# 用可视化直观理解DQN[DQN实战篇]

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

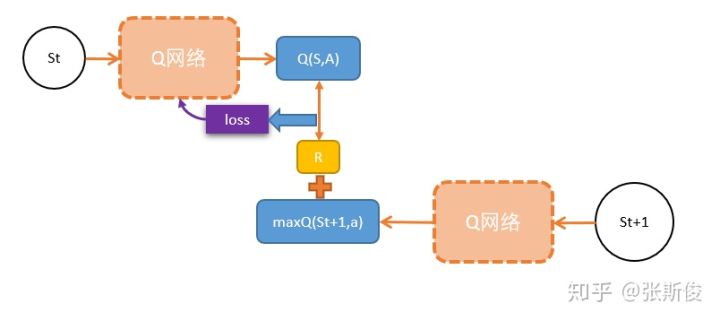
[张斯俊](https://www.zhihu.com/people/zhang-si-jun-52" \t "/home/admin/文档\\x/_blank)

愿成为一把梯子，助你跨过无数的坑。

于是可以用Qlearning算法修改一下，变成DQN的算法。

其实和Qlearning很像，修改如下.

1. 初始化一个网络，用于计算Q值。（在Qlearning，用Qtable）
2. 开始一场游戏。
3. 随机从一个状态s开始。
4. 把s输入到Q，计算s状态下，a的Q值 Q（s）
5. 选择能得到最大Q值的动作a
6. 把a输入到环境，获得新状态s’,r,done
7. 计算目标 y = r + gamma \* maxQ(s')
8. 训练Q网络，缩小Q（s，a）和y 的大小
9. 开始新一步，不断更新



如果对算法原理还不是太清楚，可以先看一下DQN理论篇：

[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)

以下，就用tensorflow的DQN代码作为示例，一起看看DQN应该如何实现。

tensorflow示例代码：

[https://github.com/tensorlayer/tensorlayer/blob/master/examples/reinforcement\_learning/tutorial\_DQN.py​github.com](https://link.zhihu.com/?target=https://github.com/tensorlayer/tensorlayer/blob/master/examples/reinforcement_learning/tutorial_DQN.py" \t "/home/admin/文档\\x/_blank)

如果看上面的代码有困难，可以看我的带注释版本。希望能帮助到你。

[https://github.com/louisnino/RLcode/blob/master/tutorial\_DQN.py​github.com](https://link.zhihu.com/?target=https://github.com/louisnino/RLcode/blob/master/tutorial_DQN.py" \t "/home/admin/文档\\x/_blank)

### Epsilon-greedy

先来说一个在示例代码中的实做技巧。

还记得之前在Qlearning使用了noisy-greedy的方式，保持探索和开发之间的平衡么？

智能体会根据Q值表选择Q值最大的动作。但在选择动作之前，会先给Q值加上一个随机的噪音，使得最终的最大值带有一定的随机性；随着游戏进行，噪音会逐渐减少，最终噪音将会小得不再影响智能体的决定。

在DQN，你同样可以使用增加噪音的方式，不过，在示例代码中，用另外一种方式处理。这就是更常用的Epsilon-greedy。其目的是相同的，就是为了保证大部分的state都能被探索到的基础上，最终也能够按照智能体学习到的方式取进行。

请大家留意示例代码中：epsilon = 0.1，这里的epsilon 就代表Epsilon，可以理解为一个门槛。

在选择动作的时候，会先随机一个[0,1]之间的值： 如果随机出来的值，高于这个门槛，智能体将会忽略掉Q值，而直接随机一个动作。这就是**探索性的动作**。

如果随机出来的值，低于这个门槛，智能体将会按照原来的**贪婪算法**，直接选择最大的Q值的动作。这就是开发性的动作。

Epsilon-greedy用大白话说就是：如果随机出来的值小于Epsilon这个门槛，就用greedy算法吧！

看示例代码中，每一次迭代（每一次游戏）进行会调整一次e。

epsilon = 1. / ((i / 50) + 10)

随着i越来越大，e将会越来越小。也就是说，门槛会随着迭代次数，越来越小，执行greedy算法的机会将会越来越多。让智能体逐渐从**探索**变为**开发**。

### 具体算法的解释：

与Qlearning的更新流程一样。其实，如果你已经明白了深度神经网络和Qlearning，那么代码看起来是很简单了。

1、先建立一个

### 建立网络

def get\_model(inputs\_shape):

ni = tl.layers.Input(inputs\_shape, name='observation')

nn = tl.layers.Dense(4, act=None, W\_init=tf.random\_uniform\_initializer(0, 0.01), b\_init=None, name='q\_a\_s')(ni)

return tl.models.Model(inputs=ni, outputs=nn, name="Q-Network")

示例代码用上了tensorlayer，一般来说用tensorflow也很方便。这里必须吐槽一下，tensorlayer和tensorflow，同样的层，同样功能的参数，名字居然不一样！还有keras里面的layer,各种坑。 但后面的代码都用上了tensorlayer，so。。。

输入层：示例代码中用ni表示。inputs\_shape，就是输入矩阵的形状。在强化学习中，需要把状态输入到网络中，所以inputs\_shape就是状态的形状了。

qnetwork = get\_model([None, 16])

后面调用的时候，输入的是[None, 16]。注意，前面是状态的数量，也就是说，会输入一个batch大小的数

nn = tl.layers.Dense(4, act=None, W\_init=tf.random\_uniform\_initializer(0, 0.01), b\_init=None, name='q\_a\_s')(ni)

和之前Fasion MNIST一样，用Dense作为输出。这里没有加一个Dense层，主要是因为这里的状态空间只有16个，这样的模型拟合能力已经足够。

在更新的时候，有两个地方要注意。

**一、 热独编码**

allQ = qnetwork(np.asarray([to\_one\_hot(s, 16)], dtype=np.float32)).numpy()

一、这里用了to\_one\_hot函数，把原来的编码，变成**热独编码**。

def to\_one\_hot(i, n\_classes=None):

a = np.zeros(n\_classes, 'uint8')

a[i] = 1

return a

那什么是热独编码呢？

现在有16个不同的state，如果按照一般的分类，可能会把这16个state变成，1,2,3,...,15,16 类。

在机器学习里面，0,1,2,3..这些数字其实是包含了一个大小的关系的，但在分类问题里面，其实每一类都是平等的，并没有大小关系。因此，需要把他变成热独编码(One hot)的形式。

热独编码这个名字其实很形象,就是**只有一点是热的**。

构造一个矩阵，只有该分类的标志位为1，其他全部为0.矩阵的大小，就是类别的个数。所以热独热独，热就是1表示的位置。

例如 类别1：[1 0 0 0 0 0 0 0 0....0] 类别2：[0 1 0 0 0 0 0 0 0....0] 类别16：[0 0 0 0 0 0 0 0 0....1]

回到示例代码的to\_one\_hot函数就很好理解了： 先构造一个全零的矩阵，大小是[1,n\_classes] (n\_classes类别个数)

a = np.zeros(n\_classes, 'uint8')

然后在对应的标志位上，把0变成1，并返回。

a[i] = 1

**二、数据的形状和格式**

输入：

np.asarray([to\_one\_hot(s, 16)], dtype=np.float32)

之前说过，神经网络其实可以看做是一个数据的工厂，而在实际写代码的过程中，这个数据工厂有时候并不那么透明，或者并不那么可读。可以说，数据工厂就像一个黑盒子一样，在debug中出来问题，还是比较棘手的。

所以要在输入之前，保证这个工厂的数据是标准的。最好做到以下3点：

1. 神经网络输入的形状需要和input\_shape对应;
2. 输入格式必须是一个array;
3. 数据的格式，最好都变成float32的形式。

在输入之前能先处理好这3点，能够减少很多不必要的麻烦。

输出：**.numpy()**

数据在输入神经网络这个工厂之后，将会变成另外一种格式：tensor（张量）。最终产生的数据格式也是tensor。

tensor可以理解为工厂专用的array吧。但有时候，numpy对array的操作并不能直接用到tensor上。所以，在后面加上 .numpy(),把tensor转为array的。

**三、【敲黑板】构造更新目标**

我把关键代码提出来大家看看

DQN:

allQ = qnetwork(np.asarray([to\_one\_hot(s, 16)], dtype=np.float32)).numpy()

a = np.argmax(allQ, 1)

...

Q1 = qnetwork(np.asarray([to\_one\_hot(s1, 16)], dtype=np.float32)).numpy()

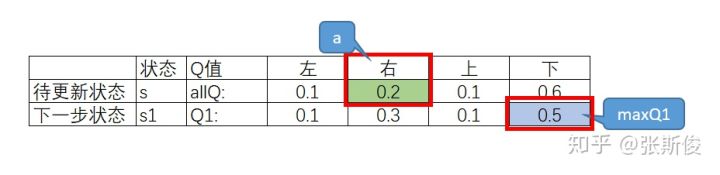
maxQ1 = np.max(Q1)

targetQ = allQ

targetQ[0, a[0]] = r + lambd \* maxQ1

举个例子： 假设需要更新s状态的Q值(allQ),allQ是所有动作的Q值[0.1 0.2 0.1 0.6]。 如果用贪婪算法，会选出的动作是【下】，因为【下】的Q值最大。但很遗憾，由于采用epsilon-greedy，这次随机并随机到a是动作2【右】

然后求出下一状态s1的Q值(Q1),现在有下图：



其中，绿色格子代表动作a。蓝色格子是Q1的最大值，也就是maxQ1 = np.max(Q1)。

现在需要构造target，因此把allQ先复制给targetQ。然后把r + lambd \* maxQ1复制给targetQ中对应a的位置。

所以，现在有allQ向targetQ更新：



可视化一下，就可以看到Q网络的调整方向：动作1,3,4将会不变，而将会把动作2的Q值向targetQ的动作2方向靠近。

### 总结

DQN = Qlearning + network

Qlearning需要依赖Qtable，所以没办法处理连续state的问题。神经网络可以理解成一个magic函数。这个函数式支持连续型的输入的。

用Qlearning的方法，最终可以把这个神经网络，拟合成一个曲面。而每个state就是曲面的一个截面。可以从截面看到每个动作的Q值分布。

而在更新的过程中，算法仍然是TD（0）算法。

但实际上，示例代码偷了个了懒，虽然DQN是可以解决连续state的问题。但用的环境，依然是离散的。大家可以试着改一下代码，完成CartPole-v0环境。

如果失败了，也没关系。后面我将会介绍几个DQN的变种，包括DoubleDQN，DuelingDQN， 。在这个过程中也会学到Experience replay和Fixed Targets Network的技术，把DQN的技术用到极致。