# 第8章 DQN 算法进阶

本章开始介绍一些基于 DQN 改进的一些算法，由于本章的算法都是网络模型、经验回放等层面的改进，整体的训练模式跟 DQN 算法是一模一样的，因此出于简洁考虑，本章算法不再贴出对应的伪代码。另一方面来讲，建议读者在阅读本章之前，牢牢掌握 DQN 算法的原理。

## 8.1 Double DQN 算法

Double DQN 算法[[1]](#footnote-1)是谷歌 DeepMind 于 2015 年 12 月提出的一篇解决 $Q$ 值过估计（overestimate）的论文。

回忆一下 DQN 算法的更新公式，如式（8.1）：

其中 是估计值，注意这里的 是目标网络（DQN 算法中有两个网络，一个是目标网络，一个是当前网络或者说策略网络）。

在 Double DQN 算法中则不再是直接在目标网络中寻找各动作最大的 Q 值，而是现在策略网络中找出最大 Q 值对应的动作，如式（8.2）：

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 Q 值，进而计算估计值，如式（8.3）：

到这里我们就讲完了 Double DQN 算法的核心，在原论文中还通过拟合曲线实验证明了过估计是真实存在且对实验结果有重大影响等相关细节，感兴趣的读者深入研究。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 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 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 Q 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。

Double DQN 算法核心代码如代码清单8-1所示，完整代码参考 JoyRL ：

代码清单8-1 Double DQN 算法核心代码

# 计算当前Q(s\_t|a=a\_t)，即Q网络的输出，这里的gather函数的作用是根据action\_batch中的值，从Q网络的输出中选出对应的Q值

q\_value\_batch = self.policy\_net(state\_batch).gather(dim=1, index=action\_batch) # shape(batchsize,1)

# 计算 Q(s\_t+1|a)

next\_q\_value\_batch = self.policy\_net(next\_state\_batch)

# 计算 Q'(s\_t+1|a)，也是与DQN不同的地方

next\_target\_value\_batch = self.target\_net(next\_state\_batch)

# 计算 Q'(s\_t+1|a=argmax Q(s\_t+1|a))

next\_target\_q\_value\_batch = next\_target\_value\_batch.gather(1, torch.max(next\_q\_value\_batch, 1)[1].unsqueeze(1)) # shape(batchsize,1)

整体代码跟 DQN 算法基本一致，只是在计算目标 Q 值的时候多了一步，即当前策略网络中找出最大 Q 值对应的动作，然后再在目标网络中计算目标 Q 值。

细心的读者可能会发现，这里似乎并没有体现出 Double DQN 算法中的 “Double”是在哪里。Double DQN 伪代码如图 8.1 所示

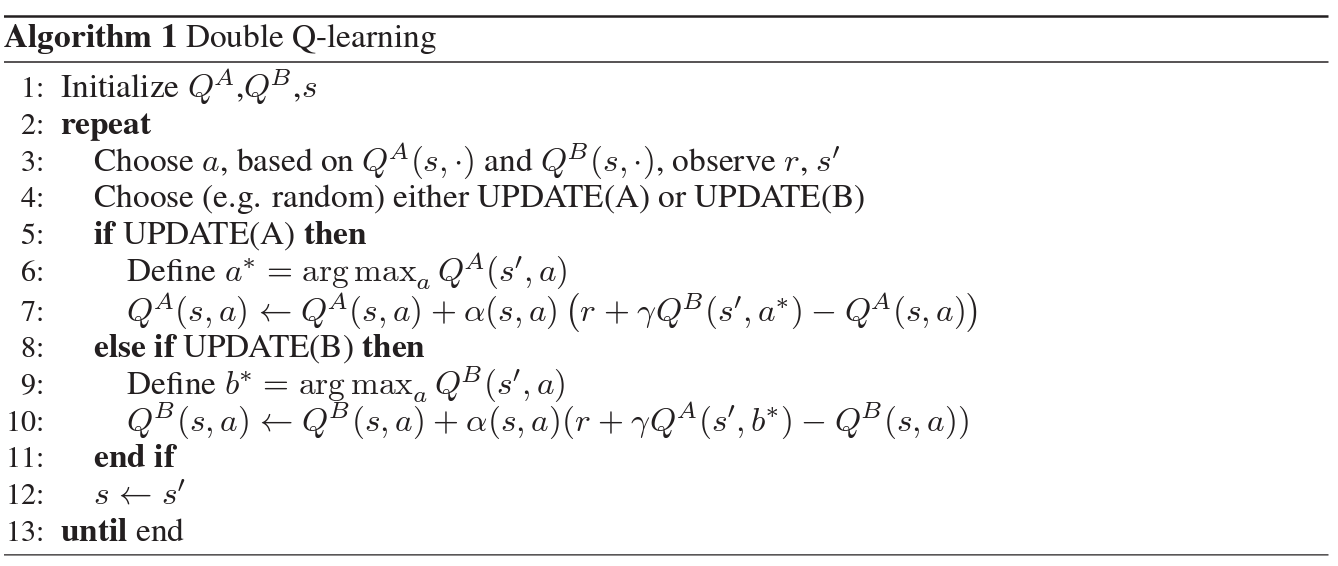


图8.1 Double DQN 伪代码

实际上在原论文中是有这样的描述的，如图 8.1 所示，它定义了两个网络，在更新的时候随机选择其中一个网络进行更新，其实跟这里的策略网络和目标网络是一样的。这是因为 Double DQN 算法论文比 Nature DQN 算法论文发表的时间更早，在后面 Nature DQN 算法论文中其实已经引入了两个网络的概念了，只是更新的方式略有差异。在 Nature DQN 算法论文中，两个网络是交替更新的，而在 Double DQN 算法论文中则是随机选择其中一个网络进行更新，实践证明交替更新的方式会更加稳定一些。但由于 Double DQN 算法的重点其实不在于 Double，而在于前面讲的估计目标 Q 值的方式，加上讲解 DQN 算法的时候已经引入了两个网络的概念，所以这里就不再赘述了。

## 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。

回忆 DQN 算法的网络结构，如图 8.2 所示，输入层的维度就是状态数，输出层的维度就是动作数。



图8.2 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，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 ，网络结构如图 8.3 所示。

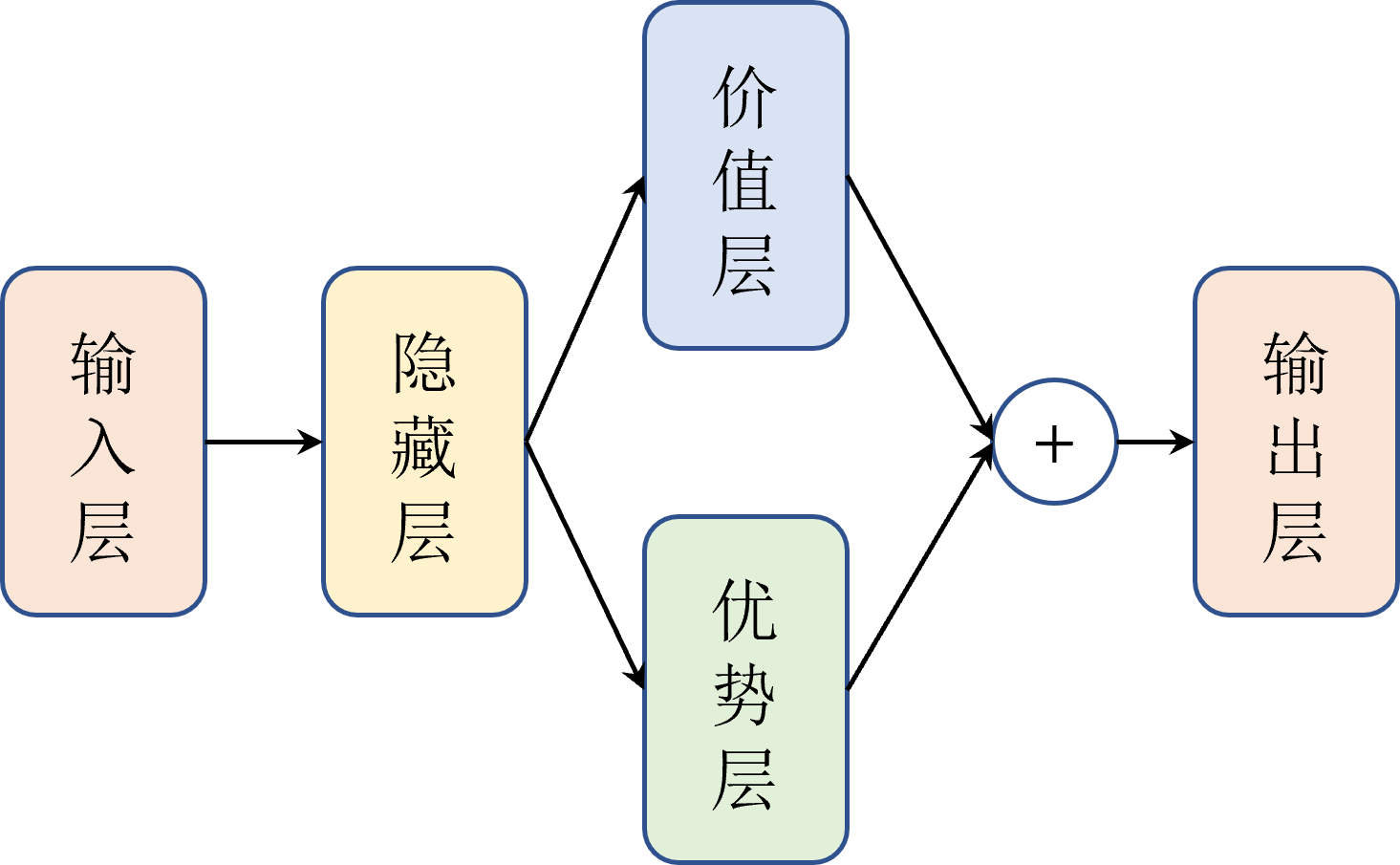


图8.3 Dueling DQN 网络结构

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如式（8.4）：

去掉这里的价值层即优势层就是普通的 Q 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（8.5）：

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（Robustness）**。

根据上面的分析，我们就可以写出 Dueling DQN 网络的代码，如代码清单8-2所示：

代码清单8-2 Dueling DQN 网络

class DuelingQNetwork(nn.Module):

def \_\_init\_\_(self, n\_states, n\_actions,hidden\_dim=128):

super(DuelingQNetwork, self).\_\_init\_\_()

# 隐藏层

self.hidden\_layer = nn.Sequential(

nn.Linear(n\_states, hidden\_dim),

nn.ReLU()

)

# 优势层

self.advantage\_layer = nn.Sequential(

nn.Linear(hidden\_dim, hidden\_dim),

nn.ReLU(),

nn.Linear(hidden\_dim, n\_actions)

)

# 价值层

self.value\_layer = nn.Sequential(

nn.Linear(hidden\_dim, hidden\_dim),

nn.ReLU(),

nn.Linear(hidden\_dim, 1)

)

def forward(self, state):

x = self.hidden\_layer(state)

advantage = self.advantage\_layer(x)

value = self.value\_layer(x)

return value + advantage - advantage.mean() # Q(s,a) = V(s) + A(s,a) - mean(A(s,a))

## 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 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，我们就讲到了探索策略以及探索-利用窘境的问题，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。

在实战中，我们首先可以定义引入了噪声层的线性层，如代码清单8-3所示：

代码清单8-2定义引入了噪声层的线性层

class NoisyLinear(nn.Module):

'''在Noisy DQN中用NoisyLinear层替换普通的nn.Linear层

'''

def \_\_init\_\_(self, in\_dim, out\_dim, std\_init=0.4):

super(NoisyLinear, self).\_\_init\_\_()

self.in\_dim = in\_dim

self.out\_dim = out\_dim

self.std\_init = std\_init

self.weight\_mu = nn.Parameter(torch.empty(out\_dim, in\_dim))

self.weight\_sigma = nn.Parameter(torch.empty(out\_dim, in\_dim))

# 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。

self.register\_buffer('weight\_epsilon', torch.empty(out\_dim, in\_dim))

self.bias\_mu = nn.Parameter(torch.empty(out\_dim))

self.bias\_sigma = nn.Parameter(torch.empty(out\_dim))

self.register\_buffer('bias\_epsilon', torch.empty(out\_dim))

self.reset\_parameters() # 初始化参数

self.reset\_noise() # 重置噪声

def forward(self, x):

if self.training:

weight = self.weight\_mu + self.weight\_sigma \* self.weight\_epsilon

bias = self.bias\_mu + self.bias\_sigma \* self.bias\_epsilon

else:

weight = self.weight\_mu

bias = self.bias\_mu

return F.linear(x, weight, bias)

def reset\_parameters(self):

mu\_range = 1 / self.in\_dim \*\* 0.5

self.weight\_mu.data.uniform\_(-mu\_range, mu\_range)

self.weight\_sigma.data.fill\_(self.std\_init / self.in\_dim \*\* 0.5)

self.bias\_mu.data.uniform\_(-mu\_range, mu\_range)

self.bias\_sigma.data.fill\_(self.std\_init / self.out\_dim \*\* 0.5)

def reset\_noise(self):

epsilon\_in = self.\_scale\_noise(self.in\_dim)

epsilon\_out = self.\_scale\_noise(self.out\_dim)

self.weight\_epsilon.copy\_(epsilon\_out.ger(epsilon\_in))

self.bias\_epsilon.copy\_(self.\_scale\_noise(self.out\_dim))

def \_scale\_noise(self, size):

x = torch.randn(size)

x = x.sign().mul(x.abs().sqrt())

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 DQN 算法中将普通的线性层替换为 NoisyLinear 层

class NoisyQNetwork(nn.Module):

def \_\_init\_\_(self, n\_states, n\_actions, hidden\_dim=128):

super(NoisyQNetwork, self).\_\_init\_\_()

self.fc1 = nn.Linear(n\_states, hidden\_dim)

self.noisy\_fc2 = NoisyLinear(hidden\_dim, hidden\_dim)

self.noisy\_fc3 = NoisyLinear(hidden\_dim, n\_actions)

def forward(self, x):

x = F.relu(self.fc1(x))

x = F.relu(self.noisy\_fc2(x))

x = self.noisy\_fc3(x)

return x

def reset\_noise(self):

self.noisy\_fc2.reset\_noise()

self.noisy\_fc3.reset\_noise()

注意在训练过程中，我们需要在每个 mini-batch 中重置噪声，更多细节请参考 JoyRL 源码。另外，我们可以直接利用 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构建 Noisy Q 网络

import torchrl

class NoisyQNetwork(nn.Module):

def \_\_init\_\_(self, n\_states, n\_actions, hidden\_dim=128):

super(NoisyQNetwork, self).\_\_init\_\_()

self.fc1 = nn.Linear(n\_states, hidden\_dim)

self.noisy\_fc2 = torchrl.NoisyLinear(hidden\_dim, hidden\_dim,std\_init=0.1)

self.noisy\_fc3 = torchrl.NoisyLinear(hidden\_dim, n\_actions,std\_init=0.1)

def forward(self, x):

x = F.relu(self.fc1(x))

x = F.relu(self.noisy\_fc2(x))

x = self.noisy\_fc3(x)

return x

def reset\_noise(self):

self.noisy\_fc2.reset\_noise()

self.noisy\_fc3.reset\_noise()

## 8.3 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 英文全称为 Prioritized Experience Replay，即优先经验回放，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 $Q$ 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，进而计算的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。

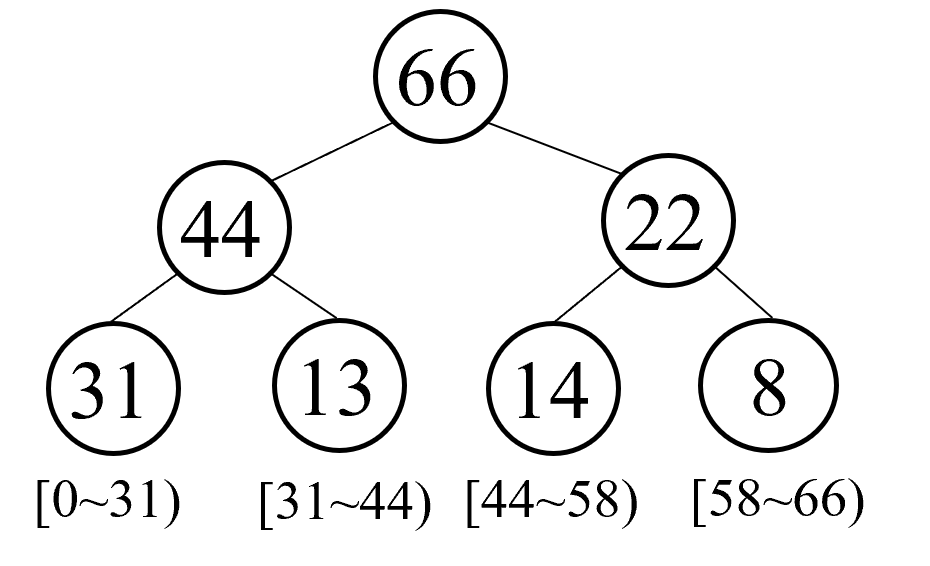


图8.4 SumTree 结构

如图 8.4 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 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 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。

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。

基于以上原理，我们可以新建一个 Python 类来实现 SumTree 结构，如代码清单8-6所示：

代码清单8-6 实现 SumTree 结构

class SumTree:

def \_\_init\_\_(self, capacity):

self.capacity = capacity

self.tree = np.zeros(2 \* capacity - 1)

self.data = np.zeros(capacity, dtype=object) # 存储样本

self.write\_idx = 0 # 写入样本的索引

self.count = 0 # 当前存储的样本数量

def add(self, priority, exps):

''' 添加一个样本到叶子节点，并更新其父节点的优先级

'''

idx = self.write\_idx + self.capacity - 1 # 样本的索引

self.data[self.write\_idx] = exps # 写入样本

self.update(idx, priority) # 更新样本的优先级

self.write\_idx = (self.write\_idx + 1) % self.capacity # 更新写入样本的索引

if self.count < self.capacity:

self.count += 1

def update(self, idx, priority):

''' 更新叶子节点的优先级，并更新其父节点的优先级

Args:

idx (int): 样本的索引

priority (float): 样本的优先级

'''

diff = priority - self.tree[idx] # 优先级的差值

self.tree[idx] = priority

while idx != 0:

idx = (idx - 1) // 2

self.tree[idx] += diff

def get\_leaf(self, v):

''' 根据优先级的值采样对应区间的叶子节点样本

'''

idx = 0

while True:

left = 2 \* idx + 1

right = left + 1

if left >= len(self.tree):

break

if v <= self.tree[left]:

idx = left

else:

v -= self.tree[left]

idx = right

data\_idx = idx - self.capacity + 1

return idx, self.tree[idx], self.data[data\_idx]

def get\_data(self, indices):

return [self.data[idx - self.capacity + 1] for idx in indices]

def total(self):

''' 返回所有样本的优先级之和，即根节点的值

'''

return self.tree[0]

def max\_prior(self):

''' 返回所有样本的最大优先级

'''

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。

基于以上的 SumTree 结构，我们可以实现优先经验回放的功能。然而，论文原作者认为，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。

**随机优先级采样**。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（8.6）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 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 ，如式（8.7）所示：

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（8.8）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。

**重要性采样**。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。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（8.9）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。我们可以定义权重如式（8.10）所示：

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（8.11）所示：

**热偏置**。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（8.12）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。

结合上面的优先级采样和重要性采样，我们可以基于 SumTree 实现一个带有优先级的经验回放，代码如代码清单8-7所示：

代码清单8-7 基于 SumTree 实现一个带有优先级的经验回放

class PrioritizedReplayBuffer:

def \_\_init\_\_(self, cfg):

self.capacity = cfg.buffer\_size

self.alpha = cfg.per\_alpha # 优先级的指数参数，越大越重要，越小越不重要

self.epsilon = cfg.per\_epsilon # 优先级的最小值，防止优先级为0

self.beta = cfg.per\_beta # importance sampling的参数

self.beta\_annealing = cfg.per\_beta\_annealing # beta的增长率

self.tree = SumTree(self.capacity)

self.max\_priority = 1.0

def push(self, exps):

''' 添加样本

'''

priority = self.max\_priority if self.tree.total() == 0 else self.tree.max\_prior()

self.tree.add(priority, exps)

def sample(self, batch\_size):

''' 采样一个批量样本

'''

indices = [] # 采样的索引

priorities = [] # 采样的优先级

exps = [] # 采样的样本

segment = self.tree.total() / batch\_size

self.beta = min(1.0, self.beta + self.beta\_annealing)

for i in range(batch\_size):

a = segment \* i

b = segment \* (i + 1)

p = np.random.uniform(a, b) # 采样一个优先级

idx, priority, exp = self.tree.get\_leaf(p) # 采样一个样本

indices.append(idx)

priorities.append(priority)

exps.append(exp)

# 重要性采样, weight = (N \* P(i)) ^ (-beta) / max\_weight

sample\_probs = np.array(priorities) / self.tree.total()

weights = (self.tree.count \* sample\_probs) \*\* (-self.beta) # importance sampling的权重

weights /= weights.max() # 归一化

indices = np.array(indices)

return zip(\*exps), indices, weights

def update\_priorities(self, indices, priorities):

''' 更新样本的优先级

'''

priorities = np.abs(priorities) # 取绝对值

for idx, priority in zip(indices, priorities):

# 控制衰减的速度, priority = (priority + epsilon) ^ alpha

priority = (priority + self.epsilon) \*\* self.alpha

priority = np.minimum(priority, self.max\_priority)

self.tree.update(idx, priority)

def \_\_len\_\_(self):

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 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 设置为 0.6， 设置为 $0.01$， 设置为 0.4， 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 实现优先级经验回放

class PrioritizedReplayBufferQue:

def \_\_init\_\_(self, cfg):

self.capacity = cfg.buffer\_size

self.alpha = cfg.per\_alpha # 优先级的指数参数，越大越重要，越小越不重要

self.epsilon = cfg.per\_epsilon # 优先级的最小值，防止优先级为0

self.beta = cfg.per\_beta # importance sampling的参数

self.beta\_annealing = cfg.per\_beta\_annealing # beta的增长率

self.buffer = deque(maxlen=self.capacity)

self.priorities = deque(maxlen=self.capacity)

self.count = 0 # 当前存储的样本数量

self.max\_priority = 1.0

def push(self,exps):

self.buffer.append(exps)

self.priorities.append(max(self.priorities, default=self.max\_priority))

self.count += 1

def sample(self, batch\_size):

priorities = np.array(self.priorities)

probs = priorities/sum(priorities)

indices = np.random.choice(len(self.buffer), batch\_size, p=probs)

weights = (self.count\*probs[indices])\*\*(-self.beta)

weights /= weights.max()

exps = [self.buffer[i] for i in indices]

return zip(\*exps), indices, weights

def update\_priorities(self, indices, priorities):

priorities = np.abs(priorities)

priorities = (priorities + self.epsilon) \*\* self.alpha

priorities = np.minimum(priorities, self.max\_priority).flatten()

for idx, priority in zip(indices, priorities):

self.priorities[idx] = priority

def \_\_len\_\_(self):

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，伪代码如下：

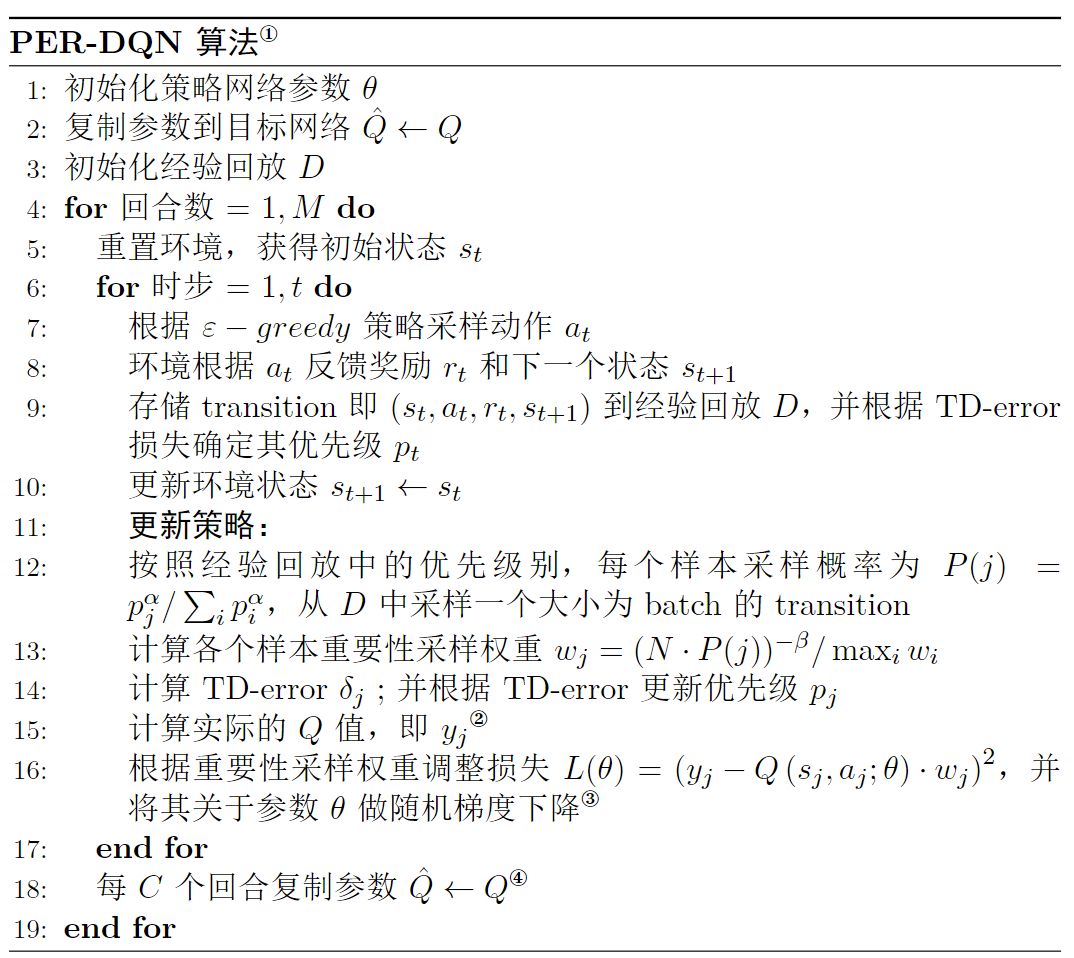


图8.4 SumTree 结构

## 8.4 C51 算法

分布式 DQN 算法，即 Distributed DQN，有时也会看到它也叫 Categorical DQN 这个名字，但最常见的名字是 C51 算法[[5]](#footnote-5) 。该算法跟 PER 算法一样，是从不同的角度改进强化学习算法，而不单单指 DQN 算法，而是能适用于任何基于 Q-learning 的强化学习算法。该算法的核心思想是将传统 DQN 算法中的值函数 换成了值分布 ，即将值函数的输出从一个数值变成了一个分布，这样就能更好地处理值函数估计不准确以及离散动作空间的问题。

在之前讲到的经典强化学习算法中我们优化的其实是值分布的均值，也就是 Q 函数，但实际上由于状态转移的随机性、函数近似等原因，智能体与环境之间也存在着随机性，这也导致了最终累积的回报也会是一个随机变量，使用一个确定的均值会忽略整个分布所提供的信息。。因此，我们可以将值函数 Q 看成是一个随机变量，它的期望值就是 Q 函数，而它的方差就是 Q 函数的不确定性，公式表示如式（8.12）所示：

其中状态分布

1. 论文链接：http://papers.neurips.cc/paper/3964-double-q-learning.pdf [↑](#footnote-ref-1)
2. Dueling DQN 算法论文：Dueling Network Architectures for Deep Reinforcement Learning [↑](#footnote-ref-2)
3. 论文链接：https://arxiv.org/abs/1706.10295 [↑](#footnote-ref-3)
4. 论文链接：https://arxiv.org/abs/1511.05952 [↑](#footnote-ref-4)
5. 论文链接：https://arxiv.org/abs/1707.06887 [↑](#footnote-ref-5)