# 第7章 DQN 算法

本章开始进入深度强化学习的部分，我们首先从 DQN 算法开始讲起。DQN 算法，英文全称 Deep Q-Network , 顾名思义就是基于深度网络模型的 Q-learning 算法，主要由 DeepMind 公司于 2013 年[[1]](#footnote-1)和 2015 年[[2]](#footnote-2)分别提出的两篇论文来实现。除了用深度网络代替 Q 之外，DQN 算法还引入了两个技巧，即经验回放和目标网络，我们将逐一介绍。

DQN 算法相对于 Q-learning 算法来说更新方法本质上是一样的，而 DQN 算法最重要的贡献之一就是本章节开头讲的，用神经网络替换表格的形式来近似动作价值函数 。

## 7.1 深度网络

在 Q-learning 算法中，我们使用 Q 表的形式来实现动作价值函数 。但是用 Q 表只适用于状态和动作空间都是离散的，并且不利于处理高维的情况，因为在高维状态空间下每次更新维护 Q 表的计算成本会高得多。因此，在 DQN 算法中，使用深度神经网络的形式来更新 Q 值，这样做的好处就是能够处理高维的状态空间，并且也能处理连续的状态空间。

如图 7.1 所示，在 DQN 的网络模型中，我们将当前状态 作为输入，并输出动作空间中所有动作（假设这里只有两个动作，即1和2）对应的 Q 值，我们记做 。对于其他状态，该网络模型同样可以输出所有动作对应的价值，这样一来神经网络近似的动作价值函数可以表示为 。其中 就是神经网络模型的参数，可以结合梯度下降的方法求解。

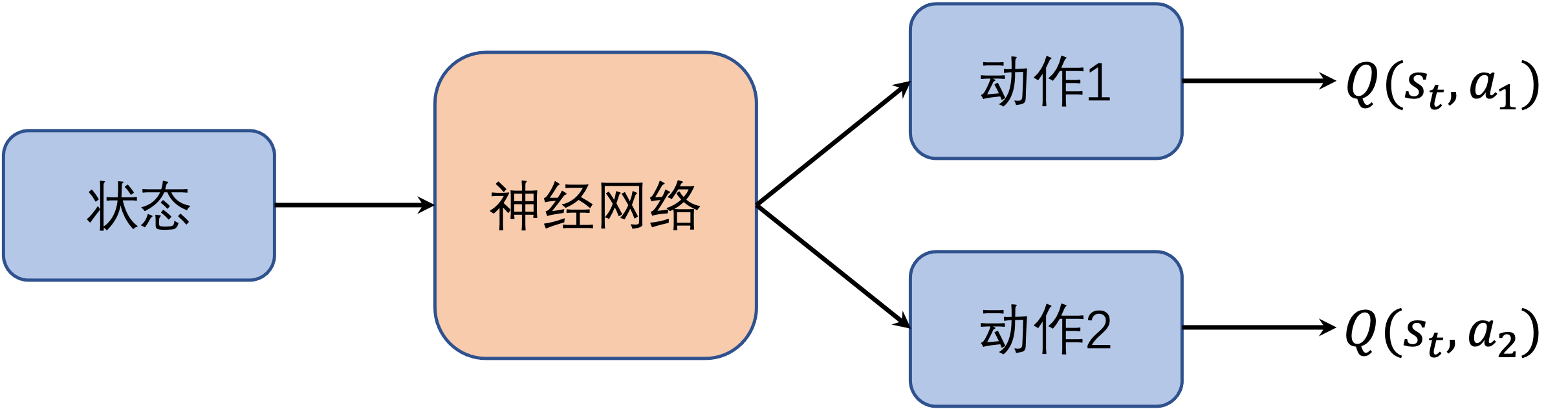


图7.1 DQN 网络结构

具体该怎么结合梯度下降来更新 Q 函数的参数呢？我们首先回顾一下 Q-learning} 算法的更新公式如式 (7.1) 所示：

我们注意到公式右边两项 和 分别表示期望的 Q 值和实际的 Q 值，其中预测的 Q 值是用目标网络中下一个状态对应 Q 值的最大值来近似的。换句话说，在更新 Q 值并达到收敛的过程中，期望的 Q 值也应该接近实际的 Q 值，即我们希望最小化 和 之间的损失，其中 是学习率，尽管优化参数的公式跟深度学习中梯度下降法优化参数的公式有一些区别（比如增加了 和 等参数）。从这个角度上来看，强化学习跟深度学习的训练方式其实是一样的，不同的地方在于强化学习用于训练的样本（包括状态、动作和奖励等等）是与环境实时交互得到的，而深度学习则是事先准备好的训练集。当然训练方式类似并不代表强化学习和深度学习之间的区别就很小，本质上来说强化学习和深度学习解决的问题是完全不同的，前者用于解决序列决策问题，后者用于解决静态问题例如回归、分类、识别等等。在 Q-learning 算法中，我们是直接优化 Q 值的，而在 DQN 中使用神经网络来近似 Q 函数，我们则需要优化网络模型对应的参数 ，如式 (7.2) 所示：

这里 DQN 算法也是基于 TD 更新的，因此依然需要判断终止状态，在 Q-learning 算法中也有同样的操作。

## 7.2 经验回放

强化学习是与环境实时交互得到样本然后进行训练的，在 Q-learning 算法中我们是每次交互一个样本，通常包含当前状态（ `state` ）、当前动作（ `action` ）、下一个状态（ `next\_state` ）、是否为终止状态（ `done` ），这样一个样本我们一般称之为一个状态转移（ transition ）。但是每次只交互一个样本并即时更新的方式在 DQN 算法中会产生一些问题。首先，对于神经网络来说，每次只喂入一个样本然后反向传播并更新参数是不稳定的。其次，连续交互产生的样本之间关联性过强，会导致深度网络更新的过程，容易陷入局部最优解。这跟深度学习中为什么采用随机梯度下降而不是单纯的顺序梯度下降的道理是类似的，只是在强化学习中问题会更为明显因为强化学习前后两个样本的关联性往往比监督学习的样本更紧密。

回顾一下在深度学习基础的章节中我们讲到梯度下降的方式，首先从样本选择方式来看分成单纯的梯度下降和随机梯度下降，随机梯度下降在样本选择过程中使用随机抽样，即每次从总样本中随机选择一些子样本处理，而不是按照固定的顺序逐个遍历总的样本，这样做的好处就是可以避免模型陷入局部最优解。在随机梯度下降的基础上，从每次抽取的样本数来看可以分为批梯度下降方法（ Batch Gradient Descent ）、（普通的）随机梯度下降（ Stochastic Gradient Descent ）和小批量梯度下降（ Mini-Batch Gradient Descent ）。

普通的随机梯度下降每一次迭代只使用一个样本来更新模型参数，尽管收敛速度快，但由于实现随机性可能会存在收敛到局部最优解的风险。批量梯度下降算法每一次迭代使用所有训练数据来更新模型参数，它的收敛速度虽然较慢，但从凸优化角度（感兴趣的读者也可以学习凸优化这门课）中可以保证收敛到全局最优解。小批量梯度下降算法每次迭代使用一定数量的样本来更新模型参数，介于批量梯度下降和随机梯度下降之间，可以在保证收敛性的同时提高计算效率。再回头看看我们前面说到的 Q-learning 算法更新方式在深度网络下遇到的两个问题，每次只连续地喂入一个样本相当于是普通的顺序梯度下降的方式，这种方式其实是最糟糕的梯度下降方式，因为既不是随机梯度下降，也不是批梯度下降，因此我们希望在 DQN 算法中也能做到小批量梯度下降，这样就能保证收敛性。

如何实现类似的小批量梯度下降呢？DeepMind 公司 在论文中提出了一个经验回放的概念（ replay buffer ），这个经验回放的功能主要包括几个方面。首先是能够缓存一定量的状态转移即样本，此时 DQN 算法并不急着更新并累积一定的初始样本。然后是每次更新的时候随机从经验回放中取出一个小批量的样本并更新策略，注意这里的随机和小批量以便保证我们存储动作价值函数的网络模型是小批量随机梯度下降的。

最后与深度学习不同的是，我们要保证经验回放是具有一定的容量限制的。本质上是因为在深度学习中我们拿到的样本都是事先准备好的，即都是很好的样本，但是在强化学习中样本是由智能体生成的，在训练初期智能体生成的样本虽然能够帮助它朝着更好的方向收敛，但是在训练后期这些前期产生的样本相对来说质量就不是很好了，此时把这些样本喂入智能体的深度网络中更新反而影响其稳定。这就好比我们在小学时积累到的经验，会随着我们逐渐长大之后很有可能就变得不是很适用了，所以经验回放的容量不能太小，太小了会导致收集到的样本具有一定的局限性，也不能太大，太大了会失去经验本身的意义。从这一个细小的点上相信读者们也能体会到深度学习和强化学习的区别了，所谓管中窥豹，可见一斑。

## 7.3 目标网络

在 DQN 算法中还有一个重要的技巧，这个技巧就跟深度学习关系不大了，而是更“强化”的一个技巧。即使用了一个每隔若干步才更新的目标网络。与之相对的，会有一个每步更新的网络，即每次从经验回放中采样到样本就更新网络参数，在本书中一般称之为策略网络。策略网络和目标网络结构都是相同的，都用于近似 Q 值，在实践中每隔若干步才把每步更新的策略网络参数复制给目标网络，这样做的好处是保证训练的稳定，避免 Q 值的估计发散。

举一个典型的例子，这里的目标网络好比明朝的皇帝，而策略网络相当于皇帝手下的太监，每次皇帝在做一些行政决策时往往不急着下定论，会让太监们去收集一圈情报，然后集思广益再做决策。这样做的好处是显而易见的，比如皇帝要处决一个可能受冤的犯人时，如果一个太监收集到一个情报说这个犯人就是真凶的时候，如果皇帝是一个急性子可能就当初处决了，但如果这时候另外一个太监收集了一个更有力的证据证明刚才那个太监收集到的情报不可靠并能够证明该犯人无罪时，那么此时皇帝就已经犯下了一个无法挽回的过错。换句话说，如果当前有个小批量样本导致模型对 Q 值进行了较差的过估计，如果接下来从经验回放中提取到的样本正好连续几个都这样的，很有可能导致 Q 值的发散（它的青春小鸟一去不回来了）。

再打个比方，我们玩 RPG 或者闯关类游戏，有些人为了破纪录经常存档（ Save ）和回档（ Load ），简称 “SL” 大法。只要我出了错，我不满意我就加载之前的存档，假设不允许加载呢，就像 DQN 算法一样训练过程中会退不了，这时候是不是搞两个档，一个档每帧都存一下，另外一个档打了不错的结果再存，也就是若干个间隔再存一下，到最后用间隔若干步数再存的档一般都比每帧都存的档好些呢。当然我们也可以再搞更多个档，也就是 DQN 增加多个目标网络，但是对于 DQN 算法来说没有多大必要，因为多几个网络效果不见得会好很多。

到这里我们基本讲完了DQN 算法的内容，可以直接贴出伪代码准备进入实战了，如图 7.2 所示：

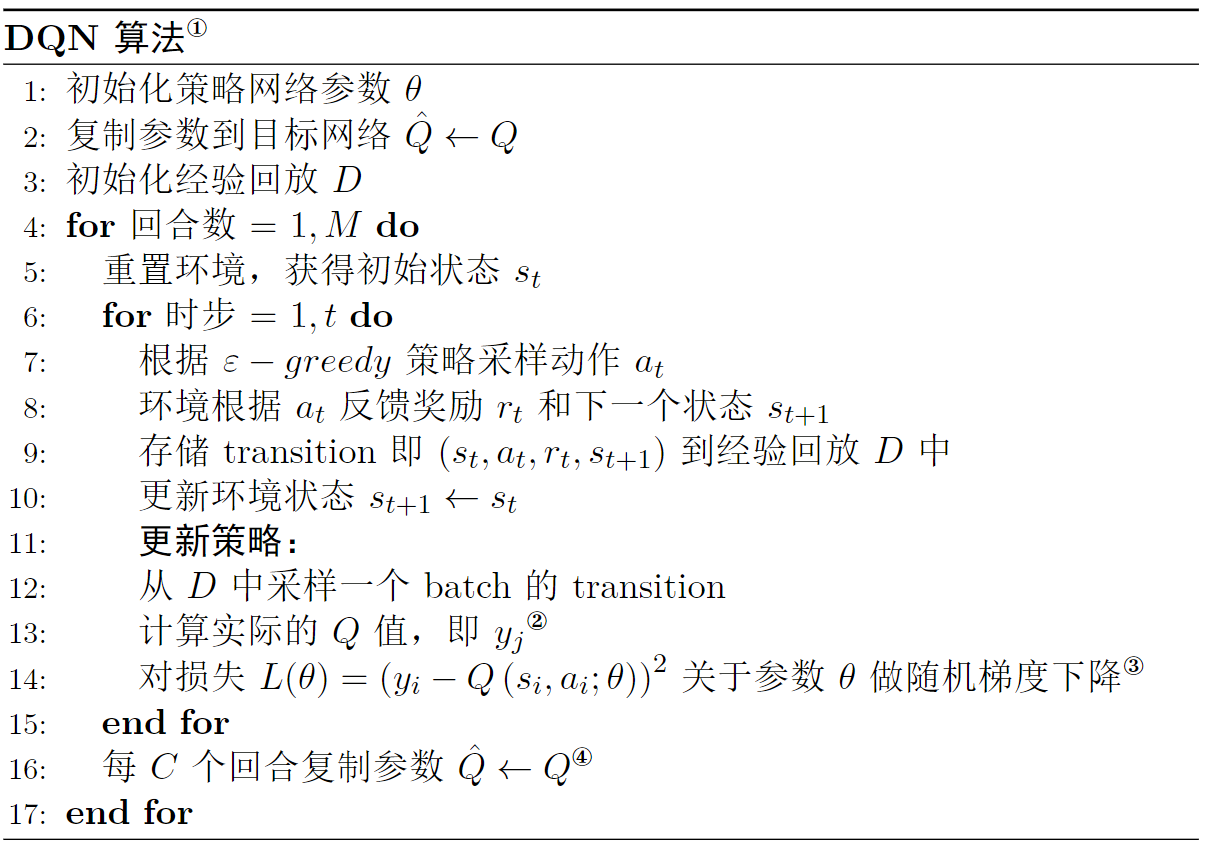


图7.2 DQN 算法伪代码

## 7.4 实战：DQN 算法

### 7.4.1 定义算法

由于 DQN 智能体包含的元素比较多，包括神经网络，经验回放等，我们接下来将逐一实现。首先需要定义一个深度网络来表示 Q 函数，目前 JoyRL 算法都是基于 Torch 框架实现的，所以需要读者们具有一定的相关基础。如下，我们定义一个全连接网络即可，输入维度就是状态数，输出的维度就是动作数，中间的隐藏层采用最常用的 ReLU 激活函数。网络定义如代码清单7-1所示：

代码清单7-1 定义一个全连接网络

class MLP(nn.Module):

def \_\_init\_\_(self, input\_dim,output\_dim,hidden\_dim=128):

""" 初始化q网络，为全连接网络

input\_dim: 输入的特征数即环境的状态维度

output\_dim: 输出的动作维度

"""

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

self.fc1 = nn.Linear(input\_dim, hidden\_dim) # 输入层

self.fc2 = nn.Linear(hidden\_dim,hidden\_dim) # 隐藏层

self.fc3 = nn.Linear(hidden\_dim, output\_dim) # 输出层

def forward(self, x):

# 各层对应的激活函数

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

x = F.relu(self.fc2(x))

return self.fc3(x)

然后我们定义经验回放，如代码清单7-2所示：

代码清单7-2 定义经验回放

class ReplayBuffer:

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

self.capacity = capacity # 经验回放的容量

self.buffer = [] # 缓冲区

self.position = 0

def push(self, state, action, reward, next\_state, done):

''' 缓冲区是一个队列，容量超出时去掉开始存入的转移(transition)

'''

if len(self.buffer) < self.capacity:

self.buffer.append(None)

self.buffer[self.position] = (state, action, reward, next\_state, done)

self.position = (self.position + 1) % self.capacity

def sample(self, batch\_size):

''' 采样

'''

batch = random.sample(self.buffer, batch\_size) # 随机采出小批量转移

state, action, reward, next\_state, done = zip(\*batch) # 解压成状态，动作等

return state, action, reward, next\_state, done

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

''' 返回当前存储的量

'''

return len(self.buffer)

前面讲到经验回放的主要功能是，存入样本然后随机采样出一个批量的样本，分别对应这里的 `push` 和 `sample` 方法，并且需要保证一定的容量（即 `capacity` ）。实现的手段有很多，也可以用 Python 队列的方式实现，这里只是一个参考。

然后我们定义智能体，跟 Q-learning 算法中类似，我们定义一个名为 `Agent` 的 Python 类，包含 `sample\_action`，`predict\_action` 和 `update` 等方法，如代码清单7-3所示：

代码清单7-3 定义智能体

class Agent:

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

# 定义策略网络

self.policy\_net = MLP(n\_states,n\_actions).to(device)

# 定义目标网络

self.target\_net = MLP(n\_states,n\_actions).to(device)

# 将策略网络参数复制到目标网络中

self.target\_net.load\_state\_dict(self.policy\_net.state\_dict())

# 定义优化器

self.optimizer = optim.Adam(self.policy\_net.parameters(), lr=learning\_rate)

# 经验回放

self.memory = ReplayBuffer(buffer\_size)

self.sample\_count = 0 # 记录采样步数

def sample\_action(self,state):

self.sample\_count += 1

# epsilon 随着采样步数衰减

self.epsilon = self.epsilon\_end + (self.epsilon\_start - self.epsilon\_end) \* math.exp(-1. \* self.sample\_count / self.epsilon\_decay)

if random.random() > self.epsilon:

with torch.no\_grad(): # 不使用梯度

state = torch.tensor(np.array(state), device=self.device, dtype=torch.float32).unsqueeze(dim=0)

q\_values = self.policy\_net(state)

action = q\_values.max(1)[1].item() # choose action corresponding to the maximum q value

else:

action = random.randrange(self.n\_actions)

def predict\_action(self,state):

with torch.no\_grad():

state = torch.tensor(np.array(state), device=self.device, dtype=torch.float32).unsqueeze(dim=0)

q\_values = self.policy\_net(state)

action = q\_values.max(1)[1].item() # choose action corresponding to the maximum q value

return action

def update(self):

pass

注意，这里所有的代码都是为了方便讲解用的演示代码，完整的代码读者可在 JoyRL 开源工具上参考。在这里我们定义了两个网络，策略网络和目标网络，在 Torch 中可以使用 `.to(device)` 来决定网络是否使用 CPU 还是 GPU 计算。 此外在初始化的时候我们需要让目标网络和策略网络的参数保持一致，可以使用 `load\_state\_dict` 方法，然后就是优化器和经验回放了。 在 DQN 算法中采样动作和预测动作跟 Q-learning 是一样的，其中 `q\_values = self.policy\_net(state)` 拿到的 Q 值是给定状态下所有动作的值，根据这些值选择最大值对应的动作即可。

DQN 算法更新本质上跟 Q-learning 区别不大，但由于读者可能第一次接触深度学习的实现方式，这里单独拎出来分析 DQN 算法的更新方式，如代码清单7-4所示：

代码清单7-4 定义DQN 算法更新方法

def update(self, share\_agent=None):

# 当经验回放中样本数小于更新的批大小时，不更新算法

if len(self.memory) < self.batch\_size: # when transitions in memory donot meet a batch, not update

return

# 从经验回放中采样

state\_batch, action\_batch, reward\_batch, next\_state\_batch, done\_batch = self.memory.sample(

self.batch\_size)

# 转换成张量（便于GPU计算）

state\_batch = torch.tensor(np.array(state\_batch), device=self.device, dtype=torch.float)

action\_batch = torch.tensor(action\_batch, device=self.device).unsqueeze(1)

reward\_batch = torch.tensor(reward\_batch, device=self.device, dtype=torch.float).unsqueeze(1)

next\_state\_batch = torch.tensor(np.array(next\_state\_batch), device=self.device, dtype=torch.float)

done\_batch = torch.tensor(np.float32(done\_batch), device=self.device).unsqueeze(1)

# 计算 Q 的实际值

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

# 计算 Q 的估计值，即 r+\gamma Q\_max

next\_max\_q\_value\_batch = self.target\_net(next\_state\_batch).max(1)[0].detach().unsqueeze(1)

expected\_q\_value\_batch = reward\_batch + self.gamma \* next\_max\_q\_value\_batch\* (1-done\_batch)

# 计算损失

loss = nn.MSELoss()(q\_value\_batch, expected\_q\_value\_batch)

# 梯度清零，避免在下一次反向传播时重复累加梯度而出现错误。

self.optimizer.zero\_grad()

# 反向传播

loss.backward()

# clip避免梯度爆炸

for param in self.policy\_net.parameters():

param.grad.data.clamp\_(-1, 1)

# 更新优化器

self.optimizer.step()

# 每C(target\_update)步更新目标网络

if self.sample\_count % self.target\_update == 0:

self.target\_net.load\_state\_dict(self.policy\_net.state\_dict())

首先由于我们是小批量随机梯度下降，所以当经验回放不满足批大小时选择不更新，这实际上是工程性问题。然后在更新时我们取出样本，并转换成 Torch 的张量，便于我们用 GPU 计算。接着计算 Q 值的估计值和实际值，并得到损失函数。在得到损失函数并更新参数时，我们在代码上会有一个固定的写法，即梯度清零，反向传播和更新优化器的过程，跟在深度学习中的写法是一样的，最后我们需要定期更新一下目标网络，这里会有一个超参数 `target\_update`, 需要读者根据经验调试。

### 7.4.2 定义环境

由于我们在 Q-learning 算法中已经讲过怎么定义训练和测试过程了，所有强化学习算法的训练过程基本上都是通用的，因此我们在这里及之后的章节中不再赘述。但由于我们在 DQN 算法中使用了跟 Q-learning 算法 中不一样的环境，但都是 OpenAI Gym 平台的，所以我们简单介绍一下该环境。环境名称叫做 Cart Pole[[3]](#footnote-3) ，中文译为推车杆游戏。如图 7.3 所示，我们的目标是持续左右推动保持倒立的杆一直不倒。

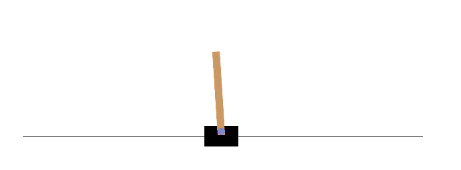


图7.3 Cart Pole游戏

环境的状态数是 4, 动作数是 2。有读者可能会奇怪，这不是比 Q-learning 算法中的 `CliffWalking-v0` 环境（状态数是 48, 动作数是 2）更简单吗，应该直接用 Q-learning 算法就能解决？实际上是不能的，因为 `Cart Pole` 的状态包括推车的位置（范围是 -4.8 到 4.8 ）、速度（范围是负无穷大到正无穷大）、杆的角度（范围是 -24 度 到 24 度）和角速度（范围是负无穷大到正无穷大）,这几个状态都是连续的值，也就是前面所说的连续状态空间，因此用 Q-learning 算法是很难解出来的。环境的奖励设置是每个时步下能维持杆不到就给一个 +1 的奖励，因此理论上在最优策略下这个环境是没有终止状态的，因为最优策略下可以一直保持杆不倒。回忆前面讲到基于 TD 的算法都必须要求环境有一个终止状态，所以在这里我们可以设置一个环境的最大步数，比如我们认为如果能在两百个时步以内坚持杆不到就近似说明学到了一个不错的策略。

### 7.4.3 设置参数

定义好智能体和环境之后就可以开始设置参数了，如代码清单7-5所示：

代码清单7-5 设置训练参数

self.epsilon\_start = 0.95 # epsilon 起始值

self.epsilon\_end = 0.01 # epsilon 终止值

self.epsilon\_decay = 500 # epsilon 衰减率

self.gamma = 0.95 # 折扣因子

self.lr = 0.0001 # 学习率

self.buffer\_size = 100000 # 经验回放容量

self.batch\_size = 64 # 批大小

self.target\_update = 4 # 目标网络更新频率

与 Q-learning 算法相比，除了 , 折扣因子以及学习率之外多了三个超参数，即经验回放的容量、批大小和目标网络更新频率。注意这里学习率在更复杂的环境中一般会设置得比较小，经验回放的容量是一个比较经验性的参数，根据实际情况适当调大即可，不需要额外花太多时间调。批大小也比较固定，一般都在 64，128，256，512 中间取值，目标网络更新频率会影响智能体学得快慢，但一般不会导致学不出来。总之，DQN 算法相对来说是深度强化学习的一个稳定且基础的算法，只要适当调整学习率都能让智能体学出一定的策略。

最后展示一下我们的训练曲线和测试曲线，分别如图 7.4 和 7.5 所示。

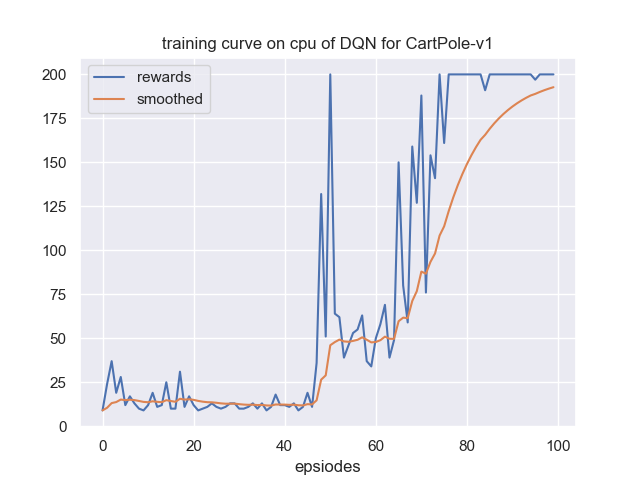


图7.4 DQN 算法训练曲线

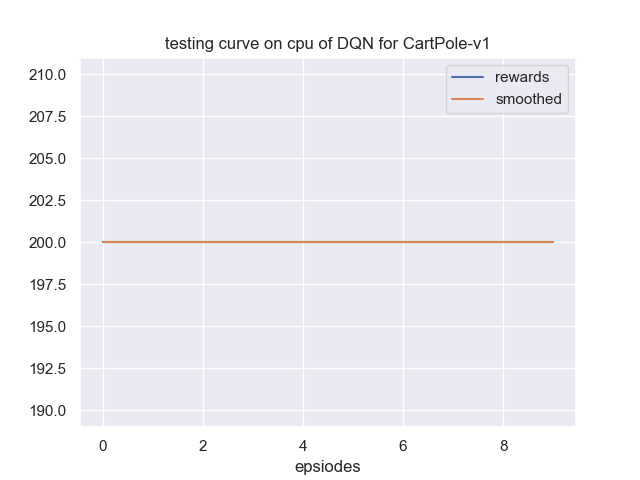


图7.4 DQN 算法测试曲线

其中我们该环境每回合的最大步数是 200，对应的最大奖励也为 200，从图中可以看出，智能体确实学到了一个最优的策略。

1. 《Playing Atari with Deep Reinforcement Learning》 [↑](#footnote-ref-1)
2. 《Human-level Control through Deep Reinforcement Learning》 [↑](#footnote-ref-2)
3. 官网环境介绍：https://gymnasium.farama.org/environments/classic\_control/cart\_pole/ [↑](#footnote-ref-3)