# Double DQN原理是什么，怎样实现？（附代码）

[](https://www.zhihu.com/people/zhang-si-jun-52)

[张斯俊](https://www.zhihu.com/people/zhang-si-jun-52" \t "/home/admin/文档\\x/_blank)

愿成为一把梯子，助你跨过无数的坑。

在上一章，虽然用了DQN，希望解决连续状态空间的问题，但环境仍然是用FrozenLake-v0。

虽然证明了可以用深度神经网络的方式，但多少有点英雄无用武之地的感觉。 在这一节，将用CartPole-v0作为环境做试验，并介绍DQN的训练的技巧和DQN的一个最重要的变种DQN。

如果对DQN的基本原理还不够了解。可以参考这篇专栏：

[张斯俊：三维可视化助你直观理解DQN算法[DQN理论篇]​](https://zhuanlan.zhihu.com/p/110620815" \t "/home/admin/文档\\x/_blank)

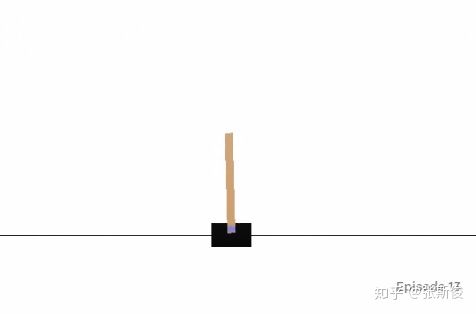
[张斯俊：用可视化直观理解DQN[DQN实战篇]​](https://zhuanlan.zhihu.com/p/110657606" \t "/home/admin/文档\\x/_blank)

由于tensorflow提供的示例代码结构比较复杂，我另外提供了一份代码以供参考。

[https://github.com/louisnino/RLcode/blob/master/double\_DQN%20%26%20dueling\_DQN.py​github.com](https://link.zhihu.com/?target=https://github.com/louisnino/RLcode/blob/master/double_DQN & dueling_DQN.py" \t "/home/admin/文档\\x/_blank)

### CartPole-v0说明

在介绍算法之前，先了解一下CartPole这个游戏的特点。



CartPole-v0游戏有一个车子，车子上面立一支杆。智能体的任务是，让车子必须左右移动来保持车上的杆保持竖直。如果杆子倾斜超过12度，则游戏结束。

每坚持一帧，智能体能获得 1 reward，如果能获得200reward，那么游戏结束。如果杆子掉下来，游戏失败。继续下一轮游戏。

之前说过，CartPole的状态是连续型的状态，所以可以用几个状态特征来表示。

状态特征：

* - 车子位置：[-2.4,2.4]
* - 车子速度：[-Inf,Inf]
* - 杆子角度：[-41.8,41.8]
* - 杆子（顶端）速度：[-Inf,Inf]

智能体可以决定做两个动作：

* - 0： 把车子往左拉
* - 1： 把车子往右拉

Reward：除最终状态外，所有状态都能获得1reward.

最终状态条件：

1. 杆子角度>= ±12°
2. 车子位置>= ±2.4
3. 坚持200步

所以CartPole游戏其实就是让智能体学会玩杂耍，坚持时间越长获得奖励越多：



### 经验回放(Experience replay)

在强化学习中，有一个问题始终绕不过。训练网络数据采集总是太慢。

当然这个慢是对比网络训练的速度。在强化学习中，网络训练经过GPU的加速，比起游戏来时快很多的。所以训练的瓶颈一般在智能体跟环境互动的过程中。

如果能把互动过程中的数据，都存起来，当数据足够多的时候，再训练网络，那么就快很多了。

**经验回放(Experience replay)**就是实现这样的过程:

把每一步的s，选择的a，进入新的状态s'，获得的奖励r，新状态是否为终止状态做为一条数据都存在一个叫回放缓存的地方(replay buffer)。

当智能体与环境互动期间，就会不断产生这样的多条数据:数据1, 数据2, 数据3, ....

当数据量足够，例如达到设定一个batch的大小，便从中抽出一个batch大小的数据，把这笔数据一起放入网络进行训练。

训练之后继续进行游戏，继续把新产生的数据添加到回放缓存里...

就这样，每次都随机抽出一个batch大小的数据训练智能体。这样，以前产生的数据同样也能用来训练数据了, 效率自然更高。

使用经验回放除了使训练更高效，同时也减少了训练产生的过度拟合的问题。

过度拟合，放到人身上就是过度依赖局部经验了。

就像孩子发现爸爸有胡子，就认为所有男人都有胡子一样。

同样，在有监督学习中，如果只给模型看少量的几张图，并且告诉模型这是猫。这样模型就只会从这几张图学习到猫的特点，而更多的猫模型可能就不认得了。说这就是过度拟合造成的，导致模型不够健壮。

所以健壮的模型，和人一样，见识丰富了就不会落入局部经验的圈套了。

经验回放具体的做法有很多，将会在稍后的示例代码中展示其中一种。

### 固定Q目标(Fixed Q-targets)

也许在上一篇DQN的介绍中，有同学已经会有这样一个疑问。

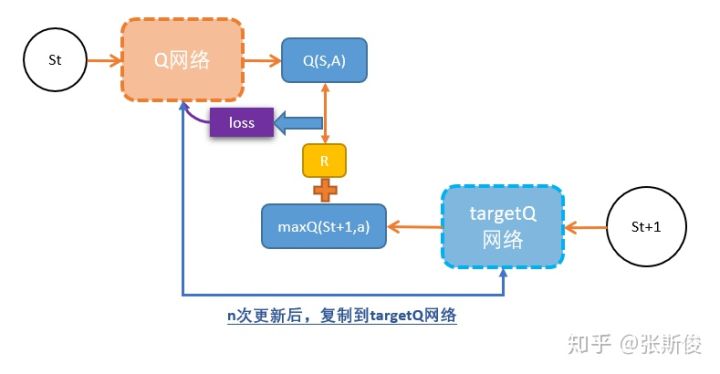
DQN的目标: **gamma \* maxQ(s') + r**

目标本身就包含一个Q网络，这样没有问题吗？

理论上是没有问题的，但，这样会造成Q网络的学习效率比较低，而且不稳定。

如果把训练神经网络比喻成射击游戏，在target中有Q网络的话，就相当于在射击一个移动靶，因为每次射击一次，靶就会挪动一次。相比起固定的靶，无疑加上了训练的难度。

那怎么解决这个问题呢？既然现在是移动靶，那么就把它弄成是固定的靶，先停止10秒。10后挪动靶再打新的靶。这就是Fixed Q-targets的思路。



在实做的时候，其实和原来的DQN一样，唯一不同点是，用两个Q网络：

一个是原来的Q网络，用于估算Q(s);

另外一个叫targetQ网络, targetQ自己并不会更新，也就是它在更新的过程中是固定的，用于计算更新目标。

y = r + gamma \* max(targetQ(s'))

进行N次更新后，就把新Q的参数赋值给旧Q。

同学们可以试试，利用以上个技术来优化自己的程序。如果有难度，也先别着急。在下面的doubleDQN的示例代码中，用了以上两个技术。DoubleDQN和DQN的差别很小，其中的差别我也会提出来让大家对比。

### Double DQN

DQN有一个显著的问题，就是DQN估计的Q值往往会偏大。这是由于Q值是以下一个s'的Q值的最大值来估算的，但下一个state的Q值也是一个估算值，也依赖它的下一个state的Q值...，这就导致了Q值往往会有偏大的的情况出现。

在同一个s'进行试探性出发，计算某个动作的Q值。然后和DQN的记过进行比较就可以得出上述结论。

这种欺上瞒下的做法，实在令人恼火。于是有人想到一个互相监察的想法。

这个思路也很直观。如果只有一个Q网络，它不是经常吹牛嘛。那我就用两个Q网络，因为两个Q网络的参数有差别，所以对于同一个动作的评估也会有少许不同。选取评估出来较小的值来计算目标。这样就能避免Q网络吹牛的情况发生了。

另外一种做法也需要用到两个Q网络。Q1网络**推荐**能够获得最大Q值的动作；Q2网络计算这个动作在Q2网络中的Q值。

恰好，如果用上Fixed Q-targets，不就是有两个Q网络了吗？

所以你可以看到，这个优化在DQN上很容易实现。这就是doubleDQN和DQN的唯一的变化。

具体的实现，请看代码分析。（ 由于tensorflow提供的示例代码对于新手来说有点复杂，我另外写了一个代码以供说明。）

### 代码分析

流程： 如果已经熟悉了之前DQN的代码，那么主流程就很容易明白了。现在主要关注 Experience replay。

我把不相关的代码隐藏掉，如下：

def train(self,episode):

...

for ep in range(episode): #1

s = self.env.reset()

...

while True:

a = self.get\_action(s) #2

s\_,r,done,\_ = self.env.step(a) #3

self.remember(s, a, s\_, r, done) #4

s = s\_

if len(self.memory)>self.batch:

loss = self.update\_Q\_network() #5

....

if (step+1)%5 == 0: #6

self.update\_epsilon()

self.update\_target\_Q()

1. 开始一次游戏，用self.env.reset()把状态设置成初始状态。
2. 把s代入get\_action函数，获得action。get\_action函数采用epsilon-greedy方法，在之前已经讲述过，这里就不再赘述。
3. 把action代入环境，获得下一个状态s\_,奖励r，结束标记done，和info（由于没有用到相关的info，所以用\_,忽略掉）
4. 用 self.remember(s, a, s\_, r, done)， 把这个过程都记录到self.memory
5. 如果的记录足够，那么就进行更新。
6. 设置一个间隔，每更新5步，就更新target\_Q，并更新epsilon

从上面可以看成，doubleDQN其实和DQN的流程没什么不同，不过加上了一步，把智能体和环境互动产生的数据remember下来，用于经验回放。

那具体怎样把数据保存起来呢？可以看一下。

首先，保存先得有容器。容器的选择有很多，通常是列表。在示例代码中用了双端deque，因为它的数据结构能够快速随机访问其中任何一个元素，利于随机获取其中的数据用于更新。

把deque最大值为2000，当插入元素超过最大值，那么就会删除最初插入的元素。这个过程deque自动进行，不需要手动管理，非常方便。

from collections import deque

...

self.memory = deque(maxlen=2000)

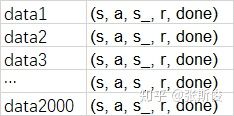
准备工作已经完成了，看remember函数，把需要存储的数据s, a, s\_, r, done传入，先用data把数据打包成一个元组，memory.append(data)，插入到队列中。

def remember(self, s, a, s\_, r, done):

data = (s, a, s\_, r, done)

self.memory.append(data)

整个结构如下图：



现在每次有state转换，都会被存储到memory这个队列中。当数据足够，例如大于batchsize，那么就从队列中，随机抽取batchsize个data，用于更新。

但buffer是不能直接更新的，最好先整理更新目标。看process\_data前半段代码。

def process\_data(self):

data = random.sample(self.memory, self.batch)

s = np.array([d[0] for d in data])

a = [d[1] for d in data]

s\_ = np.array([d[2] for d in data])

r = [d[3] for d in data]

done = [d[4] for d in data]

关注两点：

1. random.sample(self.memory, self.batch) random.sample，就是从队列中，随机抽出一定个数的数据。 例如抽取了红色部分的数据，这些新数据组成data，这部分数据就是用于这一次更新。

2. 用[d[0] for d in data]形式处理data。 构造一个循环，来访问data里面的一列数据。所以这里的d，其实相当于在memory函数里面的data，也就是每一步产出数据构成的元组。用d[0]，表示这个元组的第一个元素，也就是s。外面用中括号括起来，表示把这些数据构成一个数组。

这里s和s\_是需要输入到神经网络的，所以把数组，变成array。

下面就是这一篇的重点：如何用双Q网络更新和计算target。

在代码中，使用两个网络：

* Q\_network用于训练更新。
* target\_Q\_network 是固定网络，用于计算target。

注意: 如果用到了tensorlayer，建立网络后，这个网络是否用于训练，需要用一行代码说明，否则将会报错。例如

self.Q\_network = self.get\_model()

self.Q\_network.train() # 这个网络是可以训练的。

self.target\_Q\_network = self.get\_model()

self.target\_Q\_network.eval() # 这个网络是不可以训练的。

经过一定的episode之后，再把Q\_network的的参数，赋值给target\_Q\_network。

在赋值的时候，可以通过一个循环，逐个用assign()函数对模型的参数进行赋值。

def update\_target\_Q(self):

for i , target in zip(self.Q\_network.trainable\_weights, self.target\_Q\_network.trainable\_weights):

target.assign(i)

**计算target**

在process\_data函数中，获取一个batch的数值，继续处理一下数据。希望把state输入到网络，输出每个动作的Q值，然后和target做比较，从而算出loss。

这里可以和DQN做一下比较。如果是DQN，可以这样计算target。

在示例代码中：

1. 计算y，y是某个状态下的Q值，它的数据结构就是[q1,q2],q1,a2分别是两个不同的动作产生的q值。这相当于在Qlearning，查询qtable中，某个state那一行。
2. 用target\_Q\_network计算下一个state的Q值，数据结构仍然是[q1,q2]。这相当于在另外一个Qtable中，查询下一个state的Q值。
3. 现在逐个计算data数据里面每一行的target。如果是最终状态，那么target=r，否则用target = r + gamma \* Q(s')更新。
4. 修改y对应action的Q值。

y = self.Q\_network(np.array(s,dtype='float32')).numpy() #1

Q1 = self.target\_Q\_network(np.array(s\_,dtype='float32')) #2

for i ,(\_,a,\_,r,done) in enumerate(data): #3

if done:

target = r

else:

target = r + self.gamma \* np.max(Q1[i])

target = np.array(target,dtype='float32')

y[i][a] = target #4

现在来修改成 doubleDQN。 之前说过doubleDQN需要两个网络，一个负责提供action，一个提供该action的Q值。

在示例代码中，用Q\_network提供action(#1)，target\_Q\_network提供这个action的Q值。

也就是之前在double DQN说明中，大家可以对比一下。

y = self.Q\_network(np.array(s,dtype='float32')).numpy()

Q1 = self.target\_Q\_network(np.array(s\_,dtype='float32'))

Q2 = self.Q\_network(np.array(s\_,dtype='float32')) #1

next\_action = np.argmax(Q2,axis=1)

for i ,(\_,a,\_,r,done) in enumerate(data):

if done:

target = r

else:

#target = r + self.gamma \* np.max(Q1[i]) #DQN

target = r + self.gamma \* Q1[i][next\_action[i]] #double DQN

target = np.array(target,dtype='float32')

y[i][a] = target

计算好target，就可以更新了,详细代码我会放到我的仓库里。

### 总结

在这篇专栏，学习到2个对于DQN很重要的技巧:经验回放和固定目标网络，也用上了双Q网络计算目标。这些技巧在后续更高阶的算法也会用到。