Actor-Critic 算法

基于价值的算法如 DQN 系列算法虽然在很多任务中取得了不错的效果，但由于其只能处理确定性策略，且难以适配连续动作空间，因此在某些复杂任务中表现不佳。而纯策略梯度算法如 REINFORCE 算法虽然在一定程度上解决了这些问题，但其高方差和低采样效率的问题，往往难以在复杂环境中取得良好的效果。为了解决这些问题，研究人员提出了结合策略梯度和值函数的方法，即 Actor-Critic 算法，即不仅将策略函数进行参数化，同时也将值函数进行参数化，从而兼顾两者的优点。

纯策略梯度算法的缺点

纯策略梯度算法虽然通过直接对策略进行参数化，解决了基于价值算法难以适配连续动作空间和随机策略的问题，但也带来了新的挑战。回顾策略梯度算法通用的目标函数，如式 (1) 所示。

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{\pi_\theta}\left[\Psi_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right] \tag{1}$$

其中 Ψ_t 是某种形式的回报估计，在纯策略梯度算法中通常使用蒙特卡洛估计，即 $\Psi_t = G_t$，用来评价策略产生的轨迹或状态-动作对的好坏。

然而，由于蒙特卡洛估计本身的高方差特性，导致策略梯度估计也会具有较高的方差，从而影响训练的稳定性和收敛速度。而且，每次更新策略参数时都需要采样完整的轨迹，这进一步降低了采样效率，尤其是在复杂环境中，可能需要大量的样本才能获得可靠的梯度估计。此外，奖励稀疏的环境中，蒙特卡洛估计可能会导致梯度估计不准确，从而影响策略的改进。

Actor-Critic 原理

为了兼顾策略梯度算法的灵活性和基于价值算法的高效性，Actor-Critic 算法应运而生，即将值估计的这部分工作交给一个独立的网络（Critic），而策略部分（Actor）则专注于策略的优化。这样不仅可以利用值函数来提供更稳定的梯度估计，还能提高采样效率，从而在复杂任务中取得更好的效果。

如图 1 所示，Actor 与环境交互采样生成轨迹样本，同时 Critic 网络则利用这些样本来估计当前状态或状态-动作对的价值。更新时，Critic 网络通过时序差分方法来更新其参数，而 Actor 则利用 Critic 提供的价值估计来指导策略的更新。

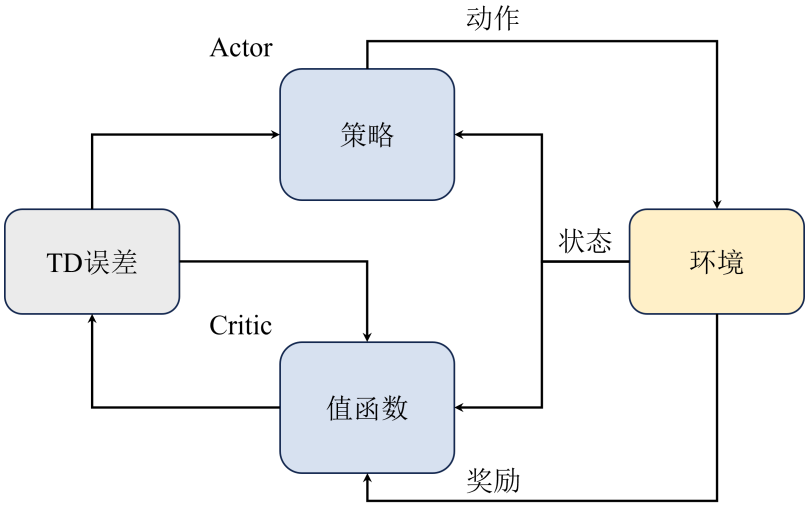

图 1: Actor-Critic 算法架构

注意，Actor-Critic 更多地是一种框架或架构，而不是具体的算法，不同算法的实现形式可能会有所不同，但都遵循 Actor-Critic 的基本思想。

例如，对于值估计部分（ Critic ）可以有多种形式，最基础的有两种：一种是使用状态价值函数 $V^{\pi}(s_t)$ 来估计当前状态的价值，另一种是使用状态-动作值函数 $Q^{\pi}(s_t, a_t)$ 来估计当前状态-动作对的价值。

使用状态价值函数来表示 Actor-Critic 算法的形式通常被称为 Value Actor-Critic 算法，如式 (2) 所示。

$$\nabla_{\theta} J(\theta) \propto \mathbb{E}_{\pi_{\theta}} \left[V_{\omega}(s_t) \nabla_{\theta} \log \pi_{\theta}(a_t \mid s_t) \right] \tag{2}$$

其中 ω 表示值函数的参数。在参数更新方面，对于策略网络（ Actor ）的参数 θ ，更新方式跟纯策略梯度算法类似，如式 (3) 所示。

$$\theta \leftarrow \theta + \alpha V_{\omega}(s_i) \nabla_{\theta} \log \pi_{\theta}(a_i \mid s_i) \tag{3}$$

对于值函数网络（ Critic ），可以先计算时序差分误差的梯度表达式，然后利用该误差来更新值函数的参数，如式 (4) 所示。

$$\nabla_{\omega} L(\omega) = (r_t + \gamma V_{\omega}(s_{t+1}) - V_{\omega}(s_t)) \nabla_{\omega} V_{\omega}(s_t) \tag{4}$$

在式 (4) 基础上，我们可以通过梯度下降的方法来更新值函数的参数 ω ，如式 (5) 所示。

$$\begin{aligned} y_i &= r_i + \gamma V_{\omega}(s_{i+1}) - V_{\omega}(s_i) \\ \omega &\leftarrow \omega + \beta y_i \nabla_{\omega} V_{\omega}(s_i) \end{aligned} \tag{5}$$

注意，跟 DQN 系列算法类似，在这里的参数更新表达式我们使用下标 i 来表示从经验回放缓冲区中采样的样本数据，以区别于与环境交互时的时间步 t 。

使用状态-动作值函数来表示 Actor-Critic 算法的形式通常被称为 Q Actor-Critic 算法，如式 (6) 所示。

$$\nabla_{\theta} J(\theta) \propto \mathbb{E}_{\pi_{\theta}} \left[Q_{\omega}(s, a) \nabla_{\theta} \log \pi_{\theta}(a \mid s) \right] \tag{6}$$

同样地，Critic 网络的参数 ω 也可以通过时序差分方法来更新，其梯度表达式如式 (7) 所示。

$$\nabla_{\omega} L(\omega) = \mathbb{E}_{\pi_{\theta}} \left[(r_t + \gamma Q_{\omega}(s_{t+1}, a_{t+1}) - Q_{\omega}(s_t, a_t)) \nabla_{\omega} Q_{\omega}(s_t, a_t) \right] \tag{7}$$

A2C 算法

与纯策略梯度算法中使用蒙特卡洛估计回报相比，使用状态价值 $V^{\pi}(s_t)$ 或状态-动作价值函数 $Q^{\pi}(s_t, a_t)$ 来估计当前状态或状态-动作对的价值，虽然能在一定程度上缓解纯策略梯度算法的高方差问题，但仍然存在一些不足。

为了进一步提高梯度估计的稳定性和准确性，我们可以引入优势函数（ advantage function ）来改进 Actor-Critic 算法，这就是 Advantage Actor-Critic （ A2C ）算法的基本思想。

优势函数 $A^{\pi}(s_t, a_t)$ 定义如式 (8) 所示。

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \tag{8}$$

其中 $Q^{\pi}(s_t, a_t)$ 表示在状态 s_t 下采取动作 a_t 的预期回报，而 $V^{\pi}(s_t)$ 则表示在状态 s_t 下的平均预期回报。优势函数 $A^{\pi}(s_t, a_t)$ 衡量了在给定状态下选择特定动作相对于平均水平的优势。

为什么引入优势函数能提高梯度估计的稳定性呢？这是因为优势函数通过减去一个基线（通常选择状态价值函数 $V^{\pi}(s_t)$ 作为基线），从而减少了梯度估计的方差。具体来说，原先对于每一个状态-动作对只能以自己为参照物估计，现在可以用平均水平作为参照物估计了，这样就能减少方差。

举例来说，小明在上次数学考试中得了 85 分，这次考试他得了 90 分。单看这次考试的成绩，似乎小明进步了 5 分。但是如果我们由于上次考试全班的平均分只有 75 分，而这次考试比较容易，全班的平均分达到了 100 分，就会发现相对于全班的平均水平，小明这次考试实际上退步了 20 分（从领先 10 分到落后 10 分）。这就是优势函数的作用，通过引入基线，我们能够更准确客观地评估小明的表现。

类似地，引入优势函数后的 A2C 算法的目标函数如式 (9) 所示。

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{\pi_\theta}\left[A^\pi(s_t, a_t)\nabla_\theta \log \pi_\theta(a_t \mid s_t)\right] \tag{9}$$

多进程

在讲 A3C 算法之前，我们先来了解一下多进程训练的基本思想。在实战中，会使用`multiprocessing`模块或者`ray`等分布式计算框架来实现多进程训练，如代码 1 所示。

代码 1: 使用 Ray 实现多进程计算

```python
import time
import ray

# 模拟耗时任务
def slow_square(x):
    time.sleep(1)
    return x * x

# Ray 版本
@ray.remote
def ray_square(x):
    time.sleep(1)
    return x * x


def run_single(nums):
    """单进程串行执行"""
    results = []
    for x in nums:
        results.append(slow_square(x))
    return results


def run_ray(nums):
    """Ray 并行执行"""


    futures = [ray_square.remote(x) for x in nums]
    results = ray.get(futures)

    ray.shutdown()
    return results


if __name__ == "__main__":
    nums = list(range(1, 9))
    print(f"任务数量: {len(nums)}")

    # --- 单进程 ---
    t0 = time.time()
    res_single = run_single(nums)
    t1 = time.time()
    print(f"单进程耗时: {t1 - t0:.2f} 秒")
```

```
    # --- Ray 并行 ---
    ray.init(ignore_reinit_error=True, num_cpus=8)
    t2 = time.time()
    res_ray = run_ray(nums)
    t3 = time.time()
    print(f"Ray 并行耗时: {t3 - t2:.2f} 秒")
```

执行结果如代码 2 所示。

代码 2: 多进程计算执行结果

```
任务数量: 8
单进程耗时: 8.01 秒
Ray 并行耗时: 2.69 秒
```

从结果可以看出，使用多进程并行计算显著减少了总的计算时间。

传统的强化学习算法通常是单进程串行训练，即一个智能体与环境交互并更新其策略和价值函数参数。然而，这种方式在某些情况下可能效率较低，尤其是在环境交互较慢或样本效率较低的情况下。而多进程训练可以让多个智能体同时与环境交互，同一时间内收集更多的样本数据，从而提高训练效率和样本利用率。

A3C 算法

基于多进程训练的思想，A3C （Asynchronous Advantage Actor-Critic）算法应运而生。A3C 算法通过引入多个并行的智能体（进程），每个智能体都拥有独立的策略和价值网络，并与环境进行交互采样数据。每个智能体在与环境交互一段时间后，会将其参数异步地更新到全局网络中，从而实现多进程的协同训练。

如图 2 所示，每一个智能体（进程）都拥有一个独立的网络和环境以供交互，并且每个进程每隔一段时间都会将自己的参数同步到全局网络中，这样就能提高训练效率。

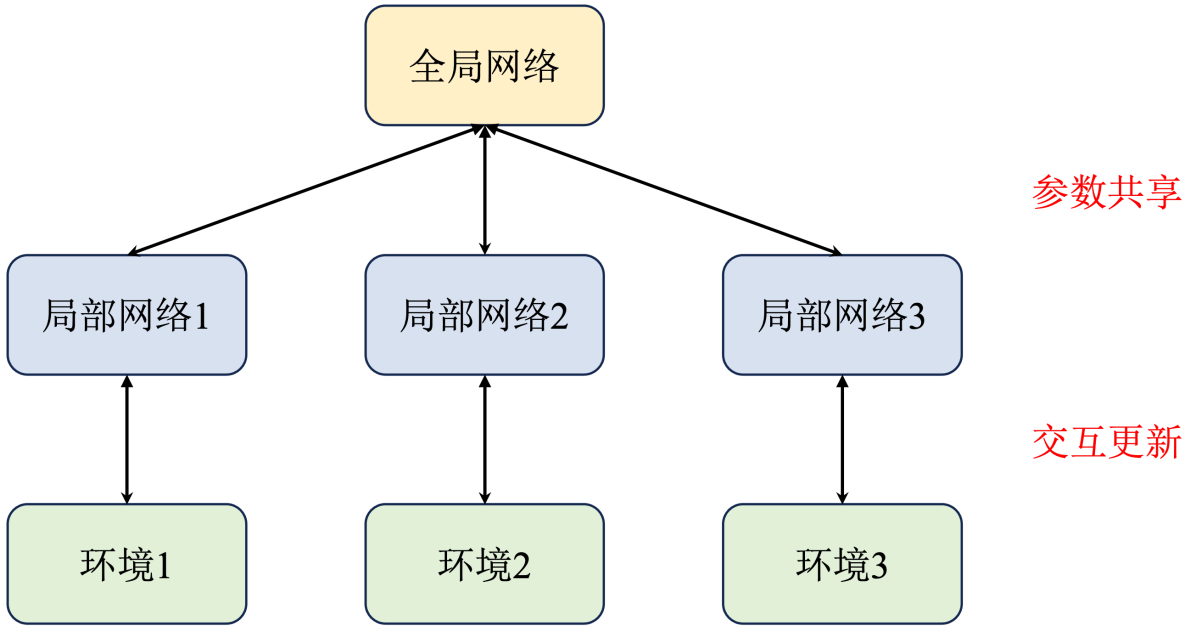

图 2: A3C 算法架构

与传统的单进程训练相比，A3C 算法除了能够提高训练效率外，还具有以下优点：

1. **增强探索能力**：多个智能体在不同的环境实例中进行探索，能够覆盖更广泛的状态空间，减少陷入局部最优的风险。

2. **稳定性提升**：异步更新机制能够减少不同智能体之间的相互干扰，从而提高训练的稳定性。

此外，多进程训练的思路也可以应用到其他强化学习算法中，包括基于价值的方法例如 DQN 系列算法，以及其他基于策略梯度的方法，例如后续章节要讲的 PPO 算法等，从而进一步提升这些算法的训练效率和性能。

广义优势估计

前面讲到通过引入优化函数来评估策略相对于平均水平的优势，如式 (10) 所示。

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \tag{10}$$

然而，在实际中我们并不知道真实的 $Q^{\pi}(s_t, a_t)$ 和 $V^{\pi}(s_t)$，所以需要估计。一种简单的方式是使用一阶时序差分（TD）误差来估计优势函数，如式 (11) 所示。

$$\hat{A}_t^{\mathrm{TD}} = r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t) \tag{11}$$

这种估计方式虽然方差较小，但是偏差较大。

也可以使用蒙特卡洛估计来估计优势函数，虽然这种方式是无偏估计，但是方差较大，如式 (12) 所示。

$$\hat{A}_t^{\mathrm{MC}} = G_t - V^{\pi}(s_t) = \sum_{l=0}^{\infty} \gamma^l r_{t+l} - V^{\pi}(s_t) \tag{12}$$

对应实现如代码 3 所示。

代码 3: 计算蒙特卡洛优势估计的示例代码

```python
def monte_carlo_advantage_estimation(rewards, values, gamma):
    """计算蒙特卡洛优势估计
    rewards: 奖励序列
    values: 价值函数估计序列
    gamma: 折扣因子
    """
    T = len(rewards)
    advantages = []
    for t in range(T):
        G = 0
        for l in range(T - t):
            G += (gamma ** l) * rewards[t + l]
        advantages.append(G - values[t])
    return advantages
# 示例数据
rewards = [1, 2, 3, 4, 5]
values = [0.5, 1.5, 2.5, 3.5, 4.5]
gamma = 0.9
advantages = monte_carlo_advantage_estimation(rewards, values, gamma)
print("蒙特卡洛优势估计:", advantages)
```

在这里我们还可以使用介于单步 TD 估计和蒙特卡洛估计之间的多步（n-step）估计来估计优势函数，如式 (13) 所示。

$$\hat{A}_t^{\mathrm{TD}(n)} = \sum_{l=0}^{n-1} \gamma^l r_{t+l} + \gamma^n V^{\pi}(s_{t+n}) - V^{\pi}(s_t) \tag{13}$$

对应实现如代码 4 所示。

代码 4: 计算 n 步优势估计的示例代码

```python
def n_step_advantage_estimation(rewards, values, gamma, n):
    """计算 n 步优势估计
    rewards: 奖励序列
    values: 价值函数估计序列
    gamma: 折扣因子
    n: 步数
    """
    T = len(rewards)
    advantages = []
    for t in range(T):
        G = 0
        for l in range(n):
            if t + l < T:  # 确保不越界
                G += (gamma ** l) * rewards[t + l]
            else:
                break
        if t + n < T:
            G += (gamma ** n) * values[t + n]
        advantages.append(G - values[t])
    return advantages

# 示例数据
rewards = [1, 2, 3, 4, 5]
values = [0.5, 1.5, 2.5, 3.5, 4.5]
gamma = 0.9
n = 3
advantages = n_step_advantage_estimation(rewards, values, gamma, n)
print("n 步优势估计:", advantages)
```

然而 n 步估计仍然存在一个问题，即 n 是一个固定的量，在实际环境中，不同状态下的轨迹长度、噪声大小、回报结构等因素可能会有所不同，因此很难让模型在每个时刻动态地选择合适的 n 值来调整方差和偏差的权衡。

为了解决这个问题，我们可以引入一个新的参数 λ，通过对不同步数的估计进行加权平均，从而形成一种新的估计方式，这就是广义优势估计（generalized advantage estimation，GAE）的方法，如式 (14) 所示。

$$\begin{aligned} A^{\mathrm{GAE}(\gamma,\lambda)}(s_t,a_t) &= \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l} \\ &= \sum_{l=0}^{\infty}(\gamma\lambda)^l\left(r_{t+l}+\gamma V^{\pi}(s_{t+l+1})-V^{\pi}(s_{t+l})\right) \end{aligned} \tag{14}$$

其中 δ_{t+l} 表示时间步 $t+l$ 时的 TD 误差，如式 (15) 所示。

$$\delta_{t+l}=r_{t+l}+\gamma V^{\pi}(s_{t+l+1})-V^{\pi}(s_{t+l}) \tag{15}$$

并且广义优势还有一个递推形式，如式 (16) 所示。

$$A_t^{\mathrm{GAE}(\gamma,\lambda)}=\delta_t+\gamma\lambda A_{t+1}^{\mathrm{GAE}(\gamma,\lambda)} \tag{16}$$

注意为了方便，这里将 $A^{\mathrm{GAE}(\gamma,\lambda)}(s_t,a_t)$ 简写为 $A_t^{\mathrm{GAE}(\gamma,\lambda)}$。

当 $\lambda = 0$ 时，根据递推公式，GAE 退化为单步 TD 误差，如式 (17) 所示。

$$A^{\mathrm{GAE}(\gamma,0)}(s_t, a_t) = \delta_t = r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t) \tag{17}$$

当 $\lambda = 1$ 时，GAE 退化为蒙特卡洛估计，如式 (18) 所示。

$$A^{\mathrm{GAE}(\gamma,1)}(s_t, a_t) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} = \sum_{l=0}^{\infty} (\gamma)^l \delta_{t+l} \tag{18}$$

通过调整 λ 的值，我们可以在方差和偏差之间进行权衡，从而获得更稳定和准确的优势估计，实现如代码 5 所示。

代码 5: 计算广义优势估计的示例代码

```python
import torch

def compute_gae(rewards, values, dones, gamma=0.99, lam=0.95):
    """
    计算 Generalized Advantage Estimation (GAE)

    参数:
        rewards (Tensor): shape = [T]，每一步的即时奖励
        values  (Tensor): shape = [T+1]，值函数 V(s_t)，最后一个是 V(s_{T+1})
        dones   (Tensor): shape = [T]，是否为终止状态（True/False）
        gamma (float): 折扣因子
        lam   (float): GAE 衰减系数 λ

    返回:
        advantages (Tensor): shape = [T]，GAE 优势值
        returns    (Tensor): shape = [T]，对应的回报值 (adv + value)
    """
    T = len(rewards)
    advantages = torch.zeros(T, dtype=torch.float32)
    last_adv = 0.0

    for t in reversed(range(T)):
        # 如果到达终止状态，下一步优势为0
        if dones[t]:
            next_non_terminal = 0.0
            next_value = 0.0
        else:
            next_non_terminal = 1.0
            next_value = values[t + 1]

        # TD 误差 δ_t
        delta = rewards[t] + gamma * next_value * next_non_terminal - values[t]
        # GAE 递推公式
        advantages[t] = delta + gamma * lam * next_non_terminal * last_adv
        last_adv = advantages[t]

    returns = advantages + values[:-1]
    return advantages, returns
# 示例数据
rewards = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0])
values = torch.tensor([0.5, 1.5, 2.5, 3.5, 4.5, 0.0])  # 注意 values 长度为 T+1
```

```
dones = torch.tensor([0, 0, 0, 0, 1], dtype=torch.bool)
advantages, returns = compute_gae(rewards, values, dones)
print("GAE 优势估计:", advantages)
print("对应回报值:", returns)
```

思考

相比于 REINFORCE 算法， A2C 主要的改进点在哪里，为什么能提高速度?

改进点主要有：**优势估计**：可以更好地区分好的动作和坏的动作，同时减小优化中的方差，从而提高了梯度的精确性，使得策略更新更有效率；**使用 Critic**： REINFORCE 通常只使用 Actor 网络，没有 Critic 来辅助估计动作的价值，效率更低；**并行化**：即 A3C ，允许在不同的环境中并行运行多个 Agent，每个 Agent 收集数据并进行策略更新，这样训练速度也会更快。

A2C 算法是 on-policy 的吗？为什么?

A2C 在原理上是一个 on-policy 算法，首先它使用当前策略的样本数据来更新策略，然后它的优势估计也依赖于当前策略的动作价值估计，并且使用的也是策略梯度方法进行更新，因此是 on-policy 的。但它可以被扩展为支持 off-policy 学习，比如引入经验回放，但注意这可能需要更多的调整，以确保算法的稳定性和性能。