神经网络-全连接层（1）

本文收录在无痛的机器学习第一季。

写在前面：感谢
@夏龙
对本文的审阅并提出了宝贵的意见。

接下来聊一聊现在大热的神经网络。最近这几年深度学习发展十分迅速，感觉已经占据了整个机器学习的“半壁江山”。各大会议也是被深度学习占据，引领了一波潮流。深度学习中目前最火热的两大类是卷积神经网络（CNN）和递归神经网络（RNN），就从这两个模型开始聊起。

当然，这两个模型所涉及到概念内容实在太多，要写的东西也比较多，所以为了能把事情讲得更清楚，这里从一些基本概念聊起，大神们不要觉得无聊啊……

今天扯的是全连接层，也是神经网络中的重要组成部分。关于神经网络是怎么发明出来的这里就不说了。全连接层一般由两个部分组成，为了后面的公式能够更加清楚的表述，以下的变量名中**上标表示所在的层，下标表示一个向量或矩阵内的行列号**：

- 线性部分：主要做线性转换，输入用X表示，输出用Z表示
- 非线性部分：那当然是做非线性变换了，输入用线性部分的输出Z表示，输出用X表示。

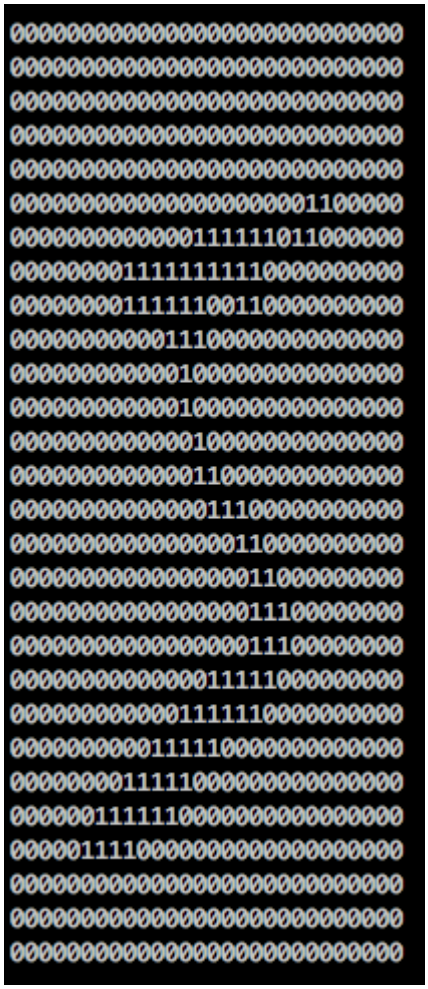
线性部分

线性部分的运算方法基本上就是线性加权求和的感觉，如果对于一个输入向量 $x=[x0,x1,...xn]Tx=[x_0,x_1,...x_n]^T$，线性部分的输出向量是

z=[z0,z1,z2,...zm]Tz=[z_0,z_1,z_2,...z_m]^T，那么线性部分的参数就可以想象一个m*n的矩阵W，再加上一个偏置项 b=[b0,...bm]Tb=[b_0,...b_m]^T，于是有：

线性部分做了什么事情呢？简单来说就是对输入数据做不同角度的分析，得出该角度下对整体输入数据的判断。

这么说有点抽象，举一个实际点的例子，就拿CNN的入门case——MNIST举例。MNIST的例子在此不多说了，它是一个手写数字的识别项目，输入是一张28*28的二值图，输出是0-9这是个数字，这里假设我们采用完全全连接的模型，那么我们的输入就是28*28=784个像素点。数据显示到屏幕上大概是这个样子：



对于我们来说，这个像素点都太过于抽象了，我们无法判断这些像素点的取值和最终识别的关系：

他们是正相关还是负相关？

很显然，像素点之间是存在相关关系的，这个关系具体是什么我们后面再说，但存在关系这件事是板上钉钉的。所以只给每一个像素点一个权重是解决不了问题的，我们需要多组权重。

我们可以

1）在第一组权重中给第一个像素一个正数，第二个也是正数，

2）在第二组权重中给第一个像素负数，而第二个还是正数……

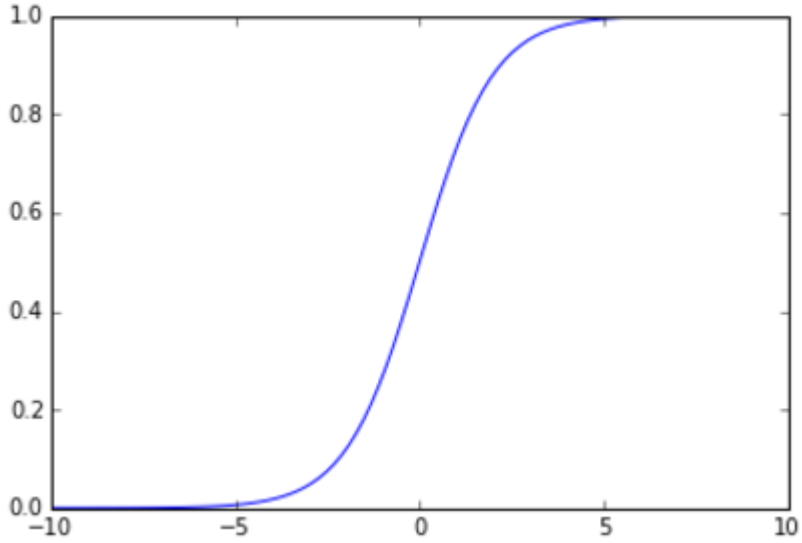
这样，我们相当于从多个角度对输入数据进行分析汇总，得到了多个输出结果，也就是对数据的多种评价。

非线性部分

非线性部分有一些“套路”函数，这里只说下其中的一个经典函数——sigmoid。它的函数形式如下所示：

$$f(x)=11+e-xf(x)=\frac{1}{1+e^{-x}}$$

图像如下所示：



这个函数的输入正是我们上一步线性部分的输出z，此时z取值范围在 $(-\infty,+\infty)(-\infty ,+\infty)$，经过了这个函数就变成了 $(0,1)(0,1)$。

那非线性部分为什么要做这个函数转换呢？以我的粗浅理解，其中的一个作用就是作数据的归一化。不管前面的线性部分做了怎样的工作，到了非线性这里，所有的数值将被限制在一个范围内，这样后面的网络层如果要基于前面层的数据继续计算，这个数值就相对可控了。不然如果每一层的数值大小都不一样，有的范围在（0，1），有的在（0，10000），做优化的时候优化步长的设定就会有麻烦。

另外一个作用，就是打破之前的线性映射关系。如果全连接层没有非线性部分，只有线性部分，我们在模型中叠加多层神经网络是没有意义的，我们假设有一个2层全连接神经网络，其中没有非线性层，那么对于第一层有：

$$W0*x0+b0=z1W^0*x^0+b^0=z^1$$

对于第二层有：

$$W1*z1+b1=z2W^1*z^1+b^1=z^2$$

两式合并，有

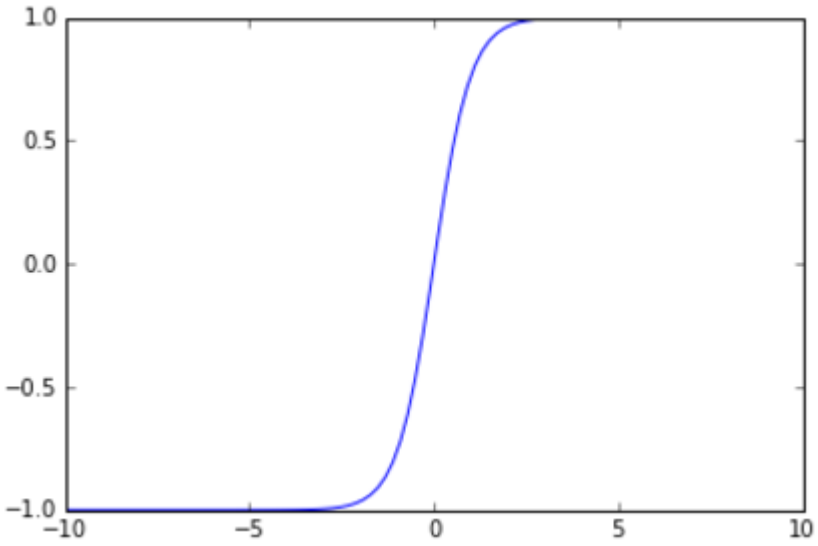
W1∗(W0∗x0+b0)+b1=z2W^1∗(W^0∗x^0+b^0)+b^1=z^2

W1∗W0∗x0+(W1∗b0+b1)=z2W^1∗W^0∗x^0+(W^1∗b^0+b^1)=z^2

所以我们只要令 W0’=W1∗W0W^{0'}=W^1∗W^0 ，b0’=W1∗b0+b1b^{0'}=W^1∗b^0+b^1，就可以用一层神经网络表示之前的两层神经网络了。所以非线性层的加入，使得多层神经网络的存在有了意义。

另外还有一个比较有名的非线性函数，叫做双曲正切函数。它的函数形式如下所示：

f(x)=ex-e-xex+e-xf(x)=\frac{e^x-e^{-x}}{e^x+e^{-x}}



这个长得很复杂的函数的范围是(-1,1)。可以看出，它的函数范围和前面的sigmoid不同，它是有正有负的，而sigmoid是全为正的。

神经网络的模样

实际上对于只有一层且只有一个输出的神经网络，如果它的非线性部分还使用sigmoid函数，那么它的形式和逻辑斯特回归（logistic

regression）是一样的。所以可以想象神经网络模型从概念上来看比逻辑斯特回归要复杂。那么它的复杂的样子是什么样呢？下面给出一段全连接层的代码，开始做实验：

```
class FC:
    def __init__(self, in_num, out_num, lr = 0.01):
        self._in_num = in_num
        self._out_num = out_num
        self.w = np.random.randn(out_num, in_num) * 10
        self.b = np.zeros(out_num)
    def _sigmoid(self, in_data):
        return 1 / (1 + np.exp(-in_data))
    def forward(self, in_data):
        return self._sigmoid(np.dot(self.w, in_data) + self.b)
```

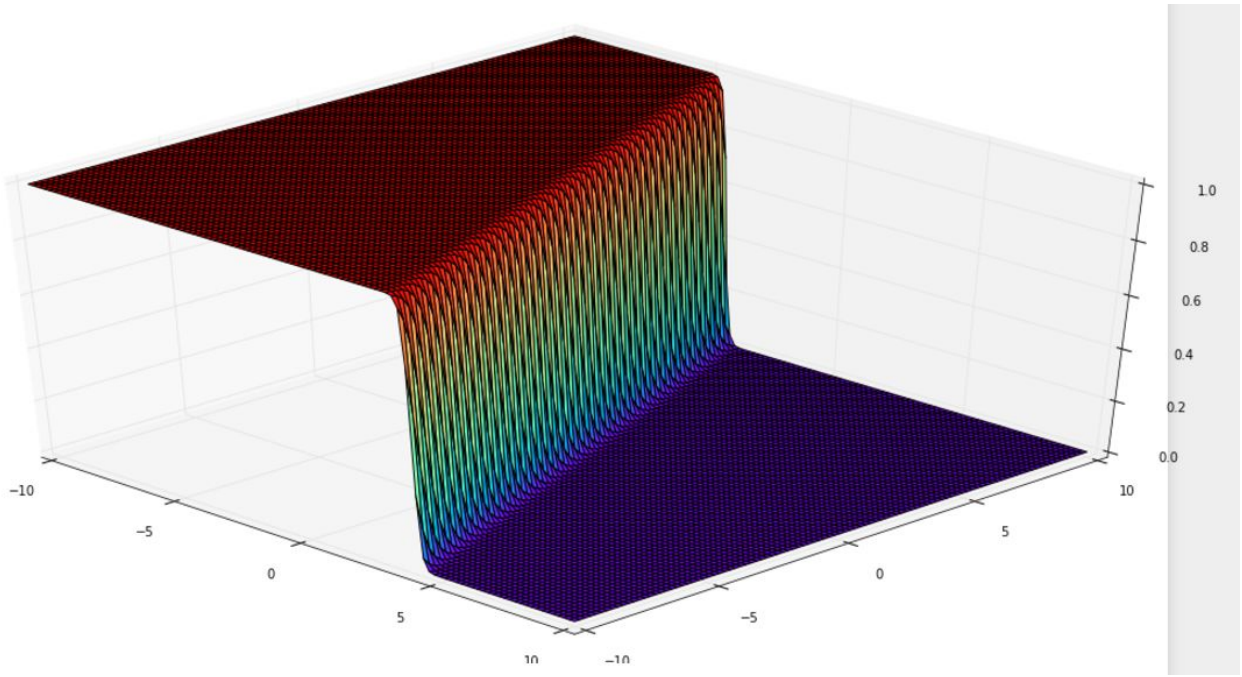
从代码上看东西并不多嘛，注意到我们会对参数中的w进行随机初始化，有时我们会让老天随机一个神经网络给我们，我们也可以看看随机大帝的旨意。

为了方便可视化，这里只做输入为2，输出为1的数据。好了，先来看1号选手：

```
x = np.linspace(-10,10,100)
y = np.linspace(-10,10,100)
X, Y = np.meshgrid(x,y)
X_f = X.flatten()
Y_f = Y.flatten()
data = zip(X_f, Y_f)

fc = FC(2, 1)
Z1 = np.array([fc.forward(d) for d in data])
Z1 = Z1.reshape((100,100))
draw3D(X, Y, Z1)
```

定睛一看这其实就是一个标准的Logistic Regression。他的图像如下所示：



经过多次随机测试，基本上它都是这个形状，只不过随着权重随机的数值变化，这个“台阶”对旋转到不同的方向，但归根结底还是一个台阶。

这也说明1层神经网络是没有出路的，它本质上还是个线性分类器的实力，那么小伙伴还给它加一层吧：

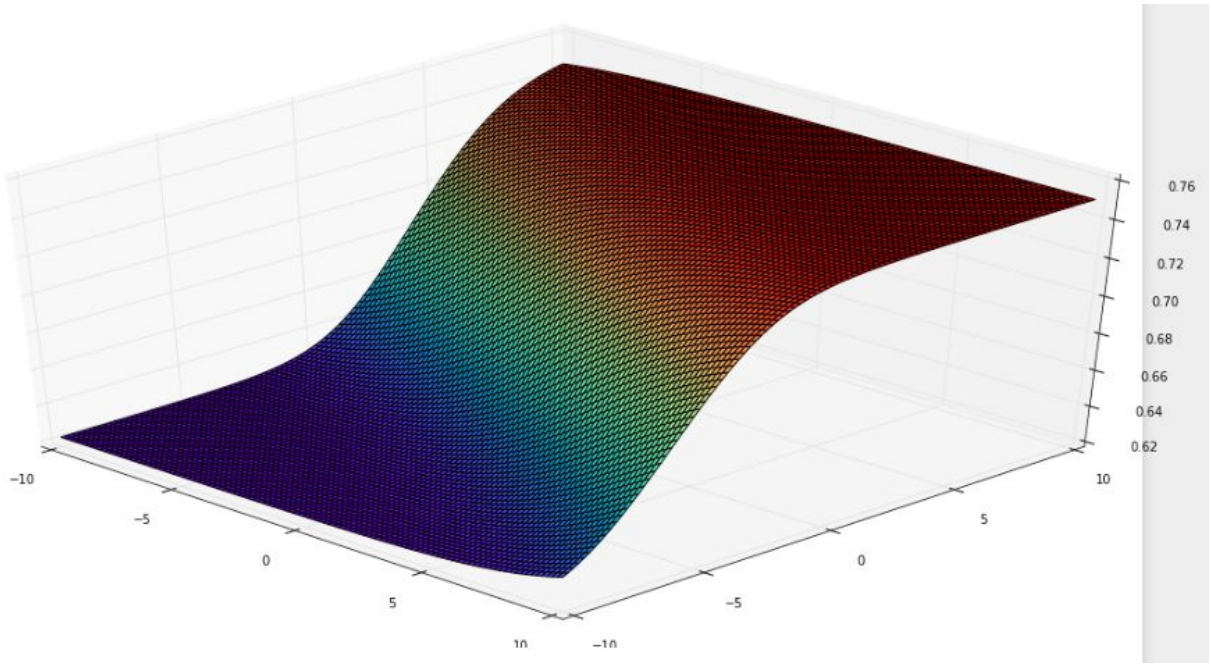
```
fc = FC(2, 3)
fc.w = np.array([[0.4, 0.6],[0.3,0.7],[0.2,0.8]])
fc.b = np.array([0.5,0.5,0.5])

fc2 = FC(3, 1)
fc2.w = np.array([0.3, 0.2, 0.1])
fc2.b = np.array([0.5])

Z1 = np.array([fc.forward(d) for d in data])
Z2 = np.array([fc2.forward(d) for d in Z1])
Z2 = Z2.reshape((100,100))

draw3D(X, Y, Z2)
```

这次我们暂时不用随机权重，而是自己设置了几个数值，可以看出，参数设置得很用心。两层全都是正数……，那么图像呢？



看上去比之前的台阶“柔软”了一些，但归根结底还是很像一个台阶……好吧，那我们加点负权重，让我们从两个方面分析输入数据：

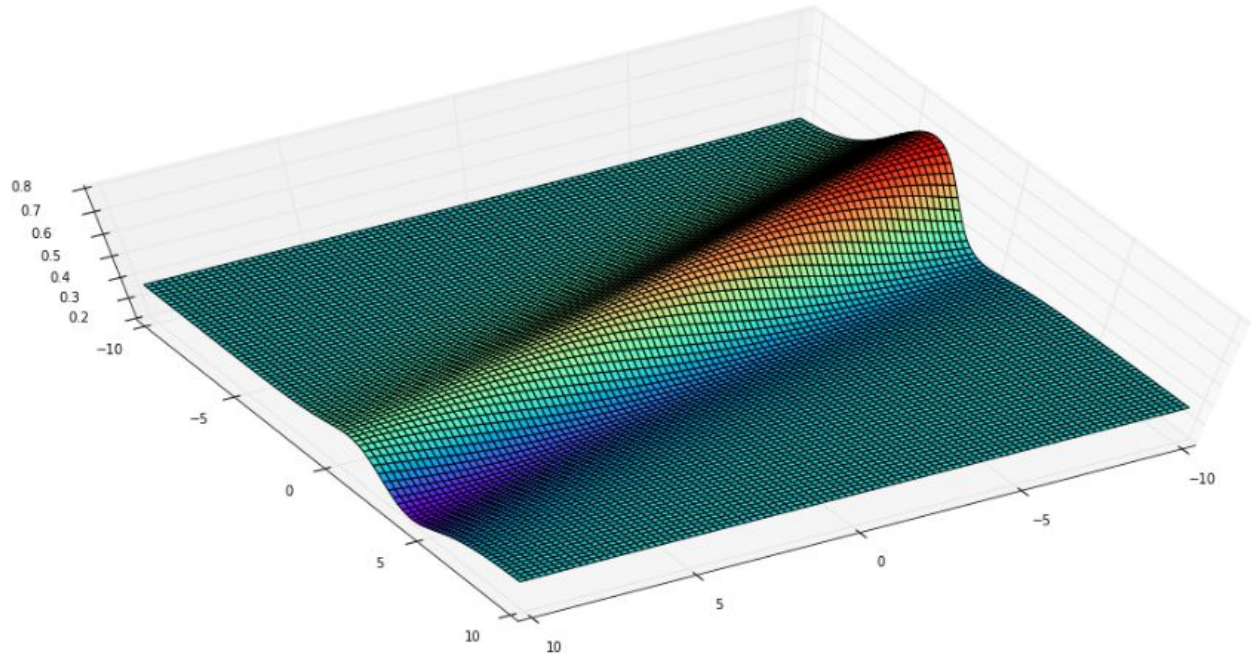
```
fc = FC(2, 3)
fc.w = np.array([[-0.4, 1.6],[-0.3,0.7],[0.2,-0.8]])
fc.b = np.array([-0.5,0.5,0.5])

fc2 = FC(3, 1)
fc2.w = np.array([-3, 2, -1])
fc2.b = np.array([0.5])

Z1 = np.array([fc.forward(d) for d in data])
Z2 = np.array([fc2.forward(d) for d in Z1])
Z2 = Z2.reshape((100,100))

draw3D(X, Y, Z2)
```

赶紧上图：

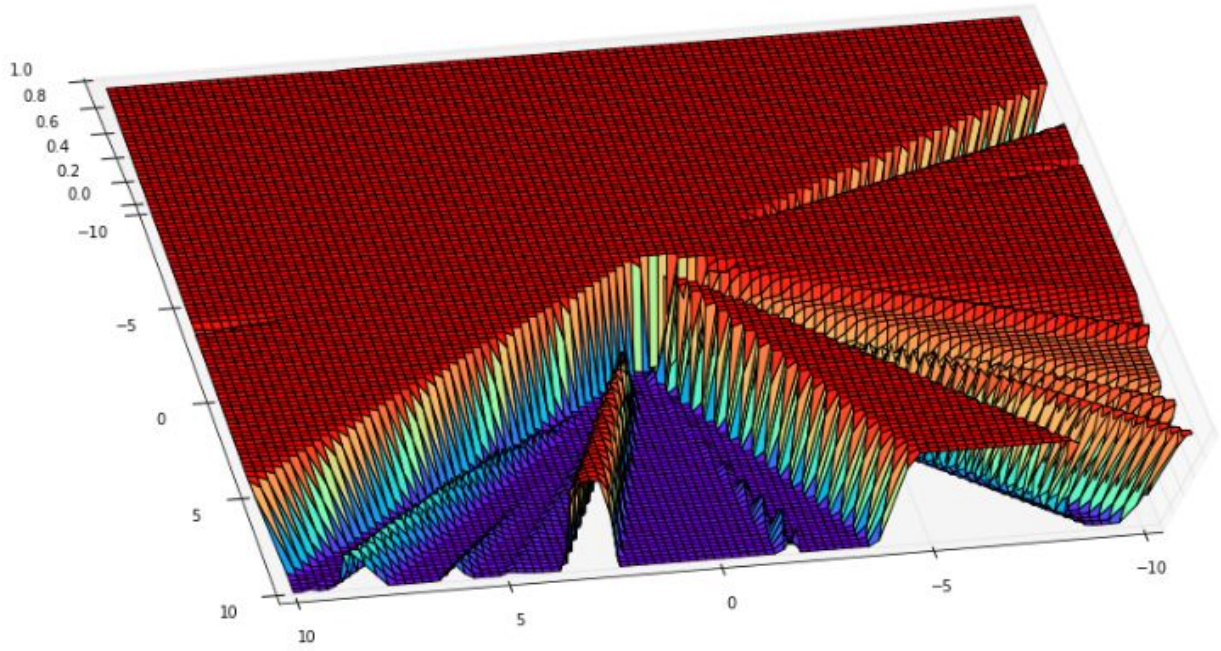


加了负权重后，看上去终于不那么像台阶了，这时候2层神经网络的非线性能力开始显现出来了。下面把权重交给随机大帝：

```
fc = FC(2, 100)
fc2 = FC(100, 1)

Z1 = np.array([fc.forward(d) for d in data])
Z2 = np.array([fc2.forward(d) for d in Z1])
Z2 = Z2.reshape((100,100))
draw3D(X, Y, Z2,(75,80))
```

上图：

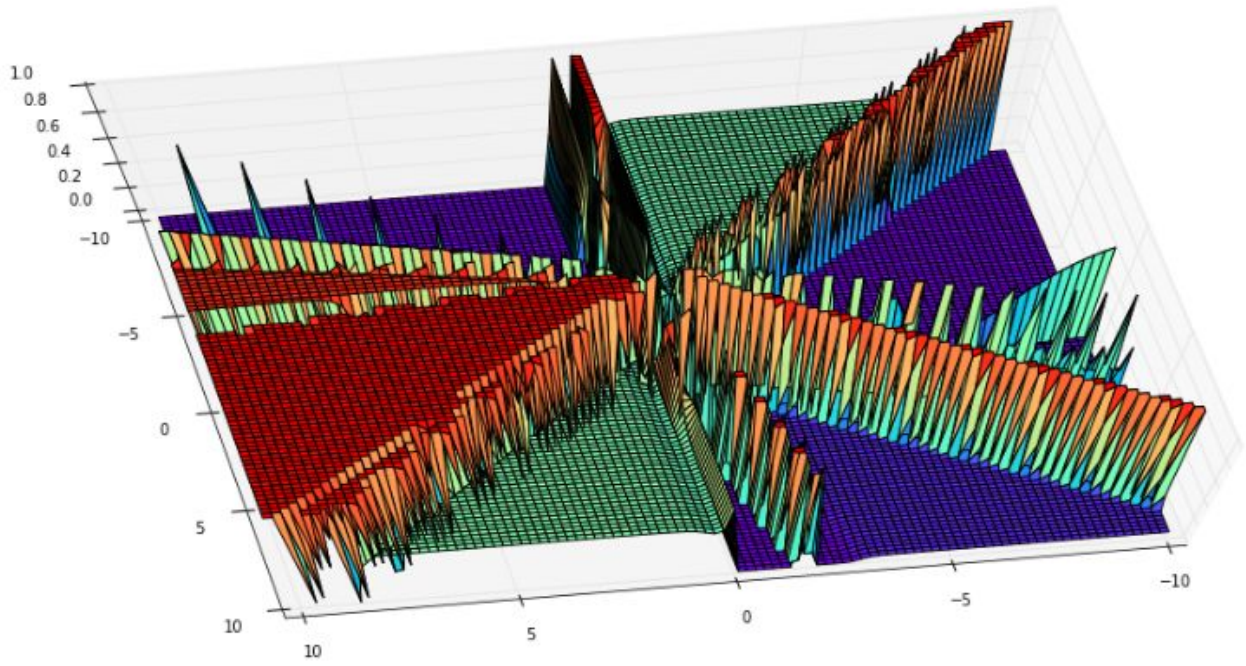


这时候的非线性已经非常明显了,我们不妨继续加几层看看DNN的厉害:

```
fc = FC(2, 10)
fc2 = FC(10, 20)
fc3 = FC(20, 40)
fc4 = FC(40, 80)
fc5 = FC(80, 1)

Z1 = np.array([fc.forward(d) for d in data])
Z2 = np.array([fc2.forward(d) for d in Z1])
Z3 = np.array([fc3.forward(d) for d in Z2])
Z4 = np.array([fc4.forward(d) for d in Z3])
Z5 = np.array([fc5.forward(d) for d in Z4])
Z5 = Z5.reshape((100,100))
draw3D(X, Y, Z5,(75,80))
```

这个图看上去又复杂了许多……



从上面的实验中可以看出，层数越高，非线性的“能力”确实越强，脑洞开得也越大。

知道了他的厉害，下回我们将详细聊下它的求解方法—反向传播（Back Propagation）。

文章代码可以在hsmyy/zhihuzhuanlan 找到