, start:])
595         self.actor_model.backward(actor_loss)
596
597         if not self.args.align_overflow:
598             value = self.critic_model.forward_value(**batch, return_value_only=True, use_cache=False)[:, :-1]
599             critic_loss = self.critic_loss_fn(value[:, start], old_values[:, start], returns, action_mask[:, start:])
600             self.critic_model.backward(critic_loss)
601
602         if self.args.align_overflow:
603             self.critic_model.step()
604
605     return actor_loss, critic_loss

```

右边子图是该方法的具体实现，其中又涉及到了四个重要的函数，第一个函数是计算 $actor$ 在策略 π_θ 下获得的奖励，第二个函数是计算策略 π_θ 下的优势，第三个函数则是计算PPO损失，第四个函数则是计算 $critic$ 损失，四个函数均定义在DeepSpeedPPOTrainer的类下，整体关系如下图所示：



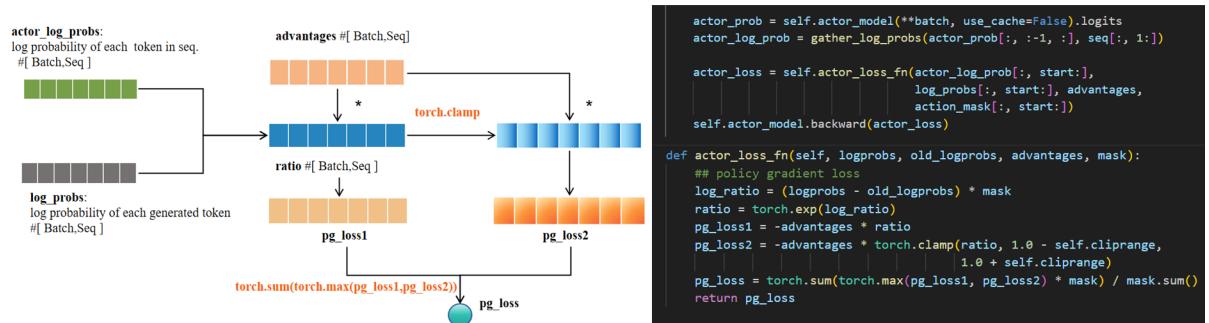
接下来看到compute_rewards和get_advantages_and_returns两个函数，分别完成了奖励计算以及优势与回报计算。rewards其实计算一个[Batch, Seq]的二维张量，Seq这个维度每一个元素需要计算actor和reference输出的KL散度即 $\mathbb{E}_{A_t \sim \pi_\theta} [\log \frac{\pi_\theta(A_t | S_t)}{\pi_{ref}(A_t | S_t)}]$ ，且最后一个位置还要加上seq序列最终的奖励分数。需要注意的是，在ppo_epochs这个循环中，一开始actor_log_probs与log_probs二者是相同的，随着循环的进行，actor不断更新后二者不再相同。



在计算完带有惩罚项的奖励以后，我们需要计算优势函数 A_t 。在广义优势估计中，有：

$$\begin{aligned}
A_t &= \sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+k} \\
&= \delta_t + (\gamma\lambda) \underbrace{\sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+1+k}}_{\gamma\lambda A_{t+1}} \\
&= \delta_t + \gamma\lambda \delta_{t+1} + (\gamma\lambda)^2 \underbrace{\sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+2+k}}_{(\gamma\lambda)^2 A_{t+2}} \\
&\quad \dots \\
\delta_t &= r_t + \gamma V(S_{t+1}) - V(S_t)
\end{aligned}$$

由上面公式可知，计算 A_t 需要知道 δ_t 和 A_{t+1} ，计算 δ_t 需要知道 r_t , $V(S_{t+1})$ 和 $V(S_t)$ ，而 r_t , $V(S_t)$ 可以由图中的rewards和old_values直接得出，所以问题在于如何求 A_{t+1} ，此时如果正向计算 $A_t, t = 0, 1, \dots, T$ 就存在一个问题，计算 A_0 需要知道 A_1 ，计算 A_1 需要知道 A_2 ，以此类推需要先把 A_t 全部都先算出来才能知道最开始的 A_0 （增加内存占用，需保存所有中间结果），因此我们通常采用倒序计算的方式以更高效地解决这个问题，即我们先算 A_T ，并基于如下公式递推计算 $A_{T-1} = \delta_{T-1} + \gamma\lambda A_T$ ，每一步复用上一步的结果，最终计算完 A_0 。计算完优势序列 A_t 后，根据PPO的公式，我们只需要再计算出新旧策略执行动作的比率 $R_t = \frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)}$ 就能计算损失函数。而比率 R_t 就是更新后的actor输出的动作序列(对数概率)除以没更新前的actor的动作序列，整个流程示意图如下所示：



计算 R_t 依赖于更新后的actor的动作概率和未更新的actor的动作概率，而actor_log_prob与log_prob是对数概率，因此有 $R_t = \exp\{\log \pi_\theta(A_t|S_t) - \log \pi_{\theta_{old}}(A_t|S_t)\}$ ，再依据PPO损失函数公式得到最后的损失pg_loss。actor会不断改进策略以执行更好的动作，与此同时critic也需要不断更新，读者可以理解为一个教练不能总是以过往的眼光来评价一个不断进步的演员。critic的损失函数计算比较简单，采用的是平方差损失：

```

282     def critic_loss_fn(self, values, old_values, returns, mask):
283         ## value loss
284         values_clipped = torch.clamp(
285             values,
286             old_values - self.cliprange_value,
287             old_values + self.cliprange_value,
288         ) #[Batch, Seq]
289         if self.compute_fp32_loss:
290             values = values.float()
291             values_clipped = values_clipped.float()
292             vf_loss1 = (values - returns)**2 #[Batch, Seq]
293             vf_loss2 = (values_clipped - returns)**2 #[Batch, Seq]
294             vf_loss = 0.5 * torch.sum(
295                 torch.max(vf_loss1, vf_loss2) * mask) / mask.sum()
296             return vf_loss

```

首先通过`torch.clamp`将`values`限制到一定范围，计算平方差损失后再取每一个位置上的最大值，如上便是整个RLHF-PPO算法的核心流程实现，值得注意的是，在`ppo_epochs`的循环中，`actor`与`critic`更新以后在558行还有一个无监督训练：

```
550     for ppo_ep in range(args.ppo_epochs):
551         for i, (exp_data, unsup_data) in enumerate(
552             zip(exp_dataset, unsup_dataset)):
553             actor_loss, critic_loss = trainer.train_rlhf(exp_data)
554             actor_loss_sum += actor_loss.item()
555             critic_loss += critic_loss.item()
556             average_reward += exp_data["rewards"].mean()
557
558             if unsupervised_training_enabled:
559                 unsup_loss = trainer.train_unsupervised(
560                     unsup_data, args.unsup_coef)
561                 unsup_loss_sum += unsup_loss.item()
562
563             inner_iter += 1
564             if args.enable_ema:
565                 moving_average(rlhf_engine.actor,
566                                 rlhf_engine.actor_ema,
567                                 zero_stage=args.actor_zero_stage)
568
569     class DeepSpeedPPOTrainerUnsupervised(DeepSpeedPPOTrainer):
570
571         def __init__(self, *args, **kwargs):
572             super().__init__(*args, **kwargs)
573
573         def train_unsupervised(self, inputs, unsup_coef):
574
575             # Train the unsupervised model here
576             self._validate_training_mode()
577
578             outputs = self.actor_model(**inputs, use_cache=False)
579             loss = outputs.loss
580             self.actor_model.backward(unsup_coef * loss)
581             self.actor_model.step()
582
583             return loss
```

进入train_unsupervised方法后可以发现其就是默认的损失，即next prediction loss，（但不是SFT形式的数据，是预训练形式的数据）目的是为了在强化学习过程中保持模型的通用领域知识，防止模型被带偏。

XX

3.3.2 小节

3.3 DPO

强化学习的优化目标可以用如下一般形式的公式进行概述：

$$\max_{\pi_\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_\theta(y|x)} [r_\phi(x, y)] - \beta \mathbb{D}_{\text{KL}} [\pi_\theta(y \mid x) \| \pi_{\text{ref}}(y \mid x)], \quad (3.3.1)$$

即让新策略在不偏离原始策略太多的情况下尽可能地生成奖励模型认为较好的内容，*DPO*将此优化目标进行变体：

$$\begin{aligned}
& \arg \max_{\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_{\theta}(y|x)} [r_{\phi}(x, y)] - \beta \mathbb{D}_{\text{KL}} [\pi_{\theta}(y|x) \| \pi_{\text{ref}}(y|x)] \\
&= \arg \max_{\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_{\theta}(y|x)} [r_{\phi}(x, y)] - \beta \pi_{\theta}(y|x) \log \frac{\pi_{\theta}(y|x)}{\pi_{\text{ref}}(y|x)} \\
&\propto \arg \max_{\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_{\theta}(y|x)} \left[\left[\frac{1}{\beta} r_{\phi}(x, y) \right] - \log \frac{\pi_{\theta}(y|x)}{\pi_{\text{ref}}(y|x)} \right] \\
&= \arg \min_{\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_{\theta}(y|x)} \left[\log \frac{\pi_{\theta}(y|x)}{\pi_{\text{ref}}(y|x)} - \frac{1}{\beta} r_{\phi}(x, y) \right]
\end{aligned} \tag{3.3.2}$$

式子 $\frac{1}{\beta} r_{\phi}(x, y)$ 和 θ 无关，我们可以将其适当地改造变形，即：

$$\frac{1}{\beta} r_{\phi}(x, y) = \log e^{r_{\phi}(x, y)/\beta} \tag{3.3.3}$$

故优化目标变为：

$$\begin{aligned}
& \arg \min_{\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_{\theta}(y|x)} \left[\log \frac{\pi_{\theta}(y|x)}{\pi_{\text{ref}}(y|x)} - \frac{1}{\beta} r_{\phi}(x, y) \right] \\
&= \arg \min_{\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_{\theta}(y|x)} \left[\log \frac{\pi_{\theta}(y|x)}{\pi_{\text{ref}}(y|x) e^{r_{\phi}(x, y)/\beta}} \right]
\end{aligned} \tag{3.3.4}$$

此时，优化目标中 log 的分母已经不再是一个概率分布了，我们可以对分母进行归一化：

$$\begin{aligned}
Z(x) &= \sum_y \pi_{\text{ref}}(y|x) e^{r_{\phi}(x, y)/\beta}, \\
&\arg \min_{\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_{\theta}(y|x)} \left[\log \frac{\pi_{\theta}(y|x)}{\pi_{\text{ref}}(y|x) e^{r_{\phi}(x, y)/\beta}} \right] \\
&= \arg \min_{\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_{\theta}(y|x)} \left[\log \frac{\pi_{\theta}(y|x)}{(\pi_{\text{ref}}(y|x) e^{r_{\phi}(x, y)/\beta} / Z(x)) Z(x)} \right] \\
&= \arg \min_{\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_{\theta}(y|x)} \left[\log \frac{\pi_{\theta}(y|x)}{\pi^*(y|x)} - \log Z(x) \right] \\
&\Leftrightarrow \arg \min_{\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_{\theta}(y|x)} \left[\log \frac{\pi_{\theta}(y|x)}{\pi^*(y|x)} \right] \\
&= \arg \min_{\theta} \mathbb{E}_{x \sim \mathcal{D}} \left[\mathbb{D}_{\text{KL}} (\pi_{\theta}(y|x) \| \pi^*(y|x)) \right]
\end{aligned} \tag{3.3.5}$$

当 $\pi_{\theta}(y|x) = \pi^*(y|x)$ 时有最小值，即最优概率分布为 π^* 。而 $\pi^*(y|x)$ 和 $r(x, y)$ 的关系为：

$$r(x, y) = \beta \log \frac{\pi^*(y|x)}{\pi_{\text{ref}}(y|x)} + \beta \log Z(x) \tag{3.3.6}$$

我们希望对于两个分得出好坏的回答 $y_1 \succ y_2$ ，奖励模型的输出有 $r(x, y_1) > r(x, y_2)$ ，Bradley-Terry 模型只考虑两个回答得分的差值，即希望 $r(x, y_1) > r(x, y_2)$ ，且好的回答大于坏的回答的几率有：

$$\begin{aligned}
p^*(y_1 \succ y_2 | x) &= \frac{1}{1 + \exp(r(x, y_1) - r(x, y_2))} \\
&= \frac{1}{1 + \exp(\beta \log \frac{\pi^*(y_1|x)}{\pi_{\text{ref}}(y_1|x)} - \beta \log \frac{\pi^*(y_2|x)}{\pi_{\text{ref}}(y_2|x)})}
\end{aligned} \tag{3.3.7}$$

我们可以通过极大似然的方式来优化策略 π_{θ} ，即：

$$\begin{aligned}\mathcal{L}(\pi_\theta; \pi_{\text{ref}}) &= -\log \prod_{x, y_w, y_l \sim \mathcal{D}} p^*(y_w \succ y_l | x) \\ &= -\mathbb{E}_{x, y_w, y_l \sim \mathcal{D}} \left[\log \sigma(\beta \log \frac{\pi_\theta(y_w | x)}{\pi_{\text{ref}}(y_w | x)} - \beta \log \frac{\pi_\theta(y_l | x)}{\pi_{\text{ref}}(y_l | x)}) \right]\end{aligned}\quad (3.3.8)$$

此时，我们只需要两个模型了，一个`actor`和一个`reference`。原来的强化学习过程就转化成了SFT的形式。

3.3.1 DeepSpeedChat源码解析

这里笔者着重讲解框架中的损失函数计算部分，分为如下三部分

3.3.1.1 构建label mask

```
460     batch_size = batch['input_ids'].shape[0] // 2 #batch:[2*B,S]
461     chosen_input_ids = batch['input_ids'][:batch_size] #[B,S]
462     rejected_input_ids = batch['input_ids'][batch_size:] #[B,S]
463     label_mask = (batch['input_ids'] != tokenizer.pad_token_id).int() #[2B,S]
464     print(f"chosen input ids:{chosen_input_ids}")
465     for i in range(batch_size):
466         divergence_ind = (chosen_input_ids[i] !=
467                            | rejected_input_ids[i]).nonzero().squeeze(-1)
468         if len(divergence_ind) > 0:
469             divergence_ind = divergence_ind[0]
470         else:
471             divergence_ind = 0
472         label_mask[i][:divergence_ind] = 0
473         label_mask[i + batch_size][:divergence_ind] = 0
```

如果开发者将batchsize设置成 B ，由于每一个`prompt`对应了一好一坏的回答，则一共会有 $2B$ 个样本在一次迭代`step`中，前 B 个样本是好的回答，后 B 个样本是被拒绝的回答。而`label_mask`则用于选择对应`token`位置的奖励，即`label_mask = 1`的位置会被考虑，其他为0的位置不会被纳入计算，由于奖励只是针对模型生成内容的部分，且对于模型的`prefilling`而言，前 k 个`token`一致时每个位置输出的logits都一致，所以要找到`chosen`与`reject`的第一个顺序遇到不同的`token`的位置索引，将`label_mask`该索引前元素都置为0，实现对应于代码第465-473行。第463行先是把不为`< pad >`的部分置1，再接着把`prompt`和`answer`中前 k 个相同`token`对应位置的`label_mask`置0。

3.3.1.2 计算对数概率

详见代码第474 – 485行与函数`def_batch_logps`。

```
474     outputs = model(**batch, use_cache=False)
475     with torch.no_grad():
476         ref_outputs = ref_model(**batch)
477
478     logits = get_batch_logps(outputs.logits, batch['input_ids'],
479                             label_mask)
480     ref_logits = get_batch_logps(ref_outputs.logits, batch['input_ids'],
481                               label_mask)
482     chosen_logits = logits[:batch_size]
483     rejected_logits = logits[batch_size:]
484     ref_chosen_logits = ref_logits[:batch_size]
485     ref_rejected_logits = ref_logits[batch_size:]
218     # Reference: https://github.com/huggingface/trl/blob/main/trl/trainer/dpo.py
219     def get_batch_logps(logits, input_ids, label_mask):
220         labels = input_ids.clone() * label_mask #只有对应的answer部分
221         assert logits.shape[:-1] == labels.shape,
222         "Logits (batch and sequence length dim) and labels must have the same shape"
223         labels = labels[:, 1:]
224         label_mask = label_mask[:, 1:]
225         logits = logits[:, :-1, :] #[B,S-1,H]
226         per_token_logps = torch.gather(Logits.log_softmax(-1),
227                                         dim=2,
228                                         index=labels.unsqueeze(2)).squeeze(2)
229         return (per_token_logps * label_mask).sum(-1)
230
231
```

我们需要先将`label`与`label_mask`对应位置元素相乘，只剩下需要考虑的位置的`token`，因此将`logits`序列对应`token`位置的`logit` $\in \mathbf{R}^{|V| \times 1}$ 向量中该`token`的索引的分量，即 $\pi_\theta(y_j | x), j = 1, \dots, |y|$ 取出，通过`torch.gather`函数实现，最终再相加作为整句话的奖励，笔者该处的符号表述多了一个下角标 j ，和`DPO`论文中的符号不一致，原因是DeepSpeed-chat框架中奖励模型不是最朴素的奖励模型，而是Outcome Reward模型，即考虑`token`位置的奖励而非只考虑整句话最后一个位置的奖励。

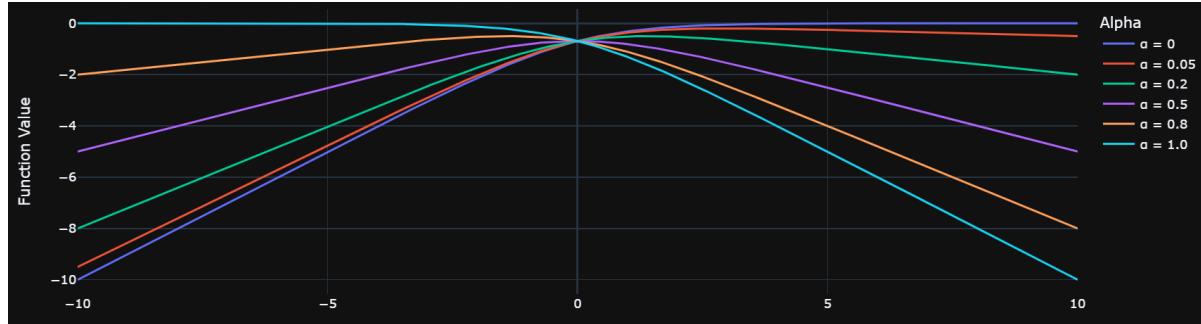
3.3.1.3 计算损失函数

```
487     logits = args.beta * ((chosen_logps - ref_chosen_logps) -  
488         (rejected_logps - ref_rejected_logps))  
489     loss = (-torch.nn.functional.logsigmoid(logits) * (1 - args.label_smoothing) - \  
490         torch.nn.functional.logsigmoid(-logits) * args.label_smoothing).mean(0)  
491     if args.print_loss:  
492         print(  
493             f"Epoch: {epoch}, Step: {step}, Rank: {torch.distributed.get_rank()}", loss = {loss}")  
494     )  
495     model.backward(loss)  
496     model.step()
```

*actor*与*reference*的对数概率变量logps和ref_logps都计算完成后就可以根据公式3.3.8带入进行损失函数计算， σ 函数内的变量对应图中代码的第487 – 489行。此外，框架中还对损失函数进行了平滑操作：

$$\log \sigma(x) \mapsto (1 - \alpha) \log \sigma(x) + \alpha \log \sigma(-x) \quad (3.3.9)$$

函数 $(1 - \alpha) \log \sigma(x) + \alpha \log \sigma(-x)$ 的图像如下：



当 $\alpha = 0$ 时，为 $\log \sigma(x) \in (-\infty, 0)$ ，而随着 α 的增大，函数右边有往下的趋势，函数左边有往上抬的趋势，当 $\alpha = 1$ 时，函数和 $\log \sigma(x)$ 图像完全反了过来（关于y轴对称）。即在训练中，虽然我们希望 $\log \frac{\pi_\theta(y_w | x)}{\pi_{\text{ref}}(y_w | x)} - \beta \log \frac{\pi_\theta(y_l | x)}{\pi_{\text{ref}}(y_l | x)}$ 大于0，但实际情况可能是小于0的，越远离0则损失函数越大，可能导致训练不稳定，因此可以通过平滑系数对原函数进行趋势的控制。

如上便是Deepspeed-Chat框架的DPO实现代码，关键部分相较于PPO便于理解，实现起来也较为简单，当然，对奖励模型损失函数的选择，并不局限于log sigmoid函数，其是无界的，即便通过平滑项进行趋势控制也难以彻底解决遇到异常样本时梯度过大导致训练不稳定的问题，我们可以选择一个有界函数使得训练过程更稳定，如保真度损失，亦或者截断梯度。

3.4 GRPO

参考文献

[X]Policy Gradients: The Foundation of RLHF

[X]Proximal Policy Optimization (PPO): The Key to LLM Alignment

[X](WIP) A Little Bit of Reinforcement Learning from Human Feedback

[x]Policy Gradient Algorithms,Weng,Lilian,liliangweng.github.io,2018

[x]深度强化学习 (三) : TRPO (Trust Region Policy Optimization , 信赖域策略优化) ,Dreammaker