



# 11

## Off-Policy 策略梯度法

我们已经介绍了很多有关策略梯度的算法，这些算法也取得了不错的成绩，但是它们仍然要面对策略梯度的两个软肋。这两个软肋已经在前面提到，笔者总结如下。

- **高方差：**由于每一次交互我们都能得到一条轨迹，基于这条轨迹的信息进行策略更新时会造成回报估计的波动，影响最终效果。为了减少方差对算法的影响，Actor Critic 方法增加了一个模型，用于估计当前状态的价值，通过引入一定的偏差换取方差的降低。
- **样本使用率：**这个问题是所有 On-Policy 算法都要解决的。前面提到策略梯度针对当前策略的价值希望进行优化，所以需要使用 On-Policy 的方法进行优化，这带来了两个问题。首先，On-Policy 方法并不是一个节约样本的方法，每一次策略发生改变，都要丢弃前面产生的样本，这将带来很大的样本浪费。对于通过计算机模拟进行的任务来说，这点浪费似乎不算什么，但是对于物理环境中的任务来说，机器人与真实环境的交互是十分耗时的，每次丢掉大量样本并不是一个好主意，所以我们需要考虑用 Off-Policy 的算法进行学习。

第 10 章介绍了用 TRPO 和 PPO 方法解决第一个问题，减少算法的波动，提高了算法的稳定性，甚至为策略模型提供了单调上升的性质；而本章我们要解决第二个问题，即使用 Off-Policy 的方法。第 7 章和第 8 章介绍了以 DQN 为核心思想的基于价值模型的 Off-Policy 算法，以及包括 Replay Buffer 在内的一系列结构，本章还会见到这些熟悉的身影。

## 11.1 Retrace

从本节开始，我们要使用由其他策略交互得到的样本对当前的模型进行训练，由于样本并非来自同一个样本，两者的概率分布也并不相同，因此我们无法直接使用这些样本。2.4 节介绍了“使用概率分布  $A$  为概率分布  $B$  采样”的方法——重要性采样，我们可以使用这样的方法解决概率分布不匹配的问题。本节，我们先来介绍如何估计 Off-Policy 的轨迹价值，并介绍 Retrace 算法。

### 11.1.1 Retrace 的基本概念

基于 Off-Policy 的价值估计方法主要使用重要性采样的方法实现，我们可以用一个  $R$  表示这一类计算方法的基本形式：

$$RQ_\pi(\mathbf{x}, \mathbf{a}) = Q_\pi(\mathbf{x}, \mathbf{a}) + E_\mu[\sum_{t \geq 0} \gamma^t (\prod_{s=1}^t c_s) (r_t + \gamma Q_\pi(\mathbf{x}_{t+1}, \cdot) - Q_\mu(\mathbf{x}_t, \mathbf{a}_t))]$$

其中  $Q(\mathbf{x}, \mathbf{a})$  表示值函数估计值， $\mu$  表示参与交互的策略， $\pi$  表示待学习的策略， $\gamma$  表示回报的折现率：

$$c_s = \frac{\pi(\mathbf{a}_s | \mathbf{x}_s)}{\mu(\mathbf{a}_s | \mathbf{x}_s)}$$

表示新旧策略的概率比率。

我们先考虑  $\mu$  和  $\pi$  完全相同的情况，此时公式中的  $\prod_{s=1}^t c_s$  等于 1。当  $t = 0$  时，上面的公式就变成了 Actor Critic 中 TD-Error 的计算公式：

$$RQ_\pi(\mathbf{x}, \mathbf{a}) = r_t + \gamma Q_\pi(\mathbf{x}_{t+1}, \cdot)$$

如果时间长度进一步拉长，我们可以得到

$$\begin{aligned} R_{t=1} Q_\pi(\mathbf{x}, \mathbf{a}) &= Q_\pi(\mathbf{x}, \mathbf{a}) + E_\pi[r_t + \gamma Q_\pi(\mathbf{x}_{t+1} | \cdot) - Q_\mu(\mathbf{x}_t, \mathbf{a}_t) + \gamma(r_{t+1} + \gamma Q_\pi(\mathbf{x}_{t+2} | \cdot) \\ &\quad - Q_\mu(\mathbf{x}_{t+1}, \mathbf{a}_{t+1}))] \\ &= Q_\pi(\mathbf{x}, \mathbf{a}) + E_\pi[\sum_{d=0}^1 \gamma^d (r_{t+d} + \gamma Q_\pi(\mathbf{x}_{t+d+1} | \cdot) - Q_\mu(\mathbf{x}_{t+d}, \mathbf{a}_{t+d}))] \end{aligned}$$

此时的公式形式和 10.2 节中 GAE 的计算公式比较接近。

当两个策略变得不同时， $\prod_{s=1}^t c_s$  这个项目就显得比较重要了。将上面的公式展开就可以得到

$$\begin{aligned}
 RQ_\pi(\mathbf{x}, \mathbf{a}) &= Q_\pi(\mathbf{x}, \mathbf{a}) + E_\pi[\sum_{t \geq 0} \gamma^t(r_t + \gamma Q_\pi(\mathbf{x}_{t+1}, \cdot) - Q_\mu(\mathbf{x}_t, \mathbf{a}_t))] \\
 &= Q_\pi(\mathbf{x}, \mathbf{a}) + \sum_{\tau} [\prod_t \pi(\mathbf{a}_t | \mathbf{s}_t) \sum_{t \geq 0} \gamma^t(r_t + \gamma Q_\pi(\mathbf{x}_{t+1}, \cdot) - Q_\mu(\mathbf{x}_t, \mathbf{a}_t))] \\
 &= Q_\pi(\mathbf{x}, \mathbf{a}) + \sum_{\tau} [\prod_t \mu(\mathbf{a}_t | \mathbf{s}_t) \frac{\prod_t \pi(\mathbf{a}_t | \mathbf{s}_t)}{\prod_t \mu(\mathbf{a}_t | \mathbf{s}_t)} \sum_{t \geq 0} \gamma^t(r_t + \gamma Q_\pi(\mathbf{x}_{t+1}, \cdot) - Q_\mu(\mathbf{x}_t, \mathbf{a}_t))] \\
 &= Q_\pi(\mathbf{x}, \mathbf{a}) + \sum_{\tau} [\prod_t \mu(\mathbf{a}_t | \mathbf{s}_t) \sum_{t \geq 0} \gamma^t \prod_t \frac{\pi(\mathbf{a}_t | \mathbf{s}_t)}{\mu(\mathbf{a}_t | \mathbf{s}_t)} (r_t + \gamma Q_\pi(\mathbf{x}_{t+1}, \cdot) - Q_\mu(\mathbf{x}_t, \mathbf{a}_t))] \\
 &= Q_\pi(\mathbf{x}, \mathbf{a}) + E_\mu[\sum_{t \geq 0} \gamma^t \prod_t \frac{\pi(\mathbf{a}_t | \mathbf{s}_t)}{\mu(\mathbf{a}_t | \mathbf{s}_t)} (r_t + \gamma Q_\pi(\mathbf{x}_{t+1}, \cdot) - Q_\mu(\mathbf{x}_t, \mathbf{a}_t))]
 \end{aligned}$$

重要性采样算法本身是无偏的，但是当两个分布相差比较大时，重要性采样算法的效果就会很差，而且在数值上可能出现一些问题。当  $\mu(\mathbf{a}_s | \mathbf{x}_s)$  非常小而  $\pi(\mathbf{a}_s | \mathbf{x}_s)$  非常大时，这个比值就会变得非常大，从而使模型产生较大的波动。所以直接使用重要性采样进行计算不能保证算法的稳定性，我们需要对这个比率做更多的限制。

为了解决这个问题，Retrace( $\lambda$ ) 算法（在本节中假设  $\lambda = 1$ ）采用了对比率进行限制的方式，使算法更新变得更稳定：

$$c_t = \min(1, \frac{\pi(\mathbf{a}_t | \mathbf{s}_t)}{\mu(\mathbf{a}_t | \mathbf{s}_t)})$$

通过这样的设定，比率既存在一个上限，不至于因为比率差距过大而对模型产生过大的波动，又可以保持重要性采样的基本形式，使计算不会有太大的偏差。我们可以证明得到 Retrace 算法同样具有良好的收敛性质，并使用它进行价值更新。更多的证明过程请读者阅读论文 *Safe and efficient off-policy reinforcement learning*<sup>[1]</sup> 了解更多的细节。

### 11.1.2 Retrace 的算法实现

Retrace 算法该如何实现呢？在 Baselines 项目的 baselines/acer/acer\_simple.py 文件中可以看到它的实现。这个子项目实现了 ACER 算法的全过程，其中包含了 Retrace 的实现。在 Retrace 的代码中，我们可以看到如下参数介绍：

```

def q_retrace(R, D, q_i, v, rho_i, nenvs, nsteps, gamma):
    """
    nenvs: 表示参与计算的episode数量，由于算法涉及多进程，所以实际上有nenvs
    个进程进行交互采样，每个进程完成1个episode的采样，这样就有nenvs个
    episode
    nsteps: 表示每个episode的长度
    R: 记录了每一时刻的Reward，维度为(nenvs, nsteps)
    D: 记录了每一时刻episode是否已经完成，维度为(nenvs, nsteps)
    q_i: 由模型计算得到的每一时刻的Q(s,a)
    v: 由模型计算得到的每一时刻的V(s)，计算方法为sum([p_i[i] * q_i[i] for i
        in actions]), p_i表示某个行动的发生概率
    rho_i: 两个策略的比率

```

函数返回：经过Q\_retrace计算得到的Return值

代码中已经假设 Retrace 算法中的  $\lambda$  值为 1，代码中  $\text{Retrace}(\lambda = 1)$  算法和论文中提到的公式有一些不同，我们需要重新做推导来说明实现过程采用的公式。首先，我们使用 Expected SARSA 的价值更新公式：

$$Q_{\pi}^{\text{ret}}(s_t, a_t) \leftarrow E_{a \sim \pi}[r_t + \gamma Q_{\pi}^{\text{ret}}(s_{t+1}, a_{t+1})]$$

SARSA 是一个 On-Policy 的算法，我们可以通过重要性采样的方法将其变成一个 Off-Policy 的方法：

$$Q_{\pi}^{\text{ret}}(s_t, a_t) = E_{a \sim \mu} \left[ \frac{\pi(a_{t+1}|s_{t+1})}{\mu(a_{t+1}|s_{t+1})} (r_t + \gamma Q_{\pi}^{\text{ret}}(s_{t+1}, a_{t+1})) \right]$$

用  $\rho_{t+1}$  代替  $\frac{\pi(a_{t+1}|s_{t+1})}{\mu(a_{t+1}|s_{t+1})}$ ，并将原本的值函数包含在公式中，可以得到

$$\begin{aligned} Q_{\pi}^{\text{ret}}(s_t, a_t) &= E_{a \sim \mu} [\rho_{t+1} (r_t + \gamma Q_{\pi}^{\text{ret}}(s_{t+1}, a_{t+1}))] + V_{\pi}(s_t) - V_{\pi}(s_t) \\ &= E_{a \sim \mu} [\rho_{t+1} (r_t + \gamma Q^{\text{ret}}(s_{t+1}, a_{t+1}) - Q_{\mu}(s_t))] + V_{\pi}(s_t) \end{aligned}$$

在计算值函数时我们就可以使用这个公式进行更新。下面就来考虑具体实现中的一些问题。现实中，我们需要对模型实际的轨迹长度进行限制，即使真实的轨迹长度超过了我们的限定，我们也会将其提前终止，所以对于每一个状态，我们必须使用额外的变量  $d_t$  表示时刻  $t$  的状态是否为终止状态。其中  $d_t = 1$  表示该轨迹终止， $d_t = 0$  表示

该轨迹仍在继续，那么我们就可以把公式拆解成两个部分：

$$\tilde{Q}^{\text{ret}}(s_t, a_t) = r_t + \gamma Q^{\text{ret}}(s_{t+1}, a_{t+1}) \times (1 - d_t) \quad (\text{第一部分})$$

$$Q^{\text{ret}}(s_t, a_t) = E_{a \sim \mu}[\rho_{t+1}(\tilde{Q}^{\text{ret}}(s_t, a_t) - Q_\mu(s_t, a_t))] + V_\pi(s_t) \quad (\text{第二部分})$$

实际上第二部分的计算是为第一部分服务的，最终要保留的是第一部分的结果。了解了这个原理，就可以把这两部分应用到下面的代码中。以下是对代码的详细介绍：

```
# 对 rho 进行限定，使其最小值为 1
# 代码中的batch_to_seq可以理解为将一个原本为(nenv, nsteps)的矩阵变换为
# 一个长度为nsteps的序列，里面的每一个元素都是长度为nenv的数组
rho_bar = batch_to_seq(tf.minimum(1.0, rho_i), nenvs, nsteps, True)
rs = batch_to_seq(R, nenvs, nsteps, True)
ds = batch_to_seq(D, nenvs, nsteps, True)
q_is = batch_to_seq(q_i, nenvs, nsteps, True)
vs = batch_to_seq(v, nenvs, nsteps + 1, True)
# 我们将最后一个时刻的值函数直接设置为模型计算得到的v
v_final = vs[-1]
qret = v_final
qrets = []
# 开始迭代计算
for i in range(nsteps - 1, -1, -1):
    # 对应第一部分的公式
    qret = rs[i] + gamma * qret * (1.0 - ds[i])
    qrets.append(qret)
    # 对应第二部分的公式
    qret = (rho_bar[i] * (qret - q_is[i])) + vs[i]
# 将计算得到的值按顺序翻转
qrets = qrets[::-1]
# 重新变回维度为(nenvs, nsteps)的格式
qret = seq_to_batch(qrets, flat=True)
return qret
```

这样我们就完成了 Retrace 的计算过程。

## 11.2 ACER

本节我们介绍的方法叫作 Actor Critic with Experience Replay (简称 ACER)，来自论文 *Sample Efficient Actor-Critic with Experience Replay*<sup>[3]</sup>，这是一种 Off-Policy 的 Actor-Critic 算法。由于 Off-Policy 在一些约定上与第 9 章及更前面的约定不同，同时也为了让读者对前面的内容做一个回顾，本节我们将重新对问题进行定义。

### 11.2.1 Off-Policy Actor-Critic

本节我们依然沿用第 6 章介绍的 MDP 框架。对于一个强化学习任务，我们用  $S$  表示状态集合， $A$  表示行动集合，那么状态转移概率分布可以表示为  $P: |S| \times |S| \times |A| \rightarrow [0, 1]$ ，在公式中我们用  $p(s'|s, a)$  表示由状态  $s$ 、行动  $a$  转移到状态  $s'$  的概率，用  $r$  表示 Agent 与环境交互过程中获得的反馈与回报。 $\gamma$  表示长期回报中未来回报对当前的打折率。状态价值可以用下面的公式表示：

$$V_\pi(s_t) = E[r_{t+1} + \dots + r_{t+T} | s_t = s]$$

这里我们假设任务从状态  $s_t$  出发，经过有限的时间长度  $T$ ，最终到达任务的终点。根据状态价值，我们还可以得出状态行动价值公式：

$$Q_\pi(s_t, a_t) = \sum_{s' \in S} p(s'|s_t, a_t)[r_{t+1} + \gamma V_\pi(s')]$$

Off-Policy 的策略梯度法涉及两个不同的策略。我们定义  $\mu$  为交互的序列样本  $\{s_0, a_0, r_0, \mu_0, \dots\}$  使用的策略，这些样本在交互后保存下来，并在未来的时刻应用于对某个策略  $\pi$  的训练中。

我们再定义状态出现的频率  $d^\mu(s) = \lim_{t \rightarrow \infty} p(s_t = s | s_0, \mu)$ ，它表示了在遵循策略  $\mu$  时，从任务的起始状态  $s_0$  出发，任意时刻到达状态  $s$  的概率之和。这样我们就可以定义目标：最大化从交互样本的每一状态出发的长期回报，令  $\theta$  为策略  $\pi$  的参数，那么目标公式可以写作

$$J(\theta) = \sum_{s \in S} d^\mu(s) V_\pi(s)$$

如果我们的问题是一个 On-Policy 的问题，那么目标公式中的  $d^\mu(s)$  将被换作  $d^\pi(s)$ ，如 9.1 节所示，我们已经了解过它的求解算法。这里使用与 9.1 节类似的求解方法，对

目标函数直接求导，可以得到

$$\begin{aligned}\nabla_{\theta} J(\theta) &= \nabla_{\theta} \left[ \sum_{s \in S} d^{\mu}(s) \sum_{a \in A} \pi(a|s) Q_{\pi}(s, a) \right] \\ &= \sum_{s \in S} d^{\mu}(s) \sum_{a \in A} [\nabla_{\theta} \pi(a|s) Q_{\pi}(s, a) + \pi(a|s) \nabla_{\theta} Q_{\pi}(s, a)]\end{aligned}$$

对公式来说， $\nabla_{\theta} Q_{\pi}(s, a)$  是一个比较难求解的项目，尤其是在它和  $d^{\mu}(s)$  混合表示的情况下。因此，对 Off-Policy 的方法来说，我们可以忽略这一项，于是公式就变为

$$\nabla_{\theta} J(\theta) \simeq g(\theta) = \sum_{s \in S} d^{\mu}(s) \sum_{a \in A} [\nabla_{\theta} \pi(a|s) Q_{\pi}(s, a)]$$

实际上即使忽略掉那一部分的公式，我们依然可以在一定的条件下确保策略的提升。我们定义参数的更新公式为

$$\theta' \leftarrow \theta + \alpha g(\theta)$$

当  $\alpha$  被限定为一个比较小的值时，例如存在一个  $\epsilon > 0$ ，使得对所有的  $\alpha < \epsilon$ ，依然可以得到

$$J(\theta') \geq J(\theta)$$

这个结论要如何证明呢？我们知道现在的近似梯度  $g(\theta)$  实际上求解了策略  $\pi$  的梯度。也就是说，在  $\theta$  附近的区域内，我们可以沿着  $g(\theta)$  的方向使策略得到提高。于是我们可以通过设置  $\alpha$ ，满足  $\pi(a|s; \theta') \geq \pi(a|s; \theta)$ ，于是就可以得到

$$\begin{aligned}J(\theta) &= \sum_{s \in S} d^{\mu}(s) \sum_{a \in A} \pi(a|s; \theta) Q_{\pi}(s, a) \\ &\leq \sum_{s \in S} d^{\mu}(s) \sum_{a \in A} \pi(a|s; \theta') Q_{\pi}(s, a)\end{aligned}$$

进一步替换，还可以得到

$$\begin{aligned}&= \sum_{s \in S} d^{\mu}(s) \sum_{a \in A} \pi(a|s; \theta') \sum_{s' \in S} p(s'|s_t, a_t) [r_{t+1} + \gamma V_{\pi}(s')] \\ &= \sum_{s \in S} d^{\mu}(s) \sum_{a \in A} \pi(a|s; \theta') \sum_{s' \in S} p(s'|s_t, a_t) [r_{t+1} + \gamma \sum_{a' \in A} \pi(a'|s'; \theta) Q_{\pi}(s', a')] \\ &\leq \sum_{s \in S} d^{\mu}(s) \sum_{a \in A} \pi(a|s; \theta') \sum_{s' \in S} p(s'|s_t, a_t) [r_{t+1} + \gamma \sum_{a' \in A} \pi(a'|s'; \theta) Q_{\pi}(s', a')]\end{aligned}$$

这样不断地展开，就可以得到

$$\begin{aligned} &\leq \sum_{s \in S} d^\mu(s) \sum_{a \in A} \pi(a|s; \theta') Q_{\pi_{\theta'}}(s, a) \\ &= J(\theta') \end{aligned}$$

基于这样的证明，我们发现这个简化的梯度同样可以对目标函数进行优化，因此我们可以选择这个方法。将梯度公式进一步变换，可以得到更易于计算的形式，用  $\beta$  表示  $d^\mu$ ，可以得到

$$\begin{aligned} g(\theta) &= \sum_{s \in S} \beta(s) \sum_{a \in A} [\nabla_\theta \pi(a|s) Q_\pi(s, a)] \\ &= E_{s \sim \beta} [\sum_{a \in A} \nabla_\theta \pi(a|s) Q_\pi(s, a)] \\ &= E_{s \sim \beta} [\sum_{a \in A} \mu(a|s) \frac{\pi(a|s)}{\mu(a|s)} \frac{\nabla_\theta \pi(a|s)}{\pi(a|s)} Q_\pi(s, a)] \\ &= E_{s \sim \beta, a \sim \mu(\cdot|s)} [\rho(s, a) \nabla_\theta \log \pi(a|s) Q_\pi(s, a)] \end{aligned}$$

其中  $\rho(s, a) = \frac{\pi(a|s)}{\mu(a|s)}$ ，表示了两个策略的比值。现在公式中的状态和行动完全遵循策略  $\mu$ ，我们就可以采用蒙特卡罗法进行采样，并使用上面的公式进行梯度计算。

看上去核心工作已经完成，而实际上同 11.1 节类似，两个策略的比值可能对训练造成不好的影响。如果两个策略的概率比值差异很大，那么梯度更新的步长也会很大，模型就会产生较大的波动，接下来介绍的 ACER 算法就是要解决这方面的问题。

### 11.2.2 ACER 算法

在 11.1 节，我们介绍了 Retrace 算法，它通过对新旧策略比值进行限定减少重要性采样带来的不稳定性。ACER 算法也要使用类似的方式，首先，我们将公式分解成 On-Policy 的部分和 Off-Policy 的部分。公式计算如下所示：

$$\begin{aligned} g^{\text{marg}} &= E_{s_t \sim \beta, a_t \sim \mu} [\rho_t \nabla_\theta \log \pi_\theta(a_t|s_t) Q^\pi(s_t, a_t)] \\ &= E_{s_t \sim \beta, a_t \sim \mu} [((1 - \frac{c}{\rho_t} + \frac{c}{\rho_t}) \rho_t \nabla_\theta \log \pi_\theta(a_t|s_t) Q^\pi(s_t, a_t))] \\ &= E_{s_t \sim \beta} [E_{a_t \sim \mu} [((1 - \frac{c}{\rho_t} + \frac{c}{\rho_t}) \rho_t \nabla_\theta \log \pi_\theta(a_t|s_t) Q^\pi(s_t, a_t)]]] \\ &= E_{s_t \sim \beta} [E_{a_t \sim \mu} [\frac{c}{\rho_t} \rho_t \nabla_\theta \log \pi_\theta(a_t|s_t) Q^\pi(s_t, a_t)]] \end{aligned}$$

$$\begin{aligned}
& + E_{a_t \sim \mu} [(1 - \frac{c}{\rho_t}) \rho_t \nabla_\theta \log \pi_\theta(a_t | s_t) Q^\pi(s_t, a_t)] \\
& = E_{s_t \sim \beta} [E_{a_t \sim \mu} [\frac{c}{\rho_t} \rho_t \nabla_\theta \log \pi_\theta(a_t | s_t) Q^\pi(s_t, a_t)] \\
& \quad + E_{a_t \sim \pi} [(\frac{\rho_t - c}{\rho_t}) \nabla_\theta \log \pi_\theta(a_t | s_t) Q^\pi(s_t, a_t)]]
\end{aligned}$$

如果我们用一个经过限定的比率  $\tilde{\rho}_t$  替换其中一部分的  $\rho_t$ , 同时限定  $\tilde{\rho}_t = \max(\rho_t, c)$ , 就有

$$\begin{aligned}
\frac{c}{\tilde{\rho}_t} &= \min(\frac{c}{\rho_t}, 1), \quad \frac{c}{\tilde{\rho}_t} \rho_t = \min(c, \rho_t) \\
\tilde{\rho}_t - c &= \max(\rho_t - c, 0), \quad \frac{\tilde{\rho}_t - c}{\rho_t} = \max(\frac{\rho_t - c}{\rho_t}, 0)
\end{aligned}$$

我们就可以将上面的公式替换为 ACER 算法的公式:

$$\begin{aligned}
& = E_{s_t \sim \beta} [E_{a_t \sim \mu} [\min(c, \rho_t) \nabla_\theta \log \pi_\theta(a_t | s_t) Q^\pi(s_t, a_t)] \\
& \quad + E_{a_t \sim \pi} [(\max(\frac{\rho_t - c}{\rho_t}, 0) \nabla_\theta \log \pi_\theta(a_t | s_t) Q^\pi(s_t, a_t)]]
\end{aligned}$$

这个公式中的两个项目对算法起到不同的作用。第一项被称为截断的重要性采样 (Truncated Importance Sampling), 和 Retrace 算法的思想类似, 它限制了重要性采样的比率上限, 我们可以确保模型不会造成太大的波动。第二项被称为偏差纠正项 (Bias Correction for the Truncation), 作为前面一项的纠正项, 它可以保证算法是无偏的, 这样算法就在偏差和方差之间得到了一定的平衡。实际上, 只有  $\rho_t > c$  时, 第二项才会发挥作用, 而此时第一项的比率被限定为  $c$ , 这时第二项会对第一项的限制做一定的补充, 从而保证算法没有较大的偏差。

完成了这一步的变换, 下面就要对值函数做一定的替换。对上面公式的第一项, 由于是 Off-Policy 的期望计算, 可以用 11.1 节介绍的 Retrace( $\lambda$ ) 算法计算价值估计值  $Q^{\text{ret}}(s_t, a_t)$ , 而第二项由于是 On-Policy 的价值计算, 因此直接使用价值模型的估计值即可, 模型的参数为  $\theta_v$ 。这样公式就变成了下面的形式:

$$\begin{aligned}
& = E_{s_t \sim \beta} [E_{a_t \sim \mu} [\min(c, \rho_t) \nabla_\theta \log \pi_\theta(a_t | s_t) Q^{\text{ret}}(s_t, a_t)] \\
& \quad + E_{a_t \sim \pi} [(\max(\frac{\rho_t - c}{\rho_t}, 0) \nabla_\theta \log \pi_\theta(a_t | s_t) Q_{\theta_v}(s_t, a_t)]]
\end{aligned}$$

最后一步是 Actor-Critic 算法中常见的步骤, 为算法添加一个 Baselines, 以降低目标函数的方差, 我们可以用价值模型估计当前的状态值函数  $V(s)$ , 最终的模型变为

$$\begin{aligned}
&= E_{s_t \sim \beta} [E_{a_t \sim \mu} [\min(c, \rho_t) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) (Q^{\text{ret}}(s_t, a_t) - V(s_t))] \\
&\quad + E_{a_t \sim \pi} [(\max(\frac{\rho_t - c}{\rho_t}, 0) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) (Q_{\theta_v}(s_t, a_t) - V_{\theta_v}(s_t))]]
\end{aligned}$$

完成了上面对目标函数的定义，下面就来介绍求解的具体过程。10.1 节已经介绍了 TRPO 方法，它能够比较好地解决 Actor-Critic 算法中模型效果波动较大的问题，于是我们也采用同样思想的方法进行求解。但是由于两个算法要解决的问题不同，同时希望加快模型训练的速度，ACER 算法采取了两个改变：采用更保守的参数更新方式和相对简单的目标函数。

第一个改变是参数更新方式。ACER 维护了一个滑动平均的策略网络，每一次模型更新时，新的参数仅会以一个很小比例进行更新。这样我们就可以确保每一次更新后的参数和之前的参数保持较近的距离。这样算法可以以较小的代价实现类似 TRPO 约束的效果。我们令平均策略的参数为  $\theta_a$ ，优化后的策略参数为  $\theta$ ，那么平均策略的更新公式为

$$\theta_a \leftarrow \alpha \theta + (1 - \alpha) \theta_a$$

第二个改变是目标函数。ACER 目标函数中计算一阶梯度的部分与 TRPO 比较相似，只要将前面得到的公式求导就可以得到

$$\begin{aligned}
\hat{g}_t^{\text{acer}} &= \min(c, \rho_t) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) [Q^{\text{ret}}(s_t, a_t) - V_{\theta_v}(s_t)] \\
&\quad + E_{a_t \sim \pi} [(\max(\frac{\rho_t - c}{\rho_t}, 0) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) (Q_{\theta_v}(s_t, a_t) - V_{\theta_v}(s_t))]
\end{aligned}$$

ACER 的约束条件与 TRPO 不同，没有使用二阶导数做约束，而是使用了 KL 散度的一阶导数，这样目标函数可以变为下面的形式：

$$\begin{aligned}
&\text{minimize}_{z} \frac{1}{2} \|\hat{g}_t^{\text{acer}} - z\|_2^2 \\
&\text{s.t. } \nabla_{\pi_{\theta}(x_t)} D_{\text{KL}}[\pi_{\theta_a} \| \pi_{\theta}]^T z < \delta
\end{aligned}$$

可以看出，这个目标函数虽然也拥有基于 KL 散度的约束条件，但是它与 TRPO 的形式完全不同。经过这样的变换，我们不再需要计算复杂的 Fisher 信息矩阵。最终的目标是求解的参数更新量  $z$ ，我们希望它在满足约束条件的同时尽可能地靠近  $\hat{g}_t^{\text{acer}}$  值。

上面提到的对目标函数的两个改变实际上为更新量的求解提供了很大的便利。为了后面求解公式时的便利，我们做如下的变量替换：

$$\hat{g}_t^{\text{acer}} = g$$

$$\nabla_{\phi_\theta(x_t)} D_{\text{KL}}[\pi_{\theta_a} \| \pi_\theta] = k$$

这个目标函数实际上是一个二次规划的问题。由于问题中目标函数的二阶导为半正定矩阵（在这个问题中，二阶导实际上是一个对角阵），我们可以证明这个问题满足 KKT 条件，它的对偶问题具有强对偶的特性，原问题和它的对偶问题的解相同。于是我们就可以构造这个问题的拉格朗日乘子法形式：

$$\underset{\mathbf{z}}{\text{minimize}} H(\mathbf{z}, \lambda) = \frac{1}{2} \|\mathbf{g} - \mathbf{z}\|_2^2 + \lambda(\mathbf{k}^T \mathbf{z} - \delta)$$

首先对  $\mathbf{z}$  进行求导，可以得到

$$\frac{\partial H(\mathbf{z}, \lambda)}{\partial \mathbf{z}} = \mathbf{z} - \mathbf{g} + \lambda \mathbf{k}$$

令导数为 0，可以得到  $\mathbf{z}$  的最优值

$$\mathbf{z}^* = \mathbf{g} - \lambda \mathbf{k}$$

将其带回上面的拉格朗日公式中，可以得到

$$\begin{aligned} H(\lambda) &= \frac{1}{2} \lambda^2 \|\mathbf{k}\|_2^2 + \lambda(\mathbf{k}^T (\mathbf{g} - \lambda \mathbf{k}) - \delta) \\ &= \frac{1}{2} \|\mathbf{k}\|_2^2 \lambda^2 + \lambda \mathbf{k}^T \mathbf{g} - \lambda^2 \|\mathbf{k}\|_2^2 - \lambda \delta \\ &= -\frac{1}{2} \|\mathbf{k}\|_2^2 \lambda^2 + (\mathbf{k}^T \mathbf{g} - \delta) \lambda \end{aligned}$$

对其进行求导，可以得到

$$\frac{\partial H(\lambda)}{\partial \lambda} = -\|\mathbf{k}\|_2^2 \lambda + (\mathbf{k}^T \mathbf{g} - \delta)$$

令导数为 0，可以得到

$$\lambda^* = \frac{\mathbf{k}^T \mathbf{g} - \delta}{\|\mathbf{k}\|_2^2}$$

因为朗格朗日的乘子被限定为不小于 0，所以  $\lambda^*$  不小于 0，将其带回前面  $\mathbf{z}^*$  表示的公式，可以得到

$$\mathbf{z}^* = \mathbf{g} - \max(0, \frac{\mathbf{k}^T \mathbf{g} - \delta}{\|\mathbf{k}\|_2^2}) \mathbf{k}$$

从结果可以看出，当约束条件可以得到满足时，最终的更新值就是梯度  $\mathbf{g}$ ；如果约束无法满足，那么  $\mathbf{z}$  就会沿着  $\mathbf{k}$  的方向进行一定的缩减，使更新量满足约束，这样就得到了我们想要的更新量。

### 11.2.3 ACER 的实现

ACER 的完整算法流程如下所示，流程主要包含两个子过程，一个是采用 On-Policy 的优化，也就是基于当前的策略与环境交互采集样本；另一个是采用 Off-Policy 的优化，直接使用保存起来的样本进行训练。这个训练流程和 DQN 的训练方法十分类似，ACER 和 DQN 都需要定期使用新的策略采集样本，以保证 Replay Buffer 中采样样本的策略和当前的策略差距不大。

由于在 ACER 中需要计算两个策略的概率比率，因此在 Replay Buffer 中需要将采样策略的行动概率值保存，因此 Replay Buffer 中就需要额外保存这个信息。

---

#### 算法 ACER 算法的主体

---

**Repeat**

    Call ACER (on-policy = True)  
     **for**  $i \in \{1, \dots, n\}$  **do**  
         Call ACER (on-policy = False)  
     **end for**

**until** 最大的迭代轮数满足

---

#### 子算法 离散行动的 ACER (on-policy)

---

重置参数梯度： $d\theta \leftarrow 0$ ,  $d\theta_v \leftarrow 0$

初始化参数： $\theta' \leftarrow \theta$ ,  $\theta'_v \leftarrow \theta_v$

**If** on-policy==True:

    利用 policy 和环境交互得到  $\{s_t, a_t, r_t, \mu_t, d_t\}_{t=1}^k$ , 并将其放入 Replay Buffer, 其中：  
 $\mu(\cdot|s_i) \leftarrow \pi(a|s_i; \theta')$

**else:**

    从 Replay Buffer 中采样得到  $\{s_t, a_t, r_t, \mu_t, d_t\}_{t=1}^k$

**end if**

**for**  $i \in \{0, \dots, k\}$  **do**

    计算： $\pi(a|s_t; \theta')$ 、 $Q(s_t, \cdot; \theta'_v)$  和  $\pi(a|s_t; \theta_a)$

$\bar{\rho}_t \leftarrow \min\{1, \frac{\pi(a_t|s_t; \theta')}{\mu(a_t|s_t)}\}$

**end for**

---

---

$Q^{\text{ret}} \leftarrow \begin{cases} 0 & \text{终止状态 } s_t \\ \sum_a Q(s_t, a|\theta'_v) \pi(a|s_t; \theta') & \text{其他情况} \end{cases}$   
**for**  $t \in \{k-1, \dots, 0\}$  **do**  
     $Q^{\text{ret}} \leftarrow r_i + \gamma Q^{\text{ret}}$   
     $V_t \leftarrow \sum_a Q(s_t, a; \theta'_v) \pi(a|s_t; \theta')$   
    计算 Trust Region 目标需要的部分公式：  
         $g \leftarrow \min(c, \rho_t) \nabla_{\theta'} \log \pi_{\theta'}(a_t|s_t) [Q^{\text{ret}}(s_t, a_t) - V_{\theta'_v}(s_t)] + \sum_a [(\max(\frac{\rho_t - c}{\rho_t}, 0) \nabla_{\theta'} \log \pi_{\theta'}(a_t|s_t) (Q_{\theta'_v}(s_t, a_t) - V_{\theta'_v}(s_t))]^2$   
         $k \leftarrow \nabla_{\pi_{\theta'}} D_{\text{KL}}[\pi_{\theta_a} \| \pi_{\theta'}]$   
        计算策略参数  $\theta'$  的更新量： $d\theta' \leftarrow d\theta + g - \max(0, \frac{k^T g - \delta}{\|k\|_2^2}) k$   
        计算价值参数  $\theta'_v$  的更新量： $d\theta'_v \leftarrow d\theta'_v + \nabla_{\theta'_v} (Q^{\text{ret}} - Q_{\theta'_v}(s_t, a))^2$   
        更新 Retrace 目标： $Q^{\text{ret}} \leftarrow \bar{\rho}_t (Q^{\text{ret}} - Q(s_t, a_t; \theta'_v)) + V_t$   
**end for**  
    将  $d\theta'$  和  $d\theta'_v$  更新到原始的参数  $\theta$  和  $\theta_v$  上  
    更新平均策略参数： $\theta_a \leftarrow \alpha \theta_a + (1 - \alpha) \theta$

---

介绍完 ACER 的算法流程，再介绍它在 Baselines 中的实现。这部分代码在 baselines/acer 项目中，核心代码在 acer\_simple.py 中。由于涉及的内容比较多，模型架构相对复杂，如图 11-1 和图 11-2 所示。

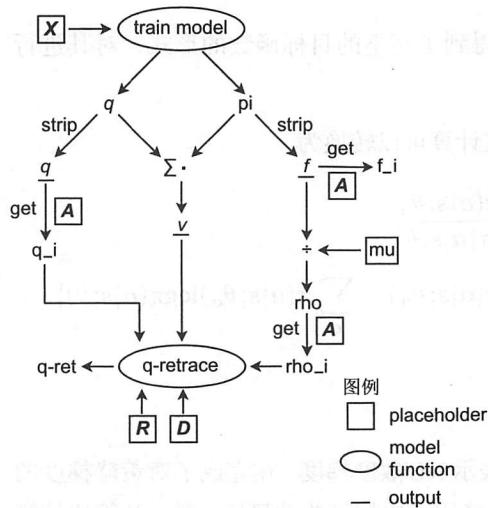


图 11-1 ACER 的流程图 1

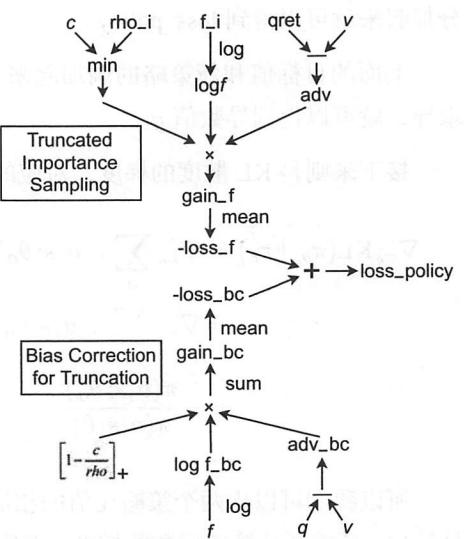


图 11-2 ACER 的流程图 2

图 11-1 所示为  $Q^{\text{ret}}$  的计算过程。算法的模型部分和 A2C 的模型有些不同，它要输出  $\pi(a|s)$  和  $q(s, a)$  两个值，在图中分别用  $\text{pi}$  和  $q$  表示。这两个值进行乘加操作，可以得到价值  $v$ 。代码中有一个 `strip` 操作，用于将数据转换成算法中易于处理的形式， $\text{pi}$  变形成  $f$ ，并与 Replay Buffer 中存储的代表  $\mu(a|s)$  的  $\text{mu}$  相除，得到策略的比值  $\text{rho}$ 。

当我们知道了回报  $R$ ，终止状态  $D$ ，价值  $q$ 、 $v$ ，策略比值  $\text{rho}$  后，就可以利用 Retrace 方法计算出  $Q^{\text{ret}}$  值  $q_{\text{ret}}$ 。

图 11-2 所示为 ACER 计算公式中的两部分。首先是截断的重要性采样部分，它由  $Q^{\text{ret}}$  和价值的估计值的差、策略的对数似然和比率阶段值三部分组成，公式为

$$\min(c, \rho_t) \nabla_{\theta'} \log \pi_{\theta'}(a_t | s_t) [Q^{\text{ret}}(s_t, a_t) - V_{\theta'_v}(s_t)]$$

其中后两项的计算结果分别为  $\log f$  和  $\text{adv}$ ，由于这三项中只有第二项参与梯度计算，因此第一、第三项需要被 `tf.stop_gradient` 包含，以关闭它们的反向计算。

ACER 公式的第二部分，也就是偏差纠正部分，同样是由比率值、对数似然和公式及价值差组成，对应的公式为

$$\sum_a (\max(\frac{\rho_t - c}{\rho_t}, 0) \nabla_{\theta'} \log \pi_{\theta'}(a_t | s_t) (Q_{\theta'_v}(s_t, a_t) - V_{\theta'_v}(s_t)))$$

其中后两项的计算结果分别为  $\log_f_{\text{bc}}$  和  $\text{adv}_{\text{bc}}$ ，这个公式同样只有第二项参与梯度计算，因此处理方式与前面一致。这样得到了两部分的 loss 值： $\text{loss\_f}$  与  $\text{loss\_bc}$ ，将两部分加起来就可以得到  $\text{loss\_policy}$ 。

上面的目标值和新策略的熵加起来，就得到了完整的目标函数的形式，对其进行求导，就可以得到导数值  $g$ 。

接下来则是 KL 散度的梯度，对应的公式计算可以转换为

$$\begin{aligned} \nabla_{\pi_\theta} \text{KL}(\pi_{\theta_a} \| \pi_\theta) &= \nabla_{\pi_\theta} \sum_a \pi(a|s; \theta_a) \log \frac{\pi(a|s; \theta_a)}{\pi(a|s; \theta)} \\ &= \nabla_{\pi_\theta} [\sum_a \pi(a|s; \theta_a) \log \pi(a|s; \theta_a) - \sum_a \pi(a|s; \theta_a) \log \pi(a|s; \theta)] \\ &= -\frac{\pi(a|s; \theta_a)}{\pi(a|s; \theta)} \end{aligned}$$

所以我们可以用两个策略比值的相反数表示 KL 散度梯度。在完成了对策略梯度的计算后，还需要计算值函数的梯度，而值函数使用了比较简单的目标函数，计算比较简单，具体过程请读者自行阅读。

可以看出，只要能够理解 Retrace 算法和 ACER 中形似 TRPO 目标函数，整体的算法流程是比较容易理解的。前面我们介绍的只是离散行动的解法，关于连续行动空间的解法，欢迎读者阅读论文进行了解，其主体思想与离散方法基本一致，只是在一些细节上有所不同。

## 11.3 DPG

11.2 节从 Experience Replay 的角度实现了 Actor-Critic 方法，本节要介绍另外一个 Off-Policy 的算法，这个方法被称为 Deterministic Policy Gradient (简称 DPG)，这个算法经过几篇论文的不断打磨改进，效果变得越来越好。这个算法的名称看上去和 Policy Gradient 相似，但其实际计算过程并不相同，从名字中可以看出，这个方法使用了确定的策略，这不同于之前提到的输出分布的策略形式。下面我们就来看看算法的计算过程。

### 11.3.1 连续空间的策略优化

在前面的章节中，我们主要介绍了离散行动空间的任务，例如 Atari 游戏，由于游戏手柄的操作形式有限，我们可以使用探索算法尽可能地将状态行动枚举出来。对于行动连续的任务，想要枚举所有的行动变得更困难，而如何将所有可行的行动逐一尝试出来也变得不那么可能。我们先回顾前面介绍过的两种强化学习算法的计算公式。首先是 DQN 算法的更新公式：

$$\Delta q(s, a) = r(s') + \max_{a'} q^{T-1}(s', a')$$

DQN 算法遵循了泛化策略梯度的思想，先完成策略评估的工作，再进行策略改进。这个过程需要计算出下一时刻状态下所有行动的价值，并从中选出最优的行动价值。如果行动数量是有限的，那么这个公式比较好计算；如果行动空间是连续的，我们该如何找出价值最优的行动呢？这就成为了一个问题。

接下来是策略梯度法，它的核心计算公式为

$$\nabla_{\theta} J(\theta) = E_{\tau \sim \pi_{\theta}(\tau)} \left[ \left( \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t} | s_{i,t}) \right) \left( \sum_{t=0}^T r(s_{i,t}, a_{i,t}) \right) \right]$$

策略梯度法由于直接对轨迹的价值期望求导，因此它不需要完成最优行动选取的

操作。因此连续型行动空间的问题可以使用策略梯度算法求解，但是策略梯度法是一个 On-Policy 的方法，它非常依赖与环境交互的过程，而 DQN 方法直接对值函数进行优化，可以使用 Off-Policy 的方法进行训练。所以，如果我们想在连续行动空间使用 Off-Policy 算法进行优化，可以考虑结合两种算法的特点，创造出一种全新的算法。不同于 11.2 节使用重要性采样进行求解，本节介绍的 DPG 算法将采用另外一种方式。

在与环境交互时，DQN 算法一般使用  $\epsilon$ -greedy 的策略，策略梯度是从一个概率分布中采样得到的，而 DPG 的交互方式结合了前面两种算法。从形式上看，DPG 使用了  $\epsilon$ -greedy 的策略，以一定的概率使用随机策略，而在剩下的情况下使用最优行动；从策略产生的行动上看，DPG 将先得到一个确定的行动，这个行动由确定的策略得到，不需要从概率分布中采样，相当于当前状态下的最优行动。如果决定使用随机策略，那么就在求出的确定行动基础上加上一定的噪声，反之则没有噪声。

虽然确定策略的思想和 DQN 相近，但实际上，DPG 也可以看作是策略梯度法的一种特殊情况。我们知道随机策略梯度的输出是行动分布形式，对于离散行动空间，模型输出的是一个 Category 的分布，也就是每一个取值的概率；而对于连续行动空间，我们一般会输出一个高斯分布，其中一部分值表示分布的均值，另一部分值表示分布的方差，然后我们可以使用这些分布的参数采样出行动值。DPG 的输出也可以想象成一个连续的分布，只不过这个分布的方差为 0。这样我们就把 DPG 和策略梯度法统一起来了，前面介绍的策略梯度法也可以称为随机策略梯度法。

下面我们就利用这个思路建立目标。DPG 的算法本身还是采用策略梯度法的方法，直接对轨迹价值求导。但是由于策略产生的行动是确定的，于是行动就可以直接被替换为策略函数，公式将变为下面的形式：

$$\nabla_{\theta} v_{\mu}(s) = \nabla_{\theta}[q_{\mu}(s, \mu(s))], \forall s \in S$$

其中  $\mu$  表示生成确定行动的策略函数。将上面的公式进一步推导，利用链式法则可以得到

$$= \nabla_{\mu(s)} q_{\mu}(s, \mu(s)) \nabla_{\theta} \mu(s)$$

同随机策略梯度法对比，这个梯度公式中没有了与行动相关的期望求解项。下面我们就围绕这个公式进行计算。

### 11.3.2 策略模型参数的一致性

在进一步介绍确定策略的目标推导前，我们需要简单介绍一个概念：策略模型参

数的一致性。在随机策略梯度方法的介绍过程中，我们曾提到直接使用一个函数来近似状态行动价值，即定义一个函数  $f_w : S \times A \rightarrow R$ ，用来近似真实的值函数  $Q^\pi$ ，函数的参数为  $w$ 。如果我们用 L2 损失作为近似函数  $f_w$  和真实价值  $Q^\pi$  的损失函数，那么近似函数的梯度为

$$\nabla_w \text{Loss} \propto \nabla_w E_{s,a}[\hat{Q}^\pi(s,a) - f_w(s,a)]^2 = E_{s,a \sim \tau}[(\hat{Q}^\pi(s,a) - f_w(s,a)) \nabla_w f_w(s,a)]$$

其中  $\hat{Q}^\pi(s,a)$  是状态行动的目标价值。如果函数  $f_w$  最终收敛，即下面的公式成立：

$$\sum_s d^\pi(s) \sum_a \pi(s,a) [\hat{Q}^\pi(s,a) - f_w(s,a)] \nabla_w f_w(s,a) = 0$$

那么，在论文 *Policy Gradient Methods for Reinforcement Learning with Function Approximation*<sup>[6]</sup> 中，作者证明了，当  $f_w$  满足上面的公式，同时满足和策略函数参数的某种一致性，即  $f_w$  关于参数的梯度等于策略的对数似然关于参数的梯度：

$$\nabla_w f_w(s,a) = \nabla_\theta \log \pi(s,a) = \nabla_\theta \pi(s,a) \frac{1}{\pi(s,a)}$$

那么，在计算梯度时我们就可以用  $f_w$  代替  $Q^\pi$ ，并得到相同的结果。具体的计算过程为

$$\begin{aligned} \nabla_\theta v^\pi(s) &= \sum_s d^\pi(s) \sum_a \nabla_\theta \pi(s,a) Q^\pi(s,a) - 0 \\ &= \sum_s d^\pi(s) \sum_a \nabla_\theta \pi(s,a) Q^\pi(s,a) \\ &\quad - \sum_s d^\pi(s) \sum_a \pi(s,a) [Q^\pi(s,a) - f_w(s,a)] \nabla_w f_w(s,a) \\ &= \sum_s d^\pi(s) \sum_a \nabla_\theta \pi(s,a) Q^\pi(s,a) \\ &\quad - \sum_s d^\pi(s) \sum_a \pi(s,a) [Q^\pi(s,a) - f_w(s,a)] \nabla_\theta \pi(s,a) \frac{1}{\pi(s,a)} \\ &= \sum_s d^\pi(s) \sum_a \nabla_\theta \pi(s,a) Q^\pi(s,a) \\ &\quad - \sum_s d^\pi(s) \sum_a \nabla_\theta \pi(s,a) [Q^\pi(s,a) - f_w(s,a)] \\ &= \sum_s d^\pi(s) \sum_a \nabla_\theta \pi(s,a) [Q^\pi(s,a) - Q^\pi(s,a) + f_w(s,a)] \\ &= \sum_s d^\pi(s) \sum_a \nabla_\theta \pi(s,a) f_w(s,a) \end{aligned}$$

上面定理实现的关键是参数的一致性，这一点容易做到吗？由于策略模型的计算结果是对数似然概率，所以这样的形式很容易被满足。我们令策略模型是一个最简单的线性模型，当模型经过 Softmax 层计算后，可以得到

$$\pi(s, a) = \frac{e^{\theta^T \phi_{s,a}}}{\sum_b e^{\theta^T \phi_{s,b}}}$$

对其取对数求导，可以得到

$$\begin{aligned}\nabla_w f_w(s, a) &= \nabla_\theta \log \pi(s, a) \\&= \nabla_\theta \log [\exp[\theta^T \phi_{s,a}] - \sum_b \exp[\theta^T \phi_{s,b}]] \\&= \nabla_\theta [\theta^T \phi_{s,a}] - \nabla_\theta \log [\sum_b \exp[\theta^T \phi_{s,b}]] \\&= \phi_{s,a} - \frac{1}{\sum_b \exp[\theta^T \phi_{s,b}]} \nabla_\theta [\sum_b \exp[\theta^T \phi_{s,b}]] \\&= \phi_{s,a} - \frac{1}{\sum_b \exp[\theta^T \phi_{s,b}]} \sum_b \exp[\theta^T \phi_{s,b}] \phi_{s,b} \\&= \phi_{s,a} - \sum_b \frac{\exp[\theta^T \phi_{s,b}]}{\sum_b \exp[\theta^T \phi_{s,b}]} \phi_{s,b} \\&= \phi_{s,a} - \sum_b \pi(s, b) \phi_{s,b} \\&= \phi_{s,a} - E_{a \sim \pi(s)} [\phi_{s,a}]\end{aligned}$$

上式的第二项是一个期望项，在实际中它是一个常量，我们可以将其忽略，只要将输入数据进行归一化即可抵消掉这一项。因此为了满足参数的一致性，我们可以将价值估计函数的形式设定为

$$f_w(s, a) = w^T \hat{\phi}_{s,a}$$

实际上，神经网络同样可以以这样的形式表示出来。因此，采用神经网络同样可以达到这样的一致性效果。读者看到这里一定充满疑惑，这个证明看上去有些无聊，那么它的意义何在呢？对于 DPG，作者在论文中给出了类似的证明与结论。

给定一个值函数  $Q^w(s, a)$  与确定的策略函数  $\mu_\theta(s)$ ，当下面两个条件满足时：

$$(1) \nabla_a Q^w(s, a)|_{a=\mu_\theta(s)} = \nabla_\theta \mu_\theta(s)^T w$$

(2) 参数  $w$  是近似值函数的局部最优值，它使得下面的目标函数取得最小值：

$$\text{MSE}(\theta, w) = E[\epsilon(s; \theta, w)^T \epsilon(s; \theta, w)]$$

其中

$$\epsilon(s; \theta, w) = \nabla_a Q^w(s, a)|_{a=\mu_\theta(s)} - \nabla_a Q^\mu(s, a)|_{a=\mu_\theta(s)}$$

我们可以得到下面的定理，也就是说可以用值函数近似真实的价值并得到正确的策略梯度：

$$\nabla_\theta J_\beta(\theta) = E[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s, a)|_{a=\mu_\theta(s)}] = E[\nabla_\theta \mu_\theta(s) \nabla_a Q^w(s, a)|_{a=\mu_\theta(s)}]$$

我们可以简单证明一下，根据前面的一致性定义，可以得到

$$\nabla_w \epsilon(s; \theta, w) = \nabla_\theta \mu_\theta(s)$$

继续展开前面的公式可以得到当  $w$  等于局部最优时，目标函数的导数为 0：

$$\nabla_w \text{MSE}(\theta, w) = E[\nabla_w [\epsilon(s; \theta, w)^T \epsilon(s; \theta, w)]] = 0$$

接着展开可以得到

$$E[2\epsilon(s; \theta, w) \nabla_w \epsilon(s; \theta, w)] = 0$$

$$E[\epsilon(s; \theta, w) \nabla_\theta \mu_\theta(s)] = 0$$

$$E[(\nabla_a Q^w(s, a)|_{a=\mu_\theta(s)} - \nabla_a Q^\mu(s, a)|_{a=\mu_\theta(s)}) \nabla_\theta \mu_\theta(s)] = 0$$

$$E[\nabla_\theta \mu_\theta(s) \nabla_a Q^w(s, a)|_{a=\mu_\theta(s)}] = E[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s, a)|_{a=\mu_\theta(s)}] = \nabla_\theta J_\beta(\theta)$$

于是得证。实际上通过前面大段的证明，我们可以得出一个结论：对于目标函数中的价值估计部分，我们可以使用一个值函数模型进行拟合，这样价值模型不需要遵从某个策略。这样我们就可以使用 Off-Policy 的方法进行计算了。

### 11.3.3 DDPG 算法

接下来介绍 DPG 的实现。实际上，DPG 的思想仅仅是使用确定的策略输出，算法并不包含其他的限定，例如是否为 On-Policy 或者 Off-Policy。对于 On-Policy 的 Deterministic Actor-Critic 算法，值函数为  $Q^w(s, a)$ ，确定策略为  $\mu_\theta(s)$ ，我们可以建立如下目

标函数：

$$J(w) = \text{minimize}_w E_\pi \left[ \frac{1}{2} (r_t + \gamma Q^w(s_{t+1}, a_{t+1}) - Q^w(s_t, a_t))^2 \right]$$

$$J(\theta) = \text{maximize}_\theta E_\pi [Q^w(s_t, \mu_\theta(s_t))]$$

对其进行求解，可以得到

$$\begin{aligned}\Delta w &= \alpha E_\pi [(r_t + \gamma Q^w(s_{t+1}, a_{t+1}) - Q^w(s_t, a_t)) \nabla_w Q^w(s_t, a_t)] \\ \Delta \theta &= \alpha E_\pi [\nabla_w Q^w(s_t, a_t)|_{a=\mu_\theta(s)} \nabla_\theta \mu_\theta(s_t)]\end{aligned}$$

对于 Off-Policy 的算法，我们同样可以建立目标函数。由于我们使用了确定的策略，同时值函数不依赖任何策略，那么在计算时我们就不需要向随机策略那样进行重要性采样计算。假设样本来自策略  $\beta$ ，我们的目标函数为

$$J(w) = \text{minimize}_w E_\beta \left[ \frac{1}{2} (r_t + \gamma Q^w(s_{t+1}, a_{t+1}) - Q^w(s_t, a_t))^2 \right]$$

$$J(\theta) = \text{maximize}_\theta E_\beta [Q^w(s_t, \mu_\theta(s_t))]$$

对其进行求解，可以得到类似的结果：

$$\begin{aligned}\Delta w &= \alpha E_\beta [(r_t + \gamma Q^w(s_{t+1}, a_{t+1}) - Q^w(s_t, a_t)) \nabla_w Q^w(s_t, a_t)] \\ \Delta \theta &= \alpha E_\beta [\nabla_w Q^w(s_t, a_t)|_{a=\mu_\theta(s)} \nabla_\theta \mu_\theta(s_t)]\end{aligned}$$

由于本章的主题是介绍 Off-Policy 的算法，因此接下来我们只关心这部分的算法。虽然 DPG 从名字上看属于策略梯度法的一种，但是它的求解过程有很多与 DQN 相近的地方，正如论文 *Continuous control with deep reinforcement learning*<sup>[6]</sup> 中介绍的，DQN 的优化过程中的两个常见的优化方法也被应用到了 Deep DPG 上。

## 1. Replay-Buffer

由于连续的序列中存在的相关性会使 DQN 的优化充满不稳定性，因此 DQN 采用了 Replay-Buffer，将一些采样样本收集起来，每次优化时从中随机取出一部分进行优化，就可以减少一些不稳定性。这个方法同样适用于 DPG。

## 2. Target Network

这也是 DQN 中常见的方法，通过设置一个不会频繁或大幅更新的模型，使模型计算的值函数在一定程度上减少波动，令计算更稳定。在论文中，作者介绍采用滑动平均的方法更新 Target Network:  $\theta_{t+1} \leftarrow \tau\theta_t + (1 - \tau)\theta'_t$ ,  $\tau$  一般设置为非常接近 1 的数，这样 Target 网络的参数  $\theta$  不会发生太大的变化，每次只会受一点训练模型  $\theta'$  的影响。这个更新方法和 11.2 节的 ACER 算法十分相似。

除了这两个方法外，作者还对模型的探索做了一些改变。由于 DPG 采用确定策略，如果它在与环境进行交互时只采用确定的策略，那么必然会导致对环境的探索不够充分，因此需要为策略增加一定的探索性。前面已经提到 DPG 可以使用基于  $\epsilon$ -greedy 的探索方法，通过为确定的行动增加噪声提高探索性。在 DDPG 模型中，作者采用 Ornstein-Uhlenbeck 噪声增加模型的探索能力。Ornstein-Uhlenbeck 噪声是一种基于 Ornstein-Uhlenbeck 过程的随机变量，它可以被用于模拟与时间有关联的噪声数据。它的生成公式为

$$dx_t = \theta(\mu - x_t) + \sigma W_t$$

其中  $x$  是要生成的数据， $\mu$  是设定的随机变量的期望值， $W$  是一个由 Wiener 过程生成的随机变量，一般我们用一个简单的随机函数代替就可以， $\theta$  和  $\sigma$  是随机过程中的参数， $dx$  是每一时刻数据的变化量，真正的采样值等于上一时刻的采样值加上求出的变化量。从公式中可以看出，每一时刻数据的变化量和当前时刻存在关联，公式右边的第一项将为随机数据提供朝向均值  $\mu$  的变化，第二项才是常见的随机变化。

为了更直观地理解这个噪声生成器，我们举一个例子。设定  $\theta = 0.01$ ,  $\mu = 0$ ,  $\sigma = 0.02$ ,  $W$  由均值为 0, 方差为 1 的高斯随机函数生成。我们设定三个随机变量，它们的初始值为 2、0 和 -2，对其进行一定时间的采样，得到图 11-3 所示的结果。

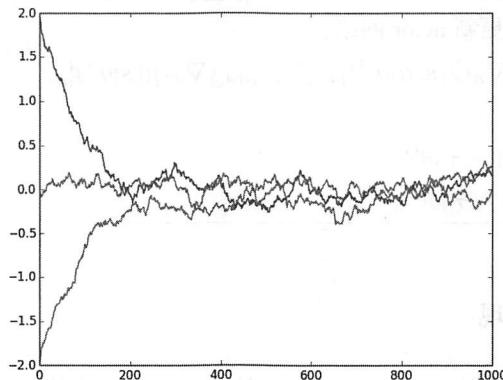


图 11-3 Ornstein-Uhlenbeck 的采样结果图

从结果可以看出，三个随机变量在波动中靠近了均值 0。这就可以确保噪声的增加不会对策略产生太大的影响，一旦噪声在某个维度造成了影响，一定会尽快给予补偿。

除了上面提到的一些改进方法，我们还可以从前面章节的内容中找到更多改进模型的灵感。第 8 章介绍了许多改进 DQN 的方法，这些方法都可以以某种形式应用到 Deep DPG 中。例如，Priority Replay Buffer、Pre-train from Demonstration 等。读者可以阅读论文 *Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards*，进一步了解关于这部分的改进过程。

完整的 DDPG 算法如下所示。

---

### 算法 DDPG 算法

---

初始化 critic 网络  $Q(s, a | \theta^Q)$ , actor 网络  $\mu(s | \theta^\mu)$

用上面的两个网络初始化对应的目标网络  $Q' \leftarrow Q$ ,  $\mu' \leftarrow \mu$

初始化 Replay Buffer:  $R$

**for** episode = 1,  $M$  **do**

    初始化噪声分布  $N$

$s_1 = \text{env.reset}()$

**for**  $t = 1, T$  **do**

$a_t = \mu(s_t | \theta^\mu) + N_t$

$s_{t+1}, \text{reward}_t, \text{terminate}_t = \text{env.step}(a_t)$

$R.\text{save}((s_t, a_t, r_t, s_{t+1}))$

        // 训练

$(s_i, a_i, r_i, s_{i+1}) = R.\text{sample}(N)$

$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1} | \theta^{\mu'}) | \theta^{Q'})$

        根据 critic loss 更新 critic 网络:  $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i | \theta^Q))^2$

        根据 actor 的梯度更新 actor 网络:

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

        更新目标网络:

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

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

---

#### 11.3.4 DDPG 的实现

在 Baselines 项目中的 baselines/ddpg 文件夹中，可以看到 DDPG 的实现。文件夹中各文件的含义如下所示。

- ddpg.py: DDPG 模型的实现。
- main.py: 启动的主程序。
- memory.py: Replay Buffer 的实现。
- models.py: Actor 和 Critic 的实现。
- noise.py: 随机噪声的实现。
- training.py: 训练过程的实现。

我们可以从一些独立的部分入手进行分析，首先是 noise.py，其中介绍了几种噪声生成方法。

- NormalActionNoise: 从高斯分布  $N(\mu, \sigma)$  中采样噪声。
- AdaptiveNoise: 自适应方差的噪声，可以根据随机的效果调整采样的方差。
- OrnsteinUhlenbeckNoise: 前面提到的噪声实现。

memory.py 中包含 Replay Buffer 的实现，这里实现了最基本的 Replay Buffer。在阅读了前面几章的代码后，我们对这部分的代码已经很熟悉了。

在 models.py 中介绍了两个模型，由于我们解决的问题是连续的状态和行动空间，不像前面介绍的 Atari 游戏那样以画面为状态输入，因此模型只需要全连接层就可以。两个模型都采用 3 层全连接层实现，其中 Actor 模型的输出和行动的维度相同，Critic 模型的输出维度为 1。

ddpg.py 中介绍了训练的主要内容。由于我们采用了回归模型进行模型训练，数值的范围变得不太可控，为了让训练过程变得更稳定、更快捷，需要对输入和输出进行归一化。对于输入的观测值，我们首先经过一个滑动平均的结构记录输入值的均值和方差，然后在输入值进入模型前对其进行归一化，也就是减去它的均值和方差。同样，对于输出值，我们得到的也是被归一化之后的数值，在模型输出后，这个数值再被恢复到原来的数值，这个过程如图 11-4 所示。

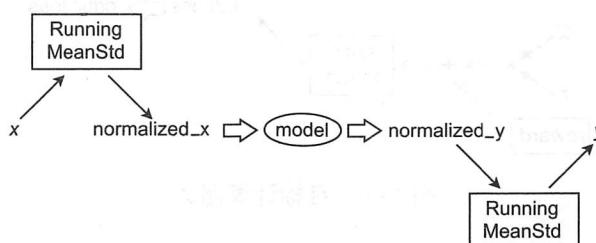


图 11-4 输入和输出的归一化与恢复

模型在训练时需要从 Replay Buffer 中读取数据，我们首先使用 Q-Learning 计算更新后的价值。我们从数据中找到  $t + 1$  时刻的状态  $obs1$ 、回报  $reward$  和状态是否为终结状态的标识符  $terminate$ ，使用 Target Network 中的  $target\_actor$  和  $target\_critic$  两个模型就可以计算得到：

```
target_Q = reward + (1-terminate)* gamma * target_critic(obs1,target_actor(obs1))
```

得到它之后，就可以计算价值模型的目标了。我们再使用  $t$  时刻的状态  $obs0$ ，就可以得到：

```
critic_loss = sqrt(square(target_Q - critic(obs0, action)))
```

我们的目标就是最小化上面的公式。上面的公式里我们忽略了归一化相关的操作，在代码中我们会看到这部分的内容。而对策略模型，我们的目标为：

```
actor_loss = -critic(obs0,actor(obs0))
```

只要对这些目标进行求导就可以了，这部分的计算如图 11-5 和图 11-6 所示。

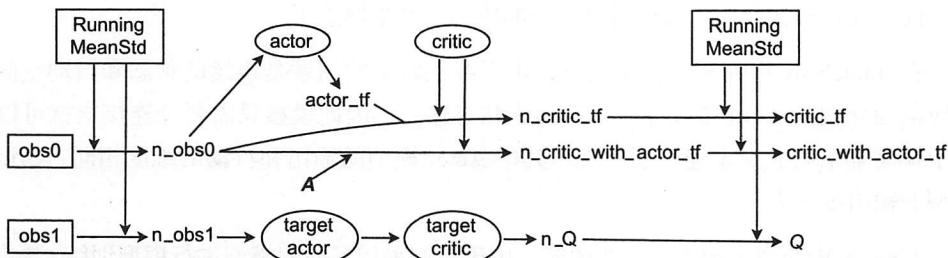


图 11-5 目标计算图 1

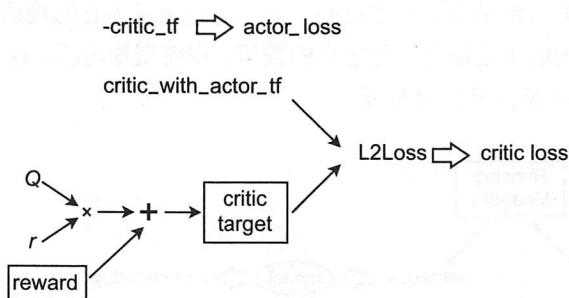


图 11-6 目标计算图 2

代码中使用了 MPI 进行梯度的通信，这样增强了模型的并发度。以上就是代码的核心部分，对具体内容感兴趣的读者可以阅读其中的细节。

## 11.4 总结

本章我们主要介绍了基于 Off-Policy 的 Actor Critic 算法，在此我们总结如下。

(1) ACER 算法继承了 Off-Policy Actor Critic 的算法，通过对重要性采样比例的限制与偏差纠正，减少了因重要性采样造成的优化波动，同时使用 Retrace 方法减少价值估计的波动。

(2) DDPG 算法吸收了 DQN 的思想，采用确定的策略函数，使问题在高维度的连续空间上能够发挥更好的效果。同时，DDPG 算法吸收了大量 DQN 的改进方案，使模型在训练频率和效果上有了保证。

## 11.5 参考资料

- [1] Munos R, Stepleton T, Harutyunyan A, et al. Safe and Efficient Off-Policy Reinforcement Learning[J]. 2016.
- [2] Degris T, White M, Sutton R S. Off-Policy Actor-Critic[J]. 2012.
- [3] Wang Z, Bapst V, Heess N, et al. Sample Efficient Actor-Critic with Experience Replay[J]. 2016.
- [4] Silver D, Lever G, Heess N, et al. Deterministic policy gradient algorithms[C]// International Conference on International Conference on Machine Learning. JMLR.org, 2014:387-395.
- [5] Lillicrap T P, Hunt J J, Pritzel A, et al. Continuous control with deep reinforcement learning[J]. Computer Science, 2015, 8(6):A187.
- [6] Večerík M, Hester T, Scholz J, et al. Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards[J]. 2017.