# 第8章 DQN算法进阶

本章将介绍一些基于DQN改进的一些算法。这些算法改进的角度各有不同，例如 Double DQN以及Dueling DQN等算法主要从网络模型层面改进，而 PER DQN 则从经验回放的角度来改进。尽管这些算法看起来各有不同，但是本质上都是通过提高预测的精度和控制过程中的探索度来改善DQN算法的性能。并且这些算法用到的技巧也都是比较通用的，读者可以根据自己的需求进行灵活的组合。

## 8.1 Double DQN 算法

Double DQN算法[[1]](#footnote-1)是谷歌DeepMind于2015年2月提出的一篇论文，主要贡献是通过引入两个网络用于解决值过估计（overestimate）的问题。顺便说一句，这里两个网络其实跟前面 DQN算法讲的目标网络是类似的，读者可能会产生混淆。

实际上它们之间的关系是这样的，我们知道DQN分别于2013和2015年提出了两个版本，后者就是目前较为成熟的Nature DQN版本，前者就是单纯在Q-learning算法基础上引入了深度网络而没有额外的技巧。而在中间的过程中Double DQN算法被提出，因此Nature DQN在发表的时候也借鉴了Double DQN的思想，所以才会有目标网络的概念。尽管如此， Double DQN算法仍然有其独特的地方，因此我们还是将其单独拿出来讲。

先回顾一下 DQN算法的更新公式，如式 (8.1) 所示。

其中是估计值，注意这里的指的是目标网络。这个意思就是直接拿目标网络中各个动作对应的最大的值来当作估计值，这样一来就会存在过估计的问题。为了解决这个问题， Double DQN算法提出了一个很简单的思路，就是现在当前网络中找出最大值对应的动作，然后再将这个动作代入到目标网络中去计算值，如式 (8.2) 所示。

然后将这个找出来的动作代入到目标网络里面去计算目标的值，进而计算估计值，如式 (8.3) 所示。

这样做相当于是把动作选择和动作评估这两个过程分离开来，从而减轻了过估计问题。为了方便读者理解，我们接着用皇帝与大臣的例子来举例说明为什么Double DQN算法能够估计得更准确。我们知道在Nature DQN算法中策略网络直接与环境交互相当于大臣们搜集情报，然后定期更新到目标网络的过程相当于大臣向皇帝汇报然后皇帝做出最优决策。

在Nature DQN算法中，大臣是不管好的还是坏的情报都会汇报给皇帝的，而在Double DQN算法中大臣会根据自己的判断将自己认为最优的情报汇报给皇帝，即先在策略网络中找出最大值对应的动作。这样一来皇帝这边得到的情报就更加精简并且质量更高了，以便于皇帝做出更好的判断和决策，也就是估计得更准确了。

注意，虽然Double DQN算法和Nature DQN算法都用了两个网络，但实际上Double DQN 算法训练方式是略有不同的。Double DQN并不是每隔C步复制参数到目标网络，而是每次随机选择其中一个网络选择动作进行更新。假设两个网络分别为和 ，那么在更新的时候就用把当作目标网络估计动作值，同时也是用来选择动作的，如式 (8.4) 所示，反之亦然。

但其实这种训练方式的效果跟单纯每隔C步复制参数到目标网络的方式差不多，而且后者更加简单，所以实践中一般都是采用后者。

## 8.2 Dueling DQN 算法

在Double DQN算法中我们是通过改进目标Q值的计算来优化算法的，而在 Dueling DQN算法[[2]](#footnote-2)中则是通过优化神经网络的结构来优化算法的。

回顾我们在DQN算法所使用的最基础的网络结构，如图8-1所示，它是一个全连接网络，包含一个输入层、一个隐藏层和输出层。输入层的维度为状态的维度，输出层的维度为动作的维度。



图8-1 DQN网络结构

而Dueling DQN算法中则是在输出层之前分流（ dueling）出了两个层，如图8-2所示，一个是优势层（advantage layer），用于估计每个动作带来的优势，输出维度为动作数一个是价值层（value layer），用于估计每个状态的价值，输出维度为1。

标志上写着字

描述已自动生成

图8-2 Dueling DQN网络结构

在 DQN 算法中我们用表示 一个Q网络，而在这里优势层可以表示为 ，这里 表示共享隐藏层的参数，表示优势层自己这部分的参数，相应地价值层可以表示为 。这样 Dueling DQN算法中网络结构可表示为式 (8.5) 。

去掉这里的价值层即优势层就是普通的网络了，另外我们会对优势层做一个中心化处理，即减掉均值，如式 (8.6)所示。

其实Dueling DQN的网络结构跟我们后面要讲的 Actor-Critic算法是类似的，这里优势层相当于Actor，价值层相当于Critic，不同的是在Actor-Critic算法中Actor和Critic是独立的两个网络，而在这里是合在一起的，在计算量以及拓展性方面都完全不同，具体我们会在后面的Actor-Critic算法对应章节中展开。

总的来讲， Dueling DQN算法在某些情况下相对于DQN是有好处的，因为它分开评估每个状态的价值以及某个状态下采取某个动作的Q值。当某个状态下采取一些动作对最终的回报都没有多大影响时，这个时候Dueling DQN这种结构的优越性就体现出来了。或者说，它使得目标值更容易计算，因为通过使用两个单独的网络，我们可以隔离每个网络输出上的影响，并且只更新适当的子网络，这有助于降低方差并提高学习鲁棒性。

## 8.3 Noisy DQN 算法

Noisy DQN算法[[3]](#footnote-3)也是通过优化网络结构的方法来提升DQN算法的性能，但与Dueling DQN算法不同的是，它的目的并不是为了提高Q值的估计，而是增强网络的探索能力。

从Q-learning算法开始，我们就讲到了探索-利用平衡的问题，常见的策略是从智能体与环境的交互过程改善探索能力，以避免陷入局部最优解。而在深度强化学习中，由于引入了深度学习，深度学习本身也会因为网络模型限制或者梯度下降方法陷入局部最优解问题。也就是说，深度强化学习既要考虑与环境交互过程中的探索能力，也要考虑深度模型本身的探索能力，从而尽量避免陷入局部最优解的困境之中，这也是为什么经常有人会说强化学习比深度学习更难“炼丹”的原因之一。

回归正题， Noisy DQN算法其实是在DQN算法基础上在神经网络中引入了噪声层来提高网络性能的，即将随机性应用到神经网络中的参数或者说权重，增加了Q网络对于状态和动作空间的探索能力，从而提高收敛速度和稳定性。在实践上也比较简单，就是通过添加随机性参数到神经网络的线性层，对应的Q值则可以表示为，注意不要把这里的跟策略中的混淆了。虽然都叫做epsilon，但这里 是由高斯分布生成的总体分类噪声参数。

## 8.4 PER DQN 算法

在 DQN算法章节中我们讲到经验回放，从另一个角度来说也是为了优化深度网络中梯度下降的方式，或者说网络参数更新的方式。在本节要讲的PER DQN算法[[4]](#footnote-4)中，进一步优化了经验回放的设计从而提高模型的收敛能力和鲁棒性。PER可以翻译为优先经验回放（prioritized experience replay），跟数据结构中优先队列与普通队列一样，会在采样过程中赋予经验回放中样本的优先级。

具体以什么为依据来为经验回放中的样本赋予不同优先级呢？答案是TD误差。TD误差最开始我们是在时序差分方法的提到的，广义的定义是值函数（包括状态价值函数和动作价值函数）的估计值与实际值之差，在DQN算法中就是目标网络计算的Q值和策略网络（当前网络）计算的值之差，也就是DQN网络中损失函数的主要构成部分。

我们每次从经验回访中取出一个批量的样本，进而计算TD误差一般是不同的，对于DQN网络反向传播的作用也是不同的。**TD误差越大，损失函数的值也越大，对于反向传播的作用也就越大**。这样一来如果TD误差较大的样本更容易被采到的话，那么我们的算法也会更加容易收敛。因此我们只需要设计一个经验回放，根据经验回放中的每个样本计算出的TD误差赋予对应的优先级，然后在采样的时候取出优先级较大的样本。

原理听上去比较简单，但具体如何实现呢？在实践中，我们通常用SumTree这样的二叉树结构来实现。这里建议没有了解过数据结构或者二叉树的读者先花个十几分钟的时间快速了解一下二叉树的基本概念，比如根节点、叶节点、父节点与子节点等等。

如图8-3所示，每个父节点的值等于左右两个子节点值之和。在强化学习中，所有的样本只保存在最下面的叶子节点中，并且除了保存样本数据之外，还会保存对应的优先级，即对应叶子节点中的值（例如图中的 31,13,14以及8等，也对应样本的TD误差）。并且根据叶子节点的值，我们从0开始依次划分采样区间。然后在采样中，例如这里根节点值为66，那么我们就可以在 [0,66) 这个区间均匀采样，采样到的值落在哪个区间中，就说明对应的样本就是我们要采样的样本。例如我们采样到了25这个值，即对应区间 [0,31) ，那么我们就采样到了第一个叶子节点对应的样本。

注意到，第一个样本对应的区间也是最长的，这意味着第一个样本的优先级最高，也就是TD误差最大，反之第四个样本的区间最短，优先级也最低。这样一来，我们就可以通过采样来实现优先经验回放的功能。

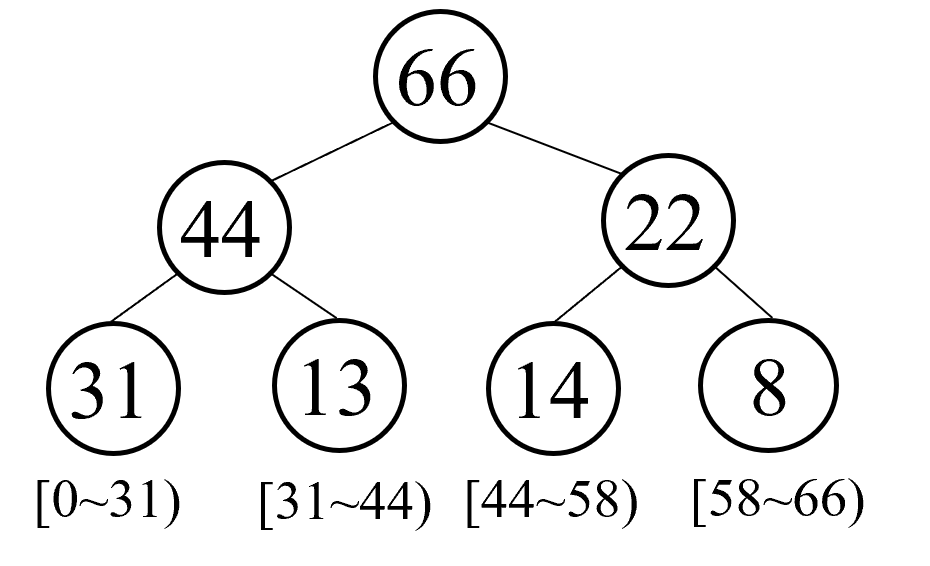


图8-3 SumTree结构

每个叶节点的值就是对应样本的TD误差（例如途中的）。我们可以通过根节点的值来计算出每个样本的TD误差占所有样本TD误差的比例，这样就可以根据比例来采样样本。在实际的实现中，我们可以将每个叶节点的值设置为一个元组，其中包含样本的TD误差和样本的索引，这样就可以通过索引来找到对应的样本。具体如何用Python类来实现SumTree结构，读者可以参考后面的实战内容。

尽管 SumTree结构可以实现优先经验回放的功能。然而直接使用 TD误差作为优先级存在一些问题。首先，考虑到算法效率问题，我们在每次更新时不会把经验回放中的所有样本都计算 TD误差并更新对应的优先级，而是只更新当前取到的一定批量的样本。这样一来，每次计算的 TD误差是对应之前的网络，而不是当前待更新的网络。

换句话说，如果某批量样本的TD误差较低，只能说明它们对于之前的网络来说“信息量”不大，但不能说明对当前的网络“信息量”不大，因此单纯根据 TD误差进行优先采样有可能会错过对当前网络“信息量”更大的样本。其次，被选中样本的 TD误差会在当前更新后下降，然后优先级会排到后面去，下次这些样本就不会被选中，这样来来回回都是那几个样本，很容易出现“旱的旱死，涝的涝死”的情况，导致过拟合。

为了解决上面提到的两个问题，我们首先引入**随机优先级采样**（stochastic prioritization）的技巧。即在每次更新时，我们不再是直接采样TD误差最大的样本，而是定义一个采样概率，如式 (8.7) 所示。

其中，是样本的优先级，是一个超参数，用于调节优先采样的程序，通常在 (0,1) 的区间内。当时，采样概率为均匀分布；当时，采样概率为优先级的线性分布。同时，即使对于最低优先级的样本，我们也不希望它们的采样概率为0，因此我们可以在优先级上加上一个常数，即式 (8.8) 。

其中，是样本的TD误差。当然，我们也可以使用其他的优先级计算方式，如式 (8.9) 所示。

其中是样本的优先级排名，这种方式也能保证每个样本的采样概率都不为0 ，但在实践中，我们更倾向于直接增加一个常数的方式。

除了随机优先级采样之外，我们还引入了另外一个技巧，在讲解该技巧之前，我们需要简单了解一下**重要性采样**，这个概念在后面的PPO算法也会用到，读者需要重点掌握。重要性采样（importance sampling）是一种用于估计某一分布性质的方法，它的基本思想是，我们可以通过与待估计分布不同的另一个分布中采样，然后通过采样样本的权重来估计待估计分布的性质，数学表达式如式 (8.10) 所示。

其中是待估计分布，是采样分布，是待估计分布的性质。在前面我们讲到，每次计算的 TD误差是对应之前的网络，而不是当前待更新的网络。也就是说，我们已经从之前的网络中采样了一批样本，也就是已知，然后只要找到之前网络分布与当前网络分布之前的权重，就可以利用重要性采样来估计出当前网络的性质。我们可以定义权重为式 (8.11) 。

其中，是经验回放中的样本数量，是样本的采样概率。同时，为了避免出现权重过大或过小的情况，我们可以对权重进行归一化处理，如式 (8.12) 所示。

注意到，我们引入了一个超参数，用于调节重要性采样的程度。当时，重要性采样的权重为1，即不考虑重要性采样；当时，重要性采样的权重为，即完全考虑重要性采样。在实践中，我们希望从0开始，随着训练步数的增加而逐渐增加，以便更好地利用重要性采样，这就是热偏置（annealing the bias）的思想。数学表达式如式 (8.13) 所示。

其中，是每个训练步数对应的的增量。在实践中，我们可以将设置为一个很小的常数，如 0.0001。这样一来，我们就可以在训练刚开始时，使用随机优先级采样，以便更快地收敛；在训练后期，使用重要性采样，以便更好地利用经验回放中的样本。

## 8.5实战：Double DQN 算法

由于本章都是基于DQN 改进的算法，整体训练方式跟 DQN 是一样的，也就是说伪代码基本都是一致的，因此不再赘述，只讲算法的改进部分。而 Double DQN 算法跟 DQN算法的区别在于目标值的计算方式，如代码清单 8-1 所示。

代码清单8-1 Double DQN目标值的计算

1. # 计算当前网络的Q值，即Q(s\_t+1|a)
2. next\_q\_values = self.policy\_net(next\_states)
3. # 计算目标网络的Q值，即Q'(s\_t+1|a)
4. next\_target\_q\_values = self.target\_net(next\_states)
5. # 计算 Q'(s\_t+1|a=argmax Q(s\_t+1|a))
6. next\_target\_q\_values = next\_target\_q\_values.gather(1, torch.max(next\_q\_values, 1)[1].unsqueeze(1))

最后与 DQN算法相同，可以得到 Double DQN算法在CartPole环境下的训练结果，如图 8-4 所示，完整的代码可以参考本书的代码仓库。

形状

中度可信度描述已自动生成

图8-4 CartPole 环境 Double DQN 算法训练曲线

与 DQN算法的训练曲线对比可以看出，在实践上Double DQN算法的效果并不一定比 DQN算法好，比如在这个环境下其收敛速度反而更慢了，因此读者需要多多实践才能摸索并体会到这些算法适合的场景。

## 8.6实战：Dueling DQN 算法

Dueling DQN算法主要是改了网络结构，其他地方跟DQN算法是一模一样的，如代码清单8-2所示。

代码清单8-2 Dueling DQN 网络结构

* 1. class DuelingQNetwork(nn.Module):
  2. def \_\_init\_\_(self, state\_dim, action\_dim,hidden\_dim=128):
  3. super(DuelingQNetwork, self).\_\_init\_\_()
  4. # 隐藏层
  5. self.hidden\_layer = nn.Sequential(
  6. nn.Linear(state\_dim, hidden\_dim),
  7. nn.ReLU()
  8. )
  9. # 优势层
  10. self.advantage\_layer = nn.Sequential(
  11. nn.Linear(hidden\_dim, hidden\_dim),
  12. nn.ReLU(),
  13. nn.Linear(hidden\_dim, action\_dim)
  14. )
  15. # 价值层
  16. self.value\_layer = nn.Sequential(
  17. nn.Linear(hidden\_dim, hidden\_dim),
  18. nn.ReLU(),
  19. nn.Linear(hidden\_dim, 1)
  20. )
  22. def forward(self, state):
  23. x = self.hidden\_layer(state)
  24. advantage = self.advantage\_layer(x)
  25. value = self.value\_layer(x)
  26. return value + advantage - advantage.mean() # Q(s,a) = V(s) + A(s,a) - mean(A(s,a))

最后我们展示一下它在CartPole环境下的训练结果，如图8-5所示，完整的代码同样可以参考本书的代码仓库。

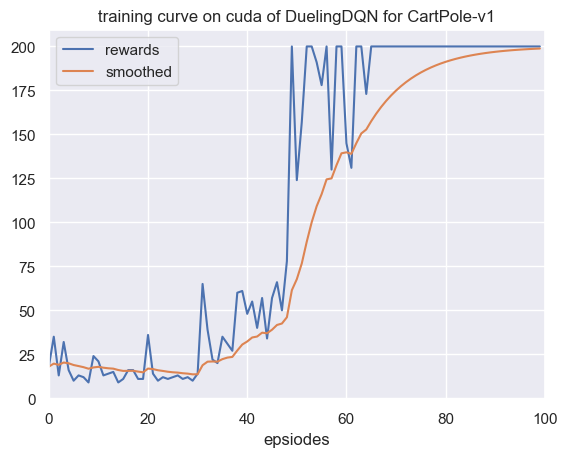


图8-5 CartPole 环境 Dueling DQN 算法训练曲线

由于环境比较简单，暂时还看不出来 Dueling DQN算法的优势，但是在复杂的环境下，比如 Atari游戏中， Dueling DQN算法的效果就会比DQN算法好很多，读者可以在 JoyRL 仓库中找到更复杂环境下的训练结果便于更好地进行对比。

## 8.7实战：Noisy DQN 算法

Noisy DQN算法的核心思想是将 DQN算法中的线性层替换成带有噪声的线性层，如代码清单 8-3所示。

代码清单8-3带有噪声的线性层网络

* 1. class NoisyLinear(nn.Module):
  2. '''在Noisy DQN中用NoisyLinear层替换普通的nn.Linear层
  3. '''
  4. def \_\_init\_\_(self, input\_dim, output\_dim, std\_init=0.4):
  5. super(NoisyLinear, self).\_\_init\_\_()
  6. self.input\_dim = input\_dim
  7. self.output\_dim = output\_dim
  8. self.std\_init = std\_init
  9. self.weight\_mu = nn.Parameter(torch.empty(output\_dim, input\_dim))
  10. self.weight\_sigma = nn.Parameter(torch.empty(output\_dim, input\_dim))
  11. # 将一个 tensor 注册成 buffer，使得这个 tensor 不被当做模型参数进行优化。
  12. self.register\_buffer('weight\_epsilon', torch.empty(output\_dim, input\_dim))
  14. self.bias\_mu = nn.Parameter(torch.empty(output\_dim))
  15. self.bias\_sigma = nn.Parameter(torch.empty(output\_dim))
  16. self.register\_buffer('bias\_epsilon', torch.empty(output\_dim))
  18. self.reset\_parameters() # 初始化参数
  19. self.reset\_noise() # 重置噪声
  21. def forward(self, x):
  22. if self.training:
  23. weight = self.weight\_mu + self.weight\_sigma \* self.weight\_epsilon
  24. bias = self.bias\_mu + self.bias\_sigma \* self.bias\_epsilon
  25. else:
  26. weight = self.weight\_mu
  27. bias = self.bias\_mu
  28. return F.linear(x, weight, bias)
  30. def reset\_parameters(self):
  31. mu\_range = 1 / self.input\_dim \*\* 0.5
  32. self.weight\_mu.data.uniform\_(-mu\_range, mu\_range)
  33. self.weight\_sigma.data.fill\_(self.std\_init / self.input\_dim \*\* 0.5)
  34. self.bias\_mu.data.uniform\_(-mu\_range, mu\_range)
  35. self.bias\_sigma.data.fill\_(self.std\_init / self.output\_dim \*\* 0.5)
  37. def reset\_noise(self):
  38. epsilon\_in = self.\_scale\_noise(self.input\_dim)
  39. epsilon\_out = self.\_scale\_noise(self.output\_dim)
  40. self.weight\_epsilon.copy\_(epsilon\_out.ger(epsilon\_in))
  41. self.bias\_epsilon.copy\_(self.\_scale\_noise(self.output\_dim))
  43. def \_scale\_noise(self, size):
  44. x = torch.randn(size)
  45. x = x.sign().mul(x.abs().sqrt())
  46. return x

根据写好的NoisyLinear层，我们可以在DQN 算法中将普通的线性层替换为 NoisyLinear 层，如代码清单 8-4 所示。

代码清单8-4带噪声层的全连接网络

1. class NoisyQNetwork(nn.Module):
2. def \_\_init\_\_(self, state\_dim, action\_dim, hidden\_dim=128):
3. super(NoisyQNetwork, self).\_\_init\_\_()
4. self.fc1 = nn.Linear(state\_dim, hidden\_dim)
5. self.noisy\_fc2 = NoisyLinear(hidden\_dim, hidden\_dim)
6. self.noisy\_fc3 = NoisyLinear(hidden\_dim, action\_dim)
8. def forward(self, x):
9. x = F.relu(self.fc1(x))
10. x = F.relu(self.noisy\_fc2(x))
11. x = self.noisy\_fc3(x)
12. return x
13. def reset\_noise(self):
14. self.noisy\_fc2.reset\_noise()
15. self.noisy\_fc3.reset\_noise()

注意在训练过程中，我们需要在每次更新后重置噪声，这样有助于提高训练的稳定性，更多细节请参考 JoyRL源码。另外，我们也可以直接利用torchrl模块中中封装好的 NoisyLinear 层来构建 Noisy Q 网络，跟我们自己定义的功能是一样的，如代码清单 8-5 所示。

代码清单8-5使用 torchrl 模块构造的 Noisy Q 网络

1. import torchrl
2. class NoisyQNetwork(nn.Module):
3. def \_\_init\_\_(self, state\_dim, action\_dim, hidden\_dim=128):
4. super(NoisyQNetwork, self).\_\_init\_\_()
5. self.fc1 = nn.Linear(state\_dim, hidden\_dim)
6. self.noisy\_fc2 = torchrl.NoisyLinear(hidden\_dim, hidden\_dim,std\_init=0.1)
7. self.noisy\_fc3 = torchrl.NoisyLinear(hidden\_dim, action\_dim,std\_init=0.1)
9. def forward(self, x):
10. x = F.relu(self.fc1(x))
11. x = F.relu(self.noisy\_fc2(x))
12. x = self.noisy\_fc3(x)
13. return x
14. def reset\_noise(self):
15. self.noisy\_fc2.reset\_noise()
16. self.noisy\_fc3.reset\_noise()

同样我们展示一下它在 CartPole 环境下的训练结果，如图 8-6所示。

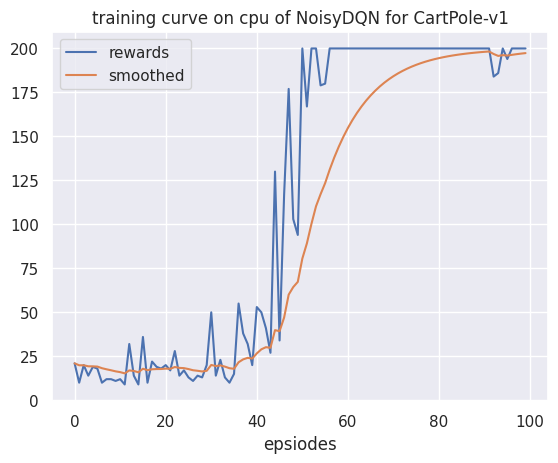


图8-6 CartPole 环境 Noisy DQN 算法训练曲线

## 8.8实战：PER DQN 算法

### 8.8.1伪代码

PER DQN算法的核心看起来简单，就是把普通的经验回放改进成了优先级经验回放，但是实现起来却比较复杂，因为我们需要实现一个SumTree结构，并且在模型更新的时候也需要一些额外的操作，因此我们先从伪代码开始，如图 8-7 所示。

图表

描述已自动生成

图8-6 CartPole 环境 Noisy DQN 算法训练曲线

### 8.8.2 SumTree结构

如代码清单8-6 所示，我们可以先实现SumTree 结构。

代码清单8-6 SumTree 结构

1. class SumTree:
2. def \_\_init\_\_(self, capacity):
3. self.capacity = capacity
4. self.tree = np.zeros(2 \* capacity - 1)
5. self.data = np.zeros(capacity, dtype=object) # 存储样本
6. self.write\_idx = 0 # 写入样本的索引
7. self.count = 0 # 当前存储的样本数量
9. def add(self, priority, exps):
10. ''' 添加一个样本到叶子节点，并更新其父节点的优先级
11. '''
12. idx = self.write\_idx + self.capacity - 1 # 样本的索引
13. self.data[self.write\_idx] = exps # 写入样本
14. self.update(idx, priority) # 更新样本的优先级
15. self.write\_idx = (self.write\_idx + 1) % self.capacity # 更新写入样本的索引
16. if self.count < self.capacity:
17. self.count += 1
19. def update(self, idx, priority):
20. ''' 更新叶子节点的优先级，并更新其父节点的优先级
21. Args:
22. idx (int): 样本的索引
23. priority (float): 样本的优先级
24. '''
25. diff = priority - self.tree[idx] # 优先级的差值
26. self.tree[idx] = priority
27. while idx != 0:
28. idx = (idx - 1) // 2
29. self.tree[idx] += diff
31. def get\_leaf(self, v):
32. ''' 根据优先级的值采样对应区间的叶子节点样本
33. '''
34. idx = 0
35. while True:
36. left = 2 \* idx + 1
37. right = left + 1
38. if left >= len(self.tree):
39. break
40. if v <= self.tree[left]:
41. idx = left
42. else:
43. v -= self.tree[left]
44. idx = right
45. data\_idx = idx - self.capacity + 1
46. return idx, self.tree[idx], self.data[data\_idx]
47. def get\_data(self, indices):
48. return [self.data[idx - self.capacity + 1] for idx in indices]
50. def total(self):
51. ''' 返回所有样本的优先级之和，即根节点的值
52. '''
53. return self.tree[0]
55. def max\_prior(self):
56. ''' 返回所有样本的最大优先级
57. '''
58. return np.max(self.tree[self.capacity-1:self.capacity+self.write\_idx-1])

其中，除了需要存放各个节点的值（tree）之外，我们需要定义要给 data 来存放叶子节点的样本。此外， add 函数用于添加一个样本到叶子节点，并更新其父节点的优先级； update 函数用于更新叶子节点的优先级，并更新其父节点的优先级； get\_leaf 函数用于根据优先级的值采样对应区间的叶子节点样本； get\_data 函数用于根据索引获取对应的样本； total 函数用于返回所有样本的优先级之和，即根节点的值； max\_prior 函数用于返回所有样本的最大优先级。

### 8.8.3优先级经验回放

基于 SumTree 结构，并结合优先级采样和重要性采样的技巧，如代码清单 8-7 所示。

代码清单8-7优先级经验回放结构

1. class PrioritizedReplayBuffer:
2. def \_\_init\_\_(self, cfg):
3. self.capacity = cfg.buffer\_size
4. self.alpha = cfg.per\_alpha # 优先级的指数参数，越大越重要，越小越不重要
5. self.epsilon = cfg.per\_epsilon # 优先级的最小值，防止优先级为0
6. self.beta = cfg.per\_beta # importance sampling的参数
7. self.beta\_annealing = cfg.per\_beta\_annealing # beta的增长率
8. self.tree = SumTree(self.capacity)
9. self.max\_priority = 1.0
11. def push(self, exps):
12. ''' 添加样本
13. '''
14. priority = self.max\_priority if self.tree.total() == 0 else self.tree.max\_prior()
15. self.tree.add(priority, exps)
17. def sample(self, batch\_size):
18. ''' 采样一个批量样本
19. '''
20. indices = [] # 采样的索引
21. priorities = [] # 采样的优先级
22. exps = [] # 采样的样本
23. segment = self.tree.total() / batch\_size
24. self.beta = min(1.0, self.beta + self.beta\_annealing)
25. for i in range(batch\_size):
26. a = segment \* i
27. b = segment \* (i + 1)
28. p = np.random.uniform(a, b) # 采样一个优先级
29. idx, priority, exp = self.tree.get\_leaf(p) # 采样一个样本
30. indices.append(idx)
31. priorities.append(priority)
32. exps.append(exp)
33. # 重要性采样, weight = (N \* P(i)) ^ (-beta) / max\_weight
34. sample\_probs = np.array(priorities) / self.tree.total()
35. weights = (self.tree.count \* sample\_probs) \*\* (-self.beta) # importance sampling的权重
36. weights /= weights.max() # 归一化
37. indices = np.array(indices)
38. return zip(\*exps), indices, weights
40. def update\_priorities(self, indices, priorities):
41. ''' 更新样本的优先级
42. '''
43. priorities = np.abs(priorities) # 取绝对值
44. for idx, priority in zip(indices, priorities):
45. # 控制衰减的速度, priority = (priority + epsilon) ^ alpha
46. priority = (priority + self.epsilon) \*\* self.alpha
47. priority = np.minimum(priority, self.max\_priority)
48. self.tree.update(idx, priority)
49. def \_\_len\_\_(self):
50. return self.tree.count

我们可以看到，优先级经验回放的核心是 SumTree，它可以在 O(log N) 的时间复杂度内完成添加、更新和采样操作。在实践中，我们可以将经验回放的容量设置为，并将 alpha 设置为 0.6， epsilon 设置为 0.01，beta设置为 0.4，beta\_annealing 设置为 0.0001。当然我们也可以利用 Python 队列的方式实现优先级经验回放，形式上会更加简洁，并且在采样的时候减少了for 循环的操作，会更加高效，如代码清单 8-8 所示。

代码清单8-8基于队列实现优先级经验回放

1. class PrioritizedReplayBufferQue:
2. def \_\_init\_\_(self, cfg):
3. self.capacity = cfg.buffer\_size
4. self.alpha = cfg.per\_alpha # 优先级的指数参数，越大越重要，越小越不重要
5. self.epsilon = cfg.per\_epsilon # 优先级的最小值，防止优先级为0
6. self.beta = cfg.per\_beta # importance sampling的参数
7. self.beta\_annealing = cfg.per\_beta\_annealing # beta的增长率
8. self.buffer = deque(maxlen=self.capacity)
9. self.priorities = deque(maxlen=self.capacity)
10. self.count = 0 # 当前存储的样本数量
11. self.max\_priority = 1.0
12. def push(self,exps):
13. self.buffer.append(exps)
14. self.priorities.append(max(self.priorities, default=self.max\_priority))
15. self.count += 1
16. def sample(self, batch\_size):
17. priorities = np.array(self.priorities)
18. probs = priorities/sum(priorities)
19. indices = np.random.choice(len(self.buffer), batch\_size, p=probs)
20. weights = (self.count\*probs[indices])\*\*(-self.beta)
21. weights /= weights.max()
22. exps = [self.buffer[i] for i in indices]
23. return zip(\*exps), indices, weights
24. def update\_priorities(self, indices, priorities):
25. priorities = np.abs(priorities)
26. priorities = (priorities + self.epsilon) \*\* self.alpha
27. priorities = np.minimum(priorities, self.max\_priority).flatten()
28. for idx, priority in zip(indices, priorities):
29. self.priorities[idx] = priority
30. def \_\_len\_\_(self):
31. return self.count

最后，我们可以将优先级经验回放和 DQN 结合起来，实现一个带有优先级的DQN算法，并展示它在 CartPole 环境下的训练结果，如图 8-7 所示。

图表

描述已自动生成

图8-7 CartPole 环境 PER DQN 算法训练曲线

## 8.9 本章小结

本章主要讲解了DQN 的一些改进算法，主要解决 Q 值过估计、探索策略差等问题，其中有些技巧是比较通用的，例如 Noisy DQN算法中再神经网络引入噪声来提高探索策略。读者在学习的过程中，一定要注意技巧本身的使用方式与泛用性，而不是作为单独的算法来看待。

## 8.10 练习题

1. DQN算法为什么会产生Q值的过估计问题？

2. 同样是提高探索，Noisy DQN和-greedy策略有什么区别？

1. Hasselt H V , Guez A , Silver D .Deep Reinforcement Learning with Double Q-learning[J].Computer ence, 2015.DOI:10.48550/arXiv.1509.06461. [↑](#footnote-ref-1)
2. Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M. & de Freitas, N. (2015). Dueling Network Architectures for Deep Reinforcement Learning.s [↑](#footnote-ref-2)
3. Fortunato M , Azar M G , Piot B ,et al.Noisy Networks for Exploration[J]. 2017.DOI:10.48550/arXiv.1706.10295. [↑](#footnote-ref-3)
4. Schaul T , Quan J , Antonoglou I ,et al.Prioritized Experience Replay[J].Computer Science, 2015.DOI:10.48550/arXiv.1511.05952. [↑](#footnote-ref-4)