从强化学习到近端策略优化（PPO）

Contents

1 引言与动机

本讲义旨在从强化学习（Reinforcement Learning, RL）的基本目标出发，系统而严谨地推导近端策略优化（Proximal Policy Optimisation, PPO）算法的完整数学框架。推导路线如下：

1. 强化学习的目标：期望累计奖励最大化
2. 策略梯度定理（Policy Gradient Theorem）与 REINFORCE 算法
3. 引入基线（Baseline）与优势函数（Advantage Function）以降低方差
4. 时序差分（TD）学习与 Actor-Critic 框架
5. 广义优势估计（GAE）及其三种等价形式
6. 重要性采样（Importance Sampling）与离策略（Off-Policy）更新
7. 代理目标（Surrogate Objective）的构造
8. PPO 的剪切目标函数与最终损失函数

本讲义假设读者具备基础的概率论、微积分和线性代数知识。

2 强化学习基础与符号约定

2.1 基本设定

考虑一个马尔可夫决策过程（Markov Decision Process, MDP），由五元组 $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$ 定义：

定义 2.1 (马尔可夫决策过程)**.** 一个 MDP 包含以下元素：

- $\mathcal{S}$：状态空间（State Space）
- $\mathcal{A}$：动作空间（Action Space）
- $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$：状态转移概率，$P(s'|s, a)$ 表示在状态 s 采取动作 a 后转移到状态 s' 的概率
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$：即时奖励函数（Reward Function）
- $\gamma \in [0, 1)$：折扣因子（Discount Factor）

定义 2.2 (策略)**.** 策略（Policy）π 是一个从状态到动作概率分布的映射。参数化策略记为 $\pi_\theta(a|s)$，表示在状态 s 下采取动作 a 的概率，其中 θ 为策略参数。

定义 2.3 (轨迹)**.** 轨迹（Trajectory）τ 是一个状态-动作-奖励序列：

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots)$$

其中 $s_0 \sim \rho_0$（初始状态分布），$a_t \sim \pi(\cdot|s_t)$，$s_{t+1} \sim P(\cdot|s_t, a_t)$，$r_t = r(s_t, a_t)$。

2.2 回报与目标函数

定义 2.4 (折扣回报)**.** 从时刻 t 开始的折扣累计回报（Discounted Return）定义为：

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

特别地，R_0 表示整条轨迹的总回报。

定义 2.5 (策略目标函数)**.** 强化学习的目标是找到最优策略参数 θ^*，使得期望回报最大化：

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R_0(\tau)] = \mathbb{E}_{s_0\sim\rho_0}\mathbb{E}_{\tau|s_0,\pi_\theta}[R_0]$$

其中 $\tau \sim \pi_\theta$ 表示轨迹 τ 由策略 π_θ 与环境交互产生。

我们的核心任务是计算 $\nabla_\theta J(\theta)$，以便通过梯度上升优化策略参数。

2.3 值函数与 Q 函数

定义 2.6 (状态值函数)**.** 策略 π 下的状态值函数（State Value Function）定义为：

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi|s_0=s}[R_0] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k}\,\middle|\, s_t = s\right]$$

定义 2.7 (动作值函数)**.** 策略 π 下的动作值函数（Action Value Function，或 Q 函数）定义为：

$$Q^\pi(s,a) = \mathbb{E}_{\tau\sim\pi|s_0=s,a_0=a}[R_0] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k}\,\middle|\, s_t = s, a_t = a\right]$$

命题 2.8 (V 与 Q 的关系)**.**

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(\cdot|s)}[Q^\pi(s,a)] = \sum_{a\in\mathcal{A}}\pi(a|s)Q^\pi(s,a) \tag{1}$$

Proof. 由全期望公式：

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[R_0|s_0 = s] \\ &= \mathbb{E}_{a_0\sim\pi(\cdot|s)}[\mathbb{E}_\pi[R_0|s_0 = s, a_0]] \\ &= \mathbb{E}_{a\sim\pi(\cdot|s)}[Q^\pi(s,a)] \end{aligned}$$

■

定义 2.9 (优势函数)**.** 优势函数（Advantage Function）定义为：

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

它衡量在状态 s 下采取动作 a 相对于平均表现的优劣程度。

注记 2.10. 由定义直接可得：

$$\mathbb{E}_{a\sim\pi(\cdot|s)}[A^\pi(s,a)] = \mathbb{E}_{a\sim\pi(\cdot|s)}[Q^\pi(s,a)] - V^\pi(s) = V^\pi(s) - V^\pi(s) = 0$$

即在策略 π 下，优势函数关于动作的期望为零。

3 策略梯度定理

本节推导策略梯度的基本形式，这是所有策略梯度方法的理论基础。

3.1 轨迹概率与似然比技巧

定义 3.1 (轨迹概率)**.** 给定策略 π_θ，轨迹 τ 的概率密度为：

$$p_\theta(\tau) = \rho_0(s_0) \prod_{t=0}^{\infty} \pi_\theta(a_t|s_t) P(s_{t+1}|s_t, a_t)$$

引理 3.2 (似然比技巧 / Log-Derivative Trick)**.** 对于任意可微的概率密度函数 $p_\theta(x)$，有：

$$\nabla_\theta p_\theta(x) = p_\theta(x) \nabla_\theta \log p_\theta(x)$$

Proof. 由对数求导的链式法则：

$$\nabla_\theta \log p_\theta(x) = \frac{\nabla_\theta p_\theta(x)}{p_\theta(x)}$$

移项即得 $\nabla_\theta p_\theta(x) = p_\theta(x) \nabla_\theta \log p_\theta(x)$。 ■

3.2 策略梯度的推导

定理 3.3 (策略梯度定理 — 基本形式)**.** 目标函数关于策略参数的梯度为：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\left(\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) R_0(\tau) \right]$$

Proof. 目标函数可写为：

$$J(\theta) = \int p_\theta(\tau) R_0(\tau) \, \mathrm{d}\tau$$

对 θ 求梯度：

$$\begin{aligned} \nabla_\theta J(\theta) &= \int \nabla_\theta p_\theta(\tau) \cdot R_0(\tau) \, \mathrm{d}\tau \\ &= \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) \cdot R_0(\tau) \, \mathrm{d}\tau \quad \text{（引理 3.2）} \\ &= \mathbb{E}_{\tau \sim \pi_\theta} [\nabla_\theta \log p_\theta(\tau) \cdot R_0(\tau)] \end{aligned}$$

接下来化简 $\nabla_\theta \log p_\theta(\tau)$：

$$\log p_\theta(\tau) = \log \rho_0(s_0) + \sum_{t=0}^{\infty} \log \pi_\theta(a_t|s_t) + \sum_{t=0}^{\infty} \log P(s_{t+1}|s_t, a_t)$$

注意 $\rho_0(s_0)$ 和 $P(s_{t+1}|s_t, a_t)$ 均与 θ 无关，因此：

$$\nabla_\theta \log p_\theta(\tau) = \sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

代入得：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\left(\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) R_0(\tau) \right]$$

■

3.3 因果性与 REINFORCE 形式

定理 3.3 的形式虽然正确，但可以利用因果性进一步简化。

引理 3.4 (因果性引理)**.** 对于 $t' < t$，有：

$$\mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot r_{t'}] = 0$$

即时刻 t 的动作对数梯度与之前时刻的奖励在期望下不相关。

Proof. 将期望按时间步展开。记 $\tau_{0:t-1} = (s_0, a_0, \ldots, s_{t-1}, a_{t-1})$ 为前 t 步的历史。则：

$$\begin{aligned}&\mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot r_{t'}] \\ &= \mathbb{E}_{\tau_{0:t-1}}\left[\mathbb{E}_{s_t,a_t|\tau_{0:t-1}}[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot r_{t'}]\right]\end{aligned}$$

由于 $t' < t$，奖励 $r_{t'} = r(s_{t'}, a_{t'})$ 完全由 $\tau_{0:t-1}$ 决定，故可提出内层期望：

$$= \mathbb{E}_{\tau_{0:t-1}}\left[r_{t'} \cdot \mathbb{E}_{s_t|\tau_{0:t-1}}\left[\mathbb{E}_{a_t\sim\pi_\theta(\cdot|s_t)}[\nabla_\theta \log \pi_\theta(a_t|s_t)]\right]\right]$$

关键步骤：对于任意状态 s，

$$\begin{aligned}\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[\nabla_\theta \log \pi_\theta(a|s)] &= \sum_a \pi_\theta(a|s) \cdot \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\ &= \sum_a \nabla_\theta \pi_\theta(a|s) \\ &= \nabla_\theta \sum_a \pi_\theta(a|s) \\ &= \nabla_\theta 1 = 0\end{aligned}$$

因此整个期望为零。 ■

定理 3.5 (策略梯度定理 — REINFORCE 形式)**.** 策略梯度可以表示为以下等价形式：

期望形式：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot R_t\right] \tag{2}$$

其中 $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ 是从时刻 t 开始的折扣回报。

有限轨迹的求和形式：

在实践中，轨迹长度有限。设一条轨迹的长度为 T（即包含时刻 $t = 0, 1, \ldots, T-1$），则该轨迹对梯度的贡献为：

$$\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot R_t \tag{3}$$

其中有限轨迹的回报为：

$$R_t = \sum_{k=0}^{T-1-t} \gamma^k r_{t+k} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-1-t} r_{T-1} \tag{4}$$

Proof. 从定理 3.3 出发：

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot R_0\right] \\ &= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot \sum_{t'=0}^{\infty}\gamma^{t'} r_{t'}\right]\end{aligned}$$

交换求和顺序：

$$= \sum_{t=0}^{\infty}\sum_{t'=0}^{\infty}\gamma^{t'}\mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot r_{t'}]$$

由引理 3.4，当 $t' < t$ 时上述期望为零，故只需考虑 $t' \geq t$：

$$\begin{aligned}&= \sum_{t=0}^{\infty}\sum_{t'=t}^{\infty}\gamma^{t'}\mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot r_{t'}] \\ &= \sum_{t=0}^{\infty}\mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot \sum_{t'=t}^{\infty}\gamma^{t'} r_{t'}\right] \\ &= \sum_{t=0}^{\infty}\mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot \gamma^t\sum_{k=0}^{\infty}\gamma^{k} r_{t+k}\right] \\ &= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot R_t\right]\end{aligned}$$

注：在实践中常省略 γ^t 因子（等价于使用未折扣的 "reward-to-go"），这不影响梯度方向，只影响不同时间步的相对权重。标准 REINFORCE 形式为：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot R_t\right]$$

■

4 基线与优势函数

REINFORCE 算法虽然理论上正确，但方差很高，难以实用。本节介绍如何通过引入基线来降低方差。

4.1 基线不改变梯度期望

定理 4.1 (基线定理)**.** 设 $b(s)$ 是任意仅依赖于状态 s 的函数（与动作无关），则：

$$\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[\nabla_\theta \log \pi_\theta(a|s)\cdot b(s)] = 0$$

因此，将基线 $b(s)$ 从回报中减去不会改变策略梯度的期望值。

Proof. 由于 $b(s)$ 与动作 a 无关，可将其提出：

$$\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[\nabla_\theta \log \pi_\theta(a|s) \cdot b(s)] = b(s) \cdot \mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[\nabla_\theta \log \pi_\theta(a|s)]$$

在引理 3.4 的证明中我们已经证明：

$$\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[\nabla_\theta \log \pi_\theta(a|s)] = \nabla_\theta \sum_a \pi_\theta(a|s) = \nabla_\theta 1 = 0$$

因此原式为零。 ■

推论 4.2 (带基线的策略梯度)**.** 对于任意状态相关的基线函数 $b(s_t)$，策略梯度可等价地写为：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot (R_t - b(s_t))\right]$$

Proof. 由定理 3.5：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot R_t\right]$$

我们希望证明减去基线后期望不变：

$$\begin{aligned}&\mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot b(s_t)\right] \\ &= \sum_{t=0}^{\infty} \mathbb{E}_{s_t\sim d_t^\pi}\left[\mathbb{E}_{a_t\sim\pi_\theta(\cdot|s_t)}[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot b(s_t)]\right]\end{aligned}$$

其中 d_t^π 是策略 π 下时刻 t 的状态分布。
由定理 4.1，内层期望为零，故整个表达式为零。因此：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot (R_t - b(s_t))\right]$$

■

4.2 最优基线与方差分析

虽然任何基线都不改变梯度的期望，但不同的基线会导致不同的方差。

命题 4.3 (单样本梯度估计的方差)**.** 考虑单个时间步的梯度估计 $g_t = \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot (R_t - b(s_t))$，其方差为：

$$\mathrm{Var}(g_t) = \mathbb{E}[(g_t - \mathbb{E}[g_t])^2] = \mathbb{E}[g_t^2] - (\mathbb{E}[g_t])^2$$

由于 $\mathbb{E}[g_t]$ 与 b 无关（基线定理），最小化方差等价于最小化 $\mathbb{E}[g_t^2]$。

定理 4.4 (最优基线)**.** 对于给定状态 s，使梯度估计方差最小的基线为：

$$b^*(s) = \frac{\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[\|\nabla_\theta \log \pi_\theta(a|s)\|^2 \cdot Q^\pi(s,a)]}{\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[\|\nabla_\theta \log \pi_\theta(a|s)\|^2]}$$

Proof. 记 $g(a) = \nabla_\theta \log \pi_\theta(a|s)$，$Q(a) = Q^\pi(s,a)$。我们要最小化：

$$\mathbb{E}_a[\|g(a)\|^2 (Q(a) - b)^2]$$

对 b 求导并令其为零：

$$\frac{\partial}{\partial b} \mathbb{E}_a[\|g(a)\|^2 (Q(a) - b)^2] = -2\mathbb{E}_a[\|g(a)\|^2 (Q(a) - b)] = 0$$

解得：

$$b^* = \frac{\mathbb{E}_a[\|g(a)\|^2 Q(a)]}{\mathbb{E}_a[\|g(a)\|^2]}$$

■

注记 4.5. 虽然最优基线的形式较复杂，但实践中通常采用简单而有效的选择：令 $b(s) = V^\pi(s)$，即状态值函数。这不是严格最优的，但计算简单且效果良好。

4.3 使用优势函数的策略梯度

定理 4.6 (策略梯度定理 — 优势函数形式)**.**

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot A^\pi(s_t, a_t) \right]$$

Proof. 由推论 4.2，取基线 $b(s_t) = V^\pi(s_t)$：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot (R_t - V^\pi(s_t)) \right]$$

注意到：

$$\mathbb{E}[R_t|s_t, a_t] = Q^\pi(s_t, a_t)$$

因此 R_t 是 $Q^\pi(s_t, a_t)$ 的无偏估计。由于我们关心的是期望：

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot (Q^\pi(s_t, a_t) - V^\pi(s_t)) \right] \\ &= \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot A^\pi(s_t, a_t) \right] \end{aligned}$$

这里最后一步使用了全期望公式：先对 R_t 关于 (s_t, a_t) 条件期望得到 Q^π，再对外层期望。 ■

5 时序差分学习与贝尔曼方程

本节介绍值函数的贝尔曼方程和时序差分（Temporal Difference, TD）方法，这是 Actor-Critic 框架和 GAE 的基础。

5.1 Bellman 方程

Bellman 方程是强化学习中最核心的递推关系，它将一个状态的价值表达为当前奖励加上未来状态价值的折扣和。这个递推结构让我们能够通过动态规划方法求解值函数。

5.1.1 基本直觉

在开始严格推导之前，我们先建立直觉理解。考虑你在状态 s 时的总回报：

- **立即回报**：你现在采取行动 a 得到奖励 $r(s,a)$
- **未来回报**：环境转移到新状态 s'，从 s' 开始你还能获得折扣后的未来回报 $\gamma V^\pi(s')$

因此，状态 s 的价值 = 立即回报的加权平均 + 折扣后的未来价值加权平均。这就是 Bellman 方程的核心思想。

5.1.2 状态值函数的 Bellman 方程

定理 5.1 (状态值函数的 Bellman 期望方程)**.** 对于策略 π，状态值函数满足以下递推关系：

求和形式：

$$\boxed{V^\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left[r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)\, V^\pi(s') \right]} \tag{5}$$

期望形式：

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(\cdot|s)} \left[r(s,a) + \gamma \mathbb{E}_{s'\sim P(\cdot|s,a)} [V^\pi(s')] \right] \tag{6}$$

两种形式等价，求和形式更适合计算，期望形式更适合理论分析。

Proof. 我们从 $V^\pi(s)$ 的定义出发，逐步展开。

步骤 1：从定义出发

值函数的定义是从状态 s 开始、按策略 π 行动所获得的期望总回报：

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,\middle|\, s_t = s \right]$$

步骤 2：分离第一步

将总回报拆分为第一步奖励和之后的回报：

$$V^\pi(s) = \mathbb{E}_\pi \left[r_t + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_t = s \right]$$

步骤 3：按第一步的选择展开

从状态 s 出发，第一步需要：

1. 按策略 π 选择行动 a，概率为 $\pi(a|s)$
2. 环境根据 $P(s'|s,a)$ 转移到下一状态 s'

按照全概率公式，对所有可能的 (a, s') 组合求加权和：

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} P(s'|s,a) \left[r(s,a) + \gamma \cdot \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_{t+1} = s' \right] \right]$$

步骤 4：识别递推结构

观察到方括号内的期望项正是 $V^\pi(s')$ 的定义：

$$\mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_{t+1} = s' \right] = V^\pi(s')$$

代入得：

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} P(s'|s,a) \left[r(s,a) + \gamma V^\pi(s') \right]$$

步骤 5：整理

注意到 $r(s,a)$ 不依赖于 s'，可以将其提到内层求和之外。由于 $\sum_{s'} P(s'|s,a) = 1$：

$$\begin{aligned} V^\pi(s) &= \sum_{a \in \mathcal{A}} \pi(a|s) \left[r(s,a) \cdot \underbrace{\sum_{s' \in \mathcal{S}} P(s'|s,a)}_{=1} + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\pi(s') \right] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\pi(s') \right] \end{aligned}$$

这就是方程 5。 ■

注记 5.2 (理解求和结构). 方程 5 包含两层求和：

- **外层** $\sum_a \pi(a|s)[\cdots]$：对策略可能选择的所有行动求加权平均，权重是策略概率
- **内层** $\sum_{s'} P(s'|s,a)V^\pi(s')$：对环境可能转移到的所有状态求加权平均，权重是转移概率

5.1.3 动作值函数的 Bellman 方程

定理 5.3 (动作值函数的 Bellman 期望方程)**.** 对于策略 π，动作值函数满足：

求和形式：

$$\boxed{Q^\pi(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s')\, Q^\pi(s',a')} \tag{7}$$

期望形式：

$$Q^\pi(s,a) = r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[\mathbb{E}_{a' \sim \pi(\cdot|s')} [Q^\pi(s',a')] \right] \tag{8}$$

Proof. 从 $Q^\pi(s,a)$ 的定义出发：

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,\middle|\, s_t = s, a_t = a \right]$$

与 V^π 的推导类似，分离第一步：

$$Q^\pi(s,a) = r(s,a) + \gamma \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_t = s, a_t = a \right]$$

注意这里 $r(s,a)$ 可以直接提出，因为第一步的行动 a 已经确定（不像 V^π 需要对行动求和）。

展开后续期望，对转移到的状态 s' 和在 s' 选择的行动 a' 求和：

$$Q^\pi(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s')\, Q^\pi(s',a')$$

■

注记 5.4 (Q^π 的 Bellman 方程更简洁的原因). 比较 V^π 和 Q^π 的 Bellman 方程：

- $V^\pi(s)$ 的方程中，$r(s,a)$ 在对 a 的求和内部，因为行动尚未确定
- $Q^\pi(s,a)$ 的方程中，$r(s,a)$ 直接提出来，因为行动已经给定

这也是为什么 Q 函数在某些场景下更方便使用。

5.1.4 V^π 与 Q^π 的相互表示

推论 5.5 (V 与 Q 的关系)**.** 状态值函数和动作值函数可以相互表示：**用 V 表示 Q：**

$$Q^\pi(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a)\, V^\pi(s') \tag{9}$$

用 Q 表示 V：

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s)\, Q^\pi(s,a) \tag{10}$$

Proof. **公式 9 的推导：**
从 $Q^\pi(s,a)$ 的定义，分离第一步奖励：

$$Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) \cdot \underbrace{\mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_{t+1} = s' \right]}_{=V^\pi(s')}$$

公式 10 的推导：
从 $V^\pi(s)$ 的定义展开第一步：

$$V^\pi(s) = \sum_a \pi(a|s) \cdot \underbrace{\mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,\middle|\, s_t = s, a_t = a \right]}_{=Q^\pi(s,a)}$$

■

注记 5.6 (直观理解). 这两个公式有清晰的物理含义：

- **公式 9**：Q 值 = 立即奖励 + 折扣后到达各状态的 V 值加权和
- **公式 10**：V 值 = 按策略选择各行动的 Q 值加权和

这两个公式可以组合使用。例如，将公式 10 代入公式 5 的右边，或将公式 9 代入公式 7，可以得到只含 V 或只含 Q 的递推关系。

注记 5.7 (贝尔曼方程的意义). 贝尔曼方程不仅仅是理论工具，它是强化学习算法的基础：

- **动态规划 (Dynamic Programming)**：直接求解贝尔曼方程
- **蒙特卡洛方法 (Monte Carlo)**：用采样回报估计值函数
- **时序差分学习 (Temporal Difference)**：用贝尔曼方程的采样版本更新估计
- **Q-learning, SARSA**：基于 Q 函数的贝尔曼方程

5.2 时序差分误差

定义 5.8 (TD 误差)**.** 给定值函数估计 V（可能是对真实 V^π 的近似），时序差分误差（TD Error）定义为：

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

命题 5.9 (TD 误差与优势函数的关系)**.** 若 $V = V^\pi$（真实值函数），则：

$$\mathbb{E}[\delta_t | s_t, a_t] = A^\pi(s_t, a_t)$$

即 TD 误差是优势函数的无偏估计。

Proof.

$$\begin{aligned}\mathbb{E}[\delta_t|s_t,a_t] &= \mathbb{E}[r_t+\gamma V^\pi(s_{t+1})-V^\pi(s_t)|s_t,a_t]\\ &= r(s_t,a_t)+\gamma\mathbb{E}_{s_{t+1}\sim P(\cdot|s_t,a_t)}[V^\pi(s_{t+1})]-V^\pi(s_t)\end{aligned}$$

由公式 9：

$$r(s_t,a_t)+\gamma\mathbb{E}_{s_{t+1}}[V^\pi(s_{t+1})]=Q^\pi(s_t,a_t)$$

因此：

$$\mathbb{E}[\delta_t|s_t,a_t]=Q^\pi(s_t,a_t)-V^\pi(s_t)=A^\pi(s_t,a_t)$$

■

5.3 多步回报与偏差-方差权衡

定义 5.10 (n 步回报)**.** 从时刻 t 开始的 n 步回报（n-step Return）定义为：

$$R_t^{(n)}=\sum_{k=0}^{n-1}\gamma^k r_{t+k}+\gamma^n V(s_{t+n})$$

其中前 n 步使用真实奖励，之后用值函数 V 进行自举（Bootstrap）。

命题 5.11 (n 步回报的性质)**.** 对于不同的 n：

1. 当 $n=1$ 时：$R_t^{(1)}=r_t+\gamma V(s_{t+1})$（1 步 TD 目标）
2. 当 $n\to\infty$ 时：$R_t^{(\infty)}=\sum_{k=0}^{\infty}\gamma^k r_{t+k}=R_t$（蒙特卡洛回报）
3. 若 $V=V^\pi$，则对所有 n：$\mathbb{E}[R_t^{(n)}|s_t,a_t]=Q^\pi(s_t,a_t)$

Proof. 第三条的证明：

$$\begin{aligned}\mathbb{E}[R_t^{(n)}|s_t,a_t] &= \mathbb{E}\left[\sum_{k=0}^{n-1}\gamma^k r_{t+k}+\gamma^n V^\pi(s_{t+n})\,\middle|\,s_t,a_t\right]\\ &= \mathbb{E}\left[\sum_{k=0}^{n-1}\gamma^k r_{t+k}+\gamma^n\mathbb{E}\left[\sum_{j=0}^{\infty}\gamma^j r_{t+n+j}\,\middle|\,s_{t+n}\right]\,\middle|\,s_t,a_t\right]\\ &= \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k}\,\middle|\,s_t,a_t\right]=Q^\pi(s_t,a_t)\end{aligned}$$

■

注记 5.12 (偏差-方差权衡). 这个权衡是 GAE 设计的核心动机:

- 小 n（如 $n=1$）：低方差（因为自举减少了随机性），但有偏（若 $V\neq V^\pi$）
- 大 n（趋向蒙特卡洛）：无偏（若 $V=V^\pi$），但高方差（累积了多步随机性）

6 广义优势估计（GAE）

广义优势估计（Generalized Advantage Estimation, GAE）通过引入参数 $\lambda \in [0,1]$ 来平滑地在偏差与方差之间进行权衡。

6.1 定义与动机

定义 6.1 (广义优势估计)**.** GAE 定义为 TD 误差的指数加权和：

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

其中 $\delta_{t+l} = r_{t+l} + \gamma V(s_{t+l+1}) - V(s_{t+l})$。

注记 6.2 (极端情况)**.** 当 λ 为 0 或 1 时：

- $\lambda = 0$：$\hat{A}_t = \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$（1 步 TD）
- $\lambda = 1$：$\hat{A}_t = \sum_{l=0}^{\infty} \gamma^l \delta_{t+l}$

命题 6.3 ($\lambda = 1$ 时 GAE 等于蒙特卡洛优势)**.** 当 $\lambda = 1$ 时：

$$\hat{A}_t^{\text{GAE}(\gamma,1)} = R_t - V(s_t)$$

其中 $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ 是蒙特卡洛回报。

Proof.

$$\begin{aligned}
\hat{A}_t^{\text{GAE}(\gamma,1)} &= \sum_{l=0}^{\infty} \gamma^l \delta_{t+l} \\
&= \sum_{l=0}^{\infty} \gamma^l (r_{t+l} + \gamma V(s_{t+l+1}) - V(s_{t+l})) \\
&= \sum_{l=0}^{\infty} \gamma^l r_{t+l} + \sum_{l=0}^{\infty} \gamma^{l+1} V(s_{t+l+1}) - \sum_{l=0}^{\infty} \gamma^l V(s_{t+l})
\end{aligned}$$

注意后两项是错位的：

$$\sum_{l=0}^{\infty} \gamma^{l+1} V(s_{t+l+1}) = \sum_{m=1}^{\infty} \gamma^m V(s_{t+m}) \quad (\text{令} m = l+1)$$

因此：

$$\begin{aligned}
\hat{A}_t^{\text{GAE}(\gamma,1)} &= \sum_{l=0}^{\infty} \gamma^l r_{t+l} + \sum_{m=1}^{\infty} \gamma^m V(s_{t+m}) - \sum_{l=0}^{\infty} \gamma^l V(s_{t+l}) \\
&= \sum_{l=0}^{\infty} \gamma^l r_{t+l} - V(s_t) \\
&= R_t - V(s_t)
\end{aligned}$$

这里利用了两个无穷级数相消的性质，以及假设 $\lim_{l\to\infty} \gamma^l V(s_{t+l}) = 0$（折扣因子保证收敛）。 ■

6.2 GAE 的三种等价形式

本节给出 GAE 的三种常用表达形式，并在附录部分证明它们的等价性。

6.2.1 形式 A：TD 误差的指数加权和（定义式）

$$\boxed{\hat{A}_t^{\text{GAE}} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}} \tag{GAE-A}$$

这是 GAE 的原始定义，直观地将多步 TD 误差以 $\gamma\lambda$ 为衰减率进行加权求和。

6.2.2 形式 B：λ-回报与值函数之差

首先定义 λ-回报：

定义 6.4 (λ-回报)**.** λ-回报（λ-Return）是各步回报的加权平均：

$$R_t^{(\lambda)} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}$$

其中 $R_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V(s_{t+n})$ 是 n 步回报。

定理 6.5 (GAE 形式 B)**.**

$$\boxed{\hat{A}_t^{\text{GAE}} = R_t^{(\lambda)} - V(s_t)} \tag{GAE-B}$$

6.2.3 形式 C：递推形式

定理 6.6 (GAE 形式 C)**.**

$$\boxed{\hat{A}_t = \delta_t + \gamma\lambda \hat{A}_{t+1}} \tag{GAE-C}$$

边界条件：在有限轨迹中，若 T 为终止时刻，则 $\hat{A}_T = 0$。

注记 6.7 (实现中的反向计算)**.** 递推形式公式 GAE-C 在实现中非常有用。给定一条长度为 T 的轨迹，可以从后向前计算：

$$\begin{aligned}
\hat{A}_{T-1} &= \delta_{T-1} \\
\hat{A}_{T-2} &= \delta_{T-2} + \gamma\lambda \hat{A}_{T-1} \\
&\vdots \\
\hat{A}_t &= \delta_t + \gamma\lambda \hat{A}_{t+1}
\end{aligned}$$

这种反向迭代只需 $O(T)$ 时间复杂度。

例 6.8 (GAE 的代码实现)**.** 以下是 GAE 计算的 Python 实现：

Listing 1: GAE 计算

```python
import numpy as np

def compute_gae(rewards, values, gamma=0.99, lam=0.95):
    """
    计算广义优势估计（GAE）
```

```
    参数:
        rewards: 奖励序列 [r_0, r_1, ..., r_{T-1}], 长度为 T
        values:  值函数估计 [V(s_0), V(s_1), ..., V(s_T)], 长度为 T+1
                 (包含终止状态的值, 通常 V(s_T) = 0)
        gamma:   折扣因子
        lam:     GAE参数 lambda

    返回:
        advantages: GAE优势估计 [A_0, A_1, ..., A_{T-1}]
    """
    T = len(rewards)
    advantages = np.zeros(T)

    # 步骤1: 计算所有TD误差 delta_t = r_t + gamma * V(s_{t+1}) - V(s_t)
    deltas = np.zeros(T)
    for t in range(T):
        deltas[t] = rewards[t] + gamma * values[t + 1] - values[t]

    # 步骤2: 从后向前递推计算GAE
    # 边界条件: A_T = 0 (隐含在循环中)
    gae = 0  # 存储 A_{t+1}
    for t in reversed(range(T)):  # t = T-1, T-2, ..., 1, 0
        gae = deltas[t] + gamma * lam * gae  # A_t = delta_t + gamma * lam
            * A_{t+1}
        advantages[t] = gae

    return advantages
```

使用示例:

```
# 假设一条长度为5的轨迹
rewards = np.array([1.0, 0.5, 2.0, -1.0, 0.0])  # r_0 到 r_4
values = np.array([10.0, 9.5, 9.0, 8.0, 7.5, 0.0])  # V(s_0) 到 V(s_5), 终
    止状态值为0

advantages = compute_gae(rewards, values, gamma=0.99, lam=0.95)
# advantages[t] 即为 \hat{A}_t
```

向量化实现:

Listing 2: GAE 向量化计算

```
def compute_gae_vectorized(rewards, values, gamma=0.99, lam=0.95):
    """向量化的GAE计算"""
    T = len(rewards)

    # 计算所有TD误差
    deltas = rewards + gamma * values[1:] - values[:-1]

    # 反向递推 (仍需循环, 但操作更简洁)
```

```
    advantages = np.zeros(T)
    gae = 0
    for t in reversed(range(T)):
        gae = deltas[t] + gamma * lam * gae
        advantages[t] = gae

    return advantages
```

计算值函数目标：

值函数的回归目标可以直接从 GAE 得到：

```
# 值函数目标：R_t^{target} = A_t + V(s_t)
returns = advantages + values[:-1]  # lambda-回报
```

注记 6.9 (参数选择指南). γ 和 λ 有不同的作用：

- γ：控制对未来奖励的折扣，通常取 0.99 或 0.995
- λ：控制偏差-方差权衡
  - λ 接近 0：低方差，但若 V 不准确则有偏差
  - λ 接近 1：低偏差（趋向蒙特卡洛），但高方差
  - 实践中常取 $\lambda \in [0.9, 0.99]$

7 重要性采样与离策略学习

在实际训练中，我们面临一个核心问题：采集数据需要时间，但我们希望用同一批数据多次更新策略。这就产生了一个矛盾——数据是用旧策略 $\pi_{\theta_{\text{old}}}$ 采集的，但我们想要计算新策略 π_θ 的梯度。

重要性采样（Importance Sampling）正是解决这个问题的工具，它使这种离策略（Off-Policy）更新成为可能。

7.1 问题的精确表述

回顾策略梯度定理（定理 4.6）：

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim d^{\pi_\theta}, a\sim\pi_\theta(\cdot|s)}[\nabla_\theta \log \pi_\theta(a|s) \cdot A^{\pi_\theta}(s,a)] \tag{11}$$

这个公式要求我们在新策略 π_θ 诱导的分布下采样。具体来说，有两个分布依赖于 π_θ：

1. **状态分布** $d^{\pi_\theta}(s)$：按策略 π_θ 与环境长期交互时，访问各状态的频率
2. **动作分布** $\pi_\theta(a|s)$：在状态 s 下选择各动作的概率

但我们手头只有旧策略 $\pi_{\theta_{\text{old}}}$ 采集的数据，这些数据服从分布 $(s,a) \sim d^{\pi_{\theta_{\text{old}}}}(s) \cdot \pi_{\theta_{\text{old}}}(a|s)$。

7.2 重要性采样原理

定理 7.1 (重要性采样)**.** 设 p 和 q 是两个概率分布，f 是任意函数。若 $q(x) > 0$ 对所有 $p(x) > 0$ 的 x 成立，则：

$$\mathbb{E}_{x\sim p}[f(x)] = \mathbb{E}_{x\sim q}\left[\frac{p(x)}{q(x)}f(x)\right]$$

Proof.

$$\begin{aligned}\mathbb{E}_{x\sim q}\left[\frac{p(x)}{q(x)}f(x)\right] &= \int q(x)\cdot\frac{p(x)}{q(x)}f(x)\,\mathrm{d}x \\ &= \int p(x)f(x)\,\mathrm{d}x = \mathbb{E}_{x\sim p}[f(x)]\end{aligned}$$

■

定义 7.2 (重要性比率)**.** 对于策略梯度优化，定义重要性比率（Importance Ratio）为：

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

它衡量新策略相对于旧策略在给定状态-动作对上的概率比值。

7.3 离策略学习

定理 7.3 (离策略策略梯度)**.** 使用旧策略 $\pi_{\theta_{\text{old}}}$ 采样的数据，新策略 π_θ 的策略梯度可以估计为：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_{\theta_{\text{old}}}}\left[\sum_t r_t(\theta)\nabla_\theta\log\pi_\theta(a_t|s_t)\hat{A}_t\right]$$

7.3.1 对动作分布应用重要性采样

首先处理动作分布的差异。固定状态 s，我们想计算：

$$\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[f(s,a)]$$

但只有从 $\pi_{\theta_{\text{old}}}(\cdot|s)$ 采样的动作。

定理 7.4 (动作空间的重要性采样)**.** 对于任意函数 $f(s,a)$：

$$\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[f(s,a)] = \mathbb{E}_{a\sim\pi_{\theta_{\text{old}}}(\cdot|s)}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}f(s,a)\right] \tag{12}$$

应用到策略梯度，取 $f(s,a) = \nabla_\theta\log\pi_\theta(a|s)\cdot A(s,a)$：

$$\begin{aligned}&\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[\nabla_\theta\log\pi_\theta(a|s)\cdot A(s,a)] \\ &= \mathbb{E}_{a\sim\pi_{\theta_{\text{old}}}(\cdot|s)}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}\cdot\nabla_\theta\log\pi_\theta(a|s)\cdot A(s,a)\right]\end{aligned} \tag{13}$$

7.3.2 状态分布的近似

现在还剩下状态分布的问题。严格来说，完整的离策略梯度应该是：

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim d^{\pi_\theta}}\left[\mathbb{E}_{a\sim\pi_{\theta_{\text{old}}}(\cdot|s)}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}\cdot\nabla_\theta\log\pi_\theta(a|s)\cdot A(s,a)\right]\right] \tag{14}$$

但我们的数据中，状态是从 $d^{\pi_{\theta_{\text{old}}}}$ 采样的，不是从 d^{π_θ} 采样的。

为什么状态分布校正很困难？

理论上，我们可以对状态分布也做重要性采样：

$$\mathbb{E}_{s\sim d^{\pi_\theta}}[g(s)] = \mathbb{E}_{s\sim d^{\pi_{\theta_{\text{old}}}}}\left[\frac{d^{\pi_\theta}(s)}{d^{\pi_{\theta_{\text{old}}}}(s)}g(s)\right]$$

但计算状态分布的重要性权重 $d^{\pi_\theta}(s)/d^{\pi_{\theta_{\text{old}}}}(s)$ 需要知道两个策略的稳态分布，这依赖于：

- 完整的环境转移模型 $P(s'|s,a)$
- 求解马尔可夫链的稳态分布（计算复杂度高）
- 在连续或高维状态空间中几乎不可行

实践中的近似：

PPO（以及大多数策略梯度方法）采用一个关键近似：

$$\boxed{d^{\pi_\theta}(s)\approx d^{\pi_{\theta_{\text{old}}}}(s)} \tag{15}$$

即假设新旧策略访问各状态的频率大致相同。

注记 7.5 (近似的合理性). 这个近似在以下条件下是合理的：

1. **策略变化小**：如果 $\pi_\theta\approx\pi_{\theta_{\text{old}}}$，那么两个策略诱导的轨迹分布也相近，从而状态访问频率相近
2. **更新频繁**：每次只用少量数据做小幅更新，而不是用大量数据做大幅更新
3. **PPO 的设计**：裁剪机制正是为了确保 π_θ 不会偏离 $\pi_{\theta_{\text{old}}}$ 太远，从而让这个近似成立

7.3.3 最终的离策略梯度估计

在状态分布近似 (15) 下，我们得到可用旧策略数据估计的梯度：

定理 7.6 (最终的离策略梯度)**.** 使用旧策略 $\pi_{\theta_{\text{old}}}$ 采样的数据，新策略 π_θ 的策略梯度可以近似为：

期望形式：

$$\nabla_\theta J(\theta)\approx\mathbb{E}_{(s,a)\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}\cdot\nabla_\theta\log\pi_\theta(a|s)\cdot\hat{A}(s,a)\right] \tag{16}$$

样本估计形式：

给定 N 个从 $\pi_{\theta_{\text{old}}}$ 采样的状态-动作对 $\{(s_i,a_i)\}_{i=1}^N$：

$$\boxed{\widehat{\nabla_\theta J(\theta)} = \frac{1}{N}\sum_{i=1}^{N}\frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{\text{old}}}(a_i|s_i)}\cdot\nabla_\theta\log\pi_\theta(a_i|s_i)\cdot\hat{A}_i} \tag{17}$$

其中 $\hat{A}_i$ 是状态-动作对 (s_i,a_i) 的优势估计（通常用 GAE 计算）。

注记 7.7 (重要性比率的简化). 注意到：

$$\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \cdot \nabla_\theta \log \pi_\theta(a|s) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \cdot \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} = \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$$

所以公式 (17) 也可以写成：

$$\widehat{\nabla_\theta J(\theta)} = \frac{1}{N} \sum_{i=1}^{N} \frac{\nabla_\theta \pi_\theta(a_i|s_i)}{\pi_{\theta_{\text{old}}}(a_i|s_i)} \cdot \hat{A}_i$$

但在实现中，通常保留 $r_t(\theta) \cdot \nabla_\theta \log \pi_\theta$ 的形式，因为自动微分框架更容易处理 log 概率。

7.3.4 重要性采样的方差问题

重要性采样虽然提供了正确的期望估计，但可能带来严重的方差问题。

命题 7.8 (重要性采样的方差)**.** 设 $X \sim q$，用重要性采样估计 $\mathbb{E}_p[f(X)]$。则：

$$\text{Var}_q\left[\frac{p(X)}{q(X)} f(X)\right] = \mathbb{E}_q\left[\left(\frac{p(X)}{q(X)}\right)^2 f(X)^2\right] - (\mathbb{E}_p[f(X)])^2$$

当 p 和 q 差异较大时，比值 $p(X)/q(X)$ 可能非常大或非常小，导致方差爆炸。

在策略梯度中的具体表现：

- 如果新策略 π_θ 大幅增加了某个动作的概率，而旧策略 $\pi_{\theta_{\text{old}}}$ 很少选择该动作，则 $r_t(\theta) = \pi_\theta(a|s)/\pi_{\theta_{\text{old}}}(a|s)$ 会非常大
- 对于轨迹级别的重要性采样，问题更严重：

$$\frac{p_\theta(\tau)}{p_{\theta_{\text{old}}}(\tau)} = \prod_{t=0}^{T-1} \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

  多个比率相乘会使方差指数级增长

例 7.9 (方差爆炸的数值示例)**.** 假设一条长度为 $T = 100$ 的轨迹，每一步的重要性比率 $r_t(\theta)$ 在 $[0.9, 1.1]$ 范围内（看起来很温和）。

轨迹级别的重要性权重为：

$$\prod_{t=0}^{99} r_t(\theta) \in [0.9^{100}, 1.1^{100}] \approx [2.7 \times 10^{-5}, 1.4 \times 10^{4}]$$

即使每步的比率变化很小，累积效应也会产生 10^9 量级的权重差异，这会导致梯度估计极不稳定。

PPO 的解决方案：

正是因为重要性采样的方差问题，PPO 才引入裁剪机制（见下一节）。裁剪通过限制 $r_t(\theta)$ 的范围来控制方差，代价是引入一定的偏差，但换来了训练稳定性。

这体现了强化学习中常见的权衡：

无偏但高方差 $\longleftrightarrow$ 有偏但低方差

PPO 选择了后者，实践证明这是非常有效的。

8 代理目标函数与信任域方法

本节构造用于优化的代理目标函数（Surrogate Objective），并解释为什么需要限制策略更新的幅度。这是理解 PPO 设计的关键。

8.1 为什么需要代理目标？

回顾上一节的离策略梯度公式 (17)：

$$\widehat{\nabla_\theta J(\theta)} = \frac{1}{N}\sum_{i=1}^{N} \frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{\text{old}}}(a_i|s_i)} \cdot \nabla_\theta \log \pi_\theta(a_i|s_i) \cdot \hat{A}_i$$

这个公式给出了梯度的估计，但在实际优化中，我们通常希望有一个**目标函数**而不仅仅是梯度。原因包括：

- 现代深度学习框架（PyTorch、JAX 等）基于自动微分，输入是标量损失函数
- 有了目标函数，可以方便地添加约束、正则项、进行多次迭代优化
- 目标函数的值可以用于监控训练进度

因此，我们需要构造一个目标函数 $L(\theta)$，使其梯度等于策略梯度。

8.2 代理目标的构造

定义 8.1 (代理目标函数)**.** 定义代理目标函数（Surrogate Objective）为：

期望形式：

$$L^{\text{PG}}(\theta) = \mathbb{E}_{(s,a)\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}\hat{A}(s,a)\right] = \mathbb{E}_t[r_t(\theta)\hat{A}_t] \tag{18}$$

样本估计形式：

$$\hat{L}^{\text{PG}}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{\text{old}}}(a_i|s_i)}\hat{A}_i = \frac{1}{N}\sum_{i=1}^{N} r_i(\theta)\hat{A}_i \tag{19}$$

其中 $r_t(\theta) = \pi_\theta(a_t|s_t)/\pi_{\theta_{\text{old}}}(a_t|s_t)$ 是重要性比率，$\hat{A}_t$ 是优势估计。

注记 8.2 (为什么叫代理目标？)**.** 我们真正想优化的是策略的期望回报 $J(\theta)$。但 $J(\theta)$ 需要在新策略 π_θ 下采样才能估计，而我们只有旧策略的数据。

$L^{\text{PG}}(\theta)$ 是 $J(\theta)$ 的一个代理——它可以用旧数据计算，且在 $\theta = \theta_{\text{old}}$ 附近与 $J(\theta)$ 有相同的行为（相同的梯度）。

8.3 代理目标的性质

定理 8.3 (代理目标的一阶近似性质)**.** 代理目标 $L^{\text{PG}}(\theta)$ 满足：

1. **在当前点的值**：$L^{\text{PG}}(\theta_{\text{old}}) = \mathbb{E}_t[\hat{A}_t]$
2. **在当前点的梯度**：$\nabla_\theta L^{\text{PG}}(\theta)\big|_{\theta=\theta_{\text{old}}} = \nabla_\theta J(\theta)\big|_{\theta=\theta_{\text{old}}}$

即代理目标在当前策略处与真实目标有相同的梯度方向。

Proof. **第一条的证明：**
当 $\theta = \theta_{\text{old}}$ 时，重要性比率 $r_t(\theta_{\text{old}}) = \pi_{\theta_{\text{old}}}(a_t|s_t)/\pi_{\theta_{\text{old}}}(a_t|s_t) = 1$。因此：

$$L^{\text{PG}}(\theta_{\text{old}}) = \mathbb{E}_t[1 \cdot \hat{A}_t] = \mathbb{E}_t[\hat{A}_t]$$

第二条的证明：
对 $L^{\text{PG}}(\theta)$ 求梯度：

$$\begin{aligned}\nabla_\theta L^{\text{PG}}(\theta) &= \nabla_\theta \mathbb{E}_t[r_t(\theta)\hat{A}_t] \\ &= \mathbb{E}_t[\nabla_\theta r_t(\theta) \cdot \hat{A}_t] \quad (\hat{A}_t \text{ 不依赖于 } \theta)\end{aligned}$$

计算 $\nabla_\theta r_t(\theta)$：

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

在 $\theta = \theta_{\text{old}}$ 处：

$$\begin{aligned}\nabla_\theta L^{\text{PG}}(\theta)\big|_{\theta=\theta_{\text{old}}} &= \mathbb{E}_t\left[\frac{\nabla_\theta \pi_\theta(a_t|s_t)\big|_{\theta=\theta_{\text{old}}}}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \cdot \hat{A}_t\right] \\ &= \mathbb{E}_t\left[\frac{\nabla_\theta \pi_{\theta_{\text{old}}}(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \cdot \hat{A}_t\right] \\ &= \mathbb{E}_t[\nabla_\theta \log \pi_{\theta_{\text{old}}}(a_t|s_t) \cdot \hat{A}_t]\end{aligned}$$

这正是策略梯度定理（定理 4.6）在 $\theta = \theta_{\text{old}}$ 处的表达式。 ■

注记 8.4 (几何直观). 定理 8.3 说明 $L^{\text{PG}}(\theta)$ 是 $J(\theta)$ 在 θ_{old} 处的一**阶泰勒近似**（忽略常数项的差异）：

$$J(\theta) \approx J(\theta_{\text{old}}) + \nabla_\theta J(\theta_{\text{old}})^\top (\theta - \theta_{\text{old}})$$

在 θ_{old} 附近，两个函数的等高线方向一致，沿着 L^{PG} 上升的方向走，也是沿着 J 上升的方向走。但走得太远，这个近似就不再准确。

8.4 直接优化代理目标的问题

既然 $L^{\text{PG}}(\theta)$ 的梯度等于策略梯度，能否直接最大化它?

答案是：不能，或者说效果很差。

例 8.5 (代理目标的失效)**.** 考虑以下场景：

- 在某个状态 s，旧策略 $\pi_{\theta_{\text{old}}}(a^*|s) = 0.01$（很少选择动作 a^*）
- 优势估计 $\hat{A}(s, a^*) = 100$（动作 a^* 非常好）
- 该样本对代理目标的贡献是 $r(\theta) \cdot 100 = \frac{\pi_\theta(a^*|s)}{0.01} \cdot 100$

为了最大化代理目标，优化器会尽可能增大 $\pi_\theta(a^*|s)$。如果新策略变成 $\pi_\theta(a^*|s) = 0.99$，则：

$$r(\theta) = \frac{0.99}{0.01} = 99$$

这个样本的贡献从 $1 \times 100 = 100$ 变成了 $99 \times 100 = 9900$。

问题：

1. 新策略与旧策略差异巨大（概率从 1% 变成 99%）
2. 优势估计 $\hat{A} = 100$ 是基于旧策略计算的，对新策略可能完全不适用
3. 状态分布近似 $d^{\pi_\theta} \approx d^{\pi_{\theta_{\text{old}}}}$ 完全失效
4. 策略可能过拟合到这一批数据，在新数据上表现很差

这种现象在强化学习中被称为**策略崩溃（Policy Collapse）**：一次过大的更新导致策略性能急剧下降，且难以恢复。

8.5 信任域的思想

解决上述问题的核心思想是：**限制每次更新的幅度，只在代理目标可信的区域内优化**。

定义 8.6 (信任域)**.** 在优化中，**信任域（Trust Region）**是指目标函数的近似（如一阶或二阶泰勒展开）足够准确的参数区域。在这个区域内，优化近似目标等价于优化真实目标；超出这个区域，近似可能严重失真。

对于策略优化，信任域对应于新策略 π_θ 与旧策略 $\pi_{\theta_{\text{old}}}$ 足够接近的区域。

定义 8.7 (策略间的 KL 散度)**.** 两个策略在状态 s 下的 KL 散度定义为：

$$\text{KL}(\pi_{\theta_{\text{old}}}(\cdot|s)\|\pi_\theta(\cdot|s)) = \sum_{a\in\mathcal{A}} \pi_{\theta_{\text{old}}}(a|s) \log \frac{\pi_{\theta_{\text{old}}}(a|s)}{\pi_\theta(a|s)} \tag{20}$$

KL 散度（Kullback-Leibler Divergence）提供了一种度量策略之间的距离的方式。策略间的平均 KL 散度定义为：

$$\bar{D}_{\text{KL}}(\theta_{\text{old}}, \theta) = \mathbb{E}_{s\sim d^{\pi_{\theta_{\text{old}}}}}[\text{KL}(\pi_{\theta_{\text{old}}}(\cdot|s)\|\pi_\theta(\cdot|s))] \tag{21}$$

注记 8.8 (为什么用 KL 散度？)**.**

- KL 散度度量两个概率分布的差异，$\text{KL}(p\|q) = 0$ 当且仅当 $p = q$
- KL 散度与重要性比率直接相关：$\text{KL}(p\|q) = \mathbb{E}_p[\log(p/q)]$
- 理论上可以证明，控制 KL 散度能够保证策略性能的单调改进（TRPO 的理论基础）

8.6 信任域策略优化（TRPO）

定义 8.9 (信任域策略优化)**. TRPO（Trust Region Policy Optimisation）**将策略优化表述为带约束的优化问题：

$$\boxed{\begin{aligned} \max_\theta \quad & L^{\text{PG}}(\theta) = \mathbb{E}_t[r_t(\theta)\hat{A}_t] \\ \text{s.t.} \quad & \bar{D}_{\text{KL}}(\theta_{\text{old}}, \theta) \le \delta \end{aligned}} \tag{22}$$

其中 $\delta > 0$ 是信任域半径，控制允许的最大策略变化。

注记 8.10 (TRPO 的直观理解)**.** TRPO 的含义是：在新策略与旧策略的 *KL* 散度不超过 δ 的约束下，最大化代理目标。

- 当 δ 很小时，新策略被限制在旧策略附近，更新保守但稳定
- 当 δ 较大时，允许更大的策略变化，更新激进但可能不稳定
- 典型取值：$\delta \in [0.001, 0.01]$

8.6.1 TRPO 的理论保证

TRPO 有严格的理论支持，这里给出核心结论：

定理 8.11 (策略改进下界)**.** 设 $\epsilon = \max_s \mathrm{KL}(\pi_{\theta_{\text{old}}}(\cdot|s)\|\pi_\theta(\cdot|s))$，则：

$$J(\theta) \geq L^{\text{PG}}(\theta) - \frac{4\epsilon\gamma}{(1-\gamma)^2} \max_{s,a} |A^{\pi_{\theta_{\text{old}}}}(s,a)| \tag{23}$$

即真实目标 $J(\theta)$ 有一个依赖于代理目标和 KL 散度的下界。

推论 8.12 (单调改进)**.** 如果每次更新都增加代理目标 $L^{\text{PG}}(\theta)$ 且保持 KL 散度足够小，则真实目标 $J(\theta)$ 也会改进（或至少不会下降太多）。

8.6.2 TRPO 的求解方法

约束优化问题 (22) 的求解需要特殊技术：

1. **二阶近似**：将 KL 约束近似为二次型 $\frac{1}{2}(\theta - \theta_{\text{old}})^\top F(\theta - \theta_{\text{old}}) \leq \delta$，其中 F 是 Fisher 信息矩阵
2. **共轭梯度法**：由于 F 通常很大（参数维度的平方），不能直接求逆。使用共轭梯度法迭代求解 $F^{-1}g$，其中 $g = \nabla_\theta L^{\text{PG}}$
3. **线搜索**：找到满足约束的最大步长

注记 8.13 (TRPO 的实现复杂性). TRPO 的主要缺点是实现复杂：

- 需要计算 Fisher 信息矩阵与向量的乘积（Hessian-vector product）
- 共轭梯度法需要多次迭代
- 线搜索增加了计算开销
- 代码实现容易出错，调试困难

这些复杂性促使了 PPO 的诞生——用更简单的方法达到类似的效果。

9 PPO：近端策略优化

近端策略优化（Proximal Policy Optimisation, PPO）是一种简单而有效的策略梯度方法，它通过裁剪目标函数来隐式地约束策略更新幅度。

9.1 裁剪代理目标

定义 9.1 (裁剪操作)**.** 裁剪函数定义为：

$$\text{clip}(x, a, b) = \max(\min(x, b), a) = \begin{cases} a & \text{if } x < a \\ x & \text{if } a \leq x \leq b \\ b & \text{if } x > b \end{cases}$$

定义 9.2 (PPO 裁剪目标)**.** PPO 的裁剪代理目标定义为：

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

其中 $\epsilon > 0$ 是裁剪参数（通常取 $\epsilon = 0.1$ 或 0.2）。

9.2 裁剪机制的分析

定理 9.3 (PPO 裁剪的行为)**.** 设 $L_t^{\text{CLIP}}(\theta) = \min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)$。则：

情况 1： 当 $\hat{A}_t > 0$（动作优于平均）时：

$$L_t^{\text{CLIP}}(\theta) = \min(r_t(\theta), 1+\epsilon) \cdot \hat{A}_t$$

目标函数对 $r_t(\theta) > 1+\epsilon$ 的区域不提供额外激励。

情况 2： 当 $\hat{A}_t < 0$（动作劣于平均）时：

$$L_t^{\text{CLIP}}(\theta) = \max(r_t(\theta), 1-\epsilon) \cdot \hat{A}_t$$

目标函数对 $r_t(\theta) < 1-\epsilon$ 的区域不提供额外惩罚。

Proof. **情况 1：** $\hat{A}_t > 0$

记 $r = r_t(\theta)$，$A = \hat{A}_t > 0$。

裁剪后的项为：

$$\text{clip}(r, 1-\epsilon, 1+\epsilon) \cdot A = \begin{cases} (1-\epsilon)A & \text{if } r < 1-\epsilon \\ r \cdot A & \text{if } 1-\epsilon \le r \le 1+\epsilon \\ (1+\epsilon)A & \text{if } r > 1+\epsilon \end{cases}$$

比较 rA 与裁剪后的项：

- 若 $r < 1-\epsilon$：$rA < (1-\epsilon)A$，故 $\min(rA, (1-\epsilon)A) = rA$
- 若 $1-\epsilon \le r \le 1+\epsilon$：$\min(rA, rA) = rA$
- 若 $r > 1+\epsilon$：$rA > (1+\epsilon)A$，故 $\min(rA, (1+\epsilon)A) = (1+\epsilon)A$

合并：$L_t^{\text{CLIP}} = \min(r, 1+\epsilon) \cdot A$。

情况 2： $\hat{A}_t < 0$

设 $A = \hat{A}_t < 0$（为负数）。

裁剪后的项为：

$$\text{clip}(r, 1-\epsilon, 1+\epsilon) \cdot A = \begin{cases} (1-\epsilon)A & \text{if } r < 1-\epsilon \\ r \cdot A & \text{if } 1-\epsilon \le r \le 1+\epsilon \\ (1+\epsilon)A & \text{if } r > 1+\epsilon \end{cases}$$

由于 $A < 0$，大小关系反转：

- 若 $r < 1-\epsilon$：$(1-\epsilon)A > rA$，故 $\min(rA, (1-\epsilon)A) = rA$
- 若 $1-\epsilon \le r \le 1+\epsilon$：$\min(rA, rA) = rA$
- 若 $r > 1+\epsilon$：$(1+\epsilon)A < rA$，故 $\min(rA, (1+\epsilon)A) = (1+\epsilon)A$

等价地写成：$L_t^{\text{CLIP}} = \max(r, 1-\epsilon) \cdot A$（利用 $\min(ra, ba) = \max(r, b) \cdot a$ 当 $a < 0$）。 ■

注记 9.4 (直观理解). Clip 机制悲观地限制了策略改变的激励，从而保持更新稳定。

- 当 $\hat{A}_t > 0$：我们希望增加该动作的概率（使 $r_t > 1$），但裁剪限制了增益上限为 $(1+\epsilon)\hat{A}_t$
- 当 $\hat{A}_t < 0$：我们希望减少该动作的概率（使 $r_t < 1$），但裁剪限制了减弱下限为 $(1-\epsilon)\hat{A}_t$

9.3 一阶性质的保持

命题 9.5 (PPO 保持一阶近似)**.** 在 $\theta = \theta_{\text{old}}$ 处：

$$\nabla_\theta L^{\text{CLIP}}(\theta)\big|_{\theta=\theta_{\text{old}}} = \nabla_\theta L^{\text{PG}}(\theta)\big|_{\theta=\theta_{\text{old}}} = \nabla_\theta J(\theta)\big|_{\theta=\theta_{\text{old}}}$$

Proof. 在 $\theta = \theta_{\text{old}}$ 时，$r_t(\theta_{\text{old}}) = 1 \in [1-\epsilon, 1+\epsilon]$。
因此，在邻域内 $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) = r_t(\theta)$，两个目标函数局部相同：

$$L_t^{\text{CLIP}}(\theta) = r_t(\theta)\hat{A}_t = L_t^{\text{PG}}(\theta)$$

在当前点的梯度自然相等。 ■

9.4 PPO 的完整损失函数

PPO 在训练时需要同时优化两个目标：策略网络（Actor）和值函数网络（Critic）。本节详细展开这两个损失项的完整形式。

9.4.1 策略损失

定义 9.6 (PPO 策略损失)**.** 策略损失是裁剪目标的负值（因为我们使用梯度下降最小化损失，而原目标是要最大化的）：

期望形式：

$$\mathcal{L}^{\text{policy}}(\theta) = -\mathbb{E}_t[L_t^{\text{CLIP}}(\theta)] = -\mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{24}$$

样本估计形式：
给定从旧策略采样的 N 个样本 $\{(s_i, a_i, \hat{A}_i)\}_{i=1}^N$：

$$\boxed{\hat{\mathcal{L}}^{\text{policy}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\min\left(\frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{\text{old}}}(a_i|s_i)}\hat{A}_i,\ \text{clip}\left(\frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{\text{old}}}(a_i|s_i)}, 1-\epsilon, 1+\epsilon\right)\hat{A}_i\right)} \tag{25}$$

注记 9.7 (优势估计的计算). 优势 $\hat{A}_t$ 使用 GAE 计算：

$$\hat{A}_t = \sum_{l=0}^{T-t-1}(\gamma\lambda)^l\delta_{t+l}, \quad \text{其中}\delta_{t+l} = r_{t+l} + \gamma V_{\phi_{\text{old}}}(s_{t+l+1}) - V_{\phi_{\text{old}}}(s_{t+l}) \tag{26}$$

这里 $V_{\phi_{\text{old}}}$ 是用旧参数计算的值函数估计。注意：

- $\hat{A}_t$ 在每个 PPO 更新周期开始时计算一次，之后固定不变
- 在同一批数据上进行多个 epoch 的优化时，$\hat{A}_t$ 不会重新计算

- 这是因为 $\hat{A}_t$ 应该反映旧策略下的优势，而非当前正在优化的策略

注记 9.8 (优势估计的归一化). 在实践中，通常对一个 batch 内的优势估计进行归一化：

$$\hat{A}_i^{\text{norm}} = \frac{\hat{A}_i - \mu_{\hat{A}}}{\sigma_{\hat{A}} + \epsilon_{\text{std}}} \tag{27}$$

其中 $\mu_{\hat{A}} = \frac{1}{N}\sum_{i=1}^{N} \hat{A}_i$ 是均值，$\sigma_{\hat{A}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{A}_i - \mu_{\hat{A}})^2}$ 是标准差，$\epsilon_{\text{std}} \approx 10^{-8}$ 防止除零。

为什么要归一化？

- 不同任务的奖励尺度可能差异很大（有的任务奖励在 $[-1, 1]$，有的在 $[-1000, 1000]$）
- 归一化后优势的尺度统一，便于选择通用的学习率和裁剪参数
- 减去均值使得大约一半的优势为正、一半为负，梯度更新更平衡

归一化不改变梯度的方向（只是缩放），但能显著提高训练稳定性。

9.4.2 值函数损失

值函数网络 $V_\phi(s)$ 的作用是估计状态值函数，为 GAE 提供 $V(s_t)$ 和 $V(s_{t+1})$。

定义 9.9 (值函数损失)**.** 值函数损失是预测值与目标值之间的均方误差：

期望形式：

$$\mathcal{L}^{\text{value}}(\phi) = \mathbb{E}_t\left[\left(V_\phi(s_t) - R_t^{\text{target}}\right)^2\right] \tag{28}$$

样本估计形式：

$$\boxed{\hat{\mathcal{L}}^{\text{value}}(\phi) = \frac{1}{N}\sum_{i=1}^{N}\left(V_\phi(s_i) - R_i^{\text{target}}\right)^2} \tag{29}$$

定义 9.10 (值函数的回归目标)**.** 值函数的目标 R_t^{target} 应该是从状态 s_t 出发的期望回报的估计。有两种等价的计算方式：

方式 1：直接使用 λ-回报

$$R_t^{\text{target}} = R_t^{(\lambda)} \tag{30}$$

方式 2：GAE 加旧值函数（实践中更常用，因为 GAE 已经算好了）

$$R_t^{\text{target}} = \hat{A}_t + V_{\phi_{\text{old}}}(s_t) \tag{31}$$

两种方式等价，因为由 GAE 的形式 B（定理 6.5）：

$$\hat{A}_t = R_t^{(\lambda)} - V_{\phi_{\text{old}}}(s_t) \quad \Longrightarrow \quad R_t^{(\lambda)} = \hat{A}_t + V_{\phi_{\text{old}}}(s_t) \tag{32}$$

注记 9.11 (值函数目标的直观理解).

回顾优势函数的定义：$A(s,a) = Q(s,a) - V(s)$，即这个动作比平均水平好多少。

GAE 估计的是 $\hat{A}_t \approx Q(s_t, a_t) - V(s_t)$。因此：

$$R_t^{\text{target}} = \hat{A}_t + V_{\phi_{\text{old}}}(s_t) \approx Q(s_t, a_t) \tag{33}$$

而 $Q(s_t, a_t) = r_t + \gamma V(s_{t+1})$ 的期望等于 $V(s_t)$（对动作 a_t 求期望）。所以 R_t^{target} 是 $V(s_t)$ 的一个样本估计（带有采样的动作 a_t 的影响）。

值函数训练的目标就是让 $V_\phi(s_t)$ 逼近这些样本估计的期望。

注记 9.12 (可选：值函数裁剪).

类似于策略的裁剪，有些实现对值函数也应用裁剪以限制更新幅度：

$$V_\phi^{\text{clip}}(s_t) = V_{\phi_{\text{old}}}(s_t) + \text{clip}\left(V_\phi(s_t) - V_{\phi_{\text{old}}}(s_t), -\epsilon_v, \epsilon_v\right) \tag{34}$$

$$\mathcal{L}^{\text{value,clip}}(\phi) = \mathbb{E}_t\left[\max\left((V_\phi(s_t) - R_t^{\text{target}})^2,\ (V_\phi^{\text{clip}}(s_t) - R_t^{\text{target}})^2\right)\right] \tag{35}$$

然而，值函数裁剪的效果存在争议。一些研究发现在某些环境中去除值函数裁剪反而能提高性能。因此这是一个可选的技巧，不是 PPO 的核心组成部分。

9.4.3 完整损失函数

定义 9.13 (PPO 损失函数)**.**

核心形式：

$$\boxed{\mathcal{L}^{\text{PPO}}(\theta, \phi) = \mathcal{L}^{\text{policy}}(\theta) + c_1\mathcal{L}^{\text{value}}(\phi)} \tag{36}$$

完全展开（样本估计形式）：

$$\begin{aligned}\hat{\mathcal{L}}^{\text{PPO}}(\theta, \phi) = & -\frac{1}{N}\sum_{i=1}^{N}\min\left(r_i(\theta)\hat{A}_i,\ \text{clip}(r_i(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_i\right) \\ & + \frac{c_1}{N}\sum_{i=1}^{N}\left(V_\phi(s_i) - R_i^{\text{target}}\right)^2\end{aligned} \tag{37}$$

其中：

- $r_i(\theta) = \pi_\theta(a_i|s_i)/\pi_{\theta_{\text{old}}}(a_i|s_i)$：重要性比率
- $\hat{A}_i$：第 i 个样本的 GAE 优势估计（归一化后）
- $R_i^{\text{target}} = \hat{A}_i + V_{\phi_{\text{old}}}(s_i)$：值函数回归目标
- $c_1 \in [0.5, 1]$：值函数损失的权重系数
- $\epsilon \in [0.1, 0.2]$：策略裁剪参数

注记 9.14 (参数共享与分离). 策略网络和值函数网络可以：

方式 1：完全分离

- 策略网络参数 θ，值函数网络参数 ϕ，互不干扰
- 优点：两个网络独立优化，不会相互影响
- 缺点：参数量翻倍，且无法共享特征表示

方式 2：共享底层（常见于图像输入任务）

- 共享特征提取层（如 CNN），各自有独立的输出头
- 优点：参数效率高，共享的特征可能对两个任务都有用
- 缺点：两个损失的梯度可能冲突，需要仔细调节 c_1

共享参数时，损失函数形式上变为 $\mathcal{L}^{\text{PPO}}(\theta) = \mathcal{L}^{\text{policy}}(\theta) + c_1\mathcal{L}^{\text{value}}(\theta)$。

注记 9.15 (可选：熵奖励).

有些 PPO 实现会加入熵奖励项以鼓励探索：

$$\mathcal{L}^{\text{PPO}}(\theta, \phi) = \mathcal{L}^{\text{policy}}(\theta) + c_1\mathcal{L}^{\text{value}}(\phi) - c_2\mathbb{E}_t[\mathcal{H}(\pi_\theta(\cdot|s_t))] \tag{38}$$

其中 $\mathcal{H}(\pi(\cdot|s)) = -\sum_a \pi(a|s)\log\pi(a|s)$ 是策略的熵，$c_2 \in [0.001, 0.01]$ 是熵系数。

熵奖励的作用是防止策略过早收敛到确定性策略，保持探索能力。然而：

- 熵奖励是一个通用的策略梯度技巧，并非 PPO 特有
- 在 RLHF/大语言模型对齐中通常不使用（$c_2 = 0$），因为目标是让模型给出更确定的好答案
- 是否使用取决于具体任务的探索需求

因此，熵奖励是可选的，不是 PPO 的核心组成部分。

9.4.4 损失函数各项的作用

总结 PPO 损失函数中两个核心项的作用：

损失项	作用	关键机制
策略损失 $\mathcal{L}^{\text{policy}}$	改进 Actor	裁剪限制更新幅度
值函数损失 $\mathcal{L}^{\text{value}}$	训练 Critic	为 GAE 提供准确的 $V(s)$

表格 1: PPO 损失函数各项的作用

两者的协作关系：

1. 值函数 V_ϕ 用于计算 TD 误差 $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$
2. TD 误差用于计算 GAE 优势估计 $\hat{A}_t$
3. 优势估计用于策略损失，指导策略改进
4. 更准确的 $V_\phi \Rightarrow$ 更准确的 $\hat{A}_t \Rightarrow$ 更有效的策略更新

因此，虽然值函数损失看起来是辅助的，但它对 PPO 的整体性能至关重要。

9.4.5 样本估计形式

在实际实现中，期望通过采样数据估计。

定义 9.16 (基于样本的 PPO 损失). 给定一个包含 N 个样本的 mini-batch $\mathcal{B} = \{(s_i, a_i, r_i, s_i', \hat{A}_i, R_i^{\text{target}})\}_{i=1}^N$，PPO 损失的样本估计为：

$$\begin{aligned}\hat{\mathcal{L}}^{\text{PPO}}(\theta, \phi) = & -\frac{1}{N}\sum_{i=1}^{N} \min\left(r_i(\theta)\hat{A}_i,\ \text{clip}(r_i(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_i\right) \\ & + \frac{c_1}{N}\sum_{i=1}^{N}\left(V_\phi(s_i) - R_i^{\text{target}}\right)^2 \\ & - \frac{c_2}{N}\sum_{i=1}^{N}\mathcal{H}(\pi_\theta(\cdot|s_i))\end{aligned} \tag{39}$$

注记 9.17 (期望计算的细节). 在 PPO 中，$\mathbb{E}_t[\cdot]$ 的含义是对收集到的所有时间步取平均。具体地：

1. 使用 $\pi_{\theta_{\text{old}}}$ 收集 M 条轨迹，每条长度为 T
2. 总共得到约 MT 个 (s_t, a_t, r_t, s_{t+1}) 样本
3. 计算所有样本的 δ_t、$\hat{A}_t$ 和 R_t^{target}
4. 将所有样本组成数据集，进行 K 个 epoch 的随机 mini-batch 优化

9.5 PPO 的代码实现

本节给出 PPO 核心组件的 Python 实现，帮助读者理解算法细节。

9.5.1 计算 GAE 和回报目标

首先实现 GAE 的计算，这是 PPO 的基础：

Listing 3: GAE 和回报目标的计算

```python
import numpy as np
import torch

def compute_gae_and_returns(rewards, values, dones, gamma=0.99, lam=0.95):
    """
    计算GAE优势估计和值函数回归目标

    参数:
        rewards: 奖励序列, 形状 [T]
        values:  值函数估计, 形状 [T+1] (包含最后状态的V值)
        dones:   终止标志, 形状 [T] (1表示episode结束)
        gamma:   折扣因子
        lam:     GAE参数lambda

    返回:
        advantages: GAE优势估计, 形状 [T]
        returns:    值函数回归目标, 形状 [T]
    """
    T = len(rewards)
    advantages = np.zeros(T)

    # 反向递推计算GAE
    gae = 0
    for t in reversed(range(T)):
        # 如果是终止状态, 下一状态的值为0
        next_value = values[t + 1] * (1 - dones[t])

        # TD误差: delta_t = r_t + gamma * V(s_{t+1}) - V(s_t)
        delta = rewards[t] + gamma * next_value - values[t]

        # GAE递推: A_t = delta_t + gamma * lambda * A_{t+1}
```

```
        # 如果是终止状态，不累积后续优势
        gae = delta + gamma * lam * (1 - dones[t]) * gae
        advantages[t] = gae

    # 值函数目标: R_t^target = A_t + V_old(s_t)
    returns = advantages + values[:-1]

    return advantages, returns
```

9.5.2 PPO 损失函数

实现 PPO 的核心损失计算:

Listing 4: PPO 损失函数

```
def compute_ppo_loss(policy_net, value_net, states, actions, old_log_probs,
                     advantages, returns, clip_epsilon=0.2, value_coef=0.5)
                         :
    """
    计算PPO的总损失

    参数:
        policy_net:     策略网络 (Actor)
        value_net:      值函数网络 (Critic)
        states:         状态, 形状 [N, state_dim]
        actions:        动作, 形状 [N] 或 [N, action_dim]
        old_log_probs:  旧策略的log概率, 形状 [N]
        advantages:     GAE优势估计 (已归一化), 形状 [N]
        returns:        值函数目标, 形状 [N]
        clip_epsilon:   裁剪参数 epsilon
        value_coef:     值函数损失系数 c_1

    返回:
        total_loss:  总损失
        policy_loss: 策略损失
        value_loss:  值函数损失
        info:        额外信息 (用于监控)
    """
    # ========== 策略损失 ==========
    # 获取新策略下的log概率
    dist = policy_net(states)  # 返回动作分布
    new_log_probs = dist.log_prob(actions)

    # 计算重要性比率: r_t(theta) = pi_theta(a|s) / pi_theta_old(a|s)
    # 在log空间计算更稳定: r = exp(log_pi_new - log_pi_old)
    ratio = torch.exp(new_log_probs - old_log_probs)

    # 未裁剪的目标: r_t * A_t
    surr1 = ratio * advantages
```

```
    # 裁剪后的目标: clip(r_t, 1-eps, 1+eps) * A_t
    surr2 = torch.clamp(ratio, 1 - clip_epsilon, 1 + clip_epsilon) *
        advantages

    # PPO裁剪目标: min(surr1, surr2)
    # 取负值因为我们要最小化损失
    policy_loss = -torch.min(surr1, surr2).mean()

    # ========== 值函数损失 ==========
    # 均方误差: (V_phi(s) - R^target)^2
    values = value_net(states).squeeze(-1)
    value_loss = ((values - returns) ** 2).mean()

    # ========== 总损失 ==========
    total_loss = policy_loss + value_coef * value_loss

    # 收集监控信息
    with torch.no_grad():
        # 近似KL散度 (用于监控策略变化)
        approx_kl = ((ratio - 1) - (new_log_probs - old_log_probs)).mean()
        # 裁剪比例 (被裁剪的样本占比)
        clip_fraction = ((ratio - 1).abs() > clip_epsilon).float().mean()

    info = {
        'approx_kl': approx_kl.item(),
        'clip_fraction': clip_fraction.item(),
        'ratio_mean': ratio.mean().item(),
    }

    return total_loss, policy_loss, value_loss, info
```

9.5.3 优势归一化

Listing 5: 优势归一化

```
def normalize_advantages(advantages, epsilon=1e-8):
    """
    对优势进行归一化: A_norm = (A - mean) / (std + eps)

    参数:
        advantages: 原始优势估计
        epsilon:    防止除零的小常数

    返回:
        归一化后的优势
    """
    return (advantages - advantages.mean()) / (advantages.std() + epsilon)
```

9.5.4 PPO 训练循环

将上述组件整合成完整的训练流程：

Listing 6: PPO 训练主循环

```
def train_ppo(env, policy_net, value_net, num_iterations=1000,
              steps_per_iteration=2048, num_epochs=10, batch_size=64,
              gamma=0.99, lam=0.95, clip_epsilon=0.2,
              lr_policy=3e-4, lr_value=1e-3, value_coef=0.5):
    """
    PPO训练主函数

    参数:
        env:                环境
        policy_net:         策略网络 (Actor)
        value_net:          值函数网络 (Critic)
        num_iterations:     训练迭代次数
        steps_per_iteration: 每次迭代收集的步数
        num_epochs:         每批数据训练的epoch数
        batch_size:         mini-batch大小
        gamma:              折扣因子
        lam:                GAE参数
        clip_epsilon:       PPO裁剪参数
        lr_policy:          策略网络学习率
        lr_value:           值函数网络学习率
        value_coef:         值函数损失系数
    """
    # 优化器
    policy_optimizer = torch.optim.Adam(policy_net.parameters(), lr=
        lr_policy)
    value_optimizer = torch.optim.Adam(value_net.parameters(), lr=lr_value)

    for iteration in range(num_iterations):
        # ========== 第一步：收集数据 ==========
        data = collect_trajectories(env, policy_net, value_net,
            steps_per_iteration)

        # ========== 第二步：计算GAE和回报目标 ==========
        advantages, returns = compute_gae_and_returns(
            data['rewards'], data['values'], data['dones'], gamma, lam
        )

        # 优势归一化
        advantages = normalize_advantages(advantages)

        # 转换为tensor
        states = torch.FloatTensor(data['states'])
        actions = torch.FloatTensor(data['actions'])
        old_log_probs = torch.FloatTensor(data['log_probs'])
```

```
        advantages = torch.FloatTensor(advantages)
        returns = torch.FloatTensor(returns)

        # ========== 第三步：多epoch优化 ==========
        num_samples = len(states)

        for epoch in range(num_epochs):
            # 随机打乱数据
            indices = np.random.permutation(num_samples)

            # Mini-batch训练
            for start in range(0, num_samples, batch_size):
                end = start + batch_size
                batch_indices = indices[start:end]

                # 获取mini-batch
                batch_states = states[batch_indices]
                batch_actions = actions[batch_indices]
                batch_old_log_probs = old_log_probs[batch_indices]
                batch_advantages = advantages[batch_indices]
                batch_returns = returns[batch_indices]

                # 计算损失
                total_loss, policy_loss, value_loss, info =
                    compute_ppo_loss(
                    policy_net, value_net,
                    batch_states, batch_actions, batch_old_log_probs,
                    batch_advantages, batch_returns,
                    clip_epsilon, value_coef
                )

                # 更新网络
                policy_optimizer.zero_grad()
                value_optimizer.zero_grad()
                total_loss.backward()
                policy_optimizer.step()
                value_optimizer.step()

            # 早停：如果KL散度过大，停止当前迭代
            if info['approx_kl'] > 0.02:
                print(f"Early stopping at epoch {epoch} due to large KL: {
                    info['approx_kl']:.4f}")
                break

        # 打印训练信息
        if iteration % 10 == 0:
            print(f"Iteration {iteration}: "
                  f"policy_loss={policy_loss:.4f}, "
                  f"value_loss={value_loss:.4f}, "
```

```
                  f"clip_frac={info['clip_fraction']:.2%}")

    return policy_net  # 返回训练好的策略网络
```

9.5.5 网络结构示例

最后给出一个简单的网络结构示例：

Listing 7: Actor-Critic 网络结构

```
import torch.nn as nn
from torch.distributions import Categorical, Normal

class PolicyNetwork(nn.Module):
    """策略网络（离散动作空间）"""
    def __init__(self, state_dim, action_dim, hidden_dim=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, action_dim),
        )

    def forward(self, state):
        logits = self.net(state)
        return Categorical(logits=logits)


class ValueNetwork(nn.Module):
    """值函数网络"""
    def __init__(self, state_dim, hidden_dim=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, 1),
        )

    def forward(self, state):
        return self.net(state)


class GaussianPolicyNetwork(nn.Module):
    """策略网络（连续动作空间，高斯策略）"""
    def __init__(self, state_dim, action_dim, hidden_dim=64):
```

```
        super().__init__()
        self.shared = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.Tanh(),
        )
        self.mean_head = nn.Linear(hidden_dim, action_dim)
        # log_std作为可学习参数（不依赖状态）
        self.log_std = nn.Parameter(torch.zeros(action_dim))

    def forward(self, state):
        features = self.shared(state)
        mean = self.mean_head(features)
        std = self.log_std.exp()
        return Normal(mean, std)
```

注记 9.18 (实现要点总结).

1. **数据收集**：用当前策略与环境交互，存储 $(s, a, r, s', \log \pi_{\text{old}}(a|s))$
2. **计算优势**：用 GAE 反向递推计算 $\hat{A}_t$，并进行归一化
3. **多 epoch 优化**：在同一批数据上训练多个 epoch，充分利用数据
4. **裁剪机制**：通过 `torch.clamp` 实现比率裁剪
5. **早停**：监控 KL 散度，过大时提前停止以保持稳定性
6. **分离优化器**：Actor 和 Critic 可以使用不同的学习率

注记 9.19 (训练输出).

PPO 训练完成后，只需保留策略网络 π_θ（Actor）。值函数网络 V_ϕ（Critic）仅在训练过程中用于计算优势估计，推理时不需要，可以丢弃。

10 奖励模型的数学原理

在经典强化学习中，奖励函数 $r(s, a)$ 由环境直接给出。但在许多实际应用中（如对话系统、文本生成），很难手工设计一个准确的奖励函数。RLHF（Reinforcement Learning from Human Feedback）的核心思想是：从人类偏好数据中学习奖励函数，再用这个学习到的奖励函数训练策略。

本章介绍奖励模型的数学基础，包括 Bradley-Terry 模型、从偏好数据学习奖励函数的方法，以及奖励模型的训练实现。

10.1 从人类偏好到奖励函数

10.1.1 问题设定

假设我们有一个策略 π（如语言模型），给定输入 x（如用户问题），策略生成输出 y（如模型回答）。我们的目标是让策略生成人类偏好的输出。

核心困难：人类偏好难以用显式的数学公式表达。
解决思路：

1. 让人类对多个输出进行**比较**（而非绝对评分）
2. 从比较数据中**学习**一个奖励函数 $r_\phi(x, y)$
3. 用学习到的奖励函数通过强化学习**优化**策略

10.1.2 偏好数据的形式

定义 10.1 (偏好数据)**.** 偏好数据集 $\mathcal{D}$ 由三元组 (x, y_w, y_l) 组成：

- x：输入（prompt）
- y_w：人类偏好的输出（winner，较好的回答）
- y_l：人类不偏好的输出（loser，较差的回答）

记作 $y_w \succ y_l \mid x$，表示" 给定输入 x，人类认为 y_w 比 y_l 更好"。

注记 10.2 (为什么用比较而非评分？)**.**

- **比较更容易**：让人类说"A 比 B 好" 比让人类给出"A 的分数是 7.3 分" 更自然、更一致
- **标准更统一**：不同标注者对绝对分数的理解可能不同，但比较的一致性更高
- **信息足够**：我们最终只需要知道哪个输出更好，不需要精确的分数

10.2 Bradley-Terry 模型

Bradley-Terry 模型是一个经典的概率模型，用于从成对比较数据中推断潜在的实力或质量分数。

10.2.1 模型假设

定义 10.3 (Bradley-Terry 模型)**.** 假设每个选项 y 有一个潜在的质量分数 $r(y) \in \mathbb{R}$。当比较两个选项 y_1 和 y_2 时，y_1 被偏好的概率为：

$$P(y_1 \succ y_2) = \frac{\exp(r(y_1))}{\exp(r(y_1)) + \exp(r(y_2))} = \sigma(r(y_1) - r(y_2)) \tag{40}$$

其中 $\sigma(z) = \frac{1}{1+e^{-z}}$ 是 sigmoid 函数。

注记 10.4 (模型的直观理解)**.**
Bradley-Terry 模型的核心假设是：

- 每个选项有一个内在质量 $r(y)$
- 质量差距越大，高质量选项被选中的概率越高
- 当 $r(y_1) = r(y_2)$ 时，两者被选中的概率各为 50%
- 当 $r(y_1) \gg r(y_2)$ 时，$P(y_1 \succ y_2) \to 1$

10.2.2 Sigmoid 函数的性质

由于 Bradley-Terry 模型大量使用 Sigmoid 函数，我们先回顾其关键性质。

命题 10.5 (Sigmoid 函数的性质)**.**

1. **定义**：$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z}$
2. **值域**：$\sigma(z) \in (0, 1)$，可解释为概率
3. **对称性**：$\sigma(-z) = 1 - \sigma(z)$
4. **导数**：$\sigma'(z) = \sigma(z)(1 - \sigma(z))$
5. **与对数几率的关系**：若 $p = \sigma(z)$，则 $z = \log \frac{p}{1-p}$（logit 函数）

Proof. 我们证明对称性和导数公式。

对称性：

$$\sigma(-z) = \frac{1}{1+e^{z}} = \frac{e^{-z}}{e^{-z}+1} = 1 - \frac{1}{1+e^{-z}} = 1 - \sigma(z)$$

导数：

$$\begin{aligned}\sigma'(z) &= \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) = \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} = \sigma(z) \cdot \frac{1}{1+e^{z}} \\ &= \sigma(z) \cdot (1 - \sigma(z))\end{aligned}$$

■

10.2.3 Bradley-Terry 模型的推导

Bradley-Terry 模型可以从多种角度推导，这里给出基于随机效用的推导。

定理 10.6 (Bradley-Terry 模型的随机效用推导)**.** 假设人类在比较 y_1 和 y_2 时，感知到的效用为：

$$U_1 = r(y_1) + \epsilon_1 \tag{41}$$

$$U_2 = r(y_2) + \epsilon_2 \tag{42}$$

其中 ϵ_1, ϵ_2 是独立同分布的 Gumbel 噪声。则人类选择 y_1 的概率为：

$$P(y_1 \succ y_2) = P(U_1 > U_2) = \sigma(r(y_1) - r(y_2)) \tag{43}$$

Proof. Gumbel 分布的累积分布函数为 $F(x) = e^{-e^{-x}}$。
两个独立 Gumbel 随机变量之差服从 Logistic 分布，即若 $\epsilon_1, \epsilon_2 \sim \text{Gumbel}(0,1)$ 独立，则 $\epsilon_1 - \epsilon_2 \sim \text{Logistic}(0,1)$。
Logistic 分布的累积分布函数为 $F(x) = \sigma(x)$。
因此：

$$\begin{aligned} P(y_1 \succ y_2) &= P(U_1 > U_2) = P(r(y_1) + \epsilon_1 > r(y_2) + \epsilon_2) \\ &= P(\epsilon_2 - \epsilon_1 < r(y_1) - r(y_2)) \\ &= \sigma(r(y_1) - r(y_2)) \end{aligned}$$

■

注记 10.7 (Gumbel 噪声的意义). Gumbel 噪声模型假设人类的判断存在随机性：

- 即使 $r(y_1) > r(y_2)$，人类也可能偶尔选择 y_2
- 噪声捕捉了人类判断的不确定性、主观性和不一致性
- 这比确定性模型（总是选择分数更高的）更符合实际

10.3 奖励模型的训练

10.3.1 最大似然估计

给定偏好数据集 $\mathcal{D} = \{(x^{(i)}, y_w^{(i)}, y_l^{(i)})\}_{i=1}^N$，我们用参数化的奖励模型 $r_\phi(x, y)$ 来拟合数据。

定义 10.8 (奖励模型的似然函数)**.** 根据 Bradley-Terry 模型，观测到偏好数据的似然为：

$$\mathcal{L}(\phi) = \prod_{i=1}^{N} P(y_w^{(i)} \succ y_l^{(i)} \mid x^{(i)}; \phi) = \prod_{i=1}^{N} \sigma\left(r_\phi(x^{(i)}, y_w^{(i)}) - r_\phi(x^{(i)}, y_l^{(i)})\right) \tag{44}$$

定义 10.9 (奖励模型的损失函数)**.** 取负对数似然，得到训练损失：

$$\boxed{\mathcal{L}_{\text{RM}}(\phi) = -\frac{1}{N} \sum_{i=1}^{N} \log \sigma\left(r_\phi(x^{(i)}, y_w^{(i)}) - r_\phi(x^{(i)}, y_l^{(i)})\right)} \tag{45}$$

记 $\Delta r_i = r_\phi(x^{(i)}, y_w^{(i)}) - r_\phi(x^{(i)}, y_l^{(i)})$ 为第 i 个样本的奖励差，则：

$$\mathcal{L}_{\text{RM}}(\phi) = -\frac{1}{N} \sum_{i=1}^{N} \log \sigma(\Delta r_i) \tag{46}$$

注记 10.10 (损失函数的直观理解).
损失函数 $-\log \sigma(\Delta r)$ 的行为：

- 当 $\Delta r \to +\infty$（正确预测，置信度高）时，$\sigma(\Delta r) \to 1$，损失 $\to 0$
- 当 $\Delta r = 0$（无法区分）时，$\sigma(0) = 0.5$，损失 $= \log 2 \approx 0.693$
- 当 $\Delta r \to -\infty$（错误预测）时，$\sigma(\Delta r) \to 0$，损失 $\to +\infty$

因此，最小化损失会推动模型给 y_w 更高的奖励、给 y_l 更低的奖励。

10.3.2 与二分类交叉熵的关系

命题 10.11 (奖励模型损失等价于二分类交叉熵)**.** 奖励模型损失 (45) 等价于以下二分类问题的交叉熵损失：

- 输入：奖励差 $\Delta r = r_\phi(x, y_w) - r_\phi(x, y_l)$
- 标签：$y = 1$（y_w 总是正确答案）
- 预测：$\hat{y} = \sigma(\Delta r)$

二分类交叉熵为：

$$\text{BCE}(y, \hat{y}) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})] \tag{47}$$

当 $y = 1$ 时：

$$\text{BCE}(1, \sigma(\Delta r)) = -\log \sigma(\Delta r) = \mathcal{L}_{\text{RM}} \tag{48}$$

注记 10.12 (实现上的便利)**.**

这个等价性意味着我们可以直接使用深度学习框架中的二分类交叉熵损失函数（如 `torch.nn.BCEWithLogitsLoss`）来训练奖励模型，只需将 Δr 作为 logit 输入，标签设为 1。

10.3.3 梯度分析

命题 10.13 (奖励模型损失的梯度)**.** 对于单个样本 (x, y_w, y_l)，损失关于参数 ϕ 的梯度为：

$$\nabla_\phi \mathcal{L}_{\text{RM}} = -\left(1 - \sigma(\Delta r)\right) \cdot \left(\nabla_\phi r_\phi(x, y_w) - \nabla_\phi r_\phi(x, y_l)\right) \tag{49}$$

其中 $\Delta r = r_\phi(x, y_w) - r_\phi(x, y_l)$。

Proof. 设 $\Delta r = r_\phi(x, y_w) - r_\phi(x, y_l)$，损失为 $\mathcal{L} = -\log \sigma(\Delta r)$。
由链式法则：

$$\nabla_\phi \mathcal{L} = -\frac{1}{\sigma(\Delta r)} \cdot \sigma'(\Delta r) \cdot \nabla_\phi \Delta r$$

利用 $\sigma'(z) = \sigma(z)(1 - \sigma(z))$：

$$\begin{aligned} \nabla_\phi \mathcal{L} &= -\frac{\sigma(\Delta r)(1 - \sigma(\Delta r))}{\sigma(\Delta r)} \cdot \nabla_\phi \Delta r \\ &= -(1 - \sigma(\Delta r)) \cdot (\nabla_\phi r_\phi(x, y_w) - \nabla_\phi r_\phi(x, y_l)) \end{aligned}$$

■

注记 10.14 (梯度的直观理解)**.** 梯度公式 (49) 揭示了训练动态：

- 系数 $(1 - \sigma(\Delta r))$ 是预测错误程度：
  - 若模型已经正确（Δr 大，$\sigma(\Delta r) \approx 1$），则系数接近 0，梯度小
  - 若模型预测错误（Δr 小或为负），则系数大，梯度大
- 梯度方向：增大 $r_\phi(x, y_w)$，减小 $r_\phi(x, y_l)$
- 这是一种自适应机制：模型在已学好的样本上更新少，在困难样本上更新多

10.4 奖励模型的架构

在大语言模型的 RLHF 中，奖励模型通常基于预训练语言模型构建。

10.4.1 基本架构

定义 10.15 (基于语言模型的奖励模型)**.** 给定预训练语言模型的 encoder（或 decoder 的隐藏状态），奖励模型的结构为：

$$r_\phi(x, y) = \text{Linear}(\text{LM}_\phi([x; y])_{\text{last}}) \tag{50}$$

其中：

- $[x; y]$ 表示将输入 x 和输出 y 拼接
- $\text{LM}_\phi(\cdot)_{\text{last}}$ 表示语言模型最后一个 token 的隐藏状态
- Linear 是一个将隐藏状态映射到标量的线性层

注记 10.16 (为什么用最后一个 token？).

使用最后一个 token 的隐藏状态是因为：

- 在自回归语言模型中，最后一个 token 的隐藏状态包含了整个序列的信息
- 这类似于分类任务中使用 [CLS] token
- 也可以使用所有 token 的平均池化，但实践中最后一个 token 效果通常足够好

10.4.2 奖励模型的初始化

注记 10.17 (从预训练模型初始化).

奖励模型通常从预训练语言模型（如被对齐的目标模型的初始版本）初始化：

- 语言模型部分：加载预训练权重
- 线性输出层：随机初始化（通常用较小的方差）

这种初始化方式的好处是奖励模型已经具备理解语言的能力，只需学习” 什么是好的输出”。

10.5 奖励模型的代码实现

10.5.1 模型定义

Listing 8: 奖励模型的定义

```python
import torch
import torch.nn as nn
from transformers import AutoModel

class RewardModel(nn.Module):
    """
    基于预训练语言模型的奖励模型
    """
    def __init__(self, model_name, hidden_size=None):
```

```
        super().__init__()
        # 加载预训练语言模型
        self.backbone = AutoModel.from_pretrained(model_name)

        # 获取隐藏层维度
        if hidden_size is None:
            hidden_size = self.backbone.config.hidden_size

        # 奖励输出头：将隐藏状态映射到标量奖励
        self.reward_head = nn.Linear(hidden_size, 1)

    def forward(self, input_ids, attention_mask=None):
        """
        计算输入序列的奖励值

        参数：
            input_ids:      token ids，形状 [batch_size, seq_len]
            attention_mask: 注意力掩码，形状 [batch_size, seq_len]

        返回：
            rewards: 奖励值，形状 [batch_size]
        """
        # 获取语言模型的隐藏状态
        outputs = self.backbone(input_ids=input_ids, attention_mask=
            attention_mask)
        hidden_states = outputs.last_hidden_state  # [batch_size, seq_len,
            hidden_size]

        # 获取最后一个token的隐藏状态
        # 注意：需要找到每个序列中最后一个非padding token的位置
        if attention_mask is not None:
            # 找到每个序列的实际长度
            seq_lengths = attention_mask.sum(dim=1) - 1  # [batch_size]
            batch_size = hidden_states.size(0)
            last_hidden = hidden_states[torch.arange(batch_size),
                seq_lengths]
        else:
            last_hidden = hidden_states[:, -1, :]  # [batch_size,
                hidden_size]

        # 计算奖励
        rewards = self.reward_head(last_hidden).squeeze(-1)  # [batch_size]

        return rewards
```

10.5.2 损失函数计算

Listing 9: 奖励模型的损失计算

```python
def compute_reward_model_loss(reward_model, chosen_ids, chosen_mask,
                              rejected_ids, rejected_mask):
    """
    计算奖励模型的Bradley-Terry损失

    参数:
        reward_model: 奖励模型
        chosen_ids:   偏好输出的token ids, 形状 [batch_size, seq_len]
        chosen_mask:  偏好输出的attention mask
        rejected_ids: 非偏好输出的token ids, 形状 [batch_size, seq_len]
        rejected_mask:非偏好输出的attention mask

    返回:
        loss:      损失值
        accuracy: 预测准确率 (奖励模型给chosen更高分的比例)
    """
    # 计算两个输出的奖励
    chosen_rewards = reward_model(chosen_ids, chosen_mask)      # [
        batch_size]
    rejected_rewards = reward_model(rejected_ids, rejected_mask)  # [
        batch_size]

    # 奖励差
    reward_diff = chosen_rewards - rejected_rewards  # [batch_size]

    # Bradley-Terry损失: -log(sigmoid(r_chosen - r_rejected))
    # 等价于 BCEWithLogitsLoss, 其中 logit = reward_diff, target = 1
    loss = -torch.nn.functional.logsigmoid(reward_diff).mean()

    # 计算准确率 (用于监控)
    with torch.no_grad():
        accuracy = (reward_diff > 0).float().mean()

    return loss, accuracy
```

10.5.3 训练循环

Listing 10: 奖励模型的训练循环

```python
def train_reward_model(reward_model, train_dataloader, num_epochs=1,
                       learning_rate=1e-5, device='cuda'):
    """
    训练奖励模型

    参数:
        reward_model:      奖励模型
```

```
        train_dataloader: 训练数据加载器，每个batch包含 (chosen, rejected) 对
        num_epochs:       训练轮数
        learning_rate:    学习率
        device:           计算设备
    """
    reward_model = reward_model.to(device)
    optimizer = torch.optim.AdamW(reward_model.parameters(), lr=learning_rate)

    reward_model.train()
    for epoch in range(num_epochs):
        total_loss = 0
        total_accuracy = 0
        num_batches = 0

        for batch in train_dataloader:
            # 假设batch包含: chosen_ids, chosen_mask, rejected_ids, rejected_mask
            chosen_ids = batch['chosen_ids'].to(device)
            chosen_mask = batch['chosen_mask'].to(device)
            rejected_ids = batch['rejected_ids'].to(device)
            rejected_mask = batch['rejected_mask'].to(device)

            # 计算损失
            loss, accuracy = compute_reward_model_loss(
                reward_model, chosen_ids, chosen_mask,
                rejected_ids, rejected_mask
            )

            # 反向传播
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
            total_accuracy += accuracy.item()
            num_batches += 1

        avg_loss = total_loss / num_batches
        avg_accuracy = total_accuracy / num_batches
        print(f"Epoch {epoch+1}/{num_epochs}: Loss={avg_loss:.4f}, Accuracy={avg_accuracy:.2%}")

    return reward_model
```

10.5.4 使用奖励模型

Listing 11: 使用奖励模型进行评分

```python
def score_responses(reward_model, tokenizer, prompt, responses, device='cuda'):
    """
    对多个回答进行评分

    参数:
        reward_model: 训练好的奖励模型
        tokenizer:    分词器
        prompt:       输入提示
        responses:    回答列表
        device:       计算设备

    返回:
        scores: 各回答的奖励分数
    """
    reward_model.eval()
    scores = []

    with torch.no_grad():
        for response in responses:
            # 拼接prompt和response
            text = prompt + response
            inputs = tokenizer(text, return_tensors='pt', padding=True)
            input_ids = inputs['input_ids'].to(device)
            attention_mask = inputs['attention_mask'].to(device)

            # 计算奖励
            reward = reward_model(input_ids, attention_mask)
            scores.append(reward.item())

    return scores


# 使用示例
prompt = "请解释什么是机器学习？"
responses = [
    "机器学习是人工智能的一个分支，它使计算机能够从数据中学习。",
    "不知道。",
    "机器学习让计算机通过经验自动改进，无需显式编程。"
]

scores = score_responses(reward_model, tokenizer, prompt, responses)
# 输出可能是: [0.82, -1.23, 0.95]
# 表示第三个回答最好，第二个最差
```

10.6 奖励模型的注意事项

注记 10.18 (奖励模型的局限性).

奖励模型并非完美，存在以下问题：

1. **分布外泛化**：奖励模型在训练数据分布外可能给出不可靠的分数
2. **奖励黑客（Reward Hacking）**：策略可能学会利用奖励模型的漏洞，生成高分但实际质量差的输出
3. **偏好数据的噪声**：人类标注存在不一致性，影响模型质量
4. **奖励尺度**：绝对奖励值没有意义，只有相对差异有意义

注记 10.19 (缓解 Reward Hacking 的方法).

为了缓解 Reward Hacking 问题，常用的技术包括：

- **KL 惩罚**：在 PPO 训练中添加与参考策略的 KL 散度惩罚（见下一章）
- **奖励裁剪**：限制奖励的范围，防止极端值
- **奖励模型集成**：使用多个奖励模型取平均
- **持续更新**：随着策略改进，收集新数据更新奖励模型

10.7 RLHF 中的 PPO 训练

前面章节介绍的 PPO 算法适用于一般的强化学习环境。在 RLHF（Reinforcement Learning from Human Feedback）中，我们需要将语言模型的生成过程建模为强化学习问题，并将训练好的奖励模型与 PPO 结合。

10.7.1 语言模型生成作为强化学习问题

定义 10.20 (RLHF 中的 MDP 建模)**.** 将语言模型的生成过程建模为马尔可夫决策过程：

- **状态** s_t：输入 prompt x 加上已生成的 token 序列 $y_{<t} = (y_1, \ldots, y_{t-1})$，即 $s_t = (x, y_{<t})$
- **动作** a_t：在词表 $\mathcal{V}$ 中选择下一个 token y_t
- **策略** $\pi_\theta(a_t|s_t)$：语言模型的条件概率分布 $\pi_\theta(y_t|x, y_{<t})$
- **状态转移**：确定性转移，$s_{t+1} = (x, y_{\leq t})$
- **奖励**：在生成结束时由奖励模型给出（见下文）

注记 10.21 (与经典 RL 的区别).

RLHF 中的 RL 问题有几个特殊之处：

1. **确定性转移**：给定当前状态和动作，下一状态是确定的（只是追加一个 token）
2. **稀疏奖励**：奖励只在序列生成结束时给出，中间步骤奖励为 0
3. **大动作空间**：动作空间是整个词表，通常有数万到数十万个 token
4. **变长序列**：不同的生成序列长度不同

10.7.2 奖励的设计

在 RLHF 中，奖励来自两部分：奖励模型的评分和 KL 惩罚。

定义 10.22 (RLHF 奖励函数)**.** 对于完整的生成序列 $y = (y_1, \ldots, y_T)$，总奖励定义为：

$$\boxed{R(x, y) = r_\phi(x, y) - \beta \cdot \mathrm{KL}_{\text{penalty}}(x, y)} \tag{51}$$

其中：

- $r_\phi(x, y)$：奖励模型给出的分数
- $\beta > 0$：KL 惩罚系数
- $\mathrm{KL}_{\text{penalty}}(x, y)$：策略与参考策略之间的 KL 散度惩罚

定义 10.23 (KL 惩罚的计算)**.** KL 惩罚通常在 token 级别计算，然后求和：

$$\mathrm{KL}_{\text{penalty}}(x, y) = \sum_{t=1}^{T} \log \frac{\pi_\theta(y_t|x, y_{<t})}{\pi_{\text{ref}}(y_t|x, y_{<t})} \tag{52}$$

其中 π_{ref} 是参考策略，通常是 SFT（监督微调）后的模型。
将 KL 惩罚代入总奖励：

$$R(x, y) = r_\phi(x, y) - \beta \sum_{t=1}^{T} \log \frac{\pi_\theta(y_t|x, y_{<t})}{\pi_{\text{ref}}(y_t|x, y_{<t})} \tag{53}$$

注记 10.24 (为什么需要 KL 惩罚？).
KL 惩罚的作用是防止策略 π_θ 偏离参考策略 π_{ref} 太远：

1. **防止 Reward Hacking**：没有 KL 约束时，策略可能学会利用奖励模型的漏洞，生成高分但实际质量差的文本（如重复、无意义但讨好奖励模型的内容）
2. **保持语言能力**：参考策略是经过预训练和 SFT 的模型，具有良好的语言能力。KL 惩罚确保优化后的模型不会忘记这些能力
3. **奖励模型的局限**：奖励模型只在有限的数据上训练，在分布外区域不可靠。KL 惩罚限制策略探索到奖励模型不可靠的区域

10.7.3 逐 token 奖励分配

由于 PPO 需要每个时间步的奖励，而 RLHF 的奖励是序列级别的，我们需要将奖励分配到各个 token。

定义 10.25 (逐 token 奖励分配)**.** 一种常见的做法是将奖励模型的分数放在最后一个 token，KL 惩罚逐 token 计算：

$$r_t = \begin{cases} -\beta \log \dfrac{\pi_\theta(y_t|x, y_{<t})}{\pi_{\text{ref}}(y_t|x, y_{<t})} & \text{if } t < T \\ r_\phi(x, y) - \beta \log \dfrac{\pi_\theta(y_T|x, y_{<T})}{\pi_{\text{ref}}(y_T|x, y_{<T})} & \text{if } t = T \end{cases} \tag{54}$$

即：中间 token 只有 KL 惩罚，最后一个 token 同时获得奖励模型分数和 KL 惩罚。

10.7.4 RLHF-PPO 的完整流程

定义 10.26 (RLHF-PPO 训练流程)**.**

给定：

- 参考策略 π_{ref}（通常是 SFT 模型，训练过程中冻结）
- 奖励模型 r_ϕ（预先训练好，训练过程中冻结）
- 待优化的策略 π_θ（初始化为 π_{ref} 的副本）
- 值函数 V_ψ（Critic，随机初始化或从 π_{ref} 初始化）

训练循环：

1. **采样**：从 prompt 数据集中采样一批 prompt $\{x^{(i)}\}$
2. **生成**：用当前策略 π_θ 为每个 prompt 生成回答 $y^{(i)} \sim \pi_\theta(\cdot|x^{(i)})$，同时记录每个 token 的 log 概率 $\log \pi_\theta(y_t|x, y_{<t})$
3. **计算奖励**：
   - 用奖励模型计算 $r_\phi(x^{(i)}, y^{(i)})$
   - 用参考策略计算 $\log \pi_{\text{ref}}(y_t|x, y_{<t})$
   - 计算逐 token 奖励 r_t（公式 (54)）
4. **计算优势**：用值函数 V_ψ 和 GAE 计算优势估计 $\hat{A}_t$
5. **PPO 更新**：
   - 用 PPO 裁剪目标更新策略 π_θ
   - 用均方误差损失更新值函数 V_ψ
6. 重复以上步骤

10.7.5 代码实现

Listing 12: RLHF-PPO 的核心计算

```
def compute_rlhf_rewards(policy_model, ref_model, reward_model,
                         input_ids, attention_mask, beta=0.1):
    """
    计算RLHF中的逐token奖励

    参数:
        policy_model: 当前策略模型
        ref_model:    参考模型（冻结）
        reward_model: 奖励模型（冻结）
        input_ids:    生成的完整序列 [batch_size, seq_len]
        attention_mask: 注意力掩码
        beta:         KL惩罚系数
```

```
    返回:
        rewards:      逐token奖励 [batch_size, seq_len]
        kl_div:       KL散度 (用于监控)
    """
    with torch.no_grad():
        # 计算参考模型的log概率
        ref_logits = ref_model(input_ids, attention_mask).logits
        ref_log_probs = torch.log_softmax(ref_logits, dim=-1)
        # 获取实际token的log概率
        ref_log_probs = torch.gather(ref_log_probs[:, :-1, :], 2,
                                     input_ids[:, 1:].unsqueeze(-1)).
                                        squeeze(-1)

    # 计算当前策略的log概率
    policy_logits = policy_model(input_ids, attention_mask).logits
    policy_log_probs = torch.log_softmax(policy_logits, dim=-1)
    policy_log_probs = torch.gather(policy_log_probs[:, :-1, :], 2,
                                    input_ids[:, 1:].unsqueeze(-1)).
                                       squeeze(-1)

    # 计算KL散度: log(pi_theta / pi_ref) = log_pi_theta - log_pi_ref
    kl_div = policy_log_probs - ref_log_probs  # [batch_size, seq_len-1]

    # KL惩罚作为负奖励
    kl_penalty = -beta * kl_div  # [batch_size, seq_len-1]

    with torch.no_grad():
        # 计算奖励模型分数 (整个序列)
        rm_scores = reward_model(input_ids, attention_mask)  # [batch_size]

    # 构造逐token奖励
    rewards = kl_penalty.clone()
    # 将奖励模型分数加到最后一个token
    seq_lengths = attention_mask.sum(dim=1) - 2  # 最后一个生成token的位置
    for i in range(rewards.size(0)):
        rewards[i, seq_lengths[i]] += rm_scores[i]

    return rewards, kl_div.mean()


def rlhf_ppo_step(policy_model, ref_model, reward_model, value_model,
                  prompts, tokenizer, ppo_config):
    """
    RLHF-PPO的一个训练步骤

    参数:
        policy_model: 策略模型 (待优化)
        ref_model:    参考模型 (冻结)
```

```
        reward_model: 奖励模型（冻结）
        value_model:  值函数模型（待优化）
        prompts:      输入prompt列表
        tokenizer:    分词器
        ppo_config:   PPO配置（包含beta, clip_epsilon, gamma, lam等）
    """
    # 第一步：生成回答
    prompt_ids = tokenizer(prompts, return_tensors='pt', padding=True)

    with torch.no_grad():
        # 用当前策略生成
        generated = policy_model.generate(
            prompt_ids['input_ids'],
            max_new_tokens=ppo_config.max_new_tokens,
            do_sample=True,
            temperature=ppo_config.temperature,
            pad_token_id=tokenizer.pad_token_id,
        )

    # 第二步：计算奖励
    rewards, kl = compute_rlhf_rewards(
        policy_model, ref_model, reward_model,
        generated, attention_mask=None,  # 需要正确构造attention_mask
        beta=ppo_config.beta
    )

    # 第三步：计算值函数估计和GAE
    with torch.no_grad():
        values = value_model(generated)

    advantages, returns = compute_gae_and_returns(
        rewards.cpu().numpy(),
        values.cpu().numpy(),
        dones=None,  # 需要根据实际情况处理
        gamma=ppo_config.gamma,
        lam=ppo_config.lam
    )

    # 第四步：PPO更新（多个epoch）
    # ... 与标准PPO相同，此处省略

    return {
        'mean_reward': rewards.sum(dim=1).mean().item(),
        'mean_kl': kl.item(),
    }
```

注记 10.27 (RLHF-PPO 与标准 PPO 的主要区别).

方面	标准 PPO	RLHF-PPO
奖励来源	环境直接给出	奖励模型 + KL 惩罚
奖励时机	每步都有	主要在序列末尾
额外约束	无	KL 散度惩罚
参考策略	无	需要冻结的参考模型
动作空间	通常较小	整个词表（数万维）

注记 10.28 (训练输出). RLHF-PPO 训练完成后：

- **保留**：优化后的策略模型 π_θ（即对齐后的语言模型）
- **丢弃**：值函数模型 V_ψ（仅用于训练）、参考模型 π_{ref}（可选保留用于后续训练）、奖励模型 r_ϕ（可选保留用于评估）

最终部署的只有对齐后的语言模型 π_θ。

11 附录

11.1 核心公式总结

概念	公式
折扣回报	$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$
值函数	$V^\pi(s) = \mathbb{E}_\pi[R_t \mid s_t = s]$
Q 函数	$Q^\pi(s,a) = \mathbb{E}_\pi[R_t \mid s_t = s, a_t = a]$
优势函数	$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$
TD 误差	$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$
GAE（定义）	$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$
GAE（递推）	$\hat{A}_t = \delta_t + \gamma\lambda \hat{A}_{t+1}$
重要性比率	$r_t(\theta) = \pi_\theta(a_t \mid s_t) / \pi_{\theta_{\text{old}}}(a_t \mid s_t)$
PPO 裁剪目标	$L^{\text{CLIP}} = \mathbb{E}_t[\min(r_t \hat{A}_t, \text{clip}(r_t, 1-\epsilon, 1+\epsilon)\hat{A}_t)]$

表格 2: PPO 核心公式

11.2 从 REINFORCE 到 PPO 的演进

1. **REINFORCE**：基础策略梯度，使用蒙特卡洛回报

$$\nabla_\theta J = \mathbb{E}[\sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t) R_t]$$

问题：高方差

2. **带基线的 REINFORCE**：引入状态相关基线降低方差

$$\nabla_\theta J = \mathbb{E}[\sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)(R_t - b(s_t))]$$

3. **Actor-Critic**：使用学习的值函数作为基线，用 TD 误差估计优势

4. **GAE**：平衡偏差与方差的优势估计

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

5. **TRPO**：通过 KL 约束稳定更新

6. **PPO**：用裁剪目标代替显式约束，简化实现

$$L^{\text{CLIP}} = \mathbb{E}[\min(r_t \hat{A}_t, \text{clip}(r_t, 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

11.3 补充证明

11.3.1 GAE 形式 B：λ-回报与值函数之差

$$\boxed{\hat{A}_t^{\text{GAE}} = R_t^{(\lambda)} - V(s_t)} \tag{55}$$

我们需要证明 $\sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l} = R_t^{(\lambda)} - V(s_t)$。

Proof. 首先展开 λ-回报的定义：

$$\begin{aligned} R_t^{(\lambda)} &= (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}R_t^{(n)} \\ &= (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\left(\sum_{k=0}^{n-1}\gamma^k r_{t+k} + \gamma^n V(s_{t+n})\right) \end{aligned}$$

交换求和顺序。对于奖励项：

$$\begin{aligned} &(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\sum_{k=0}^{n-1}\gamma^k r_{t+k} \\ &= (1-\lambda)\sum_{k=0}^{\infty}\gamma^k r_{t+k}\sum_{n=k+1}^{\infty}\lambda^{n-1} \\ &= (1-\lambda)\sum_{k=0}^{\infty}\gamma^k r_{t+k}\cdot\frac{\lambda^k}{1-\lambda} \\ &= \sum_{k=0}^{\infty}(\gamma\lambda)^k r_{t+k} \end{aligned}$$

对于自举项：

$$\begin{aligned} &(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\gamma^n V(s_{t+n}) \\ &= (1-\lambda)\gamma\sum_{n=1}^{\infty}(\gamma\lambda)^{n-1}V(s_{t+n}) \\ &= (1-\lambda)\gamma\sum_{m=0}^{\infty}(\gamma\lambda)^m V(s_{t+m+1}) \quad (\text{令} m = n-1) \end{aligned}$$

因此：

$$R_t^{(\lambda)} = \sum_{k=0}^{\infty}(\gamma\lambda)^k r_{t+k} + (1-\lambda)\gamma\sum_{m=0}^{\infty}(\gamma\lambda)^m V(s_{t+m+1}) \tag{56}$$

■

现在计算 $R_t^{(\lambda)} - V(s_t)$：

Proof.

$$R_t^{(\lambda)} - V(s_t) = \sum_{k=0}^{\infty} (\gamma\lambda)^k r_{t+k} + (1-\lambda)\gamma \sum_{m=0}^{\infty} (\gamma\lambda)^m V(s_{t+m+1}) - V(s_t)$$

另一方面，展开 GAE 定义式：

$$\begin{aligned}\sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} &= \sum_{l=0}^{\infty} (\gamma\lambda)^l [r_{t+l} + \gamma V(s_{t+l+1}) - V(s_{t+l})] \\ &= \sum_{l=0}^{\infty} (\gamma\lambda)^l r_{t+l} + \gamma \sum_{l=0}^{\infty} (\gamma\lambda)^l V(s_{t+l+1}) - \sum_{l=0}^{\infty} (\gamma\lambda)^l V(s_{t+l})\end{aligned}$$

对于值函数项，进行配对：

$$\begin{aligned}&\gamma \sum_{l=0}^{\infty} (\gamma\lambda)^l V(s_{t+l+1}) - \sum_{l=0}^{\infty} (\gamma\lambda)^l V(s_{t+l}) \\ &= \gamma \sum_{l=0}^{\infty} (\gamma\lambda)^l V(s_{t+l+1}) - V(s_t) - \sum_{l=1}^{\infty} (\gamma\lambda)^l V(s_{t+l}) \\ &= -V(s_t) + \sum_{l=0}^{\infty} (\gamma\lambda)^l [\gamma V(s_{t+l+1}) - \gamma\lambda V(s_{t+l+1})] \\ &= -V(s_t) + \gamma(1-\lambda) \sum_{l=0}^{\infty} (\gamma\lambda)^l V(s_{t+l+1})\end{aligned}$$

将上式代入，得：

$$\sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} = \sum_{l=0}^{\infty} (\gamma\lambda)^l r_{t+l} + \gamma(1-\lambda) \sum_{l=0}^{\infty} (\gamma\lambda)^l V(s_{t+l+1}) - V(s_t)$$

这与公式 eq. (56) 中的 $R_t^{(\lambda)} - V(s_t)$ 完全一致。 ■

11.3.2 GAE 形式 C：递推形式

$$\boxed{\hat{A}_t = \delta_t + \gamma\lambda \hat{A}_{t+1}} \tag{GAE-C}$$

Proof. 从定义式 eq. (GAE-A) 出发：

$$\begin{aligned}\hat{A}_t &= \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \\ &= \delta_t + \sum_{l=1}^{\infty} (\gamma\lambda)^l \delta_{t+l} \\ &= \delta_t + \gamma\lambda \sum_{l=1}^{\infty} (\gamma\lambda)^{l-1} \delta_{t+l} \\ &= \delta_t + \gamma\lambda \sum_{m=0}^{\infty} (\gamma\lambda)^m \delta_{t+1+m} \quad (\text{令} m = l-1) \\ &= \delta_t + \gamma\lambda \hat{A}_{t+1}\end{aligned}$$

■

11.3.3 λ-回报展开的详细推导

命题 11.1. λ-回报可以展开为：

$$R_t^{(\lambda)} = \sum_{k=0}^{\infty} (\gamma\lambda)^k r_{t+k} + (1-\lambda)\gamma \sum_{m=0}^{\infty} (\gamma\lambda)^m V(s_{t+m+1})$$

Proof. 从定义出发：

$$R_t^{(\lambda)} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}$$

其中 n 步回报为：

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V(s_{t+n})$$

将其代入：

$$R_t^{(\lambda)} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \left(\sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V(s_{t+n}) \right)$$

分离奖励项和值函数项：

奖励项：

$$(1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \sum_{k=0}^{n-1} \gamma^k r_{t+k}$$

交换求和顺序。对于固定的 k，它出现在所有 $n \geq k+1$ 的项中：

$$= (1-\lambda) \sum_{k=0}^{\infty} \gamma^k r_{t+k} \sum_{n=k+1}^{\infty} \lambda^{n-1}$$

计算内层级数：

$$\sum_{n=k+1}^{\infty} \lambda^{n-1} = \lambda^k + \lambda^{k+1} + \cdots = \frac{\lambda^k}{1-\lambda}$$

因此奖励项为：

$$(1-\lambda) \sum_{k=0}^{\infty} \gamma^k r_{t+k} \cdot \frac{\lambda^k}{1-\lambda} = \sum_{k=0}^{\infty} (\gamma\lambda)^k r_{t+k}$$

值函数项：

$$\begin{aligned}
&(1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \gamma^n V(s_{t+n}) \\
&= (1-\lambda)\gamma \sum_{n=1}^{\infty} (\gamma\lambda)^{n-1} V(s_{t+n}) \\
&= (1-\lambda)\gamma \sum_{m=0}^{\infty} (\gamma\lambda)^m V(s_{t+m+1})
\end{aligned}$$

合并两项得到结论。 ■

11.3.4 策略熵的性质

命题 11.2. 对于离散概率分布 $p = (p_1, \ldots, p_n)$，熵 $\mathcal{H}(p) = -\sum_i p_i \log p_i$ 满足：

1. 非负性：$\mathcal{H}(p) \geq 0$
2. 最大值：当且仅当 p 为均匀分布时取最大值 $\log n$
3. 最小值：当且仅当 p 为确定性分布（某个 $p_i = 1$）时取最小值 0

Proof. **非负性：**由于 $p_i \in [0, 1]$，有 $\log p_i \leq 0$，故 $-p_i \log p_i \geq 0$。
最大值：使用拉格朗日乘子法。令 $L = -\sum_i p_i \log p_i - \mu(\sum_i p_i - 1)$。

$$\frac{\partial L}{\partial p_i} = -\log p_i - 1 - \mu = 0$$

解得 $p_i = e^{-1-\mu}$ 对所有 i 相同，结合 $\sum_i p_i = 1$，得 $p_i = 1/n$。
最小值：当某个 $p_j = 1$，其余 $p_i = 0$ 时，$\mathcal{H}(p) = -1 \cdot \log 1 = 0$。 ■