# 第11章 DDPG 与 TD3 算法

自本章开始到接下来的两章，我们将介绍一些非常经典的基于策略梯度的算法，包括DDPG、PPO、SAC等算法。这些算法实现方式各不相同，也各有特色，因此每类算法都单独形成一个章节来展开。同时它们也是目前强化学习实践中十分常用的一些策略梯度算法，请读者务必熟练掌握。

本章先讲DDPG算法和TD3算法，其中后者是在前者的基础上做了一些优化。严格来说，DDPG算法被提出的初衷其实是DQN算法的一个连续动作空间版本扩展。当时Actor-Critic架构还没有广泛流行，因为A3C算法是在2016年发表的，比DDPG算法晚了一年。只是我们回看DDPG算法的时候发现其在形式上更像Actor-Critic的架构，因此将其归为Actor-Critic算法的一种。

## 11.1 DPG方法

深度确定性策略梯度算法（ deep deterministic policy gradient，DDPG），是一种确定性的策略梯度算法。为了让读者更好地理解DDPG算法，我们先把 “深度” 二字去掉，即先介绍一下DPG 算法，也是 DDPG算法的核心所在。

有了前面Actor-Critic算法的铺垫之后，从策略梯度的角度来理解DPG算法是比较容易的。首先我们知道DQN算法的一个主要缺点就是不能用于连续动作空间，这是因为在DQN算法中动作是通过贪心策略或者说argmax的方式来从Q函数间接得到，这里Q函数就相当于DDPG算法中的Critic。

而要想适配连续动作空间，我们干脆就将选择动作的过程变成一个直接从状态映射到具体动作的函数 ，其中表示模型的参数，这样一来就把求解Q函数、贪心选择动作这两个过程合并成了一个函数，也就是我们常说的Actor。注意，这里的 输出的是一个动作值，而不是像Actor-Critic章节中提到的概率分布 。

我们知道 函数实际上是有两个变量的，相当于一个曲线平面，如图11-1所示。当我们输入某个状态到Actor时，即固定 时，则相当于把曲线平面截断成一条曲线。而Actor的任务就是寻找这条曲线的最高点，并返回对应的横坐标，即最大Q值对应的动作。

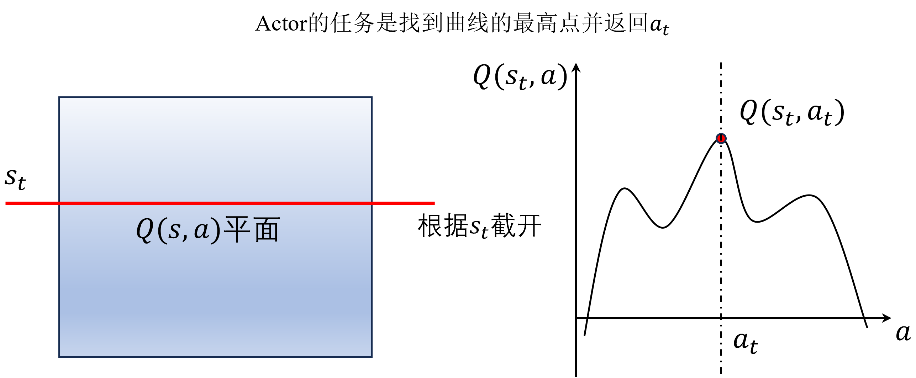


图11-1 DDPG算法中Actor的作用

所以， DDPG算法并没有做真正意义上的梯度更新，只是在寻找最大值，本质上还是DQN算法的思路。因此DDPG算法中Critic结构会同时包含状态和动作输入，而不是Actor-Critic算法中那样只包含状态，因为它本质上就是函数，如图11-2所示。

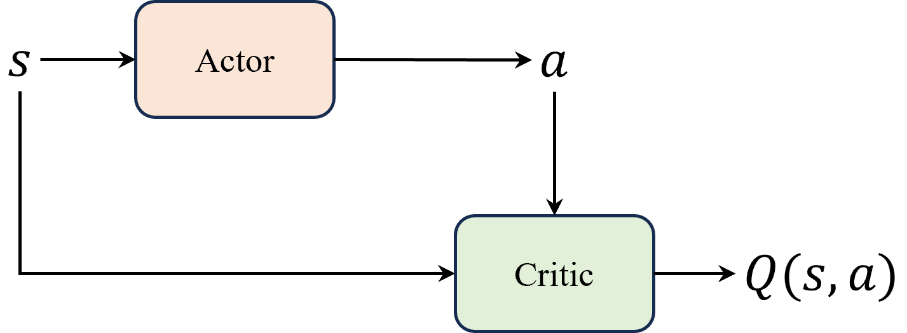


图11-2 DDPG网络结构

这里相当于是把 DQN算法中 -greedy 策略函数部分换成了Actor 。注意Actor网络 与输出概率分布的随机性策略（stochastic policy）不同，输出的是一个值，因此也叫做确定性策略（deterministic policy）。效仿策略梯度的推导，我们也可以推导出DPG算法的目标函数，如式 (11.1) 所示。

其中 是策略的初始分布，用于探索状态空间，在实际应用中相当于网络模型的初始参数，读者可以不用关注。另外注意这里的 表示的不是 Q 函数，跟 Actor-Critic 算法一样，即作为一个 Critic 网络，将状态和动作作为输入，并且输出一个值。

## 11.2 DDPG 算法

在DPG算法的基础上，再结合一些技巧，就是DDPG算法了，这些技巧既包括DQN算法中也用到的目标网络、经验回放等，也包括引入噪声来增加策略的探索性。

不知道读者有没有发现，在强化学习基础算法的研究改进当中，基本无外乎几个亘古不变的主题：一是**如何提高对值函数的估计**，保证其准确性，即尽量无偏且低方差，例如最开始的用深度神经网络替代简单的 Q 表、结合蒙特卡洛和时序差分的 、引入目标网络以及广义优势估计等等；二是如何提高探索以及平衡探索-利用的问题，尤其在探索性比较差的确定性策略中，例如 DQN和 DDPG算法都会利用各种技巧来提高探索，例如经验回放、 策略、噪声网络等等。这两个问题是强化学习算法的基础核心问题，希望能够给读者在学习和研究的过程中带来一定的启发。

回到正题，由于目标网络、经验回放前面章节都讲过了，这里就略过，我们讲讲 DDPG 引入的噪声。其实引入噪声的方法在前面 Noisy DQN 算法中就讲到了，只是 Noisy DQN 算法是在网络中引入噪声，而 DDPG 算法是在输出动作上引入噪声。本质上来讲，引入噪声的作用就是为了在不破坏系统的前提下，提高系统运行的抗干扰性。

这跟我们生活中打疫苗是类似的，通常我们会将灭活的病毒也就是疫苗注入到体内，引发免疫系统的警觉，从而提高免疫系统的抗干扰性，即提高我们身体的免疫力。这里疫苗就相当于轻微的噪声，如果免疫系统一直没见过这种噪声，那么一旦遇到真正的病毒之后是很有可能崩溃的，反之如果经常接触这种轻微的噪声，那么免疫系统就会逐渐适应，从而提高抗干扰性。又好比我们平时做的消防演练，虽然平时的演练都不是真正意义上的灾害，但经过熟练的演练之后一旦遇到真正的灾害不说从容应对，至少也不会过于慌乱了。

DDPG 算法是在输出动作上引入噪声的，由于 输出的是单个值，其实最简单的方式就是在输出的值上加上一个随机数，这个随机数可以是正态分布的（即高斯噪声），也可以是均匀分布的，只要能够保证这个随机数的值不要过大就行。

当然简单的噪声引入除了简单这一个优点之外，可能剩下的全都是缺点了，因此在 DDPG 算法中使用的其实是一种叫做 Ornstein-Uhlenbeck 的噪声，简称 OU 噪声。OU 噪声是一种具有回归特性的随机过程，其与高斯噪声相比的优点在于：

* **探索性**：OU 噪声具有持续的、自相关的特性。相比于独立的高斯噪声，OU 噪声更加平滑，并且在训练过程中更加稳定。这种平滑特性使得OU噪声有助于探索更广泛的动作空间，并且更容易找到更好的策略。
* **控制幅度**：OU 噪声可以通过调整其参数来控制噪声的幅度。在 DDPG 算法中，可以通过调整 OU 噪声的方差来控制噪声的大小，从而平衡探索性和利用性。较大的方差会增加探索性，而较小的方差会增加利用性。
* **稳定性**：OU 噪声的回归特性使得噪声在训练过程中具有一定的稳定性。相比于纯粹的随机噪声，在 DDPG 算法中使用OU 噪声可以更好地保持动作的连续性，避免剧烈的抖动，从而使得训练过程更加平滑和稳定。
* **可控性**：由于OU 噪声具有回归特性，它在训练过程中逐渐回归到均值，因此可以控制策略的探索性逐渐减小。这种可控性使得在训练的早期增加探索性，然后逐渐减小探索性，有助于更有效地进行训练。

总的来说，OU 噪声作为 DDPG 算法中的一种探索策略，具有平滑、可控、稳定等优点，使得算法能够更好地在连续动作空间中进行训练，探索更广泛的动作空间，并找到更优的策略。它是 DDPG 算法成功应用于连续动作空间问题的重要因素之一。

虽然它有这么多的优点，实际上在简单的环境中，它跟使用简单的高斯噪声甚至不用噪声的效果是差不多的，只有在复杂的环境中才会体现出来区别。因此，如果读者在实际应用中面临的问题比较简单，可以不用OU噪声，而是使用高斯噪声或者不用噪声即可，这样可以减少算法的复杂度，加快算法的收敛速度，正所谓 “杀鸡焉用牛刀”。

OU 噪声主要由两个部分组成：随机高斯噪声和回归项，其数学定义如式 (11.2) 所示。

其中 是 OU 过程在时间 t 的值，即当前的噪声值，这个 t 也是强化学习中的时步（ time step ）。 是回归到的均值，表示噪声在长时间尺度上的平均值。 是 OU 过程的回归速率，表示噪声向均值回归的速率。 是 OU 过程的扰动项，表示随机高斯噪声的标准差。 是布朗运动（ Brownian motion ）或者维纳过程（ Wiener process ），是一个随机项，表示随机高斯噪声的微小变化。

在实际应用中，我们只需要调整 和 就可以了， 通常是固定的，而 是随机项，我们也不需要关注。尽管如此，需要调整的参数还是有点多，这也是为什么 DDPG 算法的调参比较麻烦的原因之一。

## 11.3 DDPG 算法的优缺点

总的来说，DDPG 算法的优点主要有包括：

* **适用于连续动作空间**：DDPG 算法采用了确定性策略来选择动作，这使得它能够直接处理连续动作空间的问题。相比于传统的随机策略，确定性策略更容易优化和学习，因为它不需要进行动作采样，h缓解了在连续动作空间中的高方差问题。
* **高效的梯度优化**：DDPG 算法使用策略梯度方法进行优化，其梯度更新相对高效，并且能够处理高维度的状态空间和动作空间。同时，通过 Actor-Critic 结构，算法可以利用值函数来辅助策略的优化，提高算法的收敛速度和稳定性。
* **经验回放和目标网络**：这是老生常谈的内容了，经验回放机制可以减少样本之间的相关性，提高样本的有效利用率，并且增加训练的稳定性。目标网络可以稳定训练过程，避免值函数估计和目标值之间的相关性问题，从而提高算法的稳定性和收敛性。

而缺点在于以下几点。

* **只适用于连续动作空间**：这既是优点，也是缺点。
* **高度依赖超参数**：DDPG 算法中有许多超参数需要进行调整，除了一些 DQN的算法参数例如学习率、批量大小、目标网络的更新频率等，还需要调整一些 OU 噪声的参数。调整这些超参数并找到最优的取值通常是一个挑战性的任务，可能需要大量的实验和经验。
* **高度敏感的初始条件**：DDPG 算法对初始条件非常敏感。初始策略和值函数的参数设置可能会影响算法的收敛性和性能，需要仔细选择和调整。
* 容易陷入局部最优：由于采用了确定性策略，可能会导致算法陷入局部最优，难以找到全局最优策略。为了增加探索性，需要采取一些措施，如加入噪声策略或使用其他的探索方法。

可以看到，DDPG 算法的优点可能掩盖不了它众多的缺点，尤其对于初学者来说，调参是一个非常头疼的问题，因此在实际应用中，同样情况下可能会选择更加简单的 PPO 算法会来得更加容易一些。当然，对于一些熟练的调参大侠来说， DDPG 算法以及相关改进版本的算法也是值得尝试的，毕竟它们在实际应用中的效果还是非常不错的。

## 11.4 TD3 算法

我们知道 DDPG 算法的缺点太多明显，因此后来有人对其进行了改进，这就是我们接下来要介绍的 TD3 算法。TD3 算法，英文全称为 twin delayed DDPG，中文全称为双延迟确定性策略梯度算法。相对于 DDPG 算法，TD3 算法的改进主要做了三点重要的改进，一是 双 Q 网络，体现在名字中的 twin，二是 延迟更新，体现在名字中的 delayed，三是 躁声正则（ noise regularisation ），下面我们分别来看一下。

### 11.4.1 双 Q 网络

双 Q 网络的思想其实很简单，就是在 DDPG 算法中的 Critic 网络上再加一层，这样就形成了两个 Critic 网络，分别记为 和 ，其中 和 分别表示两个网络的参数。这样一来，我们就可以得到两个 Q 值，分别记为 和 $。然后我们在计算 TD 误差的时候，就可以取两个 Q 值中较小的那个，如式 11.3 所示。

其中 L 表示损失函数，同理，我们也可以得到另一个 Critic 网络的损失函数 ，如式 (11.4) 所示。

细心的读者会发现，这跟 Double DQN 的原理是一样的，只不过 Double DQN 是在 $Q$ 网络上做文章，而 TD3 算法是在 Critic 网络上做文章。这样做的好处是可以减少 Q 值的过估计，从而提高算法的稳定性和收敛性。

### 11.4.2 延迟更新

延迟更新更像是一种实验技巧，即在训练中 Actor 的更新频率要低于 Critic 的更新频率。在学习过程中，Critic 是不断更新的，可以想象一下，假设在某个时刻 Actor 好不容易达到一个最高点，这个时候 Critic 又更新了，那么 Actor 的最高点就被打破了，这样一来 Actor 就会不断地追逐 Critic，这样就会造成误差的过分累积，进而导致 Actor 的训练不稳定，甚至可能会发散。因此，为了避免这种情况，我们就可以在训练中让 Actor 的更新频率低于 Critic 的更新频率，这样一来 Actor 的更新就会比较稳定，不会受到 Critic 的影响，从而提高算法的稳定性和收敛性。

举个不太恰当的例子，Critic 就好比领导，Actor 则好比员工，领导不断给员工下达目标，员工不断地去完成目标，如果领导的决策经常失误，那么员工就很容易像无头苍蝇一样不知道该完成哪些目标。因此，一个好的解决方式就是让领导学得比员工更快，领导学得更快了之后下达目标的失误率就会更低，这样员工就能够更好地完成目标了，从而提高整个团队的效率。在实践中，Actor 的更新频率一般要比 Critic 的更新频率低一个数量级，例如 Critic 每更新 10 次，Actor 只更新 1 次。

### 11.4.3 躁声正则

更准确地说，在原论文中不叫噪声正则，而是目标策略平滑正则化（ Target Policy Smoothing Regularization ），意思都是一样的，只是笔者在表述的时候做了简化。在延迟更新中，我们的主要思想是通过提高 Critic 的更新频率来减少值函数的估计误差，也就是“降低领导决策的失误率”。但是这样其实是治标不治本的做法，因为它只是让 Critic 带来的误差不要过分地影响到了Actor，而没有考虑改进 Critic 本身的稳定性。

因此，我们也可以给 Critic 引入一个噪声提高其抗干扰性，这样一来就可以在一定程度上提高 Critic 的稳定性，从而进一步提高算法的稳定性和收敛性。注意，这里的噪声是在 Critic 网络上引入的，而不是在输出动作上引入的，因此它跟 DDPG 算法中的噪声是不一样的。具体来说，我们可以在计算 TD 误差的时候，给目标值 y 加上一个噪声，并且为了让噪声不至于过大，还增加了一个裁剪（ clip ），如式 (11.5) 所示。

其中 表示均值为 0，方差为 的高斯噪声， 表示噪声， 表示裁剪函数，即将噪声裁剪到 的范围内，c 是一个超参数，用于控制噪声的大小。可以看到，这里噪声更像是一种正则化的方式，使得值函数更新更加平滑，因此笔者称之为噪声正则。

## 11.5 实战：DDPG 算法

同之前章节一样，本书在实战中将演示一些核心的代码，完整的代码请参考JoyRL代码仓库。

### 11.5.1 DDPG 伪代码

如图11-3所示， DDPG算法的训练方式其实更像DQN算法。注意在第 15步中DDPG算法将当前网络参数复制到目标网络的方式是软更新，即每次一点点地将参数复制到目标网络中，与之对应的是 DQN算法中的硬更新。软更新的好处是更加平滑缓慢，可以避免因权重更新过于迅速而导致的震荡，同时降低训练发散的风险。

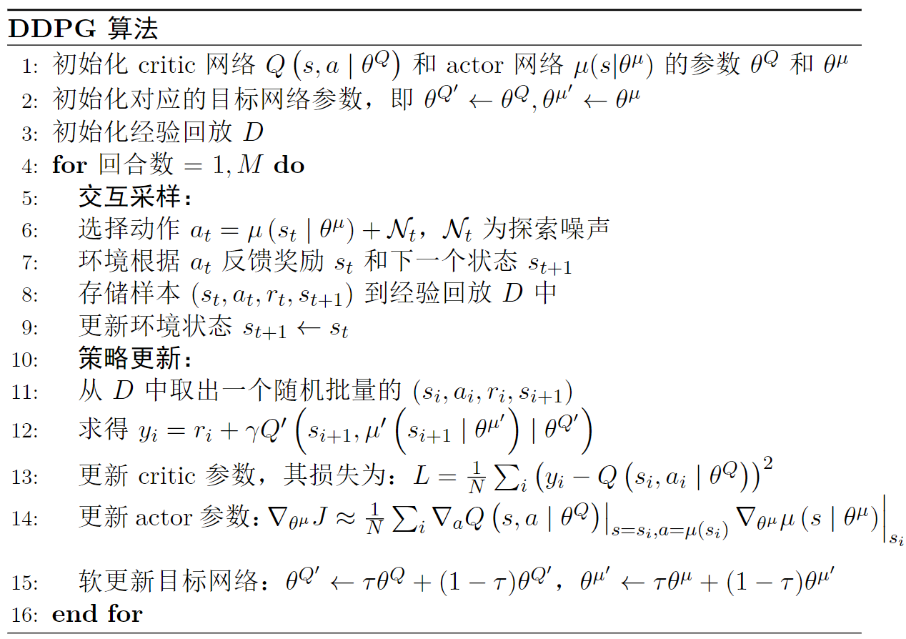


图11-3 DDPG算法伪代码

### 11.5.2 定义模型

如代码清单11-1所示，DDPG算法的模型结构跟Actor-Critic算法几乎是一样的，只是由于DDPG算法的Critic是Q函数，因此也需要将动作作为输入。除了模型之外，目标网络和经验回放的定义方式跟DQN算法一样，这里不做展开。

代码清单11-1 DDPG算法的Actor和Critic

1. import torch
2. import torch.nn as nn
3. import torch.nn.functional as F
4. class Actor(nn.Module):
5. def \_\_init\_\_(self, state\_dim, action\_dim, hidden\_dim = 256, init\_w=3e-3):
6. super(Actor, self).\_\_init\_\_()
7. self.linear1 = nn.Linear(state\_dim, hidden\_dim)
8. self.linear2 = nn.Linear(hidden\_dim, hidden\_dim)
9. self.linear3 = nn.Linear(hidden\_dim, action\_dim)
11. self.linear3.weight.data.uniform\_(-init\_w, init\_w)
12. self.linear3.bias.data.uniform\_(-init\_w, init\_w)
14. def forward(self, x):
15. x = F.relu(self.linear1(x))
16. x = F.relu(self.linear2(x))
17. x = torch.tanh(self.linear3(x)) # 输入0到1之间的值
18. return x
20. class Critic(nn.Module):
21. def \_\_init\_\_(self, state\_dim, action\_dim, hidden\_dim=256, init\_w=3e-3):
22. super(Critic, self).\_\_init\_\_()
24. self.linear1 = nn.Linear(state\_dim + action\_dim, hidden\_dim)
25. self.linear2 = nn.Linear(hidden\_dim, hidden\_dim)
26. self.linear3 = nn.Linear(hidden\_dim, 1)
27. # 随机初始化为较小的值
28. self.linear3.weight.data.uniform\_(-init\_w, init\_w)
29. self.linear3.bias.data.uniform\_(-init\_w, init\_w)
31. def forward(self, state, action):
32. # 按维数1拼接
33. x = torch.cat([state, action], 1)
34. x = F.relu(self.linear1(x))
35. x = F.relu(self.linear2(x))
36. x = self.linear3(x)
37. return x

### 11.5.3 动作采样

由于DDPG算法输出的是确定性策略，因此不需要像其他策略梯度算法那样，通过借助高斯分布来采样动作的概率分布，直接输出Actor的值即可，如代码清单11-2所示。

代码清单1-1 XXXXXXXX

1. class Agent:
2. def \_\_init\_\_(self):
3. pass
4. def update(self):
5. # 从经验回放中中随机采样一个批量的样本
6. state, action, reward, next\_state, done = self.memory.sample(self.batch\_size)
7. actor\_loss = self.critic(state, self.actor(state))
8. actor\_loss = - actor\_loss.mean()
9. next\_action = self.target\_actor(next\_state)
10. target\_value = self.target\_critic(next\_state, next\_action.detach())
11. expected\_value = reward + (1.0 - done) \* self.gamma \* target\_value
12. expected\_value = torch.clamp(expected\_value, -np.inf, np.inf)
13. actual\_value = self.critic(state, action)
14. critic\_loss = nn.MSELoss()(actual\_value, expected\_value.detach())
16. self.actor\_optimizer.zero\_grad()
17. actor\_loss.backward()
18. self.actor\_optimizer.step()
19. self.critic\_optimizer.zero\_grad()
20. critic\_loss.backward()
21. self.critic\_optimizer.step()
22. # 各自目标网络的参数软更新
23. for target\_param, param in zip(self.target\_critic.parameters(), self.critic.parameters()):
24. target\_param.data.copy\_(
25. target\_param.data \* (1.0 - self.tau) +
26. param.data \* self.tau
27. )
28. for target\_param, param in zip(self.target\_actor.parameters(), self.actor.parameters()):
29. target\_param.data.copy\_(
30. target\_param.data \* (1.0 - self.tau) +
31. param.data \* self.tau
32. )

核心代码到这里全部实现了，我们展示一下训练效果，如图11-4所示。

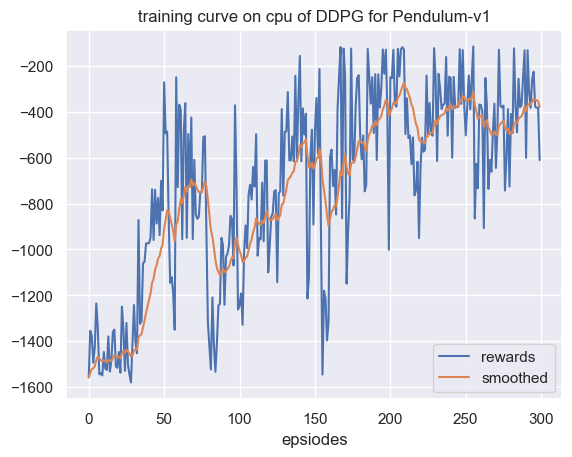


图11-4 Pendulum环境DDPG算法训练曲线

这里我们使用了一个具有连续动作空间的环境Pendulum，如图11-5所示。在该环境中，钟摆以随机位置开始，我们的目标是将其向上摆动，使其保持直立。

图表

描述已自动生成

图11-5 Pendulum环境演示

## 11.6实战：TD3 算法

如代码清单11-4所示， TD3算法只是在策略更新上与DDPG算法有所差异，其它地方基本相同。

代码清单1-1 XXXXXXXX

1. def update(self):
2. if len(self.memory) < self.explore\_steps: # 当经验回放中不满足一个批量时，不更新策略
3. return
4. state, action, reward, next\_state, done = self.memory.sample(self.batch\_size) # 从经验回放中随机采样一个批量的转移(transition)
5. # 将数据转换为tensor
6. state = torch.tensor(np.array(state), device=self.device, dtype=torch.float32)
7. action = torch.tensor(np.array(action), device=self.device, dtype=torch.float32)
8. next\_state = torch.tensor(np.array(next\_state), device=self.device, dtype=torch.float32)
9. reward = torch.tensor(reward, device=self.device, dtype=torch.float32).unsqueeze(1)
10. done = torch.tensor(done, device=self.device, dtype=torch.float32).unsqueeze(1)
11. noise = (torch.randn\_like(action) \* self.policy\_noise).clamp(-self.noise\_clip, self.noise\_clip) # 构造加入目标动作的噪声
12. # 计算加入了噪声的目标动作
13. next\_action = (self.actor\_target(next\_state) + noise).clamp(-self.action\_scale+self.action\_bias, self.action\_scale+self.action\_bias)
14. # 计算两个critic网络对t+1时刻的状态动作对的评分，并选取更小值来计算目标q值
15. target\_q1, target\_q2 = self.critic\_1\_target(next\_state, next\_action).detach(), self.critic\_2\_target(next\_state, next\_action).detach()
16. target\_q = torch.min(target\_q1, target\_q2)
17. target\_q = reward + self.gamma \* target\_q \* (1 - done)
18. # 计算两个critic网络对t时刻的状态动作对的评分
19. current\_q1, current\_q2 = self.critic\_1(state, action), self.critic\_2(state, action)
20. # 计算均方根损失
21. critic\_1\_loss = F.mse\_loss(current\_q1, target\_q)
22. critic\_2\_loss = F.mse\_loss(current\_q2, target\_q)
23. self.critic\_1\_optimizer.zero\_grad()
24. critic\_1\_loss.backward()
25. self.critic\_1\_optimizer.step()
26. self.critic\_2\_optimizer.zero\_grad()
27. critic\_2\_loss.backward()
28. self.critic\_2\_optimizer.step()
29. if self.sample\_count % self.policy\_freq == 0:
30. # 延迟策略更新，actor的更新频率低于critic
31. actor\_loss = -self.critic\_1(state, self.actor(state)).mean()
32. self.actor\_optimizer.zero\_grad()
33. actor\_loss.backward()
34. self.actor\_optimizer.step()
35. #目标网络软更新
36. for param, target\_param in zip(self.actor.parameters(), self.actor\_target.parameters()):
37. target\_param.data.copy\_(self.tau \* param.data + (1 - self.tau) \* target\_param.data)
38. for param, target\_param in zip(self.critic\_1.parameters(), self.critic\_1\_target.parameters()):
39. target\_param.data.copy\_(self.tau \* param.data + (1 - self.tau) \* target\_param.data)
40. for param, target\_param in zip(self.critic\_2.parameters(), self.critic\_2\_target.parameters()):
41. target\_param.data.copy\_(self.tau \* param.data + (1 - self.tau) \* target\_param.data)

同样我们展示一下训练效果，在合适的参数设置下它会比DDPG算法收敛的更快，如图11-6所示。

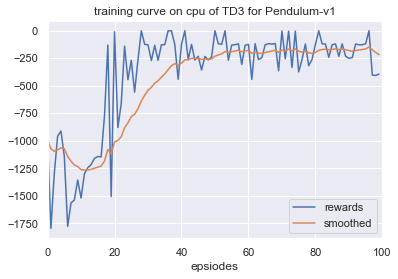


图11-4 Pendulum环境TD3算法训练曲线

## 11.7本章小结

本章主要介绍了强化学习中较为常用的一类算法，即DDPG和TD3算法，它们虽然在结构上被归类于Actor-Critic算法，但从原理上来说跟DQN算法更为接近，可以说是连续动作空间版本的DQN算法，且只适用于连续动作空间。可以说，在需要确定性策略且连续动作空间的前提下，这类算法将是比较稳定的基线算法，读者需要熟练掌握。

## 11.8练习题

1. DDPG算法是off-policy算法吗？为什么？

2. 软更新相比于硬更新的好处是什么？为什么不是所有的算法都用软更新？

3. 相比于DDPG算法，TD3算法做了哪些改进？请简要归纳。

4. TD3算法中Critic的更新频率一般要比Actor是更快还是更慢？为什么？