5 策略梯度

策略梯度方法（Policy Gradient Methods）是一类直接针对期望回报（Expected Return）通过梯度下降（Gradient Descent）进行策略优化的增强学习方法。这一类方法避免了其他传统增强学习方法所面临的一些困难，比如，没有一个准确的价值函数，或者由于连续的状态和动作空间，以及状态信息的不确定性而导致的难解性（Intractability）。在这一章中，我们会学习一系列策略梯度方法。从最基本的 REINFORCE 开始，我们会逐步介绍 Actor-Critic 方法及其分布式计算的版本、信赖域策略优化（Trust Region Policy Optimization）及其近似算法，等等。在本章最后一节，我们附上了本章涉及的所有方法所对应的伪代码，以及一个具体的实现例子。

5.1 简介

这一章主要介绍策略梯度方法。和上一章介绍的学习 Q 值函数的深度 Q-Learning 方法不同，策略梯度方法直接学习参数化的策略 π_θ。这样做的一个好处是不需要在动作空间中求解价值最大化的优化问题，从而比较适合解决具有高维或者连续动作空间的问题。策略梯度方法的另一个好处是可以很自然地对随机策略进行建模[1]。最后，策略梯度方法利用了梯度的信息来引导优化的过程。一般来讲，这样的方法有更好的收敛性保证[2]。

顾名思义，策略梯度方法通过梯度上升的方法直接在神经网络的参数上优化智能体的策略。在这一章中，我们会在 5.2 节中推导出策略梯度的初始版本算法。这个算法一般会有估计方差过高的问题。我们在 5.3 节会看到 Actor-Critic 算法可以有效地减轻这个问题。有趣的是，Actor-Critic

[1]在价值学习的设定下，智能体需要额外构造它的探索策略，比如 ϵ-贪心，以对随机性策略进行建模。

[2]但一般也仅限于局部收敛性，而不是全局收敛性。近期的一些研究在策略梯度的全局收敛性上有一些进展，但本章不讨论这一方面的工作。

和 GAN 的设计非常相像。我们会在 5.4 节比较它们的相似之处。在 5.5节、5.6 节中，我们会接着介绍 Actor-Critic 的分布式版本。最后，我们通过考虑在策略空间（而不是参数空间）中的梯度上升进一步提高策略梯度方法的性能。一个被广泛使用的方法是信赖域策略优化（Trust Region Policy Optimization，TRPO），我们会在 5.7 节和 5.8 节介绍它及其近似版本，即近端策略优化算法（Proximal Policy Optimization，PPO），以及在 5.9 节中介绍使用 Kronecker 因子化信赖域的 Actor Critic（Actor Critic using Kronecker-factored Trust Region，ACKTR）。

在本章的最后一节，即 5.10 节中，我们提供了所涉及算法的代码实现，以方便读者可以迅速上手试验。每个算法的完整实现可以在本书的代码库找到[3]。

5.2 REINFORCE：初版策略梯度

REINFORCE 算法在策略的参数空间中直观地通过梯度上升的方法逐步提高策略 π_θ 的性能。回顾一下，由式子 (2.119) 我们有

$$\nabla_\theta J(\pi_\theta)=\mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\mathrm{T}} R_t\nabla_\theta\sum_{t'=0}^{t}\log\pi_\theta(A_{t'}|S_{t'})\right]=\mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t'=0}^{\mathrm{T}}\nabla_\theta\log\pi_\theta(A_{t'}|S_{t'})\sum_{t=t'}^{\mathrm{T}} R_t\right]. \quad (5.1)$$

注 5.1 上述式子中 $\sum_{t=i}^{\mathrm{T}} R_t$ 可以看成是智能体在状态 S_i 处选择动作 A_i，并在之后执行当前策略的情况下，从第 i 步开始获得的累计奖励。事实上，$\sum_{t=i}^{\mathrm{T}} R_t$ 也可以看成 $Q_i(A_i,S_i)$，在第 i 步状态 S_i 处采取动作 A_i，并在之后执行当前策略的 Q 值。所以，一个理解 REINFORCE 的角度是：通过给不同的动作所对应的梯度根据它们的累计奖励赋予不同的权重，鼓励智能体选择那些累计奖励较高的动作 A_i。

只要把上述式子中的 T 替换成 ∞ 并赋予 R_t 以 γ^t 的权重，上述式子很容易可以扩展到折扣因子为 γ 的无限范围的设定如下。

$$\nabla J(\theta)=\mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t'=0}^{\infty}\nabla_\theta\log\pi_\theta(A_{t'}|S_{t'})\gamma^{t'}\sum_{t=t'}^{\infty}\gamma^{t-t'}R_t\right]. \quad (5.2)$$

由于折扣因子给未来的奖励赋予了较低的权重，使用折扣因子还有助于减少估计梯度时的方差大的问题。实际使用中，$\gamma^{t'}$ 经常被去掉，从而避免了过分强调轨迹早期状态的问题。

虽然 REINFORCE 简单直观，但它的一个缺点是对梯度的估计有较大的方差。对于一个长度为 L 的轨迹，奖励 R_t 的随机性可能对 L 呈指数级增长。为了减轻估计的方差太大这个问题，一个常用的方法是引进一个基准函数 $b(S_i)$。这里对 $b(S_i)$ 的要求是：它只能是一个关于状态 S_i 的函数（或者更确切地说，它不能是关于 A_i 的函数）。

[3]链接见读者服务

有了基准函数 $b(S_t)$ 之后，增强学习目标函数的梯度 $\nabla J(\theta)$ 可以表示成

$$\nabla J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t'=0}^{\infty}\nabla_\theta \log \pi_\theta(A_{t'}|\,S_{t'})\left(\sum_{t=t'}^{\infty}\gamma^{t-t'}R_t - b(S_{t'})\right)\right]. \tag{5.3}$$

这是因为

$$\mathbb{E}_{\tau,\theta}\left[\nabla_\theta \log \pi_\theta(A_{t'}|\,S_{t'})b(S_{t'})\right] = \mathbb{E}_{\tau,\theta}\left[b(S_{t'})\mathbb{E}_\theta\left[\nabla \log \pi_\theta(A_{t'}|\,S_{t'})|\,S_{t'}\right]\right] = 0. \tag{5.4}$$

上述式子的最后一个等式可以由 EGLP 引理（引理 2.2）得到。最后如算法 5.18 所示，我们得到带有基准函数的 REINFORCE 算法。

算法 5.18 带基准函数的 REINFORCE 算法

超参数: 步长 η_θ、奖励折扣因子 γ、总步数 L、批尺寸 B、基准函数 b。
输入: 初始策略参数 θ_0
初始化 $\theta = \theta_0$
for $k = 1, 2, \cdots$, **do**
　执行策略 π_θ 得到 B 个轨迹，每一个有 L 步，并收集 $\{S_{t,\ell}, A_{t,\ell}, R_{t,\ell}\}$。
　$\hat{A}_{t,\ell} = \sum_{\ell'=\ell}^{L}\gamma^{\ell'-\ell}R_{t,\ell} - b(S_{t,\ell})$
　$J(\theta) = \frac{1}{B}\sum_{t=1}^{B}\sum_{\ell=0}^{L}\log \pi_\theta(A_{t,\ell}|S_{t,\ell})\hat{A}_{t,\ell}$
　$\theta = \theta + \eta_\theta \nabla J(\theta)$
　用 $\{S_{t,\ell}, A_{t,\ell}, R_{t,\ell}\}$ 更新 $b(S_{t,\ell})$
end for
返回 θ

直观来讲，从奖励函数中减去一个基准函数这个方法是一个常见的降低方差的方法。假设需要估计一个随机变量 X 的期望 $\mathbb{E}[X]$。对于任意一个期望为 0 的随机变量 Y，我们知道 $X - Y$ 依然是 $\mathbb{E}[X]$ 的一个无偏估计。而且，$X - Y$ 的方差为

$$\mathbb{V}(X - Y) = \mathbb{V}(X) + \mathbb{V}(Y) - 2\text{cov}(X, Y). \tag{5.5}$$

式子中的 $\mathbb{V}$ 表示方差，$\text{cov}(X, Y)$ 表示 X 和 Y 的协方差。所以如果 Y 本身的方差较小，而且和 X 高度正相关，那么 $X - Y$ 会是一个方差较小的关于 $\mathbb{E}[X]$ 的无偏估计。在策略梯度方法中，基准函数的常见选择是状态价值函数 $V(S_i)$。在下一节中我们可以看到，这个算法和初版的 Actor-Critic 算法很相像。最近的一些研究工作也提出了其他不同的基准函数的选择，感兴趣的读者可以从文献 (Li et al., 2018; Liu et al., 2017; Wu et al., 2018) 中了解更多的细节。

5.3 Actor-Critic

Actor-Critic 算法 (Konda et al., 2000; Sutton et al., 2000) 是一个既基于策略也基于价值的方法。在上一节我们提到，在初版策略梯度方法中可以用状态价值函数作为基准函数来降低梯度估计的方差。Actor-Critic 算法也沿用了相同的想法，同时学习行动者（Actor）函数（也就是智能体的策略函数 $\pi(\cdot|s)$）和批判者（Critic）函数（也就是状态价值函数 $V^\pi(s)$）。此外，Actor-Critic 算法还沿用了自举法（Bootstrapping）的思想来估计 Q 值函数。REINFORCE 中的误差项 $\sum_{t=i}^{\infty}\gamma^{t-i}R_t - b(S_i)$ 被时间差分误差取代了，即 $R_i + \gamma V^\pi(S_{i+1}) - V^\pi(S_i)$。

我们这里采用 L 步的时间差分误差，并通过最小化该误差的平方来学习批判者函数 $V_\psi^{\pi_\theta}(s)$，即

$$\psi \leftarrow \psi - \eta_\psi \nabla J_{V_\psi^{\pi_\theta}}(\psi). \tag{5.6}$$

式子中 ψ 表示学习批判者函数的参数，η_ψ 是学习步长，并且

$$J_{V_\psi^{\pi_\theta}}(\psi) = \frac{1}{2}\left(\sum_{t=i}^{i+L-1}\gamma^{t-i}R_t + \gamma^L V_\psi^{\pi_\theta}(S') - V_\psi^{\pi_\theta}(S_i)\right)^2, \tag{5.7}$$

S' 是智能体在 π_θ 下 L 步之后到达的状态，所以

$$\nabla J_{V_\psi^{\pi_\theta}}(\psi) = \left(V_\psi^{\pi_\theta}(S_i) - \sum_{t=i}^{i+L-1}\gamma^{t-i}R_t - \gamma^L V_\psi^{\pi_\theta}(S')\right)\nabla V_\psi^{\pi_\theta}(S_i). \tag{5.8}$$

类似地，行动者函数 $\pi_\theta(\cdot|s)$ 决定每个状态 s 上所采取的动作或者动作空间上的一个概率分布。我们采用和初版策略梯度相似的方法来学习这个策略函数。

$$\theta = \theta + \eta_\theta \nabla J_{\pi_\theta}(\theta), \tag{5.9}$$

这里 θ 表示行动者函数的参数，η_θ 是学习步长，并且

$$\nabla J(\theta) = \mathbb{E}_{\tau,\theta}\left[\sum_{i=0}^{\infty}\nabla\log\pi_\theta(A_i|\,S_i)\left(\sum_{t=i}^{i+L-1}\gamma^{t-i}R_t + \gamma^L V_\psi^{\pi_\theta}(S') - V_\psi^{\pi_\theta}(S_i)\right)\right]. \tag{5.10}$$

注意到，我们这里分别用了 θ 和 ψ 来表示策略函数和价值函数的参数。在实际应用中，当我们选择用神经网络来表示这两个函数的时候，经常会让两个网络共享一些底层的网络层作为共同的状态表征（State Representation）。此外，AC 算法中的 L 值经常设为 1, 也就是 TD(0) 误差。AC

算法的具体步骤如算法 5.19 所示。

算法 5.19 Actor-Critic 算法

超参数: 步长 η_θ 和 η_ψ, 奖励折扣因子 γ。
输入: 初始策略函数参数 θ_0、初始价值函数参数 ψ_0。
初始化 $\theta = \theta_0$ 和 $\psi = \psi_0$。
for $t = 0, 1, 2, \cdots$ **do**
　执行一步策略 π_θ, 保存 $\{S_t, A_t, R_t, S_{t+1}\}$。
　估计优势函数 $\hat{A}_t = R_t + \gamma V_\psi^{\pi_\theta}(S_{t+1}) - V_\psi^{\pi_\theta}(S_t)$。
　$J(\theta) = \sum_t \log \pi_\theta(A_t|S_t)\hat{A}_t$
　$J_{V_\psi^{\pi_\theta}}(\psi) = \sum_t \hat{A}_t^2$
　$\psi = \psi + \eta_\psi \nabla J_{V_\psi^{\pi_\theta}}(\psi), \theta = \theta + \eta_\theta \nabla J(\theta)$
end for
返回 (θ, ψ)

值得注意的是，AC 算法也可以使用 Q 值函数作为其批判者。在这种情况下，优势函数可以用以下式子估计。

$$Q(s,a) - V(s) = Q(s,a) - \sum_a \pi(a|s)Q(s,a). \tag{5.11}$$

用来学习 Q 值函数这个批判者的损失函数为

$$J_Q = \big(R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big)^2, \tag{5.12}$$

或者

$$J_Q = \left(R_t + \gamma \sum_a \pi_\theta(a|S_{t+1})Q(S_{t+1}, a) - Q(S_t, A_t)\right)^2. \tag{5.13}$$

这里动作 A_{t+1} 由当前策略 π_θ 在状态 S_{t+1} 下取样而得。

5.4　生成对抗网络和 Actor-Critic

初看上去，生成对抗网络（Generative Adversarial Networks，GAN）(Goodfellow et al., 2014) 和 Actor-Critic 应该是截然不同的算法，用于不同的机器学习领域，一个是生成模型，而另一个是强化学习算法。但是实际上它们的结构十分类似。对于 GAN，有两个部分：用于根据某些输入生成对象的生成网络，以及紧接生成网络的用于判断生成对象真实与否的判别网络。对于 Actor-Critic 方法，也有两部分：根据状态输入生成动作的动作网络，以及一个紧接动作网络之后用价值函数

（比如下一个动作的价值或 Q 值）评估动作好坏的批判网络。

因此，GAN 和 Actor-Critic 基本遵循相同的结构。在这个结构中有两个相继的部分：一个用于生成物体，第二个用一个分数来评估生成物体的好坏；随后选择一个优化过程来使第二部分能够准确评估，并通过第二部分反向传播梯度到第一部分来保证它生成我们想要的内容，通过一个定义为损失函数的标准，也就是一个来自结构第二部分的分数或价值函数来实现。

GAN 和 Actor-Critic 的结构详细比较如图 5.1 所示。

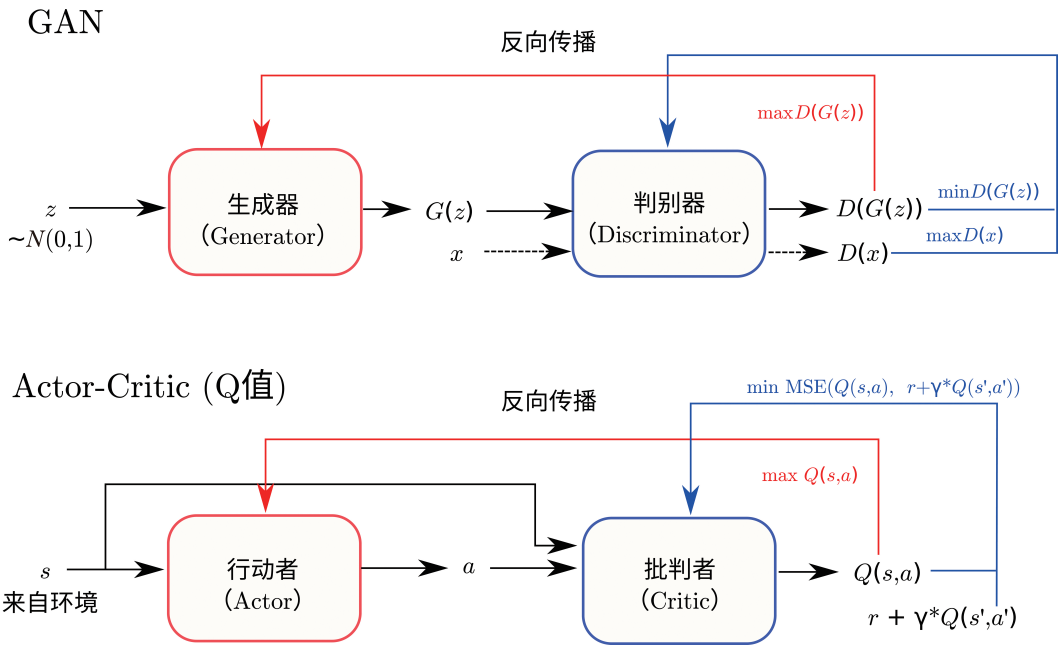

图 5.1 对比 GAN 和 Actor-Critic 的结构。在 GAN 中，z 是输入的噪声变量，它可以从如正态分布中采样，而 x 是从真实目标中采集的数据样本。在 Actor-Critic 中，s 和 a 分别表示状态和动作

- 对第一个生成物体的部分：GAN 中的生成器和 Actor-Critic 中的行动者基本一致，包括其前向推理过程和反向梯度优化过程。对于前向过程，生成器采用随机变量做输入，并输出生成的对象；对于方向优化过程，它的目标是最大化对生成对象的判别分数。行动者用状态作为输入并输出动作，对于优化来说，它的目标是最大化状态-动作对的评估值。
- 对于第二个评估物体的部分：判别器和批判者由于其功能不同而优化公式也不同，但是遵循相同的目标。判别器有来自真实对象额外输入。它的优化规则是最大化真实对象的判别值而最小化生成对象的判别值，这与我们的需要相符。对于批判者，它使用时间差分（Temporal Difference，TD）误差作为强化学习中的一种自举方法来按照最优贝尔曼方程优化价值函数。

也有一些其他模型彼此非常接近。举例来说，自动编码器（Auto-Encoder，AE）和 GAN 可

以是彼此的相反结构等。注意到，不同深度学习框架中的相似性可以帮助你获取关于现有不同领域方法共性的认识，而这有助于为未解决的问题提出新的方法。

5.5　同步优势 Actor-Critic

同步优势 Actor-Critic（Synchronous Advantage Actor-Critic，A2C）(Mnih et al., 2016) 和上一节讨论的 Actor-Critic 算法非常相似，只是在 Actor-Critic 算法的基础上增加了并行计算的设计。

如图 5.2 所示，全局行动者和全局批判者在 Master 节点维护。每个 Worker 节点的增强学习智能体通过协调器和全局行动者、全局批判者对话。在这个设计中，协调器负责收集各个 Worker 节点上与环境交互的经验（Experience），然后根据收集到的轨迹执行一步更新。更新之后，全局行动者被同步到各个 Worker 上继续和环境交互。在 Master 节点上，全局行动者和全局批判者的学习方法和 Actor-Critic 算法中行动者和批判者的学习方法一致，都是使用 TD 平方误差作为批判者的损失函数，以及 TD 误差的策略梯度来更新行动者的。

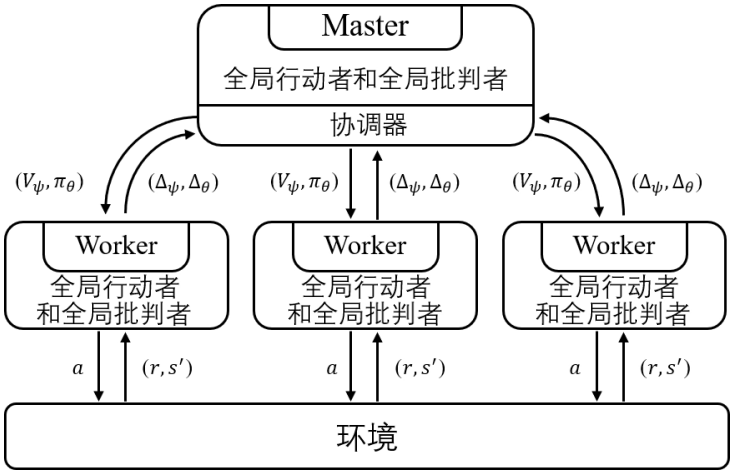

图 5.2　A2C 基本框架

在这种设计下，Worker 节点只负责和环境交互。所有的计算和更新都发生在 Master 节点。实际应用中，如果希望降低 Master 节点的计算负担，一些计算也可以转交给 Worker 节点[4]，比如说，每个 Worker 节点保存了当前全局批判者（Critic）。收集了一个轨迹之后，Worker 节点直接在本地计算给出全局行动者（Actor）和全局批判者的梯度。这些梯度信息继而被传送回 Master 节点。最后，协调器负责收集和汇总从各个 Worker 节点收集到的梯度信息，并更新全局模型。同样地，更新后的全局行动者和全局批判者被同步到各个 Worker 节点。A2C 算法的基本框架如算法 5.20 所示。

[4]这经常取决于每个 Worker 节点的计算能力，比如是否有 GPU 计算能力，等等。

算法 5.20 A2C

Master:
超参数: 步长 η_ψ 和 η_θ, Worker 节点集 $\mathcal{W}$。
输入: 初始策略函数参数 θ_0、初始价值函数参数 ψ_0。
初始化 $\theta = \theta_0$ 和 $\psi = \psi_0$
for $k = 0, 1, 2, \cdots$ **do**
  $(g_\psi, g_\theta) = 0$
  for $\mathcal{W}$ 里每一个 Worker 节点 **do**
    $(g_\psi, g_\theta) = (g_\psi, g_\theta) + \textbf{worker}(V_\psi^{\pi_\theta}, \pi_\theta)$
  end for
  $\psi = \psi - \eta_\psi g_\psi$; $\theta = \theta + \eta_\theta g_\theta$。
end for

Worker:
超参数: 奖励折扣因子 γ、轨迹长度 L。
输入: 价值函数 $V_\psi^{\pi_\theta}$、策略函数 π_θ。
执行 L 步策略 π_θ, 保存 $\{S_t, A_t, R_t, S_{t+1}\}$。
估计优势函数 $\hat{A}_t = R_t + \gamma V_\psi^{\pi_\theta}(S_{t+1}) - V_\psi^{\pi_\theta}(S_t)$。
$J(\theta) = \sum_t \log \pi_\theta(A_t|S_t)\hat{A}_t$
$J_{V_\psi^{\pi_\theta}}(\psi) = \sum_t \hat{A}_t^2$
$(g_\psi, g_\theta) = (\nabla J_{V_\psi^{\pi_\theta}}(\psi), \nabla J(\theta))$
返回 (g_ψ, g_θ)

5.6 异步优势 Actor-Critic

异步优势 Actor-Critic（Asynchronous Advantage Actor-Critic, A3C）(Mnih et al., 2016) 是上一节中 A2C 的异步版本。在 A3C 的设计中，协调器被移除。每个 Worker 节点直接和全局行动者和全局批判者进行对话。Master 节点则不再需要等待各个 Worker 节点提供的梯度信息，而是在每次有 Worker 节点结束梯度计算的时候直接更新全局 Actor-Critic。由于不再需要等待，A3C 有比 A2C 更高的计算效率。但是同样也由于没有协调器协调各个 Worker 节点，Worker 节点提供梯度信息和全局 Actor-Critic 的一致性不再成立，即每次 Master 节点从 Worker 节点得到的梯度信息很可能不再是当前全局 Actor-Critic 的梯度信息。

注 5.2 虽然 A3C 为了计算效率而牺牲 Worker 节点和 Master 节点的一致性这一点看起来有些特殊，这种异步更新的方式在神经网络的更新中其实非常常见。近期的研究 (Mitliagkas et al., 2016) 还表明，异步更新不仅加速了学习，还自动为 SGD 产生了类似于动量（Momentum）的效果。

算法 5.21 A3C

Master:
超参数: 步长 η_ψ 和 η_θ、当前策略函数 π_θ、价值函数 $V_\psi^{\pi_\theta}$。
输入: 梯度 g_ψ, g_θ。
$\psi = \psi - \eta_\psi g_\psi; \theta = \theta + \eta_\theta g_\theta$。
返回 $(V_\psi^{\pi_\theta}, \pi_\theta)$

Worker:
超参数: 奖励折扣因子 γ、轨迹长度 L。
输入: 策略函数 π_θ、价值函数 $V_\psi^{\pi_\theta}$。
$(g_\theta, g_\psi) = (0, 0)$
for $k = 1, 2, \cdots,$ **do**
  $(\theta, \psi) =$ **Master**(g_θ, g_ψ)
  执行 L 步策略 π_θ, 保存 $\{S_t, A_t, R_t, S_{t+1}\}$。
  估计优势函数 $\hat{A}_t = R_t + \gamma V_\psi^{\pi_\theta}(S_{t+1}) - V_\psi^{\pi_\theta}(S_t)$
  $J(\theta) = \sum_t \log \pi_\theta(A_t|S_t)\hat{A}_t$
  $J_{V_\psi^{\pi_\theta}}(\psi) = \sum_t \hat{A}_t^2$
  $(g_\psi, g_\theta) = (\nabla J_{V_\psi^{\pi_\theta}}(\psi), \nabla J(\theta))$
end for

5.7　信赖域策略优化

截至目前，我们在本章中介绍了初版策略梯度方法及其并行计算版本。在异步 Actor-Critic 的策略梯度中，我们更新策略如下。

$$\theta = \theta + \eta_\theta \nabla J(\theta), \tag{5.14}$$

这里

$$\nabla J(\theta) = \mathbb{E}_{\tau,\theta}\left[\sum_{i=0}^{\infty} \nabla \log \pi_\theta(A_i|\, S_i) A^{\pi_\theta}(S_i, A_i)\right], \tag{5.15}$$

其中优势函数 $A^{\pi_\theta}(s, a)$ 定义为

$$A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V_\psi^{\pi_\theta}(s). \tag{5.16}$$

和标准的梯度下降算法一样，初版策略梯度方法也有步长不好确定的缺陷。梯度 $\nabla J(\theta)$ 本身只提供了在当前 θ 下局部的一阶信息而忽略了奖励函数定义的曲面的曲度。如果在高度弯曲的区域选择了较大的步长，那么学习算法的性能可能会突然大幅下降。相反地，如果选择的步长太

小，学习的过程可能会太保守，从而非常缓慢。更甚，策略梯度方法中的梯度 $\nabla J(\theta)$ 需要从基于当前策略 π_θ 收集的样本中估计。策略性能的突然下降或者提升太过缓慢，会反过来影响收集到的样本的质量，这让学习的性能对于步长的选择更敏感。

初版策略梯度方法的另一个局限是：它的更新是在参数空间，而不是策略空间中进行的。

$$\Pi = \{\pi | \, \pi \geqslant 0, \int \pi = 1\}. \tag{5.17}$$

因为相同的步长 η_θ 可能使策略 π_θ 在策略空间中有完全不一样幅度的更新，这使得步长 η_θ 在实际应用中更加难以选择。举个例子，考虑当前的策略 $\pi = (\sigma(\theta), 1 - \sigma(\theta))$ 的两种不同情况。这里 $\sigma(\theta)$ 是 Sigmod 函数。假设在第一种情况下，θ 被从 $\theta = 6$ 更新到了 $\theta = 3$。而在另一种情况中，θ 被从 $\theta = 1.5$ 更新到了 $\theta = -1.5$。两种情况 π_θ 在参数空间中的更新幅度都是 3。然而，在第一种情况下，π_θ 在策略空间中从几乎是 $\pi \approx (1.00, 0.00)$ 变成了 $\pi \approx (0.95, 0.05)$，而在另一种情况下，$\pi = (0.82, 0.18)$ 被更新到了 $\pi = (0.18, 0.82)$。虽然两者在参数空间中的更新幅度相同，但是在策略空间中的更新幅度却完全不同。

在本节中，我们会开发一个能更好处理步长的策略梯度算法。这个算法的思想基于信赖域的想法，所以被称为信赖域策略优化算法（Trust Region Policy Optimization，TRPO）(Schulman et al., 2015)。注意到，我们的目标是找到一个比原策略 π_θ 更好的策略 π'_θ。下述引理为 π_θ 和 π'_θ 的性能提供了一个很深刻的联系：从 π_θ 到 π'_θ 在性能上的提升，可以由 π_θ 的优势函数 $A^{\pi_\theta}(s, a)$ 来计算。文献 (Kakade et al., 2002) 让 θ' 表示 π'_θ 的参数。

引理 5.1

$$J(\theta') = J(\theta) + \mathbb{E}_{\tau \sim \pi'_\theta} \left[\sum_{t=0}^{\infty} \gamma^t A^{\pi_\theta}(S_t, A_t) \right]. \tag{5.18}$$

这里 $J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t) \right]$，$\tau$ 是由 π'_θ 产生的同状态动作轨迹。

所以，学习最优的策略 π_θ 等价于最大化以下这个目标

$$\mathbb{E}_{\tau \sim \pi'_\theta} \left[\sum_{t=0}^{\infty} \gamma^t A^{\pi_\theta}(S_t, A_t) \right]. \tag{5.19}$$

然而，上述式子其实难以直接优化，因为式子中的期望是在 π'_θ 上。基于此，TRPO 优化该式子的一个近似，我们用 $\mathcal{L}_{\pi_\theta}(\pi'_\theta)$ 表示，如下式。

$$\mathbb{E}_{\tau \sim \pi'_\theta} \left[\sum_{t=0}^{\infty} \gamma^t A^{\pi_\theta}(S_t, A_t) \right] \tag{5.20}$$

$$= \mathbb{E}_{s\sim\rho_{\pi'_\theta}(s)}\left[\mathbb{E}_{a\sim\pi'_\theta(a|\,s)}\left[A^{\pi_\theta}(s,a)|\,s\right]\right] \tag{5.21}$$

$$\approx \mathbb{E}_{s\sim\rho_{\pi_\theta}(s)}\left[\mathbb{E}_{a\sim\pi'_\theta(a|\,s)}\left[A^{\pi_\theta}(s,a)|\,s\right]\right] \tag{5.22}$$

$$= \mathbb{E}_{s\sim\rho_{\pi_\theta}(s)}\left[\mathbb{E}_{a\sim\pi_\theta(a|\,s)}\left[\frac{\pi'_\theta(a|\,s)}{\pi_\theta(a|\,s)}A^{\pi_\theta}(s,a)|\,s\right]\right] \tag{5.23}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t\frac{\pi'_\theta(A_t|\,S_t)}{\pi_\theta(A_t|\,S_t)}A^{\pi_\theta}(S_t,A_t)\right]. \tag{5.24}$$

用 $\mathcal{L}_{\pi_\theta}(\pi'_\theta)$ 表示上述等式的最后一个式子，即

$$\mathcal{L}_{\pi_\theta}(\pi'_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t\frac{\pi'_\theta(A_t|\,S_t)}{\pi_\theta(A_t|\,S_t)}A^{\pi_\theta}(S_t,A_t)\right].$$

在上面的式子中，我们直接用 $\rho_{\pi_\theta}(s)$ 来近似 $\rho_{\pi'_\theta}(s)$。这个近似虽然看似粗糙，但下面的定理在理论上证明了，当 π_θ 和 π'_θ 相似的时候，这个近似并不差。

定理 5.1 让 $D_{\mathrm{KL}}^{\max}(\pi_\theta\|\pi'_\theta) = \max_s D_{\mathrm{KL}}(\pi_\theta(\cdot|s)\|\pi'_\theta(\cdot|s))$, 那么

$$|J(\theta') - J(\theta) - \mathcal{L}_{\pi_\theta}(\pi'_\theta)| \leqslant CD_{\mathrm{KL}}^{\max}(\pi_\theta\|\pi'_\theta). \tag{5.25}$$

这里 C 是和 π'_θ 无关的常数。

因此，如果 $D_{\mathrm{KL}}^{\max}(\pi_\theta\|\pi'_\theta)$ 很小，那么 $\mathcal{L}_{\pi_\theta}(\pi'_\theta)$ 可以合理地被作为一个优化目标。这便是 TRPO 的想法。实际中，TRPO 试图在平均 KL 散度的约束下优化 $\mathcal{L}_{\pi_\theta}(\pi'_\theta)$，如下所示。

$$\begin{aligned}\max_{\pi'_\theta}\quad & \mathcal{L}_{\pi_\theta}(\pi'_\theta) \\ \text{s.t.}\quad & \mathbb{E}_{s\sim\rho_{\pi_\theta}}\left[D_{\mathrm{KL}}(\pi_\theta\|\pi'_\theta)\right] \leqslant \delta.\end{aligned} \tag{5.26}$$

我们进一步讨论如何解 TRPO 中的这个优化问题。这里我们利用目标函数的一阶近似和约束的二阶近似。事实上，$\mathcal{L}_{\pi_\theta}(\pi'_\theta)$ 在策略 π_θ 处的梯度和 Actor-Critic 中一样。

$$g = \nabla_\theta\mathcal{L}_{\pi_\theta}(\pi'_\theta)|_\theta = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t\frac{\nabla_\theta\pi'_\theta(A_t|\,S_t)}{\pi_\theta(A_t|\,S_t)}A^{\pi_\theta}(S_t,A_t)\right]\Bigg|_\theta \tag{5.27}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t\nabla_\theta\log\pi_\theta(A_t|\,S_t)\Big|_\theta A^{\pi_\theta}(S_t,A_t)\right]. \tag{5.28}$$

此外，让 H 表示 $\mathbb{E}_{s\sim\rho_{\pi_\theta}}\left[D_{\mathrm{KL}}(\pi_\theta\|\pi'_\theta)\right]$ 的 Hessian 矩阵，那么，TRPO 在当前的 π_θ 求解如下优化问题。

$$\begin{aligned}\boldsymbol{\theta}' = \arg\max_{\boldsymbol{\theta}'} \quad & \boldsymbol{g}^\top(\boldsymbol{\theta}' - \boldsymbol{\theta}) \\ \text{s.t.} \quad & (\boldsymbol{\theta}' - \boldsymbol{\theta})^\top \boldsymbol{H}(\boldsymbol{\theta}' - \boldsymbol{\theta}) \leqslant \delta.\end{aligned} \tag{5.29}$$

易见这个问题的解析解存在：

$$\boldsymbol{\theta}' = \boldsymbol{\theta} + \sqrt{\frac{2\delta}{\boldsymbol{g}^\top \boldsymbol{H}^{-1}\boldsymbol{g}}}\boldsymbol{H}^{-1}\boldsymbol{g}. \tag{5.30}$$

实际中，我们使用共轭梯度算法来近似 $\boldsymbol{H}^{-1}\boldsymbol{g}$[5]。我们选择合适的步长来保证满足样本上的 KL 散度约束。最后，价值函数的学习通过最小化 MSE 误差达到。基于论文 (Schulman et al., 2015) 的完整的 TRPO 算法在算法 5.22 中。

注 5.3 负值 Hessian 矩阵 $-\boldsymbol{H}$ 也被称为 Fisher 信息矩阵。事实上，在批优化中，将 Fisher 信息矩阵应用到梯度下降算法中已经有不少的研究，被称为自然梯度（Nature Gradient）下降。这个方法的一个好处是，它对于再参数化是不变的，也即，不管函数参数化的方法是什么，该梯度保持不变。想了解更多关于自然梯度的细节，请参考论文 (Amari, 1998)。

5.8 近端策略优化

上一节我们介绍了信赖域策略优化算法（TRPO）。TRPO 的实现较为复杂，而且计算自然梯度的计算复杂度也较高。即使是用共轭梯度法来近似 $\boldsymbol{H}^{-1}\boldsymbol{g}$，每一次更新参数也需要多步的共轭梯度算法。在这一节中，我们介绍另一个策略梯度方法，即近端策略优化（Proximal Policy Optimization, PPO）。PPO 用一个更简单有效的方法来强制 π_θ 和 π'_θ 相似 (Schulman et al., 2017)。

回顾 TRPO 中的优化问题式 (5.26)：

$$\max_{\pi'_\theta} \quad \mathcal{L}_{\pi_\theta}(\pi'_\theta) \tag{5.35}$$

$$\text{s.t.} \quad \mathbb{E}_{s\sim\rho_{\pi_\theta}}\left[D_{\mathrm{KL}}(\pi_\theta\|\pi'_\theta)\right] \leqslant \delta. \tag{5.36}$$

与其优化一个带约束的优化问题，PPO 直接优化它的正则化版本。

$$\max_{\pi'_\theta} \mathcal{L}_{\pi_\theta}(\pi'_\theta) - \lambda\mathbb{E}_{s\sim\rho_{\pi_\theta}}\left[D_{\mathrm{KL}}(\pi_\theta\|\pi'_\theta)\right]. \tag{5.37}$$

[5]一般来讲，计算 $\boldsymbol{H}^{-1}$ 需要计算复杂度 $O(N^3)$。这在实际应用中一般代价十分昂贵，因为这里的 N 是模型参数的个数。

算法 5.22 TRPO

超参数: KL-散度上限 δ、回溯系数 α、最大回溯步数 K。
输入: 回放缓存 $\mathcal{D}_k$、初始策略函数参数 θ_0、初始价值函数参数 ϕ_0。
for episode $= 0, 1, 2, \cdots$ **do**
在环境中执行策略 $\pi_k = \pi(\theta_k)$ 并保存轨迹集 $\mathcal{D}_k = \{\tau_i\}$。
计算将得到的奖励 $\hat{G}_t$。
基于当前的价值函数 V_{ϕ_k} 计算优势函数估计 $\hat{A}_t$ （使用任何估计优势的方法）。
估计策略梯度

$$\hat{\boldsymbol{g}}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{\mathrm{T}} \nabla_\theta \log \pi_\theta(A_t|S_t)\big|_{\boldsymbol{\theta}_k} \hat{A}_t \tag{5.31}$$

使用共轭梯度算法计算

$$\hat{\boldsymbol{x}}_k \approx \hat{\boldsymbol{H}}_k^{-1} \hat{\boldsymbol{g}}_k \tag{5.32}$$

这里 $\hat{\boldsymbol{H}}_k$ 是样本平均 KL 散度的 Hessian 矩阵。
通过回溯线搜索更新策略：

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha^j \sqrt{\frac{2\delta}{\hat{\boldsymbol{x}}_k^{\mathrm{T}} \hat{\boldsymbol{H}}_k \hat{\boldsymbol{x}}_k}} \hat{\boldsymbol{x}}_k \tag{5.33}$$

这里 j 是 $\{0, 1, 2, \cdots K\}$ 中提高样本损失并且满足样本 KL 散度约束的最小值。
通过使用梯度下降的算法最小化均方误差来拟合价值函数：

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{\mathrm{T}} \left(V_\phi(S_t) - \hat{G}_t \right)^2 \tag{5.34}$$

end for

这里 λ 是正则化系数。对于式 (5.26) 每一个 δ 值，都有一个相对应的 λ 使得两个优化问题有相同的解。然而，λ 的值依赖于 π_θ。基于此，在式 (5.37) 使用一个适应性的 λ 更合理。在 PPO 中，我们通过检验 KL 散度的值来决定 λ 的值应该增大还是减小。这个版本的 PPO 算法称为 PPO-Penalty。这个版本的实现如算法 5.23 所示 (Heess et al., 2017; Schulman et al., 2017)。

另一个方法是直接剪断用于策略梯度的目标函数，从而得到更保守的更新。让 $\ell_t(\theta')$ 表示两个策略的比值 $\frac{\pi'_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)}$。经验表明，下述目标函数可以让策略梯度方法有稳定的学习性能：

$$\mathcal{L}^{\text{PPO-Clip}}(\pi'_\theta) = \mathbb{E}_{\pi_\theta} \left[\min \left(\ell_t(\theta') A^{\pi_\theta}(S_t, A_t), \text{clip}(\ell_t(\theta'), 1-\epsilon, 1+\epsilon) A^{\pi_\theta}(S_t, A_t) \right) \right]. \tag{5.38}$$

这里 $\text{clip}(x, 1-\epsilon, 1+\epsilon)$ 将 x 截断在 $[1-\epsilon, 1+\epsilon]$ 中。这个版本的算法被称为 PPO-Clip，如算法 5.24 所示 (Schulman et al., 2017)。更具体，PPO-Clip 先将 $\ell_t(\theta')$ 截断在 $[1-\epsilon, 1+\epsilon]$ 中来保证 π'_θ 和 π_θ

算法 5.23 PPO-Penalty

超参数: 奖励折扣因子 γ，KL 散度惩罚系数 λ，适应性参数 $a = 1.5, b = 2$, 子迭代次数 M, B。
输入: 初始策略函数参数 θ、初始价值函数参数 ϕ。
for k $= 0, 1, 2, \cdots$ **do**
  执行 T 步策略 π_θ，保存 $\{S_t, A_t, R_t\}$。
  估计优势函数 $\hat{A}_t = \sum_{t\prime>t} \gamma^{t\prime-t} R_{t\prime} - V_\phi(S_t)$。
  $\pi_{\text{old}} \leftarrow \pi_\theta$
  for $m \in \{1, \cdots, M\}$ **do**
    $J_{\text{PPO}}(\theta) = \sum_{t=1}^{\text{T}} \frac{\pi_\theta(A_t|S_t)}{\pi_{\text{old}}(A_t|S_t)} \hat{A}_t - \lambda \hat{\mathbb{E}}_t \left[D_{\text{KL}}(\pi_{\text{old}}(\cdot|S_t) \| \pi_\theta(\cdot|S_t)) \right]$
    使用梯度算法基于 $J_{\text{PPO}}(\theta)$ 更新策略函数参数 θ。
  end for
  for $b \in \{1, \cdots, B\}$ **do**
    $L(\phi) = -\sum_{t=1}^{\text{T}} \left(\sum_{t\prime>t} \gamma^{t\prime-t} R_{t\prime} - V_\phi(S_t) \right)^2$
    使用梯度算法基于 $L(\phi)$ 更新价值函数参数 ϕ。
  end for
  计算 $d = \hat{\mathbb{E}}_t \left[D_{\text{KL}}(\pi_{\text{old}}(\cdot|S_t) \| \pi_\theta(\cdot|S_t)) \right]$
  if $d < d_{\text{target}}/a$ **then**
    $\lambda \leftarrow \lambda/b$
  else if $d > d_{\text{target}} \times a$ **then**
    $\lambda \leftarrow \lambda \times b$
  end if
end for

相似。最后，取截断的目标函数和未截断的目标函数中较小的一方作为学习的最终目标函数。所以，PPO-Clip 可以理解为在最大化目标函数的同时将从 π_θ 到 π'_θ 的更新保持在可控范围内。

5.9 使用 Kronecker 因子化信赖域的 Actor-Critic

使用 Kronecker 因子化信赖域的 Actor-Critic（Actor Critic using Kronecker-factored Trust Region，ACKTR）(Wu et al., 2017) 是降低 TRPO 计算负担的另一个方法。ACKTR 的想法是通过 Kronecker 因子近似曲度方法（Kronecker-Factored Approximated Curvature，K-FAC）(Grosse et al., 2016; Martens et al., 2015) 来计算自然梯度。在这一节中，我们介绍如何用 ACKTR 来学习 MLP 策略网络。

注意到

$$\mathbb{E}_{s\sim\rho_{\pi_{\text{old}}}} \left[\frac{\partial^2}{\partial^2 \theta} D_{\text{KL}}(\pi_{\text{old}} \| \pi_\theta) \right] \tag{5.45}$$

$$= -\mathbb{E}_{s\sim\rho_{\pi_{\text{old}}}} \left[\sum_a \pi_{\text{old}}(a|s) \frac{\partial^2}{\partial^2 \theta} \log \pi_\theta(a|s) \right] \tag{5.46}$$

算法 5.24 PPO-Clip

超参数: 截断因子 ϵ，子迭代次数 M, B。
输入: 初始策略函数参数 θ、初始价值函数参数 ϕ。
for k $= 0, 1, 2, \cdots$ **do**
　在环境中执行策略 π_{θ_k} 并保存轨迹集 $\mathcal{D}_k = \{\tau_i\}$。
　计算将得到的奖励 $\hat{G}_t$。
　基于当前的价值函数 V_{ϕ_k} 计算优势函数 $\hat{A}_t$（基于任何优势函数的估计方法）。
　for $m \in \{1, \cdots, M\}$ **do**

$$\ell_t(\theta') = \frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{\text{old}}}(A_t|S_t)} \tag{5.39}$$

　　采用 Adam 随机梯度上升算法最大化 PPO-Clip 的目标函数来更新策略：

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{\mathrm{T}} \min(\ell_t(\theta') A^{\pi_{\theta_{\text{old}}}}(S_t, A_t), \tag{5.40}$$

$$\text{clip}(\ell_t(\theta'), 1-\epsilon, 1+\epsilon) A^{\pi_{\theta_{\text{old}}}}(S_t, A_t)) \tag{5.41}$$

　end for
　for $b \in \{1, \cdots, B\}$ **do**
　　采用梯度下降算法最小化均方误差来学习价值函数：

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

　end for
end for

$$= -\mathbb{E}_{s\sim\rho_{\pi_{\text{old}}}} \left[\mathbb{E}_{a\sim\pi_{\text{old}}} \left[\frac{\partial^2}{\partial^2 \theta} \log \pi_\theta(a|s) \right] \right] \tag{5.47}$$

$$= \mathbb{E}_{s\sim\rho_{\pi_{\text{old}}}} \left[\mathbb{E}_{a\sim\pi_{\text{old}}} \left[\left(\nabla_\theta \log \pi_\theta(a|s)\right) \left(\nabla_\theta \log \pi_\theta(a|s)\right)^\top \right] \right]. \tag{5.48}$$

在 TRPO 中，我们需要使用多步的共轭梯度方法来近似 $\boldsymbol{H}^{-1}\boldsymbol{g}$。在 ACKTR 中，我们用一个分块对角矩阵来来近似 $\boldsymbol{H}^{-1}$。矩阵的每一块对应神经网络每一层的 Fisher 信息矩阵。假设网络的第 ℓ 层为 $\boldsymbol{x}_{\text{out}} = \boldsymbol{W}_\ell \boldsymbol{x}_{\text{in}}$。这里 $\boldsymbol{W}_\ell$ 的维度为 $d_{\text{out}} \times d_{\text{in}}$。我们来介绍 ACKTR 分解的想法。注意到这一层的梯度 $\nabla_{\boldsymbol{W}_\ell} L$ 是 $(\nabla_{\boldsymbol{x}_{\text{out}}} L)$ 和 $\boldsymbol{x}_{\text{in}}$ 的外积 $(\nabla_{\boldsymbol{x}_{\text{out}}} L)\boldsymbol{x}_{\text{in}}^\top$。所以

$$\left(\nabla_\theta \log \pi_\theta(a|s)\right) \left(\nabla_\theta \log \pi_\theta(a|s)\right)^\top = \boldsymbol{x}_{\text{in}} \boldsymbol{x}_{\text{in}}^\top \otimes (\nabla_{\boldsymbol{x}_{\text{out}}} L)(\nabla_{\boldsymbol{x}_{\text{out}}} L)^\top, \tag{5.49}$$

这里 $\otimes$ 是 Kronecker 乘积。进一步

$$\Big(\big(\nabla_\theta \log \pi_\theta(a|s)\big)\big(\nabla_\theta \log \pi_\theta(a|s)\big)^\top\Big)^{-1} \boldsymbol{g} \tag{5.50}$$

$$= \Big(\boldsymbol{x}_{\text{in}}\boldsymbol{x}_{\text{in}}^\top \otimes (\nabla_{\boldsymbol{x}_{\text{out}}} L)(\nabla_{\boldsymbol{x}_{\text{out}}} L)^\top\Big)^{-1} \boldsymbol{g} \tag{5.51}$$

$$= \left[\Big(\boldsymbol{x}_{\text{in}}\boldsymbol{x}_{\text{in}}^\top\Big)^{-1} \otimes \Big((\nabla_{\boldsymbol{x}_{\text{out}}} L)(\nabla_{\boldsymbol{x}_{\text{out}}} L)^\top\Big)^{-1}\right] \boldsymbol{g} \tag{5.52}$$

所以，与其对一个 $(d_{\text{in}}d_{\text{out}}) \times (d_{\text{in}}d_{\text{out}})$ 的矩阵求逆，从而需要 $O(d_{\text{in}}^3 d_{\text{out}}^3)$ 计算复杂度，ACKTR 只需要对两个维度为 $d_{\text{in}} \times d_{\text{in}}$ 和 $d_{\text{out}} \times d_{\text{out}}$ 的矩阵求逆，从而计算复杂度只有 $O(d_{\text{in}}^3 + d_{\text{out}}^3)$。

ACKTR 算法的实现如算法 5.25 所示。ACKTR 算法也可以被用于学习价值网络。感兴趣的读者可以参考论文 (Wu et al., 2017) 了解更多的细节，我们这里不做详细解释。

算法 5.25 ACKTR

超参数: 步长 $\eta_{\max}$、KL-散度上限 δ。
输入: 空回放缓存 $\mathcal{D}$、初始策略函数参数 θ_0、初始价值函数参数 ϕ_0
for k $= 0, 1, 2, \cdots$。 **do**
在环境中执行策略 $\pi_k = \pi(\boldsymbol{\theta}_k)$ 并保存轨迹集 $\mathcal{D}_k = \{\tau_i | i = 0, 1, \cdots\}$。
计算累积奖励 G_t。
基于当前的价值函数 V_{ϕ_k} 计算优势函数 $\hat{A}_t$（基于任何优势函数的估计方法）。
估计策略梯度。

$$\hat{\boldsymbol{g}}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{\mathrm{T}} \nabla_\theta \log \pi_\theta(A_t|S_t)\big|_{\boldsymbol{\theta}_k} \hat{A}_t \tag{5.42}$$

for $l = 0, 1, 2, \cdots$ **do**
$\text{vec}(\Delta\boldsymbol{\theta}_k^l) = \text{vec}(\boldsymbol{A}_l^{-1} \nabla_{\boldsymbol{\theta}_k^l} \hat{\boldsymbol{g}}_k \boldsymbol{S}_l^{-1})$
这里 $\boldsymbol{A}_l = \mathbb{E}[\boldsymbol{a}_l \boldsymbol{a}_l^{\mathrm{T}}]$, $\boldsymbol{S}_l = \mathbb{E}[(\nabla_{\boldsymbol{s}_l} \hat{\boldsymbol{g}}_k)(\nabla_{\boldsymbol{s}_l} \hat{\boldsymbol{g}}_k)^{\mathrm{T}}]$（$\boldsymbol{A}_l, \boldsymbol{S}_l$ 通过计算片段的滚动平均值所得），$\boldsymbol{a}_l$ 是第 l 层的输入激活向量，$\boldsymbol{s}_l = \boldsymbol{W}_l \boldsymbol{a}_l$，$\text{vec}(\cdot)$ 是把矩阵变换成一维向量的向量化变换。
end for
由 K-FAC 近似自然梯度来更新策略：

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \eta_k \Delta\boldsymbol{\theta}_k \tag{5.43}$$

这里 $\eta_k = \min(\eta_{\max}, \sqrt{\frac{2\delta}{\boldsymbol{\theta}_k^{\mathrm{T}} \hat{\boldsymbol{H}}_k \boldsymbol{\theta}_k}})$，$\hat{\boldsymbol{H}}_k^l = \boldsymbol{A}_l \otimes \boldsymbol{S}_l$。
采用 Gauss-Newton 二阶梯度下降方法（并使用 K-FAC 近似）最小化均方误差来学习价值函数：

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{\mathrm{T}} \big(V_\phi(S_t) - G_t\big)^2 \tag{5.44}$$

end for

5.10　策略梯度代码例子

在前几节中，我们在理论角度介绍了几个基于策略梯度算法的伪代码，介绍的内容包括 REINFORCE（初版策略梯度）、Actor-Critic（AC）、同步优势 Actor-Critic（A2C）、异步优势 Actor-Critic（A3C）、信赖域策略优化（TRPO）、近端策略优化（PPO）、使用 Kronecker 因子化信赖域的 Actor Critic（ACKTR）。在本节中，我们将提供以上部分算法的 Python 代码例子。例子中以 OpenAI Gym 作为游戏环境。我们会先简单地介绍一下在例子中用到的环境，之后详细介绍各算法的实现。虽然本章中介绍的多数算法都能应用于离散和连续的环境，但在实现中对于离散和连续环境的处理有一些不同。这里我们提供的例子只是作为演示，只能应用在同一种动作空间的特定环境中。不过读者可以通过简单地修改就能使代码应用于不同动作空间的其他环境中。完整代码在 GitHub 库中[6]，例子参考并改编自许多开源资料，感兴趣的读者可以参考各代码简介注释中 Reference 部分所提及的内容进行扩展学习。

5.10.1　相关的 Gym 环境

在以下几节中提供例子的环境都基于 OpenAI Gym 环境。这些环境可以被分为离散动作空间的环境和连续动作空间的环境。

```
import gym
env = gym.make('Pong-V0')
print(env.action_space)
```

上述代码建立了一个 ID 为 Pong-V0 的环境，并且打印出了它的动作空间。将 Pong-V0 这个 ID 换成其他诸如 CartPole-V1 或者 Pendulum-V0 的 ID 可以建立相应的环境。

以下几节中的代码会用到一些开源库。这里通过如下代码引入它们。

```
import numpy as np
import tensorflow as tf
import tensorflow_probability as tfp
import tensorlayer as tl
...
```

离散动作空间环境：Pong 与 CartPole

这里将介绍两个 OpenAI Gym 中使用离散动作空间的游戏：Pong 和 CartPole。

[6]链接见读者服务

Pong

在 Pong 游戏中（如图 5.3 所示），我们控制绿色的板子上下移动来弹球。这里使用了 Pong-V0 版本。在这个版本中，状态空间是一个 RGB 图像向量，形状为 (210, 160, 3)。需要输入的动作是一个在 0,1,2,3,4,5 中的整数，分别对应如下动作：0 空动作，1 开火，2 右，3 左，4 右 + 开火，5 左 + 开火。

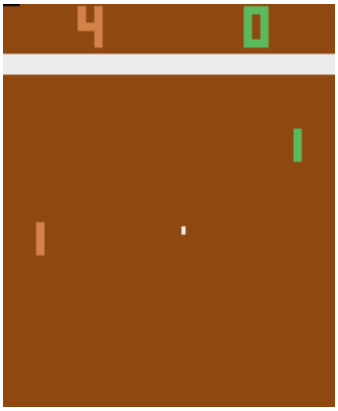

图 5.3 Pong

CartPole

CartPole（如图 5.4 所示）是一个经典的倒立摆环境。我们通过控制小车进行左右移动，来使杆子保持直立。在 CartPole-V0 环境中，观测空间是一个 4 维向量，分别表示小车的速度、小车的位置、杆子的角度、杆子顶端的速度。需要输入的动作是一个为 0 或者 1 的整数，分别控制小车左移和右移。

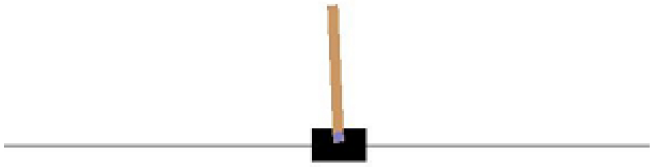

图 5.4 CartPole

连续动作空间环境：BipedalWalker-V2 与 Pendulum-V0

本节中，我们将介绍使用连续动作空间的环境：BipedalWalker-V2 和 Pendulum-V0。

BipedalWalker-V2

BipedalWalker-V2 是一个双足机器人仿真环境（如图 5.5 所示）。在环境中，我们要控制机器人在相对平坦的地面上行走，并最终到达目的地。其状态空间是一个 24 维向量，分别表示速度、角度信息，以及前方视野情况（详见表 5.1）。环境的动作空间是一个 4 维的连续动作空间，分别控制机器人的 2 个膝关节、2 个臀关节，一共 4 个关节进行旋转。

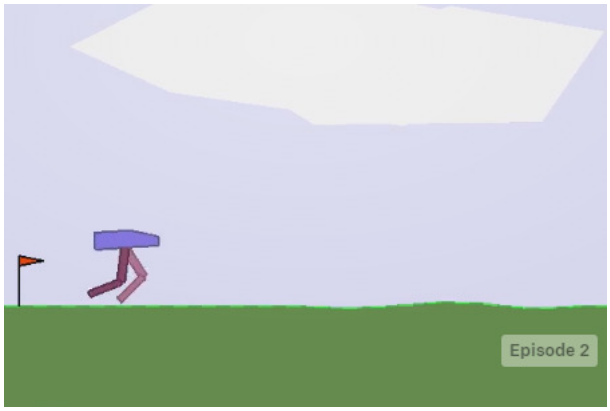

图 5.5　BipedalWalker-V2

表 5.1　BipedalWalker-V2 各维度状态意义简介

索引	简介	索引	简介
0	壳体角度	8	1 号腿触地状态
1	壳体角速度	9	2 号臀关节角度
2	壳体 x 方向速度	10	2 号臀关节速度
3	壳体 y 方向速度	11	2 号膝关节角度
4	1 号臀关节角度	12	2 号膝关节速度
5	1 号臀关节速度	13	2 号腿触地状态
6	1 号膝关节角度	14–23	10 位前方雷达测距值
7	1 号膝关节速度		

Pendulum-V0

Pendulum-V0 也是一个经典的倒立摆环境（如图 5.6 所示）。在环境中，我们需要控制杆子旋转来让其直立。环境的状态空间是一个 3 维向量，分别代表 $\cos(\theta)$、$\sin(\theta)$ 和 $\Delta(\theta)$。其中 θ 是杆子和垂直向上方向的角度。环境的动作是一维的动作，来控制杆子的旋转力矩。

图 5.6 Pendulum-V0

值得注意的是，该环境中没有终止状态。这里的意思是，必须人为设置游戏的结束。在默认情况下，环境的最大运行步长被限制为 200 步。当运行超过 200 步时，`step()` 函数返回的 Done 变量将为 True。由于有这个限制，当我们每个回合片段运行超过 200 步时，代码逻辑会因为收到 done 信号而退出该回合。通过如下代码可以移除这个限制。

```
import gym
env = gym.make('Pendulum-V0')
env = env.unwrapped # 解除最大步长的限制
```

5.10.2 REINFORCE: Atari Pong 和 CartPole-V0

Pong

开始之前，我们需要准备一下环境、模型、优化器，并初始化一些之后会用上的变量。

```
env = gym.make("Pong-V0") # 创建环境
observation = env.reset() # 重置环境
prev_x = None
running_reward = None
reward_sum = 0
episode_number = 0

# 准备收集数据
xs, ys, rs = [], [], []
epx, epy, epr = [], [], []

model = get_model([None, D]) # 创建模型
train_weights = model.trainable_weights

optimizer = tf.optimizers.RMSprop(lr=learning_rate, decay=decay_rate) # 创建优化器
```

```
model.train() # 设置模型为训练模式（防止模型被加上 DropOut）

start_time = time.time()
game_number = 0
```

在完成准备工作之后，就可以运行主循环了。首先，我们需要对观测数据进行预处理，并将处理后的数据传递给变量 x。在将 x“喂”入网络之后，我们将从网络得到每个动作的执行概率。

为了简化难度，在这里只用到了 3 个动作：空动作、上、下。在 REINFORCE 算法中，使用了 Softmax 函数输出动作概率，最后通过概率选择动作。

```
while True:
    if render:
        env.render()

    cur_x = prepro(observation)
    x = cur_x - prev_x if prev_x is not None else np.zeros(D, dtype=np.float32)
    x = x.reshape(1, D)
    prev_x = cur_x

    _prob = model(x)
    prob = tf.nn.softmax(_prob)

    # 动作 1:  空动作 2:  上 3:  下
    action = tl.rein.choice_action_by_probs(prob[0].numpy(), [1, 2, 3])
```

现在基于当前状态选出了一个动作。接下来要用该动作和环境进行交互。环境根据当前收到的动作执行到下一步，并返回观测数据、奖励、结束状态和额外信息（对应代码中的变量 _）。我们将这些数据存储起来用于之后的更新。

```
    observation, reward, done, _ = env.step(action)

    reward_sum += reward
    xs.append(x) # 一个片段内的所有观测数据
    ys.append(action - 1) # 一个片段内的所有伪标签（由于动作从 1 开始，所以这里减 1）
    rs.append(reward) # 一个片段内的所有奖励
```

如果 `step()` 返回的结束状态为 True，说明当前片段结束。我们可以重置环境并开始一个

新的片段。但在那之前，我们需要将刚刚采集的本片段的数据进行处理，之后存入跨片段数据列表中。

```
if done:
    episode_number += 1
    game_number = 0

    epx.extend(xs)
    epy.extend(ys)
    disR = tl.rein.discount_episode_rewards(rs, gamma)
    disR -= np.mean(disR)
    disR /= np.std(disR)
    epr.extend(disR)
    xs, ys, rs = [], [], []
```

智能体在进行了很多局游戏，并收集了足够的数据之后，就可以开始更新了。我们使用交叉熵损失和梯度下降方法来计算各参数的梯度，之后将梯度应用在相应的参数上，并结束更新。

```
    if episode_number
        print('batch over...... updating parameters......')
        with tf.GradientTape() as tape:
            _prob = model(epx)
            _loss = tl.rein.cross_entropy_reward_loss(_prob, epy, disR)
        grad = tape.gradient(_loss, train_weights)
        optimizer.apply_gradients(zip(grad, train_weights))

        epx, epy, epr = [], [], []
```

以上内容描述了主要工作，之后的代码主要用于显示训练相关数据，以便更好地观察训练走势。我们可以使用滑动平均来计算每个片段的运行奖励，以降低数据抖动的程度，方便观察趋势。最后，做完这些内容后别忘了重置环境，因为此时当前片段已经结束了。

```
    # if episode_number
    # tl.files.save_npz(network.all_params, name=model_file_name + '.npz')
    running_reward = reward_sum if running_reward is None else running_reward * 0.99
        + reward_sum * 0.01
    print('resetting env. episode reward total was {}. running mean:
        {}'.format(reward_sum, running_reward))
```

```
        reward_sum = 0
        observation = env.reset()
        prev_x = None

    if reward != 0:
        print(
            (  'episode
                (episode_number, game_number, time.time() - start_time, reward)
            ), ('' if reward == -1 else ' !!!!!!!!')
        )
        start_time = time.time()
        game_number += 1
```

CartPole

这个例子中，算法和 Pong 的一样。我们可以考虑将整个算法放入一个类中，并将各部分代码写入对应的函数。这样可以使得代码更为简洁易读。PolicyGradient 类的结构如下所示：

```
class PolicyGradient:
    def __init__(self, state_dim, action_num, learning_rate=0.02, gamma=0.99):
        # 类初始化。创建模型、优化器和需要的变量
        ......
    def get_action(self, s, greedy=False): # 基于动作分布选择动作
        ......
    def store_transition(self, s, a, r): # 存储从环境中采样的交互数据
        ......
    def learn(self): # 使用存储的数据进行学习和更新
        ......
    def _discount_and_norm_rewards(self): # 计算折扣化回报并进行标准化处理
        ......
    def save(self): # 存储模型
        ......
    def load(self): # 载入模型
        ......
```

初始化函数先后创建了一些变量、模型并选择 Adam 作为策略优化器。在代码中，我们可以看出这里的策略网络只有一层隐藏层。

```
def __init__(self, state_dim, action_num, learning_rate=0.02, gamma=0.99):
    self.gamma = gamma

    self.state_buffer, self.action_buffer, self.reward_buffer = [], [], []

    input_layer = tl.layers.Input([None, state_dim], tf.float32)
    layer = tl.layers.Dense(
        n_units=30, act=tf.nn.tanh, W_init=tf.random_normal_initializer(mean=0,
            stddev=0.3),
        b_init=tf.constant_initializer(0.1)
    )(input_layer)
    all_act = tl.layers.Dense(
        n_units=action_num, act=None, W_init=tf.random_normal_initializer(mean=0,
            stddev=0.3),
        b_init=tf.constant_initializer(0.1)
    )(layer)

    self.model = tl.models.Model(inputs=input_layer, outputs=all_act)
    self.model.train()
    self.optimizer = tf.optimizers.Adam(learning_rate)
```

在初始化策略网络之后，我们可以通过 `get_action()` 函数计算某状态下各动作的概率。通过设置'greedy=True'，可以直接输出概率最高的动作。

```
def get_action(self, s, greedy=False):
    _logits = self.model(np.array([s], np.float32))
    _probs = tf.nn.softmax(_logits).numpy()
    if greedy:
        return np.argmax(_probs.ravel())
    return tl.rein.choice_action_by_probs(_probs.ravel())
```

但此时，我们选择的动作可能并不好。只有通过不断学习之后，网络才能做出越来越好的判断。每次的学习过程由 `learn()` 函数完成，这部分函数的代码基本也和 Pong 例子中一样。我们使用标准化后的折扣化奖励和交叉熵损失来更新模型。在每次更新后，学过的转移数据将被丢弃。

```
def learn(self):
    # 计算标准化后的折扣化奖励
    discounted_ep_rs_norm = self._discount_and_norm_rewards()
```

```
        with tf.GradientTape() as tape:
            _logits = self.model(np.vstack(self.ep_obs))
            neg_log_prob = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=_logits,
                labels=np.array(self.ep_as))
            loss = tf.reduce_mean(neg_log_prob * discounted_ep_rs_norm)

        grad = tape.gradient(loss, self.model.trainable_weights)
        self.optimizer.apply_gradients(zip(grad, self.model.trainable_weights))

        self.ep_obs, self.ep_as, self.ep_rs = [], [], [] # 清空片段数据
        return discounted_ep_rs_norm
```

learn() 函数需要使用智能体与环境交互得到的采样数据。因此我们需要使用 store_transition() 来存储交互过程中的每个状态、动作和奖励。

```
    def store_transition(self, s, a, r):
        self.ep_obs.append(np.array([s], np.float32))
        self.ep_as.append(a)
        self.ep_rs.append(r)
```

策略梯度算法使用蒙特卡罗方法。因此，我们需要计算折扣化回报，并对回报进行标准化，也有助于学习。

```
    def _discount_and_norm_rewards(self):
        # 计算折扣化片段奖励
        discounted_ep_rs = np.zeros_like(self.ep_rs)
        running_add = 0
        for t in reversed(range(0, len(self.ep_rs))):
            running_add = running_add * self.gamma + self.ep_rs[t]
            discounted_ep_rs[t] = running_add
        # 标准化片段奖励
        discounted_ep_rs -= np.mean(discounted_ep_rs)
        discounted_ep_rs /= np.std(discounted_ep_rs)
        return discounted_ep_rs
```

和 Pong 的代码一样，我们先准备好环境和算法。在创建好环境之后，我们产生一个名为 agent 的 PolicyGradient 类的实例。

```
    env = gym.make(ENV_ID).unwrapped
```

```
# 通过设置随机种子，可以复现一些运行情况
np.random.seed(RANDOM_SEED)
tf.random.set_seed(RANDOM_SEED)
env.seed(RANDOM_SEED)
agent = PolicyGradient(
    action_num=env.action_space.n,
    state_dim=env.observation_space.shape[0],
)
t0 = time.time()
```

在训练模式中，我们使用模型输出的动作来和环境进行交互，之后存储转移数据并在每个片段更新策略。为了简化代码，智能体将在每局结束时直接进行更新。

```
if args.train:
    all_episode_reward = []
    for episode in range(TRAIN_EPISODES):
        # 重置环境
        state = env.reset()
        episode_reward = 0

        for step in range(MAX_STEPS): # 在一个片段中
            if RENDER:
                env.render()
            # 选择动作
            action = agent.get_action(state)
            # 与环境交互
            next_state, reward, done, info = env.step(action)
            # 存储转移数据
            agent.store_transition(state, action, reward)

            state = next_state
            episode_reward += reward
            # 如果环境返回 done 为 True，则跳出循环
            if done:
                break
        # 在每局游戏结束时进行更新
        agent.learn()
        print(
```

```
            'Training | Episode: {}/{} | Episode Reward: {:.0f} | Running Time:
                {:.4f}'.format(
                episode + 1, TRAIN_EPISODES, episode_reward,
                time.time() - t0))
```

我们可以在每局游戏结束后的部分增加一些代码，以便更好地显示训练过程。我们显示每个回合的总奖励和通过滑动平均计算的运行奖励。之后可以绘制运行奖励以便更好地观察训练趋势。最后，存储训练好的模型。

```
    agent.save()
    plt.plot(all_episode_reward)
    if not os.path.exists('image'):
        os.makedirs('image')
    plt.savefig(os.path.join('image', 'pg.png'))
```

如果我们使用测试模式，则过程更为简单，只需要载入预训练的模型，再用它和环境进行交互即可。

```
if args.test:
    # 进行测试
    agent.load()
    for episode in range(TEST_EPISODES):
        state = env.reset()
        episode_reward = 0
        for step in range(MAX_STEPS):
            env.render()
            state, reward, done, info = env.step(agent.get_action(state, True))
            episode_reward += reward
            if done:
                break
        print(
            'Testing | Episode: {}/{} | Episode Reward: {:.0f} | Running Time:
                {:.4f}'.format(
                episode + 1, TEST_EPISODES, episode_reward,
                time.time() - t0))
```

5.10.3 AC: CartPole-V0

Actor-Critic 算法通过 TD 方法计算基准，能在每次和环境交互后立刻更新策略，和 MC 非常不同。

在 Actor-Critic 算法中，我们建立了 2 个类：`Actor` 和 `Critic`，其结构如下所示。

```
class Actor(object):
    def __init__(self, state_dim, action_num, lr=0.001): # 类初始化。创建模型、优化器及其
                                                          # 所需变量
        ...
    def learn(self, state, action, td_error): # 更新模型
        ...
    def get_action(self, state, greedy=False): # 通过概率分布或者贪心方法选择动作
        ...
    def save(self): # 存储训练模型
        ...
    def load(self): # 载入训练模型
        ...

class Critic(object):
    def __init__(self, state_dim, lr=0.01): # 类初始化。创建模型、优化器及其所需变量
        ...
    def learn(self, state, reward, state_): # 更新模型
        ...
    def save(self): # 存储训练模型
        ...
    def load(self): # 载入训练模型
        ...
```

`Actor` 类的部分和策略梯度算法很像。唯一的区别是 `learn()` 函数使用了 TD 误差作为优势估计值进行更新，而不是使用折扣化奖励。

```
    def learn(self, state, action, td_error):
        with tf.GradientTape() as tape:
            _logits = self.model(np.array([state]))
            _exp_v = tl.rein.cross_entropy_reward_loss(logits=_logits, actions=[action],
                rewards=td_error[0])
        grad = tape.gradient(_exp_v, self.model.trainable_weights)
        self.optimizer.apply_gradients(zip(grad, self.model.trainable_weights))
```

```
        return _exp_v
```

和 PG 算法不同，AC 算法有一个带有价值网络的批判者，它能估计每个状态的价值。所以它初始化函数十分清晰，只需要创建网络和优化器即可。

```
class Critic(object):
    def __init__(self, state_dim, lr=0.01):
        input_layer = tl.layers.Input([1, state_dim], name='state')
        layer = tl.layers.Dense(
            n_units=30, act=tf.nn.relu6, W_init=tf.random_uniform_initializer(0, 0.01),
                name='hidden'
        )(input_layer)
        layer = tl.layers.Dense(n_units=1, act=None, name='value')(layer)
        self.model = tl.models.Model(inputs=input_layer, outputs=layer, name="Critic")
        self.model.train()

        self.optimizer = tf.optimizers.Adam(lr)
```

在初始化函数之后，我们有了一个价值网络。下一步就是建立 learn() 函数。learn() 函数任务非常简单，通过公式 $\delta = R + \gamma V(s') - V(s)$ 计算 TD 误差 δ，之后将 TD 误差作为优势估计来计算损失。

```
    def learn(self, state, reward, state_, done):
        d = 0 if done else 1
        v_ = self.model(np.array([state_]))
        with tf.GradientTape() as tape:
            v = self.model(np.array([state]))
            # TD_error = r + d * lambda * V(newS) - V(S)
            td_error = reward + d * LAM * v_ - v
            loss = tf.square(td_error)
        grad = tape.gradient(loss, self.model.trainable_weights)
        self.optimizer.apply_gradients(zip(grad, self.model.trainable_weights))
        return td_error
```

存储和载入函数与往常一样。我们也可以将网络参数存储为.npz 格式的文件。

```
    def save(self): # 存储模型
        if not os.path.exists(os.path.join('model', 'ac')):
            os.makedirs(os.path.join('model', 'ac'))
```

```
        tl.files.save_npz(self.model.trainable_weights, name=os.path.join('model', 'ac',
            'model_critic.npz'))

    def load(self): # 载入模型
        tl.files.load_and_assign_npz(name=os.path.join('model', 'ac',
            'model_critic.npz'), network=self.model)
```

训练循环的代码和之前的代码非常相似。唯一的不同是更新的时机不同。使用 TD 误差的情况下，我们可以在每步进行更新。

```
if args.train:
    all_episode_reward = []
    for episode in range(TRAIN_EPISODES):
        # 重置环境
        state = env.reset().astype(np.float32)
        step = 0 # 片段中的步数
        episode_reward = 0 # 整个片段的奖励
        while True:
            if RENDER: env.render()
            # 选择动作，并与环境交互
            action = actor.get_action(state)
            state_new, reward, done, info = env.step(action)
            state_new = state_new.astype(np.float32)

            if done: reward = -20 # reward shaping trick
            episode_reward += reward

            # 在和环境交互后，更新模型
            td_error = critic.learn(state, reward, state_new, done)
            actor.learn(state, action, td_error)

            state = state_new
            step += 1

            # 一直运行，直到环境返回 done 为 True，或者达到最大步数限制
            if done or step >= MAX_STEPS:
                break
```

显示信息、绘图和测试部分的代码和策略梯度的代码一样，这里就不再赘述了。

5.10.4　A3C: BipedalWalker-v2

在这里的 A3C 实现中，有个全局的 AC 和许多 Worker。全局 AC 的功能是使用 Worker 节点采集的数据更新网络。每个 Worker 节点都有自己的 AC 网络，用来和环境交互。Worker 节点并将采集的数据传给全局 AC，之后从全局 AC 获取最新的网络参数，再替换自己本地的参数并接着采集数据。Worker 类的结构如下所示：

```
class Worker(object):
    def __init__(self, name): # 初始化
        ...
    def work(self, globalAC): # 主要的功能函数
        ...
```

如上所说，每个 Worker 节点都有自己的行动者网络和批判者网络。所以在初始化函数中，我们通过实例化 ACNet 类来创建模型。

```
class Worker(object):
    def __init__(self, name):
        self.env = gym.make(GAME)
        self.name = name
        self.AC = ACNet(name)
```

work() 函数是 Worker 类的主要函数。它和之前代码中的主循环相似，但在更新的地方有所不同。和往常一样，这里循环的主要内容是从智能体取得动作，并与环境交互。

```
def work(self, globalAC):
    global GLOBAL_RUNNING_R, GLOBAL_EP
    total_step = 1
    buffer_s, buffer_a, buffer_r = [], [], []

    while not COORD.should_stop() and GLOBAL_EP < MAX_GLOBAL_EP:

        # 重置环境
        s = self.env.reset()
        ep_r = 0

        while True:
```

```
            # 在训练过程中，将 Worker0 可视化
            if self.name == 'Worker_0' and total_step
                self.env.render()

            # 选择动作并与环境交互
            s = s.astype('float32')
            a = self.AC.choose_action(s)
            s_, r, done, _info = self.env.step(a)
            s_ = s_.astype('float32')

            # 将机器人摔倒的奖励设置为 -2，代替原来的 -100
            if r == -100: r = -2

            ep_r += r

            # 存储转移数据
            buffer_s.append(s)
            buffer_a.append(a)
            buffer_r.append(r)
```

当智能体采集足够的数据时，将开始更新全局网络。在那之后，本地网络的参数将被替换为更新后的最新全局网络参数。

```
if total_step
    if done:
        v_s_ = 0 # 终止情况下
    else:
        v_s_ = self.AC.critic(s_[np.newaxis, :])[0,0] # 修正数据维度

    # 折扣化奖励
    buffer_v_target = []
    for r in buffer_r[::-1]:
        v_s_ = r + GAMMA * v_s_
        buffer_v_target.append(v_s_)
    buffer_v_target.reverse()
    buffer_s = tf.convert_to_tensor(np.vstack(buffer_s))
    buffer_a = tf.convert_to_tensor(np.vstack(buffer_a))
    buffer_v_target = tf.convert_to_tensor(np.vstack(buffer_v_target).astype('float32'))
```

```
    # 更新全局网络
    self.AC.update_global(buffer_s, buffer_a, buffer_v_target.astype('float32'),
        globalAC)
    buffer_s, buffer_a, buffer_r = [], [], []

    # 同步本地网络
    self.AC.pull_global(globalAC)

s = s_
total_step += 1
if done:
    if len(GLOBAL_RUNNING_R) == 0: # 存储运行过程中的奖励
        GLOBAL_RUNNING_R.append(ep_r)
    else: # 使用滑动平均
        GLOBAL_RUNNING_R.append(0.95 * GLOBAL_RUNNING_R[-1] + 0.05 * ep_r)

    print('Training | {}, Episode: {}/{} | Episode Reward: {:.4f} | Running Time:
        {:.4f}'\
    .format(self.name, GLOBAL_EP, MAX_GLOBAL_EP, ep_r, time.time()-T0 ))
    GLOBAL_EP += 1
    break
```

在上述代码中用到的 ACNet 类包含行动者和批判者。它的结构如下所示：

```
class ACNet(object):
    def __init__(self, scope): # 初始化
        ...
    def update_global(self, buffer_s, buffer_a, buffer_v_target, globalAC):
        # 更新全局网络
        ...
    def pull_global(self, globalAC): # 本地网络同步全局网络
        ...
    def get_action(self, s, greedy=False): # 本地网络采集动作
        ...
    def save(self): # 存储训练模型
        ...
    def load(self): # 载入训练模型
```

```
    ...
```

update_global() 函数是其中最重要的函数之一，从如下代码可以看出，使用了采样数据来计算梯度，但是将梯度应用到全局网络，在那之后，再从全局网络更新数据，并继续循环。在这个模式下，可以异步更新多个 Worker 节点。

```
def update_global(
        self, buffer_s, buffer_a, buffer_v_target, globalAC
): # 通过采样更新全局 AC 网络
    # 更新全局批判者
    with tf.GradientTape() as tape:
        self.v = self.critic(buffer_s)
        self.v_target = buffer_v_target
        td = tf.subtract(self.v_target, self.v, name='TD_error')
        self.c_loss = tf.reduce_mean(tf.square(td))
    self.c_grads = tape.gradient(self.c_loss, self.critic.trainable_weights)
    OPT_C.apply_gradients(zip(self.c_grads, globalAC.critic.trainable_weights))
        # 将本地梯度应用在全局网络上
    # 更新全局行动者
    with tf.GradientTape() as tape:
        self.mu, self.sigma = self.actor(buffer_s)
        self.test = self.sigma[0]
        self.mu, self.sigma = self.mu * A_BOUND[1], self.sigma + 1e-5

        normal_dist = tfd.Normal(self.mu, self.sigma) # tf2.0 中没有 tf.contrib
        self.a_his = buffer_a
        log_prob = normal_dist.log_prob(self.a_his)
        exp_v = log_prob * td # td 在 critic 用过了，这里没有梯度
        entropy = normal_dist.entropy() # 鼓励探索
        self.exp_v = ENTROPY_BETA * entropy + exp_v
        self.a_loss = tf.reduce_mean(-self.exp_v)
    self.a_grads = tape.gradient(self.a_loss, self.actor.trainable_weights)
    OPT_A.apply_gradients(zip(self.a_grads, globalAC.actor.trainable_weights))
        # 将本地梯度应用在全局网络上
    return self.test # 返回测试用数据
```

更新本地网络的函数非常简单，只要将本地网络的参数替换为全局网络的参数即可。

```
def pull_global(self, globalAC): # 本地运行，从全局网络同步数据
```

```
        for l_p, g_p in zip(self.actor.trainable_weights,
            globalAC.actor.trainable_weights):
            l_p.assign(g_p)
        for l_p, g_p in zip(self.critic.trainable_weights,
            globalAC.critic.trainable_weights):
            l_p.assign(g_p)
```

最后，准备工作都完成后，在主函数中逐一启动各个线程即可。

```
env = gym.make(GAME)
N_S = env.observation_space.shape[0]
N_A = env.action_space.shape[0]

A_BOUND = [env.action_space.low, env.action_space.high]
A_BOUND[0] = A_BOUND[0].reshape(1, N_A)
A_BOUND[1] = A_BOUND[1].reshape(1, N_A)
with tf.device("/cpu:0"):
    GLOBAL_AC = ACNet(GLOBAL_NET_SCOPE) # 这里的全局网络只用来存储参数

T0 = time.time()
if args.train:
    with tf.device("/cpu:0"):
        OPT_A = tf.optimizers.RMSprop(LR_A, name='RMSPropA')
        OPT_C = tf.optimizers.RMSprop(LR_C, name='RMSPropC')

        workers = []
        for i in range(N_WORKERS):
            i_name = "Worker_%i" %i # worker name
            workers.append(Worker(i_name, GLOBAL_AC))

    COORD = tf.train.Coordinator()

    # 启动 TF 线程
    worker_threads = []
    for worker in workers:
        # t = threading.Thread(target=worker.work)
        job = lambda: worker.work(GLOBAL_AC)
        t = threading.Thread(target=job)
```

```
        t.start()
        worker_threads.append(t)
    COORD.join(worker_threads)

    GLOBAL_AC.save()
    plt.plot(GLOBAL_RUNNING_R)
    if not os.path.exists('image'):
        os.makedirs('image')
    plt.savefig(os.path.join('image', 'a3c.png'))
```

5.10.5 TRPO: Pendulum-V0

TRPO 以信赖域方法使用在 KL 散度约束下的最大更新步长。例子中也使用了通用优势估计器（Generalized Advantage Estimator, GAE）。我们先看一下 `GAE_Buffer` 如何实现。

```
class GAE_Buffer:
    def __init__(self, obs_dim, act_dim, size, gamma=0.99, lam=0.95): # 初始化缓存
        ...
    def store(self, obs, act, rew, val, logp, mean, log_std): # 存储数据
        ...
    def finish_path(self, last_val=0): # 通过 GAE-Lambda 计算优势估计
        ...
    def _discount_cumsum(self, x, discount): # 机选折扣化累积和
        ...
    def is_full(self): # 查看缓存是否已满
        ...
    def get(self): # 从缓存中取出数据
        ...
```

我们在初始化函数中建立之后要用到的变量。

```
class GAE_Buffer:
    def __init__(self, obs_dim, act_dim, size, gamma=0.99, lam=0.95):
        self.obs_buf = np.zeros((size, obs_dim), dtype=np.float32)
        self.act_buf = np.zeros((size, act_dim), dtype=np.float32)
        self.adv_buf = np.zeros(size, dtype=np.float32)
        self.rew_buf = np.zeros(size, dtype=np.float32)
        self.ret_buf = np.zeros(size, dtype=np.float32)
```

```
        self.val_buf = np.zeros(size, dtype=np.float32)
        self.logp_buf = np.zeros(size, dtype=np.float32)
        self.mean_buf = np.zeros(size, dtype=np.float32)
        self.log_std_buf = np.zeros(size, dtype=np.float32)
        self.gamma, self.lam = gamma, lam
        self.ptr, self.path_start_idx, self.max_size = 0, 0, size
```

在 store() 函数中，我们将数据存入对应的缓存中，再移动指针。

```
    def store(self, obs, act, rew, val, logp, mean, log_std):
        assert self.ptr < self.max_size # 确保有存储空间
        self.obs_buf[self.ptr] = obs
        self.act_buf[self.ptr] = act
        self.rew_buf[self.ptr] = rew
        self.val_buf[self.ptr] = val
        self.logp_buf[self.ptr] = logp
        self.mean_buf[self.ptr] = mean
        self.log_std_buf[self.ptr] = log_std
        self.ptr += 1
```

finish_path() 函数在每个轨迹的结尾或者一个回合结束时会被调用。它提取当前轨迹并计算 GAE-Lambda 优势和价值函数会用到的累积回报。

```
    def finish_path(self, last_val=0):
        path_slice = slice(self.path_start_idx, self.ptr)
        rews = np.append(self.rew_buf[path_slice], last_val)
        vals = np.append(self.val_buf[path_slice], last_val)
        # 下面两行计算了 GAE-Lambda 优势
        deltas = rews[:-1] + self.gamma * vals[1:] - vals[:-1]
        self.adv_buf[path_slice] = self._discount_cumsum(deltas, self.gamma * self.lam)

        # 下一行计算了折扣化奖励，它将作为价值函数的目标
        self.ret_buf[path_slice] = self._discount_cumsum(rews, self.gamma)[:-1]

        self.path_start_idx = self.ptr
```

在之前代码中用到的 _discount_cumsum() 函数如下所示。这里使用了 scipy（一个开源库）的内建函数来实现。

```
    def _discount_cumsum(self, x, discount):
        return scipy.signal.lfilter([1], [1, float(-discount)], x[::-1], axis=0)[::-1]
```

is_full() 函数只是简单确认一下指针是否移动到底。

```
    def is_full(self):
        return self.ptr == self.max_size
```

当缓存满了的时候，我们将取出数据并重置指针。这里使用了优势标准化技术。

```
    def get(self):
        assert self.ptr == self.max_size # 取数据之前，缓存必须是满的
        self.ptr, self.path_start_idx = 0, 0

        # 下两行实现的是优势标准化技术
        adv_mean, adv_std = np.mean(self.adv_buf), np.std(self.adv_buf)
        self.adv_buf = (self.adv_buf - adv_mean) / adv_std
        return [self.obs_buf, self.act_buf, self.adv_buf, self.ret_buf, self.logp_buf,
            self.mean_buf, self.log_std_buf]
```

接下来我们将介绍 TRPO，其结构如下所示：

```
class TRPO:
    def __init__(self, state_dim, action_dim, action_bound): # 创建网络、优化器及变量
        ...
    def get_action(self, state, greedy=False): # 获取动作和其他变量
        ...
    def pi_loss(self, states, actions, adv, old_log_prob): # 计算策略损失
        ...
    def gradient(self, states, actions, adv, old_log_prob): # 计算策略网络梯度
        ...
    def train_vf(self, states, rewards_to_go): # 训练价值网络
        ...
    def kl(self, states, old_mean, old_log_std): # 计算 KL 散度
        ...
    def _flat_concat(self, xs): # 展平变量
        ...
    def get_pi_params(self): # 获取策略网络的参数
        ...
```

```
    def set_pi_params(self, flat_params): # 设置策略网络的参数
        ...
    def save(self): # 存储网络参数
        ...
    def load(self): # 载入网络参数
        ...
    def cg(self, Ax, b): # 共轭梯度算法
        ...
    def hvp(self, states, old_mean, old_log_std, x): # Hessian 向量积（Hessian-vector
        product）
        ...
    def update(self): # 更新全部网络
        ...
    def finish_path(self, done, next_state): # 结束一段轨迹
        ...
```

和往常一样，我们在初始化函数中先设置网络、优化器和其他变量。这里的动作分布是由一个均值和一个标准差描述的高斯分布。策略网络只输出了每个动作维度的均值，所有动作共用一个变量来作为对数标准差。

```
class TRPO:
    def __init__(self, state_dim, action_dim, action_bound):
        # critic
        with tf.name_scope('critic'):
            layer = input_layer = tl.layers.Input([None, state_dim], tf.float32)
            for d in HIDDEN_SIZES:
                layer = tl.layers.Dense(d, tf.nn.relu)(layer)
            v = tl.layers.Dense(1)(layer)
        self.critic = tl.models.Model(input_layer, v)
        self.critic.train()

        # actor
        with tf.name_scope('actor'):
            layer = input_layer = tl.layers.Input([None, state_dim], tf.float32)
            for d in HIDDEN_SIZES:
                layer = tl.layers.Dense(d, tf.nn.relu)(layer)
            mean = tl.layers.Dense(action_dim, tf.nn.tanh)(layer)
            mean = tl.layers.Lambda(lambda x: x * action_bound)(mean)
            log_std = tf.Variable(np.zeros(action_dim, dtype=np.float32))
```

```
        self.actor = tl.models.Model(input_layer, mean)
        self.actor.trainable_weights.append(log_std)
        self.actor.log_std = log_std
        self.actor.train()

        self.buf = GAE_Buffer(state_dim, action_dim, BATCH_SIZE, GAMMA, LAM)
        self.critic_optimizer = tf.optimizers.Adam(learning_rate=VF_LR)
        self.action_bound = action_bound
```

有了网络，我们就可以通过如下函数取得对应状态下的动作。除此之外，我们需要计算一些额外数据存入 GAE 缓存中。

```
    def get_action(self, state, greedy=False):
        state = np.array([state], np.float32)
        mean = self.actor(state)
        log_std = tf.convert_to_tensor(self.actor.log_std)
        std = tf.exp(log_std)
        std = tf.ones_like(mean) * std
        pi = tfp.distributions.Normal(mean, std)

        if greedy:
            action = mean
        else:
            action = pi.sample()
        action = np.clip(action, -self.action_bound, self.action_bound)
        logp_pi = pi.log_prob(action)

        value = self.critic(state)
        return action[0], value, logp_pi, mean, log_std
```

如下代码显示了如何计算策略损失。我们先计算替代优势，这是一个描述当前策略在之前策略采样的数据中表现如何的数据。之后使用负的替代优势作为子策略损失。

```
    def pi_loss(self, states, actions, adv, old_log_prob):
        mean = self.actor(states)
        pi = tfp.distributions.Normal(mean, tf.exp(self.actor.log_std))
        log_prob = pi.log_prob(actions)[:, 0]
        ratio = tf.exp(log_prob - old_log_prob)
```

```
        surr = tf.reduce_mean(ratio * adv)
        return -surr
```

通过调用之前定义的 pi_loss() 函数，我们可以很简单地计算梯度。

```
    def gradient(self, states, actions, adv, old_log_prob):
        pi_params = self.actor.trainable_weights
        with tf.GradientTape() as tape:
            loss = self.pi_loss(states, actions, adv, old_log_prob)
        grad = tape.gradient(loss, pi_params)
        gradient = self._flat_concat(grad)
        return gradient, loss
```

训练价值网络的方法如下所示。只要通过回归减少均方差即可拟合价值函数。

```
    def train_vf(self, states, rewards_to_go):
        with tf.GradientTape() as tape:
            value = self.critic(states)
            loss = tf.reduce_mean((rewards_to_go - value[:, 0]) ** 2)
        grad = tape.gradient(loss, self.critic.trainable_weights)
        self.critic_optimizer.apply_gradients(zip(grad, self.critic.trainable_weights))
```

计算 KL 散度的过程如下所示。我们先基于均值和标准差产生动作分布，然后计算两个分布的 KL 散度。

```
    def kl(self, states, old_mean, old_log_std):
        old_mean = old_mean[:, np.newaxis]
        old_log_std = old_log_std[:, np.newaxis]
        old_std = tf.exp(old_log_std)
        old_pi = tfp.distributions.Normal(old_mean, old_std)

        mean = self.actor(states)
        std = tf.exp(self.actor.log_std)*tf.ones_like(mean)
        pi = tfp.distributions.Normal(mean, std)

        kl = tfp.distributions.kl_divergence(pi, old_pi)
        all_kls = tf.reduce_sum(kl, axis=1)
        return tf.reduce_mean(all_kls)
```

在这个代码例子中，许多参数都使用 _flat_concat() 函数展平，这样能简化很多计算过程。

```
def _flat_concat(self, xs):
    return tf.concat([tf.reshape(x, (-1,)) for x in xs], axis=0)
```

如下的 get_pi_params() 和 set_pi_params() 函数用于获得和设置行动者网络的参数。在获取和设置参数的过程中需要进行一些简单的处理。

```
def get_pi_params(self):
    pi_params = self.actor.trainable_weights
    return self._flat_concat(pi_params)

def set_pi_params(self, flat_params):
    pi_params = self.actor.trainable_weights
    flat_size = lambda p: int(np.prod(p.shape.as_list())) # the 'int' is important
        for scalars
    splits = tf.split(flat_params, [flat_size(p) for p in pi_params])
    new_params = [tf.reshape(p_new, p.shape) for p, p_new in zip(pi_params, splits)]
    return tf.group([p.assign(p_new) for p, p_new in zip(pi_params, new_params)])
```

存储和载入函数和之前一样。

```
def save(self):
    path = os.path.join('model', 'trpo')
    if not os.path.exists(path):
        os.makedirs(path)
    tl.files.save_weights_to_hdf5(os.path.join(path, 'actor.hdf5'), self.actor)
    tl.files.save_weights_to_hdf5(os.path.join(path, 'critic.hdf5'), self.critic)

def load(self):
    path = os.path.join('model', 'trpo')
    tl.files.load_hdf5_to_weights_in_order(os.path.join(path, 'actor.hdf5'),
        self.actor)
    tl.files.load_hdf5_to_weights_in_order(os.path.join(path, 'critic.hdf5'),
        self.critic)
```

如下代码实现的是共轭梯度算法[7]。使用这个函数可以不通过计算和存储整个矩阵来直接计算矩阵向量积。

[7]链接见读者服务

```
    def cg(self, Ax, b):
        x = np.zeros_like(b)
        r = copy.deepcopy(b) # 注意，这里应该是'b - Ax(x)'，但 x=0 时，Ax(x)=0。如果想热启
                             # 动可以进行修改
        p = copy.deepcopy(r)
        r_dot_old = np.dot(r, r)
        for _ in range(CG_ITERS):
            z = Ax(p)
            alpha = r_dot_old / (np.dot(p, z) + EPS)
            x += alpha * p
            r -= alpha * z
            r_dot_new = np.dot(r, r)
            p = r + (r_dot_new / r_dot_old) * p
            r_dot_old = r_dot_new
        return x
```

如下代码显示了通过使用公式 $\boldsymbol{Hx} = \nabla_\theta \left(\left(\nabla_\theta \bar{D}_{KL}(\theta \| \theta_k) \right)^{\mathrm{T}} \boldsymbol{x} \right)$ 计算 Hessian 向量积的过程。这里使用阻尼系数来改变计算 $\boldsymbol{Hx} \to (\alpha \boldsymbol{I} + \boldsymbol{H})\boldsymbol{x}$ 的过程，可以获得更好的数值稳定性。

```
    def hvp(self, states, old_mean, old_log_std, x):
        pi_params = self.actor.trainable_weights
        with tf.GradientTape() as tape1:
            with tf.GradientTape() as tape0:
                d_kl = self.kl(states, old_mean, old_log_std)
            g = self._flat_concat(tape0.gradient(d_kl, pi_params))
            l = tf.reduce_sum(g * x)
        hvp = self._flat_concat(tape1.gradient(l, pi_params))

        if DAMPING_COEFF > 0:
            hvp += DAMPING_COEFF * x
        return hvp
```

有了如上准备，我们最后可以开始更新了。首先，通过 GAE 采集数据并计算梯度和损失。接着我们使用共轭梯度算法来计算变量 $\boldsymbol{x}$，它对应公式 $\hat{\boldsymbol{x}}_k \approx \hat{\boldsymbol{H}}_k^{-1} \hat{\boldsymbol{g}}_k$ 中的 $\hat{\boldsymbol{x}}_k$。然后，我们计算公式 $\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{\boldsymbol{x}}_k^{\mathrm{T}} \hat{\boldsymbol{H}}_k \hat{\boldsymbol{x}}_k}} \hat{\boldsymbol{x}}_k$ 中的 $\sqrt{\frac{2\delta}{\hat{\boldsymbol{x}}_k^{\mathrm{T}} \hat{\boldsymbol{H}}_k \hat{\boldsymbol{x}}_k}}$ 部分。之后，我们使用回溯线搜索来更新策略网络。最后，通过 MES 损失更新价值网络。

```
    def update(self):
```

```
    states, actions, adv, rewards_to_go, logp_old_ph, old_mu, old_log_std =
        self.buf.get()
    g, pi_l_old = self.gradient(states, actions, adv, logp_old_ph)

    Hx = lambda x: self.hvp(states, old_mu, old_log_std, x)
    x = self.cg(Hx, g)

    alpha = np.sqrt(2 * DELTA / (np.dot(x, Hx(x)) + EPS))
    old_params = self.get_pi_params()

    def set_and_eval(step):
        params = old_params - alpha * x * step
        self.set_pi_params(params)
        d_kl = self.kl(states, old_mu, old_log_std)
        loss = self.pi_loss(states, actions, adv, logp_old_ph)
        return [d_kl, loss]

    # 回溯线搜索，固定 KL 限制
    for j in range(BACKTRACK_ITERS):
        kl, pi_l_new = set_and_eval(step=BACKTRACK_COEFF ** j)
        if kl <= DELTA and pi_l_new <= pi_l_old:
            # 接受一步线搜索中更新的新参数
            break
    else:
        # 线搜索失败，保持旧参数
        set_and_eval(step=0.)

    # 价值网络更新
    for _ in range(TRAIN_V_ITERS):
        self.train_vf(states, rewards_to_go)
```

这里在轨迹要被切断或者回合结束的时候，也会需要使用 `finish_path()` 函数。如果轨迹由于智能体到达终止状态而结束，那么最后的价值将被设置为 0。

```
def finish_path(self, done, next_state):
    if not done:
        next_state = np.array([next_state], np.float32)
        last_val = self.critic(next_state)
```

```
        else:
            last_val = 0
        self.buf.finish_path(last_val)
```

代码的主循环如下所示。我们先创建环境、智能体和一些后面会用上的变量。

```
env = gym.make(ENV_ID).unwrapped

# 设置随机种子以便复现效果
np.random.seed(RANDOM_SEED)
tf.random.set_seed(RANDOM_SEED)
env.seed(RANDOM_SEED)

state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
action_bound = env.action_space.high

agent = TRPO(state_dim, action_dim, action_bound)
t0 = time.time()
```

在训练模式下，我们将智能体与环境产生的交互数据存入缓存，当缓存满了的时候则进行一次更新。

```
if args.train: # train
    all_episode_reward = []
    for episode in range(TRAIN_EPISODES):
        state = env.reset()
        state = np.array(state, np.float32)
        episode_reward = 0
        for step in range(MAX_STEPS):
            if RENDER:
                env.render()
            action, value, logp, mean, log_std = agent.get_action(state)
            next_state, reward, done, _ = env.step(action)
            next_state = np.array(next_state, np.float32)
            agent.buf.store(state, action, reward, value, logp, mean, log_std)
            episode_reward += reward
            state = next_state
            if agent.buf.is_full():
```

```
                agent.finish_path(done, next_state)
                agent.update()
            if done:
                break
        agent.finish_path(done, next_state)
        if episode == 0:
            all_episode_reward.append(episode_reward)
        else:
            all_episode_reward.append(all_episode_reward[-1] * 0.9 + episode_reward *
                0.1)
        print(
            'Training | Episode: {}/{} | Episode Reward: {:.4f} | Running Time:
                {:.4f}'.format(
                episode+1, TRAIN_EPISODES, episode_reward,
                time.time() - t0
            )
        )
        if episode
            agent.save()
    agent.save()
```

接着我们可以增加一些绘图的代码，以便于观察训练过程。

```
    plt.plot(all_episode_reward)
    if not os.path.exists('image'):
        os.makedirs('image')
    plt.savefig(os.path.join('image', 'trpo.png'))
```

当训练完成后，我们可以开始测试。

```
if args.test:
    # test
    agent.load()
    for episode in range(TEST_EPISODES):
        state = env.reset()
        episode_reward = 0
        for step in range(MAX_STEPS):
            env.render()
            action, *_ = agent.get_action(state, greedy=True)
```

```
            state, reward, done, info = env.step(action)
            episode_reward += reward
            if done:
                break
        print(
            'Testing | Episode: {}/{} | Episode Reward: {:.4f} | Running Time:
                {:.4f}'.format(
                episode + 1, TEST_EPISODES, episode_reward,
                time.time() - t0))
```

5.10.6　PPO: Pendulum-V0

PPO 是一种一阶方法，与 TRPO 这样的二阶算法不同。

在 PPO-Penalty 中，是通过给目标函数增加一个 KL 散度惩罚项的，以解决像 TRPO 这样带 KL 约束的更新问题。PPO 类的结构如下所示：

```
class PPO(object):
    def __init__(self, state_dim, action_dim, action_bound, method='clip'): # 初始化
        ...
    def train_actor(self, state, action, adv, old_pi): # 行动者训练函数
        ...
    def train_critic(self, reward, state): # 批判者训练函数
        ...
    def update(self): # 主更新函数
        ...
    def get_action(self, s, greedy=False): # 选择动作
        ...
    def save(self): # 存储网络
        ...
    def load(self): # 载入网络
        ...
    def store_transition(self, state, action, reward): # 存储每步的状态、动作、奖励
        ...
    def finish_path(self, next_state): # 计算累积奖励
        ...
```

在 PPO 算法中，我们在初始化函数中建立行动者网络和批判者网络。PPO 有两种方法：PPO-Penalty 和 PPO-Clip。我们在选用不同的方法时，要设置其相对应的参数。由于环境是一个连续运

动控制环境，我们可以使用随机策略网络输出均值和对数标准差来描述动作分布。另外，我们在网络输出加了一个 lambda 层将均值乘以 2，这是由于'Pendulum-V0' 环境中的动作范围是 $[-2, 2]$。

```
class PPO(object):
    def __init__(self, state_dim, action_dim, action_bound, method='clip'):
        # Critic
        with tf.name_scope('critic'):
            inputs = tl.layers.Input([None, state_dim], tf.float32, 'state')
            layer = tl.layers.Dense(64, tf.nn.relu)(inputs)
            layer = tl.layers.Dense(64, tf.nn.relu)(layer)
            v = tl.layers.Dense(1)(layer)
        self.critic = tl.models.Model(inputs, v)
        self.critic.train()

        # Actor
        with tf.name_scope('actor'):
            inputs = tl.layers.Input([None, state_dim], tf.float32, 'state')
            layer = tl.layers.Dense(64, tf.nn.relu)(inputs)
            layer = tl.layers.Dense(64, tf.nn.relu)(layer)
            a = tl.layers.Dense(action_dim, tf.nn.tanh)(layer)
            mean = tl.layers.Lambda(lambda x: x * action_bound, name='lambda')(a)
            logstd = tf.Variable(np.zeros(action_dim, dtype=np.float32))
        self.actor = tl.models.Model(inputs, mean)
        self.actor.trainable_weights.append(logstd)
        self.actor.logstd = logstd
        self.actor.train()
        self.actor_opt = tf.optimizers.Adam(LR_A)
        self.critic_opt = tf.optimizers.Adam(LR_C)

        self.method = method
        if method == 'penalty':
            self.kl_target = KL_TARGET
            self.lam = LAM
        elif method == 'clip':
            self.epsilon = EPSILON

        self.state_buffer, self.action_buffer = [], []
        self.reward_buffer, self.cumulative_reward_buffer = [], []
```

```
        self.action_bound = action_bound
```

train_actor() 函数负责使用 PPO 方法更新行动者。PPO 使用特定的目标函数来防止新策略远离旧策略。

```
    def train_actor(self, state, action, adv, old_pi):
        with tf.GradientTape() as tape:
            mean, std = self.actor(state), tf.exp(self.actor.logstd)
            pi = tfp.distributions.Normal(mean, std)

            ratio = tf.exp(pi.log_prob(action) - old_pi.log_prob(action))
            surr = ratio * adv
            if self.method == 'penalty': # ppo penalty
                kl = tfp.distributions.kl_divergence(old_pi, pi)
                kl_mean = tf.reduce_mean(kl)
                aloss = -(tf.reduce_mean(surr - self.lam * kl))
            else: # ppo clip
                aloss = -tf.reduce_mean(
                    tf.minimum(surr,
                               tf.clip_by_value(ratio, 1. - self.epsilon, 1. + self.epsilon)
                                   * adv)
                )
        a_gard = tape.gradient(aloss, self.actor.trainable_weights)
        self.actor_opt.apply_gradients(zip(a_gard, self.actor.trainable_weights))

        if self.method == 'kl_pen':
            return kl_mean
```

train_critic() 函数负责对批判者进行更新，代码如下所示。过程就是计算优势并最小化损失 $\sum_t \hat{A}_t^2$。

```
    def train_critic(self, reward, state):
        reward = np.array(reward, dtype=np.float32)
        with tf.GradientTape() as tape:
            advantage = reward - self.critic(state)
            loss = tf.reduce_mean(tf.square(advantage))
        grad = tape.gradient(loss, self.critic.trainable_weights)
        self.critic_opt.apply_gradients(zip(grad, self.critic.trainable_weights))
```

在 update() 函数中，我们先计算旧策略的分布，之后再进行更新。如果我们使用 PPO-Penalty 方法，则我们还需要在更新行动者之后，根据 KL 散度来更新 lambda 值。

```
def update(self):
    s = np.array(self.state_buffer, np.float32)
    a = np.array(self.action_buffer, np.float32)
    r = np.array(self.cumulative_reward_buffer, np.float32)
    mean, std = self.actor(s), tf.exp(self.actor.logstd)
    pi = tfp.distributions.Normal(mean, std)
    adv = r - self.critic(s)

    # update actor
    if self.method == 'kl_pen':
        for _ in range(A_UPDATE_STEPS):
            kl = self.a_train(s, a, adv, pi)
        if kl < self.kl_target / 1.5:
            self.lam /= 2
        elif kl > self.kl_target * 1.5:
            self.lam *= 2
    else:
        for _ in range(A_UPDATE_STEPS):
            self.a_train(s, a, adv, pi)

    # update critic
    for _ in range(C_UPDATE_STEPS):
        self.c_train(r, s)

    self.state_buffer.clear()
    self.action_buffer.clear()
    self.cumulative_reward_buffer.clear()
    self.reward_buffer.clear()
```

get_action() 函数就是简单地使用均值和标准差来描述动作分布，并且从中采样动作。如果我们想要一个没有探索的动作，就只需要输出均值即可。

```
def get_action(self, s, greedy=False):
    state = state[np.newaxis, :].astype(np.float32)
    mean, std = self.actor(state), tf.exp(self.actor.logstd)
    if greedy:
```

```
            action = mean[0]
        else:
            pi = tfp.distributions.Normal(mean, std)
            action = tf.squeeze(pi.sample(1), axis=0)[0]
        return np.clip(action, -self.action_bound, self.action_bound)
```

save()、load()、store_transition() 函数和之前的代码类似，这里不做展开。finish_path() 函数负责在游戏结束或者采集好了一批数据的时候计算累计奖励。

```
    def finish_path(self, next_state, done):
        if done:
            v_s_ = 0
        else:
            v_s_ = self.critic(np.array([next_state], np.float32))[0, 0]
        discounted_r = []
        for r in self.reward_buffer[::-1]:
            v_s_ = r + GAMMA * v_s_
            discounted_r.append(v_s_)
        discounted_r.reverse()
        discounted_r = np.array(discounted_r)[:, np.newaxis]
        self.cumulative_reward_buffer.extend(discounted_r)
        self.reward_buffer.clear()
```

主函数也和之前的十分相似。首先建立环境和 PPO 智能体。

```
env = gym.make(ENV_ID).unwrapped

# 设置随机种子，可以更好地复现效果
env.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
tf.random.set_seed(RANDOM_SEED)

state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
action_bound = env.action_space.high

agent = PPO(state_dim, action_dim, action_bound)
t0 = time.time()
```

接着使用智能体和环境进行交互，并存储数据。在游戏结束或者收集足够的数据时，执行 finish_path() 函数计算累计奖励。在采集好一批数据时更新智能体。在经历过很多次学习之后，智能体就能取得很好的分数了。

```
if args.train:
    all_episode_reward = []
    for episode in range(TRAIN_EPISODES):
        state = env.reset()
        episode_reward = 0
        for step in range(MAX_STEPS): # 在单个片段中
            if RENDER:
                env.render()
            action = agent.get_action(state)
            state_, reward, done, info = env.step(action)
            agent.store_transition(state, action, reward)
            state = state_
            episode_reward += reward

            # 更新 PPO
            if len(agent.state_buffer) >= BATCH_SIZE:
                agent.finish_path(state_, done)
                agent.update()
            if done:
                break
        agent.finish_path(state_, done)
        print(
            'Training | Episode: {}/{} | Episode Reward: {:.4f} | Running Time:
                {:.4f}'.format(
                episode + 1, TRAIN_EPISODES, episode_reward, time.time() - t0)
        )
        if episode == 0:
            all_episode_reward.append(episode_reward)
        else:
            all_episode_reward.append(all_episode_reward[-1] * 0.9 + episode_reward *
                0.1)

    agent.save()
    plt.plot(all_episode_reward)
```

```
    if not os.path.exists('image'):
        os.makedirs('image')
    plt.savefig(os.path.join('image', 'ppo.png'))
```

最后，像往常一样测试智能体。

```
if args.test:
    agent.load()
    for episode in range(TEST_EPISODES):
        state = env.reset()
        episode_reward = 0
        for step in range(MAX_STEPS):
            env.render()
            state, reward, done, info = env.step(agent.get_action(state, greedy=True))
            episode_reward += reward
            if done:
                break
        print(
            'Testing | Episode: {}/{} | Episode Reward: {:.4f} | Running Time:
                {:.4f}'.format(
                episode + 1, TEST_EPISODES, episode_reward,
                time.time() - t0))
```

参考文献

AMARI S I, 1998. Natural gradient works efficiently in learning[J]. Neural computation, 10(2): 251-276.

GOODFELLOW I, POUGET-ABADIE J, MIRZA M, et al., 2014. Generative Adversarial Nets[C]// Proceedings of the Neural Information Processing Systems (Advances in Neural Information Processing Systems) Conference.

GROSSE R, MARTENS J, 2016. A kronecker-factored approximate fisher matrix for convolution layers[C]//International Conference on Machine Learning (ICML). 573-582.

HEESS N, SRIRAM S, LEMMON J, et al., 2017. Emergence of locomotion behaviours in rich environments[J]. arXiv:1707.02286.

KAKADE S, LANGFORD J, 2002. Approximately optimal approximate reinforcement learning[C]// Proceedings of the International Conference on Machine Learning (ICML): volume 2. 267-274.

KONDA V R, TSITSIKLIS J N, 2000. Actor-critic algorithms[C]//Advances in Neural Information Processing Systems. 1008-1014.

LI J, WANG B, 2018. Policy optimization with second-order advantage information[J]. arXiv preprint arXiv:1805.03586.

LIU H, FENG Y, MAO Y, et al., 2017. Action-depedent control variates for policy optimization via stein's identity[J]. arXiv preprint arXiv:1710.11198.

MARTENS J, GROSSE R, 2015. Optimizing neural networks with kronecker-factored approximate curvature[C]//International Conference on Machine Learning (ICML). 2408-2417.

MITLIAGKAS I, ZHANG C, HADJIS S, et al., 2016. Asynchrony begets momentum, with an application to deep learning[C]//2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton). IEEE: 997-1004.

MNIH V, BADIA A P, MIRZA M, et al., 2016. Asynchronous methods for deep reinforcement learning[C]//International Conference on Machine Learning (ICML). 1928-1937.

SCHULMAN J, LEVINE S, ABBEEL P, et al., 2015. Trust region policy optimization[C]//International Conference on Machine Learning (ICML). 1889-1897.

SCHULMAN J, WOLSKI F, DHARIWAL P, et al., 2017. Proximal policy optimization algorithms[J]. arXiv:1707.06347.

SUTTON R S, MCALLESTER D A, SINGH S P, et al., 2000. Policy gradient methods for reinforcement learning with function approximation[C]//Advances in Neural Information Processing Systems. 1057-1063.

WU C, RAJESWARAN A, DUAN Y, et al., 2018. Variance reduction for policy gradient with action-dependent factorized baselines[J]. arXiv preprint arXiv:1803.07246.

WU Y, MANSIMOV E, GROSSE R B, et al., 2017. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation[C]//Advances in Neural Information Processing Systems. 5279-5288.