6 深度 Q 网络和 Actor-Critic 的结合

深度 Q 网络（Deep Q-Network，DQN）算法是最著名的深度强化学习算法之一，将强化学习与深度神经网络相结合以近似最优动作价值函数，只需以像素值作为输入就在绝大部分 Atari 游戏中达到了人类水平的表现。Actor-Critic 方法将 REINFORCE 算法的蒙特卡罗更新方式转化为时间差分更新方式，大幅度提高了采样效率。近年来，将深度 Q 网络算法与 Actor-Critic 方法相结合的算法愈加流行，如深度确定性策略梯度（Deep Deterministic Policy Gradient，DDPG）算法。这些算法结合了深度 Q 网络和 Actor-Critic 方法的优点，在大多数环境特别是连续动作空间的环境中表现出优越的性能。本章先简要介绍各类方法的优缺点，然后介绍一些将深度 Q 网络和 Actor-Critic 方法相结合的经典算法，如 DDPG 算法、孪生延迟 DDPG（Twin Delayed Deep Deterministic Policy Gradient，TD3）算法和柔性 Actor-Critic（Soft Actor-Critic，SAC）算法。

6.1 简介

深度 Q 网络（Deep Q-Network，DQN）(Mnih et al., 2015) 算法是一种经典的离线策略方法。它将 Q-Learning 算法与深度神经网络相结合，实现了从视觉输入到决策输出的端到端学习。该算法仅使用 Atari 游戏的原始像素作为输入，便在几十款游戏中取得了人类水平级的表现。然而，虽然深度 Q 网络的输入可以是高维的状态空间，但是它只能处理离散的、低维的动作空间。对于连续的、高维的动作空间，深度 Q 网络无法直接计算出每个动作对应的 Q 值。

Actor-Critic（AC）(Sutton et al., 2018) 方法是 REINFORCE (Sutton et al., 2018) 算法的扩展。通过引入 Critic，该方法将策略梯度算法的蒙特卡罗更新转化为时间差分更新。通过这种方式，自举法（Bootstrapping）可以灵活地运用到值估计当中，因此策略的更新不需要等得到完整的轨迹之后再进行，即不需要等到每局游戏结束。虽然时间差分更新会引入一些估计偏差，但它可以减

少估计方差从而加快学习速度。尽管如此，原始的 Actor-Critic 方法仍然是一种在线策略的算法，而在线策略方法的采样效率远低于离线策略方法。

将深度 Q 网络与 Actor-Critic 相结合可以同时利用这两种算法的优点。由于深度 Q 网络的存在，Actor-Critic 方法转化为离线策略方法，可以使用回放缓存的样本对网络进行训练，从而提高采样效率。从回放缓存中随机采样也可以打乱数据的序列关系，最小化样本之间的相关性，从而使价值函数的学习更加稳定。Actor-Critic 方法使得我们可以通过网络学习策略函数 π，便于处理深度 Q 网络很难解决的具有高维或连续动作空间的问题（表 6.1）。

表 6.1 深度 Q 网络算法与 Actor-Critic 算法的特点

算法	在线策略/离线策略	采样效率	动作空间
深度 Q 网络	离线策略	高	离散
Actor-Critic	在线策略	低	连续
深度 Q 网络 +Actor-Critic	离线策略	高	离散和连续

6.2 深度确定性策略梯度算法

深度确定性策略梯度算法可以看作是确定性策略梯度（Deterministic Policy Gradient，DPG）算法 (Silver et al., 2014) 和深度神经网络的结合，也可以看作是深度 Q 网络算法在连续动作空间中的扩展。它可以解决深度 Q 网络算法无法直接应用于连续动作空间的问题。深度确定性策略梯度算法同时建立 Q 值函数（Critic）和策略函数（Actor）。Q 值函数（Critic）与深度 Q 网络算法相同，通过时间差分方法进行更新。策略函数（Actor）利用 Q 值函数（Critic）的估计，通过策略梯度方法进行更新。

在深度确定性策略梯度算法中，Actor 是一个确定性策略函数，表示为 $\pi(s)$，待学习参数表示为 θ^π。每个动作直接由 $A_t = \pi(S_t|\theta_t^\pi)$ 计算，不需要从随机策略中采样。

这里，一个关键问题是如何平衡这种确定性策略的探索和利用（Exploration and Exploitation）。深度确定性策略梯度算法通过在训练过程中添加随机噪声解决该问题。每个输出动作添加噪声 N，此时有动作为 $A_t = \pi(S_t|\theta_t^\pi) + N_t$。其中 N 可以根据具体任务进行选择，原论文 (Uhlenbeck et al., 1930) 中使用 Ornstein-Uhlenbeck 过程（O-U 过程）添加噪声项。

O-U 过程满足以下随机微分方程：

$$\mathrm{d}X_t = \theta(\pi - X_t)\mathrm{d}t + \sigma\mathrm{d}W_t, \tag{6.1}$$

其中 X_t 是随机变量，$\theta > 0, x, \sigma > 0$ 为参数。W_t 是维纳过程或称布朗运动 (It et al., 1965)，它具有以下性质：

- W_t 是独立增量过程，表示对于时间 $T_0 < T_1 < ... < T_n$，有随机变量 $W_{T_0}, W_{T_1} - W_{T_0}, \cdots, W_{T_n} - W_{T_{n-1}}$ 都是独立的。
- 对于任意时刻 t 和增量 Δt, 有 $W(t + \Delta_t) - W(t) \sim N(0, \sigma_W^2 \Delta t)$。
- W_t 是关于 t 的连续函数。

我们知道马尔可夫决策过程是基于马尔可夫性质的，满足 $p(X_{t+1}|X_t, \cdots, X_1) = p(X_{t+1}|X_t)$，其中 X_t 是 t 时刻的随机变量，这意味着随机变量 X_t 的时间相关性只取决于上一个时刻的随机变量 X_{t-1}。而 O-U 噪声就是一个具有时间相关性的随机变量，这一点与马尔可夫决策过程的性质相符，因此很自然地被运用到随机噪声的添加中。然而，实践表明，时间不相关的零均值高斯噪声也能取得很好的效果。

回到深度确定性策略梯度算法，动作价值函数 $Q(s, a|\theta^Q)$ 和深度 Q 网络算法一样，通过贝尔曼方程（Bellman Equations）进行更新。

在状态 S_t 下，通过策略 π 执行动作 $A_t = \pi(S_t|\theta_t^\pi)$，得到下一个状态 S_{t+1} 和奖励值 R_t。我们有：

$$Q^\pi(S_t, A_t) = \mathbb{E}[r(S_t, A_t) + \gamma Q^\pi(S_{t+1}, \pi(S_{t+1}))]. \tag{6.2}$$

然后计算 Q 值：

$$Y_i = R_i + \gamma Q^\pi(S_{t+1}, \pi(S_{t+1})). \tag{6.3}$$

使用梯度下降算法最小化损失函数：

$$L = \frac{1}{N} \sum_i (Y_i - Q(S_i, A_i|\theta^Q))^2. \tag{6.4}$$

通过将链式法则应用于期望回报函数 J 来更新策略函数 π。这里，$J = \mathbb{E}_{R_i, S_i \sim E, A_i \sim \pi}[R_t]$（$E$ 表示环境），$R_t = \sum_{i=t}^{\mathrm{T}} \gamma^{(i-t)} r(S_i, A_i)$。我们有：

$$\begin{aligned} \nabla_{\theta^\pi} J \approx & \mathbb{E}_{S_t \sim \rho^\beta}[\nabla_{\theta^\pi} Q(s, a|\theta^Q)|_{s=S_t, a=\pi(S_t|\theta^\pi)}], \\ = & \mathbb{E}_{S_t \sim \rho^\beta}[\nabla_a Q(s, a|\theta^Q)|_{s=S_t, a=\pi(S_t)} \nabla_{\theta_\pi} \pi(s|\theta^\pi)|_{s=S_t}]. \end{aligned} \tag{6.5}$$

通过批量样本（Batches）的方式更新：

$$\nabla_{\theta^\pi} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=S_i, a=\pi(S_i)} \nabla_{\theta^\pi} \pi(s|\theta^\pi)|_{S_i}. \tag{6.6}$$

此外，深度确定性策略梯度算法采用了类似深度 Q 网络算法的目标网络，但这里通过指数平

滑方法而不是直接替换参数来更新目标网络：

$$\theta^{Q'} \leftarrow \rho\theta^{Q} + (1-\rho)\theta^{Q'}, \tag{6.7}$$

$$\theta^{\pi'} \leftarrow \rho\theta^{\pi} + (1-\rho)\theta^{\pi'}. \tag{6.8}$$

由于参数 $\rho \ll 1$，目标网络的更新缓慢且平稳，这种方式提高了学习的稳定性。

算法伪代码详见算法 6.26。

算法 6.26 DDPG

超参数：软更新因子 ρ，奖励折扣因子 γ。
输入：回放缓存 $\mathcal{D}$，初始化 critic 网络 $Q(s,a|\theta^Q)$ 参数 θ^Q、actor 网络 $\pi(s|\theta^\pi)$ 参数 θ^π、目标网络 Q'、π'。
初始化目标网络参数 Q' 和 π'，赋值 $\theta^{Q'} \leftarrow \theta^Q, \theta^{\pi'} \leftarrow \theta^\pi$。
for episode $= 1, M$ **do**
  初始化随机过程 $\mathcal{N}$ 用于给动作添加探索。
  接收初始状态 S_1。
  for t $= 1, T$ **do**
    选择动作 $A_t = \pi(S_t|\theta^\pi) + \mathcal{N}_t$。
    执行动作 A_t 得到奖励 R_t，转移到下一状态 S_{t+1}。
    存储状态转移数据对 $(S_t, A_t, R_t, D_t, S_{t+1})$ 到 $\mathcal{D}$。
    令 $Y_i = R_i + \gamma(1-D_t)Q'(S_{t+1}, \pi'(S_{t+1}|\theta^{\pi'})|\theta^{Q'})$
    通过最小化损失函数更新 Critic 网络：
      $L = \frac{1}{N}\sum_i (Y_i - Q(S_i, A_i|\theta^Q))^2$
    通过策略梯度的方式更新 Actor 网络：
      $\nabla_{\theta^\pi} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=S_i, a=\pi(S_i)} \nabla_{\theta^\pi}\pi(s|\theta^\pi)|_{S_i}$
    更新目标网络：
      $\theta^{Q'} \leftarrow \rho\theta^Q + (1-\rho)\theta^{Q'}$
      $\theta^{\pi'} \leftarrow \rho\theta^\pi + (1-\rho)\theta^{\pi'}$
  end for
end for

6.3 孪生延迟 DDPG 算法

孪生延迟 DDPG（Twin Delayed Deep Deterministic Policy Gradient，TD3）算法是深度确定性策略梯度算法的改进，其中运用了三个关键技术：

(1) 截断的 Double Q-Learning：通过学习两个 Q 值函数，用类似 Double Q-Learning 的方式更新 critic 网络。

(2) 延迟策略更新：更新过程中，策略网络的更新频率低于 Q 值网络。

(3) 目标策略平滑：在目标策略的输出动作中加入噪声，以此平滑 Q 值函数的估计，避免过

拟合。

对于第一个技术，我们知道在深度 Q 网络算法中 max 操作会导致 Q 值过估计的问题，这个问题同样存在于深度确定性策略梯度算法中，因为深度确定性策略梯度算法中 $Q(s,a)$ 的更新方式与深度 Q 网络算法相同：

$$Q(s,a) \leftarrow R_s^a + \gamma \max_{\hat{a}} Q(s', \hat{a}). \tag{6.9}$$

在表格学习方法（Tabular Methods）中不存在该问题，因为 Q 值是精确存储的。而当我们使用神经网络等工具作为函数近似器（Function Approximator）来处理更复杂的问题时，Q 值的估计是存在误差的，也就是说：

$$Q^{\text{approx}}(s', \hat{a}) = Q^{\text{target}}(s', \hat{a}) + Y_{s'}^{\hat{a}}, \tag{6.10}$$

其中，$Y_{s'}^{\hat{a}}$ 是零均值的噪声。但使用 max 操作，会导致 Q^{approx} 和 Q^{target} 之间存在误差。将误差表示为 Z_s，我们有：

$$\begin{aligned} Z_s \overset{\text{def}}{=} & R_s^a + \gamma \max_{\hat{a}} Q^{\text{approx}}(s', \hat{a}) - (R_s^a + \gamma \max_{\hat{a}} Q^{\text{target}}(s', \hat{a})), \\ = & \gamma(\max_{\hat{a}} Q^{\text{approx}}(s', \hat{a}) - \max_{\hat{a}} Q^{\text{target}}(s', \hat{a})). \end{aligned} \tag{6.11}$$

考虑噪声项 $Y_{s'}^{\hat{a}}$，一些 Q 值可能偏小，而另一些可能偏大。max 操作总是为每个状态选择最大的 Q 值，这将导致算法对高估动作的对应 Q 值异常敏感。在这种情况下，该噪声使得 $\mathbb{E}[Z_s] > 0$，从而导致过估计问题。

孪生延迟 DDPG 算法在深度确定性策略梯度算法中引入了 Double Q-Learning，通过建立两个 Q 值网络来估计下一个状态的值：

$$Q_{\theta_1'}(s', a') = Q_{\theta_1'}(s', \pi_{\phi_1}(s')), \tag{6.12}$$

$$Q_{\theta_2'}(s', a') = Q_{\theta_2'}(s', \pi_{\phi_1}(s')). \tag{6.13}$$

使用两个 Q 值中的最小值计算贝尔曼方程：

$$Y_1 = r + \gamma \min_{i=1,2} Q_{\theta_i'}(s', \pi_{\phi_1}(s')). \tag{6.14}$$

使用截断的 Double Q-Learning，目标网络的估值不会给 Q-Learning 的目标带来过高的估计误差。虽然此更新规则可能导致低估，但这对更新影响不大。因为与过估计的动作不同，低估的动作的 Q 值不会被显式更新 (Fujimoto et al., 2018)。

对于第二个技术，我们知道目标网络是实现深度强化学习算法稳定更新的有力工具。因为函

数逼近需要多次梯度更新才能收敛，目标网络在学习过程中给算法提供了一个稳定的更新目标。因此，如果目标网络可以用来减少多步更新的误差，且错误状态估计下的策略更新会导致发散的策略更新，那么策略网络应该以比价值网络更低的频率进行更新，以便在进行策略更新之前先最小化价值估计的误差。因此，孪生延迟 DDPG 算法降低了策略网络的更新频率，策略网络只在价值网络更新 d 次后才进行更新。这种策略更新方式可以使 Q 值函数的估计具有更小的方差，从而获得质量更高的策略更新。

对于第三个技术，确定性策略的一个问题是该类方法对于值空间中的窄峰估计可能存在过拟合。在孪生延迟 DDPG 算法原文中，作者认为相似的动作应该具有相似的值估计，因此将目标动作周围的一小块区域的值进行模糊拟合是有道理的：

$$y = r + \mathbb{E}_\epsilon[Q_{\theta'}(s', \pi_{\phi'}(s') + \epsilon)]. \tag{6.15}$$

通过在每个动作中加入截断的正态分布噪声作为正则化，可以平滑 Q 值的计算，避免过拟合。修正后的更新如下：

$$y = r + \gamma Q_{\theta'}(s', \pi_{\phi'}(s') + \epsilon), \epsilon \sim \text{clip}(N(0, \sigma), -c, c). \tag{6.16}$$

算法伪代码见算法 6.27。

算法 6.27 TD3

超参数：软更新因子 ρ、回报折扣因子 γ、截断因子 c
输入：回放缓存 $\mathcal{D}$，初始化 Critic 网络 $Q_{\theta_1}, Q_{\theta_2}$ 参数 θ_1, θ_2，初始化 Actor 网络 π_ϕ 参数 ϕ
初始化目标网络参数 $\hat{\theta}_1 \leftarrow \theta_1, \hat{\theta}_2 \leftarrow \theta_2, \hat{\phi} \leftarrow \phi$
for $t = 1$ to T do **do**
  选择动作 $A_t \sim \pi_\phi(S_t) + \epsilon, \epsilon \sim \mathcal{N}(0, \sigma)$
  接受奖励 R_t 和新状态 S_{t+1}
  存储状态转移数据对 $(S_t, A_t, R_t, D_t, S_{t+1})$ 到 $\mathcal{D}$
  从 $\mathcal{D}$ 中采样大小为 N 的小批量样本 $(S_t, A_t, R_t, D_t, S_{t+1})$
  $\tilde{a}_{t+1} \leftarrow \pi_{\phi'}(S_{t+1}) + \epsilon, \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}, -c, c))$。
  $y \leftarrow R_t + \gamma(1 - D_t) \min_{i=1,2} Q_{\theta_i'}(S_{t+1}, \tilde{a}_{t+1})$
  更新 Critic 网络 $\theta_i \leftarrow \arg\min_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(S_t, A_t))^2$
  if $t \bmod d$ **then**
    更新 ϕ：
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(S_t, A_t)|_{A_t = \pi_\phi(S_t)} \nabla_\phi \pi_\phi(S_t)$
    更新目标网络：
    $\hat{\theta}_i \leftarrow \rho\theta_i + (1 - \rho)\hat{\theta}_i$
    $\hat{\phi} \leftarrow \rho\phi + (1 - \rho)\hat{\phi}$
  end if
end for

6.4　柔性 Actor-Critic 算法

柔性 Actor-Critic（Soft Actor-Critic，SAC）算法继续采用了上一章提到的最大化熵的想法。学习的目标是最大化熵正则化的累积奖励而不只是累计奖励，从而鼓励更多的探索。

$$\max_{\pi_\theta} \mathbb{E}\left[\sum_t \gamma^t \left(r(S_t, A_t) + \alpha \mathcal{H}(\pi_\theta(\cdot|S_t))\right)\right]. \tag{6.17}$$

这里 α 是正则化系数。最大化熵增强学习这个想法已经被很多论文，包括 (Fox et al., 2016; Haarnoja et al., 2017; Levine et al., 2013; Nachum et al., 2017; Ziebart et al., 2008) 提及。在本节中，我们主要介绍柔性策略迭代（Soft Policy Iteration）算法。以这个算法为基础，我们会接着介绍 SAC。

6.4.1　柔性策略迭代

柔性策略迭代是一个有理论保证的学习最优最大化熵策略的算法。和策略迭代类似，柔性策略迭代也分为两步：柔性策略评估和柔性策略提高。

令

$$V^\pi(s) = \mathbb{E}\left[\sum_t \gamma^t \left(r(S_t, A_t) + \alpha \mathcal{H}(\pi(\cdot|S_t))\right)\right], \tag{6.18}$$

其中 $s_0 = s$，令

$$Q(s,a) = r(s,a) + \gamma \mathbb{E}\left[V(s')\right] \tag{6.19}$$

这里假设 $s' \sim \Pr\left(\cdot|s,a\right)$ 是下一个状态。可以很容易地验证以下式子成立。

$$V^\pi(s) = \mathbb{E}_{a\sim\pi}\left[Q(s,a) - \alpha \log(a|s)\right]. \tag{6.20}$$

在柔性策略评估时，定义的贝尔曼回溯算子 $\mathcal{T}$ 为

$$\mathcal{T}^\pi Q(s,a) = r(s,a) + \gamma \mathbb{E}\left[V^\pi(s')\right]. \tag{6.21}$$

和策略评估类似，我们可以证明对于任何映射 $Q^0 : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, $Q^k = \mathcal{T}^\pi Q^{k-1}$ 会收敛到 π 的柔性 Q 值。

在策略提高阶段，我们用当前的 Q 值求解以下最大化熵正则化奖励的优化问题。

$$\pi(\cdot|s) = \arg\max_\pi \mathbb{E}_{a\sim\pi}\left[Q(s,a) + \alpha \mathcal{H}(\pi)\right]. \tag{6.22}$$

求解以上这个优化问题后 (Fox et al., 2016; Nachum et al., 2017) 可以得到的解为

$$\pi(\cdot|s) = \frac{\exp\left(\frac{1}{\alpha}Q(s,\cdot)\right)}{Z(s)}. \tag{6.23}$$

这里 $Z(s)$ 是归一化常数，也即 $Z(s) = \sum_a \exp\left(\frac{1}{\alpha}Q(s,a)\right)$。如果采用的策略模型无法表达最优的策略 π，我们可以进一步求解

$$\pi(\cdot|s) = \arg\min_{\pi\in\Pi} D_{\text{KL}}\left(\pi(\cdot|s)\|\frac{\exp\left(\frac{1}{\alpha}Q(s,\cdot)\right)}{Z(s)}\right). \tag{6.24}$$

我们可以证明在学习过程，上面描述的柔性策略提高阶段也有单调提高的性质。即使在使用 KL-散度投影到 Π 之后这个性质也是成立的。这一点和上一章提到的 TRPO 类似。最后，我们可以证明柔性策略迭代和策略迭代类似收敛到最优解，如以下定理所示。

定理 6.1 *让 $\pi_0 \in \Pi$ 为初始策略。假设在柔性策略迭代算法下，π_0 会收敛到 $\pi*$，那么对任意的 $(s,a) \in \mathcal{S}\times\mathcal{A}$ 和任意的 $\pi \in \Pi$，$Q^{\pi*}(s,a) \geqslant Q^{\pi}(s,a)$。*

我们省略了这一章提到的各个结论的证明过程。感兴趣的读者可以参考论文 (Haarnoja et al., 2018)。

6.4.2 SAC

SAC 进一步把柔性策略迭代拓展到更实用的函数近似设定下，它采用在价值函数和策略函数之间进行交替优化的方式来学习，而不只是通过估计策略 π 的 Q 值来提升策略。

令 $Q_\phi(s,a)$ 表示 Q 值函数，π_θ 表示策略函数。这里我们考虑连续动作的设定并假设 π_θ 的输出为一个正态分布的期望和方差。和本书前面提到的方法类似，Q 值函数可以通过最小化柔性 Bellman 残差来学习：

$$J_Q(\phi) = \mathbb{E}\left[\left(Q(S_t,A_t) - r(S_t,A_t) - \gamma\mathbb{E}_{S_{t+1}}\left[V_{\tilde{\phi}}(S_{t+1})\right]\right)^2\right]. \tag{6.25}$$

这里 $V_{\tilde{\phi}}(s) = \mathbb{E}_{\pi_\theta}\left[Q_{\tilde{\phi}}(s,a) - \alpha\log\pi_\theta(a|s)\right]$，$Q_{\tilde{\phi}}$ 表示参数 $\tilde{\phi}$ 由 Q 值函数的参数 ϕ 的指数移动平均数得到的目标 Q 值网络。策略函数 π_θ 可以通过最小化以下的 KL-散度得到。

$$J_\pi(\theta) = \mathbb{E}_{s\sim\mathcal{D}}\left[\mathbb{E}_{a\sim\pi_\theta}\left[\alpha\log\pi_\theta(a|s) - Q_\phi(s,a)\right]\right]. \tag{6.26}$$

实际中，SAC 也使用了两个 Q 值函数（同时还有两个 Q 值目标函数）来处理 Q 值估计的偏差问题，也就是令 $Q_\phi(s,a) = \min\left(Q_{\phi_1}(s,a), Q_{\phi_2}(s,a)\right)$。注意到 $J_\pi(\theta)$ 中的期望也依赖于策略

π_θ，我们可以使用似然比例梯度估计的方法来优化 $J_\pi(\theta)$ (Williams, 1992)。在连续动作空间的设定下，我们也可以用策略网络的重参数化来优化。这样往往能够减少梯度估计的方差。再参数化的做法将 π_θ 表示成一个使用状态 s 和标准正态样本 ϵ 作为其输入的函数直接输出动作 a：

$$a = f_\theta(s, \epsilon). \tag{6.27}$$

代入 $J_\pi(\theta)$ 的式子中

$$J_\pi(\theta) = \mathbb{E}_{s\sim\mathcal{D},\epsilon\sim\mathcal{N}}\left[\alpha \log \pi_\theta(f_\theta(s,\epsilon)|s) - Q_\phi(s, f_\theta(s,\epsilon))\right]. \tag{6.28}$$

式子中 $\mathcal{N}$ 表示标准正态分布，π_θ 现在被表示为 f_θ。

最后，SAC 还提供了自动调节正则化参数 α 方法。该方法通过最小化以下损失函数实现。

$$J(\alpha) = \mathbb{E}_{a\sim\pi_\theta}\left[-\alpha \log \pi_\theta(a|s) - \alpha\kappa\right]. \tag{6.29}$$

这里 κ 是一个可以理解为目标熵的超参数。这种更新 α 的方法被称为自动熵调节方法。其背后的原理是在给定每一步平均熵至少为 κ 的约束下，原来的策略优化问题的对偶形式。对自动熵调节方法的严格表述感兴趣的读者，可以参考 SAC 的论文 (Haarnoja et al., 2018)。算法 6.28 给出了 SAC 的伪代码。

算法 6.28 Soft Actor-Critic (SAC)

超参数: 目标熵 κ, 步长 $\lambda_Q, \lambda_\pi, \lambda_\alpha$, 指数移动平均系数 τ。
输入: 初始策略函数参数 θ, 初始 Q 值函数参数 ϕ_1 和 ϕ_2。
$\mathcal{D} = \emptyset$; $\tilde{\phi}_i = \phi_i$, for $i = 1, 2$
for $k = 0, 1, 2, \cdots$ **do**
  for $t = 0, 1, 2, \cdots$ **do**
    从 $\pi_\theta(\cdot|S_t)$ 中取样 A_t, 保存 (R_t, S_{t+1})。
    $\mathcal{D} = \mathcal{D} \cup \{S_t, A_t, R_t, S_{t+1}\}$
  end for
  进行多步梯度更新：
    $\phi_i = \phi_i - \lambda_Q \nabla J_Q(\phi_i)$ for $i = 1, 2$
    $\theta = \theta - \lambda_\pi \nabla_\theta J_\pi(\theta)$
    $\alpha = \alpha - \lambda_\alpha \nabla J(\alpha)$
    $\tilde{\phi}_i = (1-\tau)\phi_i + \tau\tilde{\phi}_i$ for $i = 1, 2$
end for
返回 θ, ϕ_1, ϕ_2。

6.5 代码例子

本节将分享 DDPG、TD3、和 SAC 的代码例子。它们都是使用 Q 网络作为批判者的 Actor-Critic 方法。这里的例子都基于 OpenAI Gym 环境。由于这些算法都基于连续动作空间，我们使用了“Pendulum-V0”环境。

6.5.1 相关的 Gym 环境

之前有提到过，Pendulum-V0 是一个经典的倒立摆环境。它有 3 维观测空间和 1 维动作空间。在每步中，环境根据当前的旋转角度、速度和加速度返回一个奖励。此任务的目标是让倒立摆尽量直立不动，来获取最高分数。

6.5.2 DDPG: Pendulum-V0

DDPG 使用离线策略和 TD 方法。DDPG 类的结构如下所示。

```
class DDPG(object):
    def __init__(self, action_dim, state_dim, action_range): # 初始化
        ...
    def ema_update(self): # 指数滑动平均更新
        ...
    def get_action(self, s, greedy=False): # 获得动作
        ...
    def learn(self): # 学习和更行
        ...
    def store_transition(self, s, a, r, s_): # 存储转移数据
        ...
    def save(self): # 存储模型
        ...
    def load(self): # 载入模型
        ...
```

在初始化函数中，建立了 4 个网络，分别是行动者网络、批判者网络、行动者目标网络和批判者目标网络。目标网络的参数将被直接替换为对应网络的参数。

```
class DDPG(object):
    def __init__(self, action_dim, state_dim, action_range):
        self.memory = np.zeros((MEMORY_CAPACITY, state_dim * 2 + action_dim + 1),
            dtype=np.float32)
```

```
        self.pointer = 0
        self.action_dim, self.state_dim, self.action_range = action_dim, state_dim,
            action_range
        self.var = VAR

        W_init = tf.random_normal_initializer(mean=0, stddev=0.3)
        b_init = tf.constant_initializer(0.1)

        def get_actor(input_state_shape, name=''):
            input_layer = tl.layers.Input(input_state_shape, name='A_input')
            layer = tl.layers.Dense(n_units=64, act=tf.nn.relu, W_init=W_init,
                b_init=b_init, name='A_l1')(input_layer)
            layer = tl.layers.Dense(n_units=64, act=tf.nn.relu, W_init=W_init,
                b_init=b_init, name='A_l2')(layer)
            layer = tl.layers.Dense(n_units=action_dim, act=tf.nn.tanh, W_init=W_init,
                b_init=b_init, name='A_a')(layer)
            layer = tl.layers.Lambda(lambda x: action_range * x)(layer)
            return tl.models.Model(inputs=input_layer, outputs=layer, name='Actor' + name)

        def get_critic(input_state_shape, input_action_shape, name=''):
            state_input = tl.layers.Input(input_state_shape, name='C_s_input')
            action_input = tl.layers.Input(input_action_shape, name='C_a_input')
            layer = tl.layers.Concat(1)([state_input, action_input])
            layer = tl.layers.Dense(n_units=64, act=tf.nn.relu, W_init=W_init,
                b_init=b_init, name='C_l1')(layer)
            layer = tl.layers.Dense(n_units=64, act=tf.nn.relu, W_init=W_init,
                b_init=b_init, name='C_l2')(layer)
            layer = tl.layers.Dense(n_units=1, W_init=W_init, b_init=b_init,
                name='C_out')(layer)
            return tl.models.Model(inputs=[state_input, action_input], outputs=layer,
                name='Critic' + name)

        # 建立网络
        self.actor = get_actor([None, state_dim])
        self.critic = get_critic([None, state_dim], [None, action_dim])
        self.actor.train()
        self.critic.train()
```

```
    def copy_para(from_model, to_model):
        for i, j in zip(from_model.trainable_weights, to_model.trainable_weights):
            j.assign(i)

    # 替换参数
    self.actor_target = get_actor([None, state_dim], name='_target')
    copy_para(self.actor, self.actor_target)
    self.actor_target.eval()

    self.critic_target = get_critic([None, state_dim], [None, action_dim],
        name='_target')
    copy_para(self.critic, self.critic_target)
    self.critic_target.eval()

    self.ema = tf.train.ExponentialMovingAverage(decay=1 - TAU) # 软替换

    self.actor_opt = tf.optimizers.Adam(LR_A)
    self.critic_opt = tf.optimizers.Adam(LR_C)
```

在训练过程中，目标网络的参数将通过滑动平均来更新。

```
def ema_update(self):
    paras = self.actor.trainable_weights + self.critic.trainable_weights
    self.ema.apply(paras)
    for i, j in zip(self.actor_target.trainable_weights +
        self.critic_target.trainable_weights, paras):
        i.assign(self.ema.average(j))
```

由于策略网络是一个确定性策略网络，所以我们如果不是要贪心地选择动作，就要对动作增加一些随机。我们这里使用了一个正态分布作为随机项，它的方差会随着更新迭代而渐渐减小。这里的随机可以改成其他方式，如 O-U 噪声。不过 OpenAI[1] 推荐使用不相关的 0 均值高斯噪声，效果很好。

```
def get_action(self, state, greedy=False):
    a = self.actor(np.array([s], dtype=np.float32))[0]
    if greedy:
        return a
```

[1]链接见读者服务

```
        # 增加一些随机，来让动作采样带有一些探索
        return np.clip(np.random.normal(a, self.var),
                       -self.action_range,
                       self.action_range)
```

在 learn() 函数中，我们从回放缓存中采样离线数据，并使用贝尔曼方程来学习 Q 函数。之后，可以通过最大化 Q 值来学习策略。最后，通过 Polyak 平均 (Polyak, 1964) 来更新目标网络，其公式为 $\theta^{Q'} \leftarrow \rho\theta^{Q} + (1-\rho)\theta^{Q'}, \theta^{\pi'} \leftarrow \rho\theta^{\pi} + (1-\rho)\theta^{\pi'}$。

```
    def learn(self):
        self.var *= .9995
        indices = np.random.choice(MEMORY_CAPACITY, size=BATCH_SIZE)
        bt = self.memory[indices, :]
        bs = bt[:, :self.s_dim]
        ba = bt[:, self.s_dim:self.s_dim + self.a_dim]
        br = bt[:, -self.s_dim - 1:-self.s_dim]
        bs_ = bt[:, -self.s_dim:]

        with tf.GradientTape() as tape:
            a_ = self.actor_target(bs_)
            q_ = self.critic_target([bs_, a_])
            y = br + GAMMA * q_
            q = self.critic([bs, ba])
            td_error = tf.losses.mean_squared_error(y, q)
        c_grads = tape.gradient(td_error, self.critic.trainable_weights)
        self.critic_opt.apply_gradients(zip(c_grads, self.critic.trainable_weights))

        with tf.GradientTape() as tape:
            a = self.actor(bs)
            q = self.critic([bs, a])
            a_loss = -tf.reduce_mean(q) # 最大化 Q 值
        a_grads = tape.gradient(a_loss, self.actor.trainable_weights)
        self.actor_opt.apply_gradients(zip(a_grads, self.actor.trainable_weights))
        self.ema_update()
```

store_transition() 函数使用了回放缓存来存储每步的转移数据。

```
def store_transition(self, s, a, r, s_):
    s = s.astype(np.float32)
    s_ = s_.astype(np.float32)
    transition = np.hstack((s, a, [r], s_))
    index = self.pointer
    self.memory[index, :] = transition
    self.pointer += 1
```

主函数非常直接易懂，就是在每一步中使用智能体和环境交互，将数据存入回放缓存，再从回放缓存中随机采样数据更新网络。

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

# 设置随机种子，方便复现效果
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_range = env.action_space.high # 缩放动作 [-action_range, action_range]

agent = DDPG(action_dim, state_dim, action_range)
t0 = time.time()

if args.train: # 训练
    all_episode_reward = []
    for episode in range(TRAIN_EPISODES):
        state = env.reset()
        episode_reward = 0
        for step in range(MAX_STEPS):
            if RENDER:
                env.render()
            # 添加探索噪声
            action = agent.get_action(state)
            state_, reward, done, info = env.step(action)
            agent.store_transition(state, action, reward, state_)
```

```
            if agent.pointer > MEMORY_CAPACITY:
                agent.learn()
            state = state_
            episode_reward += reward
            if done:
                break

        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
            )
        )

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

在训练完成后，可以进行测试。

```
if args.test:
    # 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, 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))
```

6.5.3 TD3: Pendulum-V0

TD3 代码使用了这些类：ReplayBuffer、QNetwork、PolicyNetwork 和 TD3。

ReplayBuffer 类用来建立一个回放缓存，它的主要函数是 push() 和 sample() 函数。

```
class ReplayBuffer:
    def __init__(self, capacity): # 初始化函数
        ...
    def push(self, state, action, reward, next_state, done): # 存入数据
        ...
    def sample(self, batch_size): # 采样数据
        ...
    def __len__(self): # 通过重构以实现对 len 函数的支持
        ...
```

__init__ 函数负责初始化，其中只包含指针、缓存和容量值变量。

```
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
        self.position = 0
```

push() 函数负责将数据存入缓存，并且移动指针。这里的缓存是一个环形缓存。

```
    def push(self, state, action, reward, next_state, done):
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
        self.buffer[self.position] = (state, action, reward, next_state, done)
        self.position = int((self.position + 1)
```

sample() 函数负责从缓存中采样数据并返回。

```
    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = map(np.stack, zip(*batch)) # 堆叠各元素
        return state, action, reward, next_state, done
```

通过重构 __len__() 函数可以在 ReplayBuffer 类被 len() 函数调用的时候返回缓存的大小。

```
    def __len__(self):
        return len(self.buffer)
```

QNetwork 类被用于建立批判者的 Q 网络。这里使用了另一种建立网络的方法，通过继承 Model 类并重构 forward 函数来建立网络模型。

```
class QNetwork(Model):
    def __init__(self, num_inputs, num_actions, hidden_dim, init_w=3e-3):
        super(QNetwork, self).__init__()
        input_dim = num_inputs + num_actions
        w_init = tf.random_uniform_initializer(-init_w, init_w)
        self.linear1 = Dense(n_units=hidden_dim, act=tf.nn.relu, W_init=w_init,
            in_channels=input_dim, name='q1')
        self.linear2 = Dense(n_units=hidden_dim, act=tf.nn.relu, W_init=w_init,
            in_channels=hidden_dim, name='q2')
        self.linear3 = Dense(n_units=1, W_init=w_init, in_channels=hidden_dim, name='q3')

    def forward(self, input):
        x = self.linear1(input)
        x = self.linear2(x)
        x = self.linear3(x)
        return x
```

PolicyNetwork 类用于建立行动者的策略网络。它在建立网络模型的同时，也增加了 evaluate()、get_action()、sample_action() 函数。

```
class PolicyNetwork(Model):
    def __init__(self, num_inputs, num_actions, hidden_dim, action_range=1.,
        init_w=3e-3): # 初始化网络
        ...
```

```
    def forward(self, state): # 重构前向传播函数
        ...
    def evaluate(self, state, eval_noise_scale): # 进行评估
        ...
    def get_action(self, state, explore_noise_scale, greedy=False): # 获取动作
        ...
    def sample_action(self): # 采样动作
        ...
```

建立网络结构的详细过程如下所示。

```
class PolicyNetwork(Model):
    def __init__(self, num_inputs, num_actions, hidden_dim, action_range=1.,
         init_w=3e-3):
        super(PolicyNetwork, self).__init__()
        w_init = tf.random_uniform_initializer(-init_w, init_w)
        self.linear1 = Dense(n_units=hidden_dim, act=tf.nn.relu, W_init=w_init,
            in_channels=num_inputs, name='policy1')
        self.linear2 = Dense(n_units=hidden_dim, act=tf.nn.relu, W_init=w_init,
            in_channels=hidden_dim, name='policy2')
        self.linear3 = Dense(n_units=hidden_dim, act=tf.nn.relu, W_init=w_init,
            in_channels=hidden_dim, name='policy3')
        self.output_linear = Dense(n_units=num_actions, W_init=w_init,
            b_init=tf.random_uniform_initializer(-init_w, init_w),
            in_channels=hidden_dim, name='policy_output')
        self.action_range = action_range
        self.num_actions = num_actions

    def forward(self, state):
        x = self.linear1(state)
        x = self.linear2(x)
        x = self.linear3(x)
        output = tf.nn.tanh(self.output_linear(x)) # 这里的输出范围是 [-1, 1]
        return output
```

`evaluate()` 函数通过评估状态产生用于计算梯度的动作。它利用目标策略平滑技术来产生有噪声的动作。

```
    def evaluate(self, state, eval_noise_scale):
        state = state.astype(np.float32)
        action = self.forward(state)
        action = self.action_range * action
        # 添加噪声
        normal = Normal(0, 1)
        eval_noise_clip = 2 * eval_noise_scale
        noise = normal.sample(action.shape) * eval_noise_scale
        noise = tf.clip_by_value(noise, -eval_noise_clip, eval_noise_clip)
        action = action + noise
        return action
```

get_action() 函数通过状态来产生用于和环境交互的动作。

```
    def get_action(self, state, explore_noise_scale, greedy=False):
        action = self.forward([state])
        action = self.action_range * action.numpy()[0]
        if greedy:
            return action
        # 添加噪声
        normal = Normal(0, 1)
        noise = normal.sample(action.shape) * explore_noise_scale
        action += noise
        return action.numpy()
```

sample_action() 函数用于在训练开始时产生随机动作。

```
    def sample_action(self, ):
        a = tf.random.uniform([self.num_actions], -1, 1)
        return self.action_range * a.numpy()
```

接下来介绍 TD3 类，它是本例子的核心内容。

```
class TD3():
    def __init__(self, state_dim, action_dim, replay_buffer, hidden_dim, action_range,
        policy_target_update_interval=1, q_lr=3e-4, policy_lr=3e-4):
            # 创建回放缓存和网络
        ...
    def target_ini(self, net, target_net): # 初始化目标网络时用到的硬拷贝更新
```

```
    ...
  def target_soft_update(self, net, target_net, soft_tau): # 通过使用 Polyak 平均对目标
                                                            # 网络进行软更新
    ...
  def update(self, batch_size, eval_noise_scale, reward_scale=10., gamma=0.9,
    soft_tau=1e-2): # 更新 TD3 中的所有网络
    ...
  def save(self): # 存储训练参数
    ...
  def load(self): # 载入训练参数
    ...
```

初始化函数创建了 2 个 Q 网络、1 个策略网络，还建立了它们的目标网络。总共建立了 $(2+1)\times 2=6$ 个网络。

```
class TD3():
  def __init__(self, state_dim, action_dim, replay_buffer, hidden_dim, action_range,
      policy_target_update_interval=1, q_lr=3e-4, policy_lr=3e-4):
    self.replay_buffer = replay_buffer

    # 初始化所有网络
    self.q_net1 = QNetwork(state_dim, action_dim, hidden_dim)
    self.q_net2 = QNetwork(state_dim, action_dim, hidden_dim)
    self.target_q_net1 = QNetwork(state_dim, action_dim, hidden_dim)
    self.target_q_net2 = QNetwork(state_dim, action_dim, hidden_dim)
    self.policy_net = PolicyNetwork(state_dim, action_dim, hidden_dim, action_range)
    self.target_policy_net = PolicyNetwork(state_dim, action_dim, hidden_dim,
        action_range)
    print('Q Network (1,2): ', self.q_net1)
    print('Policy Network: ', self.policy_net)

    # 初始化目标网络参数
    self.target_q_net1 = self.target_ini(self.q_net1, self.target_q_net1)
    self.target_q_net2 = self.target_ini(self.q_net2, self.target_q_net2)
    self.target_policy_net = self.target_ini(self.policy_net, self.target_policy_net)

    # 设置训练模式
    self.q_net1.train()
```

```
        self.q_net2.train()
        self.target_q_net1.train()
        self.target_q_net2.train()
        self.policy_net.train()
        self.target_policy_net.train()

        self.update_cnt = 0
        self.policy_target_update_interval = policy_target_update_interval

        self.q_optimizer1 = tf.optimizers.Adam(q_lr)
        self.q_optimizer2 = tf.optimizers.Adam(q_lr)
        self.policy_optimizer = tf.optimizers.Adam(policy_lr)
```

target_ini() 函数和 target_soft_update() 函数都用来更新目标网络。不同之处在于前者是通过硬拷贝直接替换参数，而后者是通过 Polyak 平均进行软更新。

```
    def target_ini(self, net, target_net):
        for target_param, param in zip(target_net.trainable_weights,
            net.trainable_weights):
            target_param.assign(param)
        return target_net

    def target_soft_update(self, net, target_net, soft_tau):
        for target_param, param in zip(target_net.trainable_weights,
            net.trainable_weights):
            target_param.assign(target_param * (1.0 - soft_tau) + param * soft_tau) # 软更
                新
        return target_net
```

接下来将介绍关键的 update() 函数。这部分充分体现了 TD3 算法的 3 个关键技术。

在函数的开始部分，我们先从回放缓存中采样数据。

```
    def update(self, batch_size, eval_noise_scale, reward_scale=10., gamma=0.9,
        soft_tau=1e-2): # 更新 TD3 中的所有网
        络
        self.update_cnt += 1

        # 采样数据
        state, action, reward, next_state, done = self.replay_buffer.sample(batch_size)
```

```
reward = reward[:, np.newaxis] # 扩展维度
done = done[:, np.newaxis]
```

接下来，我们通过给目标动作增加噪声实现了目标策略平滑技术。通过这样跟随动作的变化，对 Q 值进行平滑，可以使得策略更难利用 Q 函数的拟合差错。这是 TD3 算法中的第三个技术。

```
# 技术三: 目标策略平滑。通过给目标动作增加噪声来实现
new_next_action = self.target_policy_net.evaluate(
    next_state, eval_noise_scale=eval_noise_scale
) # 添加了截断的正态噪声

# 通过批数据的均值和标准差进行标准化
reward = reward_scale * (reward - np.mean(reward, axis=0)) / np.std(reward,
    axis=0)
```

下一个技术是截断的 Double-Q Learning。它将同时学习两个 Q 值函数，并且选择较小的 Q 值来作为贝尔曼误差损失函数中的目标 Q 值。通过这种方法可以减轻 Q 值的过估计。这也是 TD3 算法中的第一个技术。

```
# 训练 Q 函数
target_q_input = tf.concat([next_state, new_next_action], 1) # 0 维是样本数量

# 技术一: 截断的 Double-Q Learning。这里使用了更小的 Q 值作为目标 Q 值
target_q_min = tf.minimum(self.target_q_net1(target_q_input),
    self.target_q_net2(target_q_input))

target_q_value = reward + (1 - done) * gamma * target_q_min # 如果 done==1, 则只有
                                                            # reward 值
q_input = tf.concat([state, action], 1) # 处理 Q 网络的输入

with tf.GradientTape() as q1_tape:
    predicted_q_value1 = self.q_net1(q_input)
    q_value_loss1 = tf.reduce_mean(tf.square(predicted_q_value1 - target_q_value))
q1_grad = q1_tape.gradient(q_value_loss1, self.q_net1.trainable_weights)
self.q_optimizer1.apply_gradients(zip(q1_grad, self.q_net1.trainable_weights))

with tf.GradientTape() as q2_tape:
    predicted_q_value2 = self.q_net2(q_input)
    q_value_loss2 = tf.reduce_mean(tf.square(predicted_q_value2 - target_q_value))
```

```
        q2_grad = q2_tape.gradient(q_value_loss2, self.q_net2.trainable_weights)
        self.q_optimizer2.apply_gradients(zip(q2_grad, self.q_net2.trainable_weights))
```

最后一个技术是延迟策略更新技术。这里的策略网络及其目标网络的更新频率比 Q 值网络的更新频率更小。论文 (Fujimoto et al., 2018) 中建议每 2 次 Q 值函数更新时进行 1 次策略更新。这也是 TD3 算法中提到的第二个技术。

```
        # 训练策略函数
        # 技术二:  延迟策略更新。减少策略更新的频率
        if self.update_cnt
            with tf.GradientTape() as p_tape:
                new_action = self.policy_net.evaluate(
                    state, eval_noise_scale=0.0
                ) # 无噪声, 确定性策略梯度
                new_q_input = tf.concat([state, new_action], 1)
                # 实现方法一:
                # predicted_new_q_value =
                    tf.minimum(self.q_net1(new_q_input),self.q_net2(new_q_input))
                # 实现方法二:
                predicted_new_q_value = self.q_net1(new_q_input)
                policy_loss = -tf.reduce_mean(predicted_new_q_value)
            p_grad = p_tape.gradient(policy_loss, self.policy_net.trainable_weights)
            self.policy_optimizer.apply_gradients(zip(p_grad,
                self.policy_net.trainable_weights))

            # 软更新目标网络
            self.target_q_net1 = self.target_soft_update(self.q_net1, self.target_q_net1,
                soft_tau)
            self.target_q_net2 = self.target_soft_update(self.q_net2, self.target_q_net2,
                soft_tau)
            self.target_policy_net = self.target_soft_update(self.policy_net,
                self.target_policy_net, soft_tau)
```

如下是主要训练代码。这里先创建环境和智能体。

```
# 初始化环境
env = gym.make(ENV_ID).unwrapped
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
```

```
action_range = env.action_space.high # 缩放动作 [-action_range, action_range]

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

# 初始化回放缓存
replay_buffer = ReplayBuffer(REPLAY_BUFFER_SIZE)

# 初始化智能体
agent = TD3(state_dim, action_dim, action_range, HIDDEN_DIM, replay_buffer,
        POLICY_TARGET_UPDATE_INTERVAL, Q_LR, POLICY_LR)
t0 = time.time()
```

在开始片段之前，需要做一些初始化操作。这里训练时间受总运行步数的限制，而不是最大片段迭代数。由于网络建立的方式不同，这种方式需要在使用前额外调用一次函数。

```
# 训练循环
if args.train:
    frame_idx = 0
    all_episode_reward = []
    # 这里需要进行一次额外的调用，以使内部函数进行一些初始化操作，让其可以正常使用
    # model.forward 函数
    state = env.reset().astype(np.float32)
    agent.policy_net([state])
    agent.target_policy_net([state])
```

在训练刚开始的时候，会先由智能体进行随机采样。通过这种方式可以采集到足够多的用于更新的数据。在那之后，智能体还是和往常一样与环境进行交互并采集数据，再进行存储和更新。

```
    for episode in range(TRAIN_EPISODES):
        state = env.reset().astype(np.float32)
        episode_reward = 0
        for step in range(MAX_STEPS):
            if RENDER:
                env.render()
            if frame_idx > EXPLORE_STEPS:
```

```
            action = agent.policy_net.get_action(state, EXPLORE_NOISE_SCALE)
        else:
            action = agent.policy_net.sample_action()

        next_state, reward, done, _ = env.step(action)
        next_state = next_state.astype(np.float32)
        done = 1 if done is True else 0

        replay_buffer.push(state, action, reward, next_state, done)
        state = next_state
        episode_reward += reward
        frame_idx += 1

        if len(replay_buffer) > BATCH_SIZE:
            for i in range(UPDATE_ITR):
                agent.update(BATCH_SIZE, EVAL_NOISE_SCALE, REWARD_SCALE)
        if done:
            break
```

最终，我们提供了一些可视化训练过程所需的函数，并将训练的模型进行存储。

```
    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
        )
    )
agent.save()
plt.plot(all_episode_reward)
if not os.path.exists('image'):
    os.makedirs('image')
plt.savefig(os.path.join('image', 'td3.png'))
```

6.5.4 SAC: Pendulum-v0

SAC 使用了离线策略的方式对随机策略进行优化。它最大的特点是使用了熵正则项，但也使用了一些 TD3 中的技术。其目标 Q 值的计算使用了两个 Q 网络中的最小值和策略 $\pi(\tilde{a}|s)$ 的对数概率。例子中的代码使用了这些类：ReplayBuffer、SoftQNetwork、PolicyNetwork 和 SAC。

其中 ReplayBuffer 和 SoftQNetwork 类与 TD3 中的 ReplayBuffer 和 QNetwork 类一样，这里就不再赘述，直接介绍后续的代码。

```
class ReplayBuffer: # 一个环形回放缓存，用于存储转移数据并提供数据采样
    def __init__(self, capacity):
        ......
    def push(self, state, action, reward, next_state, done):
        ......
    def sample(self, batch_size):
        ......
    def __len__(self):
        ......

class SoftQNetwork(Model): # 用于评估状态-动作值 Q(s,a) 的网络
    def __init__(self, num_inputs, num_actions, hidden_dim, init_w=3e-3):
        ......
    def forward(self, input):
        ......
```

PolicyNetwork 类也和 TD3 的十分相似。不同之处在于，SAC 使用了一个随机策略网络，而不是 TD3 中的确定性策略网络。

```
class PolicyNetwork(Model):
    def __init__(self, num_inputs, num_actions, hidden_dim, action_range=1.,
        init_w=3e-3, log_std_min=-20, log_std_max=2): # 初始化
        ......
    def forward(self, state): # 前向传播
        ......
    def evaluate(self, state, epsilon=1e-6): # 进行评估
        ......
    def get_action(self, state, greedy=False): # 获取动作
        ......
    def sample_action(self): # 采样动作
        ......
```

随机策略网络输出了动作和对数标准差来描述动作分布。因此网络有两层输出。

```
class PolicyNetwork(Model):
    def __init__(self, num_inputs, num_actions, hidden_dim, action_range=1.,
         init_w=3e-3, log_std_min=-20, log_std_max=2):
        super(PolicyNetwork, self).__init__()
        self.log_std_min = log_std_min
        self.log_std_max = log_std_max
        w_init = tf.keras.initializers.glorot_normal(seed=None)
        self.linear1 = Dense(n_units=hidden_dim, act=tf.nn.relu, W_init=w_init,
            in_channels=num_inputs, name='policy1')
        self.linear2 = Dense(n_units=hidden_dim, act=tf.nn.relu, W_init=w_init,
            in_channels=hidden_dim, name='policy2')
        self.linear3 = Dense(n_units=hidden_dim, act=tf.nn.relu, W_init=w_init,
            in_channels=hidden_dim, name='policy3')
        self.mean_linear = Dense(n_units=num_actions, W_init=w_init,
            b_init=tf.random_uniform_initializer(-init_w, init_w),
            in_channels=hidden_dim, name='policy_mean')
        self.log_std_linear = Dense(n_units=num_actions, W_init=w_init,
            b_init=tf.random_uniform_initializer(-init_w, init_w),
            in_channels=hidden_dim, name='policy_logstd')
        self.action_range = action_range
        self.num_actions = num_actions
```

这里在 forward() 函数中的对数标准差上进行截断，防止标准差过大。

```
    def forward(self, state):
        x = self.linear1(state)
        x = self.linear2(x)
        x = self.linear3(x)
        mean = self.mean_linear(x)
        log_std = self.log_std_linear(x)
        log_std = tf.clip_by_value(log_std, self.log_std_min, self.log_std_max)
        return mean, log_std
```

evaluate() 函数使用重参数技术从动作分布上采样动作，这样可以保证梯度能够反向传播。函数也计算了采样动作在原始动作分布上的对数概率。

```
def evaluate(self, state, epsilon=1e-6):
    state = state.astype(np.float32)
    mean, log_std = self.forward(state)
    std = tf.math.exp(log_std) # 评估时不进行裁剪，裁剪会影响梯度
    normal = Normal(0, 1)
    z = normal.sample(mean.shape)
    action_0 = tf.math.tanh(mean + std * z) # 动作选用 TanhNormal 分布；这里使用了重参
                                            # 数技术
    action = self.action_range * action_0
    # 根据论文原文，这里最后加了一个额外项以标准化不同动作范围
    log_prob = Normal(mean, std).log_prob(mean + std * z) - tf.math.log(1. -
        action_0 ** 2 + epsilon) - np.log(self.action_range)
    # normal.log_prob 和 -log(1-a**2) 的维度都是 (N,dim_of_action);
    # Normal.log_prob 输出了和输入特征一样的维度，而不是 1 维的概率
    # 这里需要跨维度相加，来得到 1 维的概率，或者使用多元正态分布
    log_prob = tf.reduce_sum(log_prob, axis=1)[:, np.newaxis]
    # 由于 reduce_sum 减少了 1 个维度，这里将维度扩展回来
    return action, log_prob, z, mean, log_std
```

get_action() 函数是前面函数的简单版。它只需要从动作分布上采样动作即可。

```
def get_action(self, state, greedy=False):
    mean, log_std = self.forward([state])
    std = tf.math.exp(log_std)
    normal = Normal(0, 1)
    z = normal.sample(mean.shape)
    action = self.action_range * tf.math.tanh(
        mean + std * z
    ) # 动作分布使用 TanhNormal 分布；这里使用了重参数技术

    action = self.action_range * tf.math.tanh(mean) if greedy else action
    return action.numpy()[0]
```

sample_action() 函数更加简单。它只用在训练刚开始的时候采集第一次更新所需的数据。

```
def sample_action(self, ):
    a = tf.random.uniform([self.num_actions], -1, 1)
    return self.action_range * a.numpy()
```

SAC 的结构如下：

```
class SAC():
    def __init__(self, state_dim, action_dim, replay_buffer, hidden_dim, action_range,
        soft_q_lr=3e-4, policy_lr=3e-4, alpha_lr=3e-4): # 建立网络及变量
        ......
    def target_ini(self, net, target_net): # 初始化目标网络时所需的硬拷贝更新
        ......
    def target_soft_update(self, net, target_net, soft_tau): # 更新目标网络时所用到的软更
                                                             # 新，使用了 Polyak 平均
        ......
    def update(self, batch_size, reward_scale=10., auto_entropy=True,
        target_entropy=-2, gamma=0.99, soft_tau=1e-2): # 更新 SAC 中所有的网络
        ......
    def save(self): # 存储训练参数
        ......
    def load(self): # 载入训练参数
        ......
```

SAC 算法中有 5 个网络，分别是 2 个 soft Q 网络及其目标网络，以及一个随机策略网络。另外还需要一个 alpha 变量来作为熵正则化的权衡系数。

```
class SAC():
    def __init__(self, state_dim, action_dim, replay_buffer, hidden_dim, action_range,
         soft_q_lr=3e-4, policy_lr=3e-4, alpha_lr=3e-4):
        self.replay_buffer = replay_buffer

        # 初始化所有网络
        self.soft_q_net1 = SoftQNetwork(state_dim, action_dim, hidden_dim)
        self.soft_q_net2 = SoftQNetwork(state_dim, action_dim, hidden_dim)
        self.target_soft_q_net1 = SoftQNetwork(state_dim, action_dim, hidden_dim)
        self.target_soft_q_net2 = SoftQNetwork(state_dim, action_dim, hidden_dim)
        self.policy_net = PolicyNetwork(state_dim, action_dim, hidden_dim, action_range)
        self.log_alpha = tf.Variable(0, dtype=np.float32, name='log_alpha')
        self.alpha = tf.math.exp(self.log_alpha)
        print('Soft Q Network (1,2): ', self.soft_q_net1)
        print('Policy Network: ', self.policy_net)
        # set mode
        self.soft_q_net1.train()
```

```
        self.soft_q_net2.train()
        self.target_soft_q_net1.eval()
        self.target_soft_q_net2.eval()
        self.policy_net.train()

        # 初始化目标网络的参数
        self.target_soft_q_net1 = self.target_ini(self.soft_q_net1,
            self.target_soft_q_net1)
        self.target_soft_q_net2 = self.target_ini(self.soft_q_net2,
            self.target_soft_q_net2)

        self.soft_q_optimizer1 = tf.optimizers.Adam(soft_q_lr)
        self.soft_q_optimizer2 = tf.optimizers.Adam(soft_q_lr)
        self.policy_optimizer = tf.optimizers.Adam(policy_lr)
        self.alpha_optimizer = tf.optimizers.Adam(alpha_lr)
```

这里我们介绍一下 update() 函数。其他函数和之前 TD3 的代码一样，这里不做赘述。和往常一样，在 update() 函数的开始，我们先从回放缓存中采样数据。对奖励值进行正则化，以提高训练效果。

```
    def update(self, batch_size, reward_scale=10., auto_entropy=True, target_entropy=-2,
        gamma=0.99, soft_tau=1e-2):
        state, action, reward, next_state, done = self.replay_buffer.sample(batch_size)
        reward = reward[:, np.newaxis] # 扩展维度
        done = done[:, np.newaxis]
        reward = reward_scale * (reward - np.mean(reward, axis=0)) / (
                np.std(reward, axis=0) + 1e-6
        ) # 通过批数据的均值和标准差进行标准化，并增加一个极小的数防止除以 0 导致数值溢出问题
```

在这之后，我们将基于下一个状态值计算相应的 Q 值。SAC 使用了两个目标网络输出中较小的值，这里和 TD3 相同。但是与之不同的是，SAC 在计算目标 Q 值的时候增加了熵正则项。这里的 log_prob 部分是一个权衡策略随机性的熵值。

```
        # 训练 Q 函数
        new_next_action, next_log_prob, _, _, _ = self.policy_net.evaluate(next_state)
        target_q_input = tf.concat([next_state, new_next_action], 1) # 第 0 维是样本数量
        target_q_min = tf.minimum(
            self.target_soft_q_net1(target_q_input),
```

```
            self.target_soft_q_net2(target_q_input)
    ) - self.alpha * next_log_prob
    target_q_value = reward + (1 - done) * gamma * target_q_min
        # 如果 done==1，则只有 reward 值
```

在计算 Q 值之后，训练 Q 网络就很简单了。

```
    q_input = tf.concat([state, action], 1)
    with tf.GradientTape() as q1_tape:
        predicted_q_value1 = self.soft_q_net1(q_input)
        q_value_loss1 =
            tf.reduce_mean(tf.losses.mean_squared_error(predicted_q_value1,
            target_q_value))
    q1_grad = q1_tape.gradient(q_value_loss1, self.soft_q_net1.trainable_weights)
    self.soft_q_optimizer1.apply_gradients(zip(q1_grad,
        self.soft_q_net1.trainable_weights))
    with tf.GradientTape() as q2_tape:
        predicted_q_value2 = self.soft_q_net2(q_input)
        q_value_loss2 =
            tf.reduce_mean(tf.losses.mean_squared_error(predicted_q_value2,
            target_q_value))
    q2_grad = q2_tape.gradient(q_value_loss2, self.soft_q_net2.trainable_weights)
    self.soft_q_optimizer2.apply_gradients(zip(q2_grad,
        self.soft_q_net2.trainable_weights))
```

这里的策略损失考虑了额外的熵项。通过最大化损失函数，可以训练策略来使预期回报和熵之间的权衡达到最佳。

```
    # 训练策略网络
    with tf.GradientTape() as p_tape:
        new_action, log_prob, z, mean, log_std = self.policy_net.evaluate(state)
        new_q_input = tf.concat([state, new_action], 1) # 第 0 维是样本数量
        # 实现方式一
        predicted_new_q_value = tf.minimum(self.soft_q_net1(new_q_input),
            self.soft_q_net2(new_q_input))
        # 实现方式二
        # predicted_new_q_value = self.soft_q_net1(new_q_input)
        policy_loss = tf.reduce_mean(self.alpha * log_prob - predicted_new_q_value)
    p_grad = p_tape.gradient(policy_loss, self.policy_net.trainable_weights)
```

```
        self.policy_optimizer.apply_gradients(zip(p_grad,
            self.policy_net.trainable_weights))
```

最后，我们要更新熵权衡系数 alpha 和目标网络。

```
        # 更新 alpha
        # alpha:  探索（最大化熵）和利用（最大化 Q 值）之间的权衡
        if auto_entropy is True:
            with tf.GradientTape() as alpha_tape:
                alpha_loss = -tf.reduce_mean((self.log_alpha * (log_prob +
                    target_entropy)))
            alpha_grad = alpha_tape.gradient(alpha_loss, [self.log_alpha])
            self.alpha_optimizer.apply_gradients(zip(alpha_grad, [self.log_alpha]))
            self.alpha = tf.math.exp(self.log_alpha)
        else: # 固定 alpha 值
            self.alpha = 1.
            alpha_loss = 0

        # 软更新目标价值网络
        self.target_soft_q_net1 = self.target_soft_update(self.soft_q_net1,
            self.target_soft_q_net1, soft_tau)
        self.target_soft_q_net2 = self.target_soft_update(self.soft_q_net2,
            self.target_soft_q_net2, soft_tau)
```

训练的主循环和 TD3 一样，先建立环境和智能体。

```
# 初始化环境
env = gym.make(ENV_ID).unwrapped
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
action_range = env.action_space.high # 缩放动作，[-action_range，action_range]

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

# 初始化缓存
```

```
replay_buffer = ReplayBuffer(REPLAY_BUFFER_SIZE)
# 初始化智能体
agent = SAC(state_dim, action_dim, action_range, HIDDEN_DIM,
                  replay_buffer, SOFT_Q_LR, POLICY_LR, ALPHA_LR)
t0 = time.time()
```

之后，使用智能体和环境交互，并存储用于更新的采样数据。在第一次更新之前，用随机动作来采集数据。

```
# 训练循环
if args.train:
    frame_idx = 0
    all_episode_reward = []
    # 这里需要进行一次额外的调用，来使内部函数进行一些初始化操作，让其可以正常使用
    # model.forward 函数
    state = env.reset().astype(np.float32)
    agent.policy_net([state])

    for episode in range(TRAIN_EPISODES):
        state = env.reset().astype(np.float32)
        episode_reward = 0
        for step in range(MAX_STEPS):
            if RENDER:
                env.render()
            if frame_idx > EXPLORE_STEPS:
                action = agent.policy_net.get_action(state)
            else:
                action = agent.policy_net.sample_action()
            next_state, reward, done, _ = env.step(action)
            next_state = next_state.astype(np.float32)
            done = 1 if done is True else 0
            replay_buffer.push(state, action, reward, next_state, done)
            state = next_state
            episode_reward += reward
            frame_idx += 1
```

采集到足够的数据后，我们可以开始在每步进行更新。

```
            if len(replay_buffer) > BATCH_SIZE:
                for i in range(UPDATE_ITR):
                    agent.update(
                        BATCH_SIZE, reward_scale=REWARD_SCALE,
                            auto_entropy=AUTO_ENTROPY,
                        target_entropy=-1. * action_dim
                    )
            if done:
                break
```

通过上述步骤，智能体就可以通过不断更新变得越来越强了。增加下面的代码可以更好地显示训练过程。

```
        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
            )
        )
```

最后，存储模型并且绘制学习曲线。

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

参考文献

FOX R, PAKMAN A, TISHBY N, 2016. Taming the noise in reinforcement learning via soft updates[C]// Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence. AUAI Press: 202-211.

FUJIMOTO S, VAN HOOF H, MEGER D, 2018. Addressing function approximation error in actor-critic methods[J]. arXiv preprint arXiv:1802.09477.

HAARNOJA T, TANG H, ABBEEL P, et al., 2017. Reinforcement learning with deep energy-based policies[C]//Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR. org: 1352-1361.

HAARNOJA T, ZHOU A, HARTIKAINEN K, et al., 2018. Soft actor-critic algorithms and applications[J]. arXiv preprint arXiv:1812.05905.

IT K, MCKEAN H, 1965. Diffusion processes and their sample paths[J]. Die Grundlehren der math. Wissenschaften, 125.

LEVINE S, KOLTUN V, 2013. Guided policy search[C]//International Conference on Machine Learning. 1-9.

MNIH V, KAVUKCUOGLU K, SILVER D, et al., 2015. Human-level control through deep reinforcement learning[J]. Nature.

NACHUM O, NOROUZI M, XU K, et al., 2017. Bridging the gap between value and policy based reinforcement learning[C]//Advances in Neural Information Processing Systems. 2775-2785.

POLYAK B T, 1964. Some methods of speeding up the convergence of iteration methods[J]. USSR Computational Mathematics and Mathematical Physics, 4(5): 1-17.

SILVER D, LEVER G, HEESS N, et al., 2014. Deterministic policy gradient algorithms[C].

SUTTON R S, BARTO A G, 2018. Reinforcement learning: An introduction[M]. MIT press.

UHLENBECK G E, ORNSTEIN L S, 1930. On the theory of the brownian motion[J]. Physical review, 36(5): 823.

WILLIAMS R J, 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning[J]. Machine Learning, 8(3-4): 229-256.

ZIEBART B D, MAAS A L, BAGNELL J A, et al., 2008. Maximum entropy inverse reinforcement learning.[C]//Proceedings of the AAAI Conference on Artificial Intelligence: volume 8. Chicago, IL, USA: 1433-1438.