> 《2021 PyTorch深度学习实战》
>
> Lesson 9：深层神经网络
>
> 作者：@菜菜TsaiTsai
>
> 版本号：2021/01/28 V1.0

Lesson 9 深层神经网络

在之前的介绍中，我们已经学习了三种单层神经网络，分别为实现线性方程的回归网络，实现二分类的逻辑回归（二分类网络），以及实现多分类的softmax回归（多分类网络），我们已经基本了解了构成普通神经网络的诸多元素。从本节课开始，我们将从单层神经网络展开至深层神经网络，并深入理解层、以及层上的各类函数计算对于神经网络的意义。

从单层到多层是神经网络发展史上的重大变化，层的增加彻底将神经网络的性能提升到了另一个高度，正确理解层的意义对于我们自主构建神经网络有很重要的作用，学会利用层是避免浪费计算资源以及提升神经网络效果的关键。在本节的最后，我们还将继承nn.Module类实现一个完整的深层神经网络的前向传播的过程。在这个过程中我们将会用到大量类相关的知识，如果你还不太了解类，请尽快复习B站公开课中，九天老师讲解类的创建与继承的部分：https://www.bilibili.com/video/BV1U54y1W7jw

Lesson 9 深层神经网络

一、异或门问题

还记得之前的课程中我们提到的数据“与门”（andgate）吗？与门是一组只有两个特征（x1，x2）的分类数据，当两个特征下的取值都为1时，分类标签为1，其他时候分类标签为0。

x0	x1	x2	y_and
1	0	0	0
1	1	0	0
1	0	1	0
1	1	1	1

之前我们使用tensor结构与nn.Linear类在“与门”数据上上实现了简单的二分类神经网络（逻辑回归），其中使用tensor结构定义时，我们自定义了权重向量w，并巧妙的让逻辑回归输出的结果与真实结果一致，具体代码如下：

```
import torch

X = torch.tensor([[1,0,0],[1,1,0],[1,0,1],[1,1,1]], dtype = torch.float32)
andgate = torch.tensor([0,0,0,1], dtype = torch.float32)

#定义w
w = torch.tensor([-0.2,0.15,0.15], dtype = torch.float32)

def LogisticR(X,w):
    zhat = torch.mv(X,w) #首先执行线性回归的过程，依然是mv函数，让矩阵与向量相乘得到z
    sigma = torch.sigmoid(zhat) #执行sigmoid函数，你可以调用torch中的sigmoid函数，也可以自己用torch.exp来写
    andhat = torch.tensor([int(x) for x in sigma >= 0.5], dtype = torch.float32) #设置阈值为0.5，使用列表推导式将值转化为0和1
    return sigma, andhat

sigma, andhat = LogisticR(X,w)

sigma

andhat

andgate

andgate == andhat
```

考虑到与门的数据只有两维，我们可以通过python中的matplotlib代码将数据可视化，其中，特征x_1为横坐标，特征x_2为纵坐标，紫色点代表了类别0，红色点代表类别1。

```
import matplotlib.pyplot as plt
import seaborn as sns

plt.style.use('seaborn-whitegrid') #设置图像的风格
sns.set_style("white")
plt.figure(figsize=(5,3)) #设置画布大小
plt.title("AND GATE",fontsize=16) #设置图像标题
plt.scatter(X[:,1],X[:,2],c=andgate,cmap="rainbow") #绘制散点图
plt.xlim(-1,3) #设置横纵坐标尺寸
plt.ylim(-1,3)
plt.grid(alpha=.4,axis="y") #显示背景中的网格
plt.gca().spines["top"].set_alpha(.0) #让上方和右侧的坐标轴被隐藏
plt.gca().spines["right"].set_alpha(.0);
```

在机器学习中，存在一种对数据进行分类的简单方式：就是在两类数据中间绘制一条直线，并规定直线一侧的点属于一类标签，下方另一侧的点属于另一类标签。而在这个由x_1与x_2构成的空间中，二维平面上的任意一条线可以被表示为：

$$x_1 = ax_2 + b$$

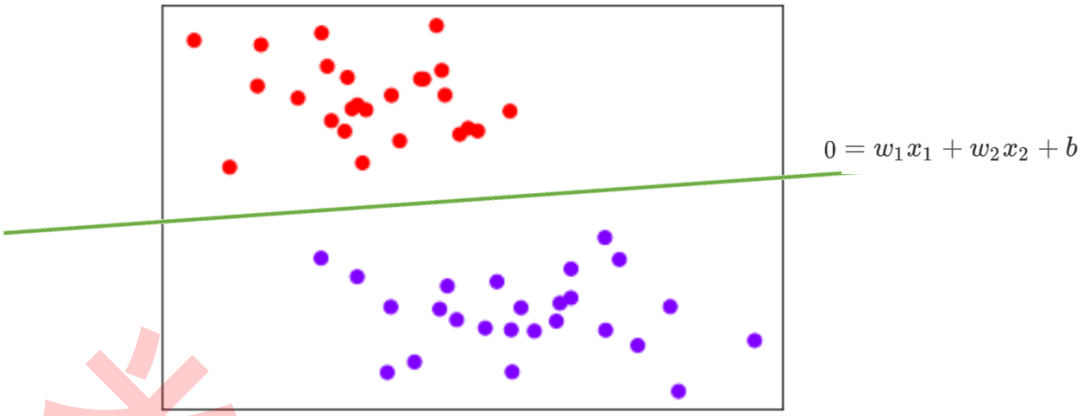
我们将此表达式变换一下：

$$0 = b - x_1 + ax_2$$

$$0 = [1, x_1, x_2] * \begin{bmatrix} b \\ -1 \\ a \end{bmatrix}$$

$$0 = \boldsymbol{Xw}$$

其中[b, -1, a]就是我们的权重向量$\boldsymbol{w}$（默认列向量），$\boldsymbol{X}$就是我们的特征张量，b是我们的截距。在一组数据下，给定固定的$\boldsymbol{w}$和b，这个式子就可以是一条固定直线，在$\boldsymbol{w}$和b不确定的状况下，这个表达式$\boldsymbol{X}\boldsymbol{w} = 0$就可以代表平面上的任意一条直线。在直线上方的点为1类（红色），在直线下方的点为0类（紫色），其颜色标注与刚才绘制的与门数据中的数据点一致。



如果在$\boldsymbol{w}$和b固定时，给定一个唯一的X的取值，这个表达式就可以表示一个固定的点。如果任取一个紫色的点X_p就可以被表示为：

$$\boldsymbol{X_p}\boldsymbol{w} = p$$

由于紫色的点所代表的标签y是0，所以**我们规定，p<=0**。同样的，对于任意一个红色的点X_r而言，我们可以将它表示为：

$$\boldsymbol{X_r}\boldsymbol{w} = r$$

由于红色点所表示的标签y是1，所以**我们规定，r>0**。由此，如果我们有新的测试数据$\boldsymbol{X_t}$，则的$\boldsymbol{X_t}$标签就可以根据以下式子来判定：

$$y = \begin{cases} 1, & if\ \boldsymbol{X_t}\boldsymbol{w} > 0 \\ 0, & if\ \boldsymbol{X_t}\boldsymbol{w} \leq 0 \end{cases}$$

在我们只有两个特征的时候，实际上这个式子就是：

$$y = \begin{cases} 1 & if\ w_1x_1 + w_2x_2 + b > 0 \\ 0 & if\ w_1x_1 + w_2x_2 + b \leq 0 \end{cases}$$

而$w_1x_1 + w_2x_2 + b$就是我们在神经网络中表示z的式子，所以这个式子实际上就是：

$$y = \begin{cases} 1 & if\ \boldsymbol{z} > 0 \\ 0 & if\ \boldsymbol{z} \leq 0 \end{cases}$$

你发现了吗？**用线划分点的数学表达和我们之前学习的阶跃函数的表示方法一模一样！**在Lesson 8中，我们试着使用阶跃函数作为联系函数替代sigmoid来对与门数据进行分类，最终的结果发现阶跃函数也可以轻松实现与sigmoid函数作为联系函数时的分类效果。我们现在把代码改写一下：

```
import torch
X = torch.tensor([[1,0,0],[1,1,0],[1,0,1],[1,1,1]], dtype = torch.float32)
andgate = torch.tensor([0,0,0,1], dtype = torch.float32)

def AND(X):
    w = torch.tensor([-0.2,0.15, 0.15], dtype = torch.float32)
    zhat = torch.mv(X,w)
    andhat = torch.tensor([int(x) for x in zhat >= 0],dtype=torch.float32)
    return andhat

andhat = AND(X)

andgate
```

实际上，阶跃函数的本质也就是用直线划分点的过程，而这条具有分类功能直线被我们称为"**决策边界**"。在机器学习中，任意分类算法都可以绘制自己的决策边界，且依赖决策边界来进行分类。（逻辑回归自然也有，不过因为sigmoid函数的决策边界绘制起来没有阶跃函数那么直接，因此我们没有选择逻辑回归的决策边界来进行绘制）。不难看出，既然决策边界的方程就是z的表达式，那在相同的数据$\boldsymbol{X}$下（即在相同的数据空间中），决策边界具体在哪里就是由我们定义的w决定的。

在之前的代码中，我们定义了$w_1 = 0.15, w_2 = 0.15, b = -0.23$，这些参数对应的直线就是$0.15x_1 + 0.15x_2 - 0.23$。我们可以用以下代码，将这条线绘制到样本点的图像上：

```
import numpy as np
x = np.arange(-1,3,0.5)
plt.plot(x,(0.23-0.15*x)/0.15 #这里是从直线的表达式变型出的x2 = 的式子
         ,color="k",linestyle="--");
```

可以看到，这是一条能够将样本点完美分割的直线。这说明，我们所设置的权重和截距绘制出的直线，可以将与门数据中的两类点完美分开。所以对于任意的数据，我们只需要找到适合的w和b就能够确认相应的决策边界，也就可以自由进行分类了。

现在，让我们使用**阶跃函数**作为线性结果z之后的函数，在其他典型数据上试试看使用决策边界进行分类的方式。比如下面的"或门"（OR GATE），特征一或特征二为1的时候标签就为1的数据。

x0	x1	x2	orgate
1	0	0	0
1	1	0	1
1	0	1	1
1	1	1	1

以及"非与门"（NAND GATE），特征一和特征二都是1的时候标签就为0，其他时候标签则为1的数据。

x0	x1	x2	nandgate
1	0	0	1
1	1	0	1
1	0	1	1
1	1	1	0

用以下代码，可以很容易就实现对这两种数据的拟合：

```
#定义数据
X = torch.tensor([[1,0,0],[1,1,0],[1,0,1],[1,1,1]], dtype = torch.float32)

#或门，或门的图像

#定义或门的标签
orgate = torch.tensor([0,1,1,1], dtype = torch.float32)

#或门的函数（基于阶跃函数）
def OR(X):
    w = torch.tensor([-0.08,0.15,0.15], dtype = torch.float32) #在这里我修改了b的数值
    zhat = torch.mv(X,w)
    yhat = torch.tensor([int(x) for x in zhat >= 0],dtype=torch.float32)
    return yhat

OR(X)

#绘制直线划分散点的图像
x = np.arange(-1,3,0.5)
plt.figure(figsize=(5,3))
plt.title("OR GATE",fontsize=16)
plt.scatter(X[:,1],X[:,2],c=orgate,cmap="rainbow")
plt.plot(x,(0.08-0.15*x)/0.15,color="k",linestyle="--")
plt.xlim(-1,3)
plt.ylim(-1,3)
plt.grid(alpha=.4,axis="y")
plt.gca().spines["top"].set_alpha(.0)
plt.gca().spines["right"].set_alpha(.0)


#非与门、非与门的图像
nandgate = torch.tensor([1,1,1,0], dtype = torch.float32)

def NAND(X):
    w = torch.tensor([0.23,-0.15,-0.15], dtype = torch.float32) #和与门、或门都不同的权重
    zhat = torch.mv(X,w)
    yhat = torch.tensor([int(x) for x in zhat >= 0],dtype=torch.float32)
    return yhat

NAND(X)

#图像
x = np.arange(-1,3,0.5)
plt.figure(figsize=(5,3))
plt.title("NAND GATE",fontsize=16)
plt.scatter(X[:,1],X[:,2],c=nandgate,cmap="rainbow")
plt.plot(x,(0.23-0.15*x)/0.15,color="k",linestyle="--")
plt.xlim(-1,3)
plt.ylim(-1,3)
plt.grid(alpha=.4,axis="y")
plt.gca().spines["top"].set_alpha(.0)
plt.gca().spines["right"].set_alpha(.0)
```

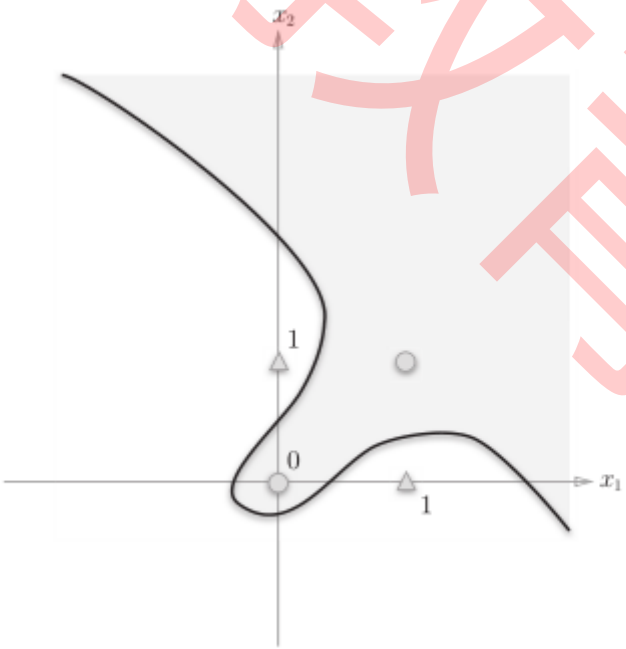
可以看到，或门和非与门的情况都可以被很简单地解决。现在，来看看下面这一组数据：

x0	x1	x2	y_xor
1	0	0	0
1	1	0	1
1	0	1	1
1	1	1	0

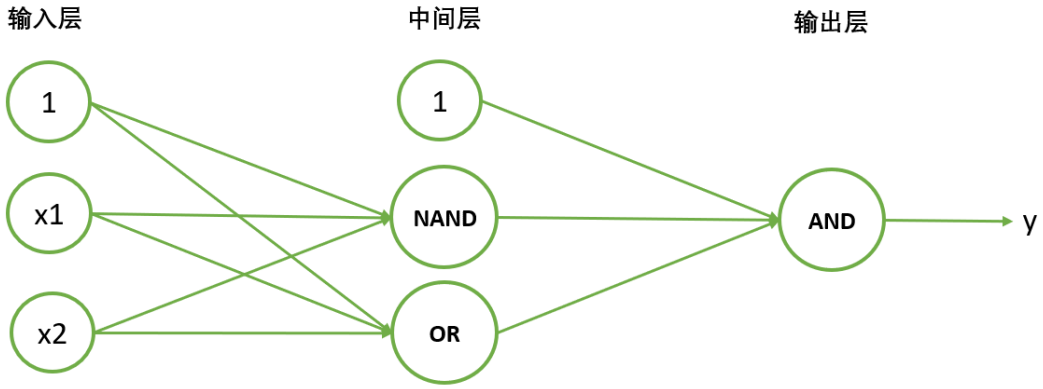
和之前的数据相比，这组数据的特征没有变化，不过标签y变成了[0,1,1,0]。这是一组被称为“异或门”（XOR GATE）的数据，可以看出，当两个特征的取值一致时，标签为0，反之标签则为1。我们同样把这组数据可视化看看：

```
xorgate = torch.tensor([0,1,1,0], dtype = torch.float32)

plt.figure(figsize=(5,3))
plt.title("XOR GATE",fontsize=16)
plt.scatter(X[:,1],X[:,2],c=y_xor,cmap="rainbow")
plt.xlim(-1,3)
plt.ylim(-1,3)
plt.grid(alpha=.4,axis="y")
plt.gca().spines["top"].set_alpha(.0)
plt.gca().spines["right"].set_alpha(.0)
```

问题出现了——很容易注意到，现在没有任意一条直线可以将两类点完美分开，所以无论我们如何调整w和b的取值都无济于事。这种情况在机器学习中非常常见，而现实中的大部分数据都是无法用直线完美分开的（相似的是，线性回归可以拟合空间中的直线，但大部分变量之间的关系都无法用直线来拟合，这也是线性模型的局限性）。此时我们会需要类似如下的曲线来对数据进行划分：



在神经网络诞生初期，无法找出曲线的决策边界是神经网络的痛，但是现在的神经网络已经可以轻松解决这个问题。那我们如何把直线的决策边界变成曲线呢？答案是将**单层神经网络变成多层**。来看下面的网格结构：

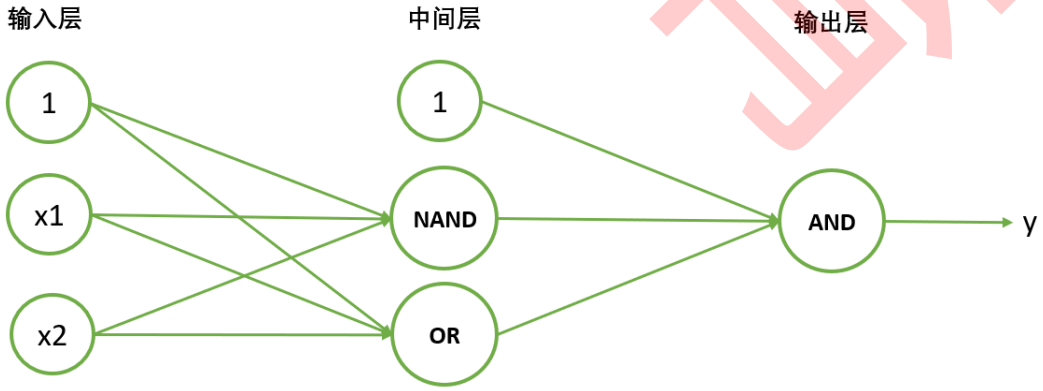


这是一个多层神经网络，除了输入层和输出层，还多了一层“中间层”。在这个网络中，数据依然是从左侧的输入层进入，特征会分别进入NAND和OR两个中间层的神经元，分别获得NAND函数的结果y_{nand}和OR函数的结果y_{or}，接着，y_{nand}和y_{or}会继续被输入下一层的神经元AND，经过AND函数的处理，成为最终结果y。让我们来使用代码实现这个结构，来看看这样的结构是否能够解决异或门的问题：

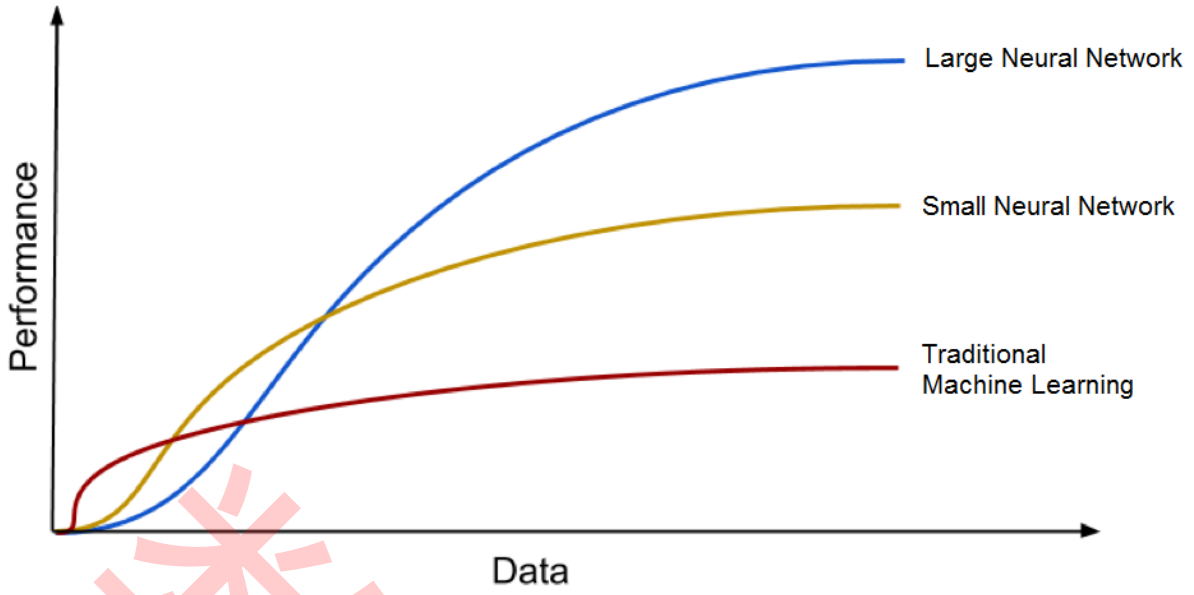
```
def XOR(X):
    #输入值：
    input_1 = X
    #中间层：
    sigma_nand = NAND(input_1)
    sigma_or = OR(input_1)
    x0 = torch.tensor([[1],[1],[1],[1]],dtype=torch.float32)
    #输出层：
    input_2 = torch.cat((x0,sigma_nand.view(4,1),sigma_or.view(4,1)),dim=1)
    y_and = AND(input_2)
    #print("NANE:",y_nand)
    #print("OR:",y_or)
    return y_and
```

可以看到，最终输出的结果和异或门要求的结果是一致的。可见，拥有更多的”层“帮助我们解决了单层神经网络无法处理的非线性问题。叠加了多层的神经网络也被称为”多层神经网络“。多层神经网络是神经网络在深度学习中的基本形态，接下来，我们就来认识多层神经网络。

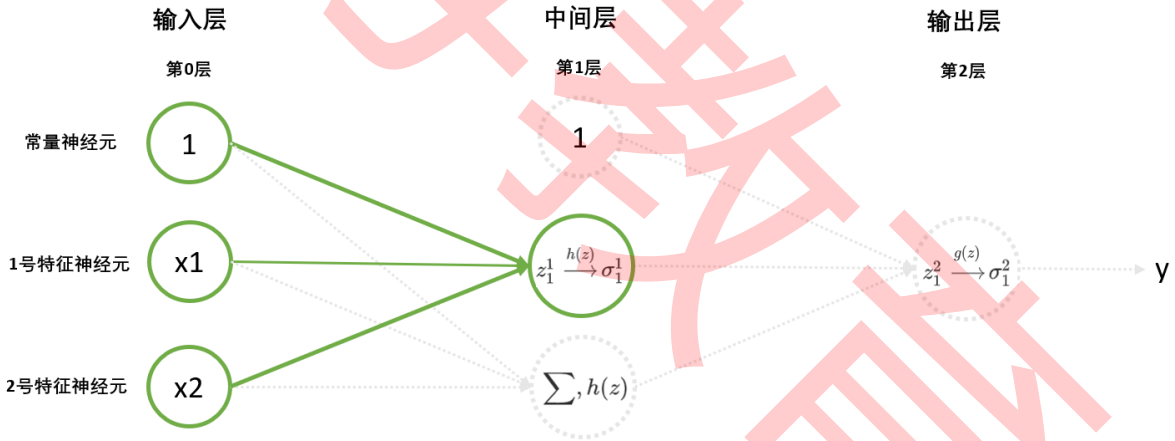
二、黑箱：深层神经网络的不可解释性



首先从结构上来看，多层神经网络比单层神经网络多出了“中间层”。中间层常常被称为隐藏层（hidden layer），理论上来说可以有无限层，所以在图像表示中经常被省略。层数越多，神经网络的模型复杂度越高，一般也认为更深的神经网络可以解决更加复杂的问题。在学习中，通常我们最多只会设置3~5个隐藏层，但在实际工业场景中会更多。还记得这张图吗？当数据量够大时，现代神经网络层数越深，效果越好。

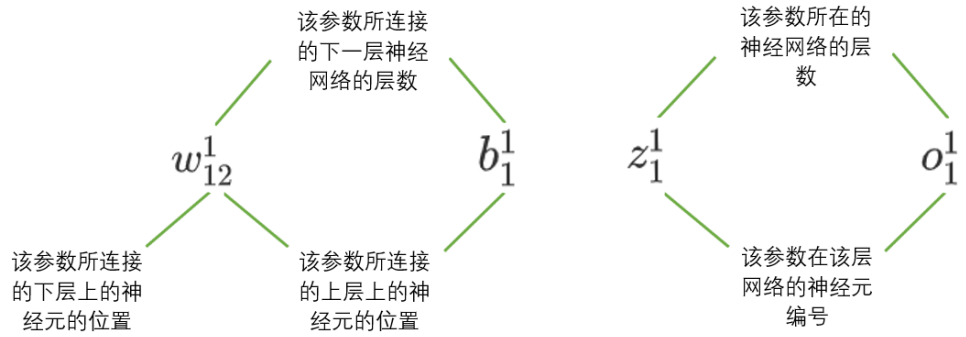


在一个神经网络中，更靠近输入层的层级相对于其他层级叫做"上层"，更靠近输出层的则相对于其他层级叫做"下层"。若从输入层开始从左向右编号，则输入层为第0层，输出层为最后一层。除了输入层以外，每个神经元中都存在着对数据进行处理的数个函数。在我们的例子异或门（XOR）中，隐藏层中的函数是NAND函数和OR函数（也就是线性回归的加和函数+阶跃函数），输出层上的函数是AND函数。对于所有神经元和所有层而言，加和函数的部分都是一致的（都得到结果z），因此我们需要关注的是加和之外的那部分函数。在隐藏层中这个函数被称为**激活函数**，符号为$h(z)$，在输出层中这个函数只是普通的连接函数，我们定义为是$g(z)$。我们的数据被逐层传递，**每个下层的神经元都必须处理上层的神经元中的$\mathbf{h(z)}$处理完毕的数据**，整个流程本质是一个嵌套计算结果的过程。

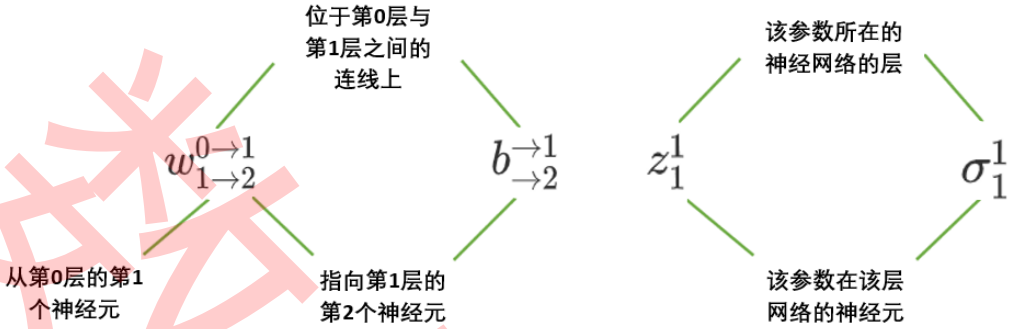


在神经网络中，任意层上都有至少一个神经元，最上面的是常量神经元，连接常量神经元的箭头上的参数是截距b，剩余的是特征神经元，连接这些神经元的箭头上的参数都是权重w。神经元是从上至下进行编号，**需要注意的是，常量神经元与特征神经元是分别编号的。**和从0开始编号的层数不同，神经元是从1开始编号的。在异或门的例子中，含有1的偏差神经元是1号偏差神经元，含有特征x_1的神经元则是1号特征神经元。

除了神经元和网络层，权重、偏差、神经元上的取值也存在编号。这些编号规律分别如下：



这些编号实在很复杂，因此在本次课程中，我将编号改写为如下情况：



有了这些编号说明，我们就可以用数学公式来表示从输入层传入到第一层隐藏层的信号了。以上一节中说明的XOR异或门为例子，对于**仅有两个特征的单一样本**而言，在第一层的第一个特征神经元中获得加和结果的式子可以表示为：

$$z_1^1 = b_{\to1}^{\to1} + x_1 w_{1\to1}^{0\to1} + x_2 w_{2\to1}^{0\to1}$$

而隐藏层中被$h(z)$处理的公式可以写作：

$$\begin{aligned}\sigma_1^1 &= h(z_1^1) \\ &= h(b_{\to1}^{\to1} + x_1 w_{1\to1}^{0\to1} + x_2 w_{2\to1}^{0\to1})\end{aligned}$$

根据我们之前写的NAND函数，这里的$h(z)$为阶跃函数。

现在，我们用矩阵来表示数据从输入层传入到第一层，并在第一层的神经元中被处理成σ的情况：

$$\mathbf{Z^1} = \mathbf{W^1} \cdot \mathbf{X} + \mathbf{B^1}$$

$$\begin{bmatrix} z_1^1 \\ z_2^1 \end{bmatrix} = \begin{bmatrix} w_{1\to1}^{0\to1} & w_{1\to2}^{0\to1} \\ w_{2\to1}^{0\to1} & w_{2\to2}^{0\to1} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_{\to1}^{\to1} \\ b_{\to2}^{\to1} \end{bmatrix}$$

矩阵结构表示为：$(2,1) = (2,2) * (2,1) + (2,1)$

$$= \begin{bmatrix} x_1 w_{1\to1}^{0\to1} + x_2 w_{1\to2}^{0\to1} \\ x_1 w_{2\to1}^{0\to1} + x_2 w_{2\to2}^{0\to1} \end{bmatrix} + \begin{bmatrix} b_{\to1}^{\to1} \\ b_{\to2}^{\to1} \end{bmatrix}$$

$$= \begin{bmatrix} x_1 w_{1\to1}^{0\to1} + x_2 w_{1\to2}^{0\to1} + b_{\to1}^{\to1} \\ x_1 w_{2\to1}^{0\to1} + x_2 w_{2\to2}^{0\to1} + b_{\to2}^{\to1} \end{bmatrix}$$

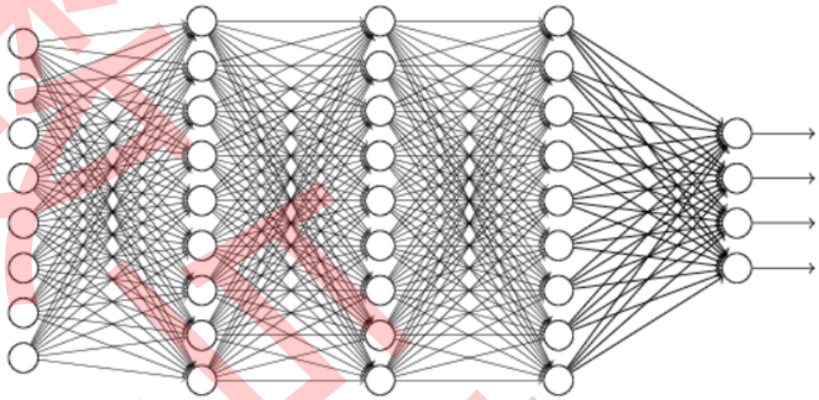
$$\begin{bmatrix} \sigma_1^1 \\ \sigma_2^1 \end{bmatrix} = \begin{bmatrix} h(x_1 w_{1\to1}^{0\to1} + x_2 w_{1\to2}^{0\to1} + b_{\to1}^{\to1}) \\ h(x_1 w_{2\to1}^{0\to1} + x_2 w_{2\to2}^{0\to1} + b_{\to2}^{\to1}) \end{bmatrix}$$

相应的从中间层最下面的神经网络会得到的结果是σ_1^2（如果是阶跃函数则是直接得到y）。σ会作为中间层的结果继续传入下一层。如果我们继续向下嵌套，则可以得到：

$$
\begin{aligned}
z_1^2 &= b_{\rightarrow 1}^{\rightarrow 2} + \sigma_1^1 w_{1\rightarrow 1}^{1\rightarrow 2} + \sigma_2^1 w_{2\rightarrow 1}^{1\rightarrow 2} \\
\sigma_1^2 &= g(z_1^2) \\
\sigma_1^2 &= g(b_{\rightarrow 1}^{\rightarrow 2} + \sigma_1^1 w_{1\rightarrow 1}^{1\rightarrow 2} + \sigma_2^1 w_{2\rightarrow 1}^{1\rightarrow 2})
\end{aligned}
$$

由于第二层就已经是输出层了，因此第二层使用的函数是$g(z)$，在这里，第二层的表达和第一层几乎一模一样。相信各种编号在这里已经让人感觉到有些头疼了，虽然公式本身并不复杂，但涉及到神经网络不同的层以及每层上的神经元之间的数据流动，公式的编号会让人有所混淆。如果神经网络的层数继续增加，或每一层上神经元数量继续增加，神经网络的嵌套和计算就会变得更加复杂。

在实际中，我们的真实数据可能有超过数百甚至数千个特征，所以真实神经网络的复杂度是非常高，计算非常缓慢的。所以，当神经网络长成如下所示的模样，我们就无法理解中间过程了。我们不知道究竟有多少个系数，如何相互作用产生了我们的预测结果，因此神经网络的过程是一个“黑箱”。



在PyTorch中实现神经网络的时候，我们一般用不到这些复杂的数学符号，也不需要考虑这些嵌套流程，毕竟这些计算非常底层。那为什么我们还要学习这些符号呢？有以下几点原因：

1、利用数学的嵌套，我们可以很容易就理解深层神经网络为什么会随着层数的增多、每层上神经元个数的增多而变得复杂，从而理解“黑箱”究竟是怎样形成的

2、多层神经网络与单层神经网络在许多关键点上其实有所区别，这种区别使用代数表示形式会更容易显示。比如，单层神经网络（线性回归、逻辑回归）中直线的表现形式都是$\boldsymbol{Xw}$，且$\boldsymbol{w}$是结构为(n_features,1)的列向量，但在多层神经网络中，随着“层”和神经元个数的增加，只有输入层与第一个隐藏层之间是特征与w的关系，隐藏层与隐藏层、隐藏层与输出层之间都是σ与w的关系。并且，即便是在输入层与第一个隐藏层之间，单个特征所对应的$\boldsymbol{w}$不再是列向量，而是结构为(上层特征神经元个数，下层特征神经元个数)的矩阵。**并且，每两层神经元之间，都会存在一个权重矩阵，权重将无法直接追踪到特征x上，这也是多层神经网络无法被解释的一个关键原因。**同时，为了让输出结果z和σ都保持列向量的形式（与神经网络的图像匹配），$\boldsymbol{X}$的结构也要顺应$\boldsymbol{w}$的变化而变化（上文的推导中我们展现的是单一样本仅有两个特征的情况，想想看多个样本多个特征会是什么样吧），相乘公式也变化为$\boldsymbol{wX}$。这些细节在由单层推广到多层时，都会成为新手容易掉入的坑，基础不牢固的情况下，新手很可能在2层推广到4层，2个特征推广到N个特征，甚至是1个样本推广到多个样本时被卡住。

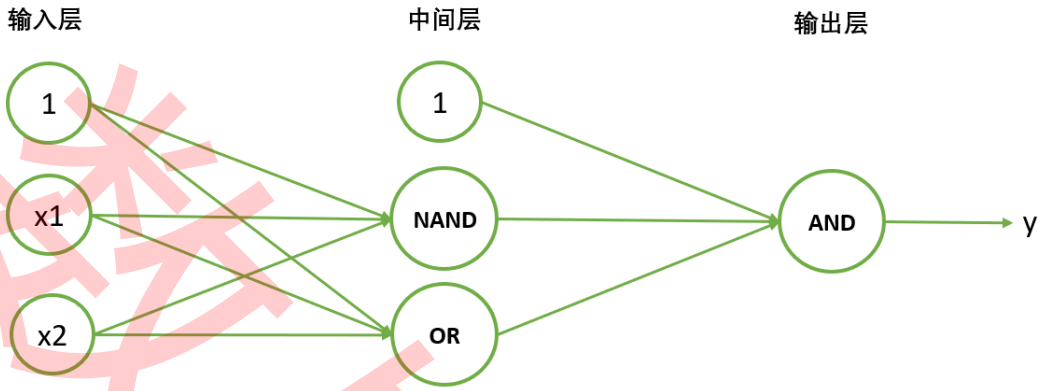
3、嵌套的数学公式可以帮助我们更好地理解反向传播，以及更好地阅读其他教材。

不过，即便神经网络是一个神秘黑箱，我们依然可以对它进行一系列的探索。基于我们已经熟悉的三层神经网络XOR的结构，我们来提出问题。

三、探索多层神经网络：层 vs h(z)

我们首先想要发问的隐藏层的作用。在之前的XOR函数中，我们提出"多层神经网络能够描绘出一条曲线作为决策边界，以此为基础处理单层神经网络无法处理的复杂问题"，这可能让许多人产生了"是增加层数帮助了神经网络"的错觉。实际上并非如此。

在神经网络的隐藏层中，存在两个关键的元素，一个是加和函数$\sum$，另一个是$h(z)$。除了输入层之外，任何层的任何神经元上都会有加和的性质，因为**神经元有"多进单出"的性质，可以一次性输入多个信号，但是输出只能有一个，因此输入神经元的信息必须以某种方式进行整合，否则神经元就无法将信息传递下去**，而最容易的整合方式就是加和$\sum$。因此我们可以认为加和$\sum$是神经元自带的性质，只要增加更多的层，就会有更多的加和。但是$h(z)$的存在却不是如此，即便隐藏层上没有$h(z)$（或$h(z)$是一个恒等函数），神经网络依然可以从第一层走到最后一层。让我们来试试看，在XOR中，假设隐藏层上没有$h(z)$的话， 会发生什么：




```
#回忆一下XOR数据的真实标签
xorgate = torch.tensor([0,1,1,0],dtype=torch.float32)

def AND(X):
    w = torch.tensor([-0.2,0.15, 0.15], dtype = torch.float32)
    zhat = torch.mv(X,w)
    #下面这一行就是阶跃函数的表达式，注意AND函数是在输出层，所以保留输出层的阶跃函数g(z)
    andhat = torch.tensor([int(x) for x in zhat >= 0],dtype=torch.float32)
    return andhat

def OR(X):
    w = torch.tensor([-0.08,0.15,0.15], dtype = torch.float32) #在这里我修改了b的数值
    zhat = torch.mv(X,w)
    #注释掉阶跃函数，相当于h(z)是恒等函数
    #yhat = torch.tensor([int(x) for x in zhat >= 0],dtype=torch.float32)
    return zhat

def NAND(X):
    w = torch.tensor([0.23,-0.15,-0.15], dtype = torch.float32)
    zhat = torch.mv(X,w)
    #注释掉阶跃函数，相当于h(z)是恒等函数
    #yhat = torch.tensor([int(x) for x in zhat >= 0],dtype=torch.float32)
    return zhat

def XOR(X):
    #输入值：
    input_1 = X
    #中间层：
    sigma_nand = NAND(input_1)
    sigma_or = OR(input_1)
    x0 = torch.tensor([[1],[1],[1],[1]],dtype=torch.float32)
    #输出层：
```

```
    input_2 = torch.cat((x0,sigma_nand.view(4,1),sigma_or.view(4,1)),dim=1)
    y_and = AND(input_2)
    #print("NANE:",y_nand)
    #print("OR:",y_or)
    return y_and

XOR(X)
```

很明显，此时XOR函数的预测结果与真实的xorgate不一致。当隐藏层的$h(z)$是恒等函数或不存在时，叠加层并不能够解决XOR这样的非线性问题。从数学上来看，这也非常容易理解。

从输入层到第1层：

从输入层到第1层：

$$\mathbf{Z}^1 = \mathbf{W^1} \cdot \mathbf{X} + \mathbf{B^1}$$

$$\mathbf{\Sigma}^1 = h(\mathbf{Z}^1)$$

$\because h(z)$为恒等函数

$$\therefore \mathbf{\Sigma}^1 = \begin{bmatrix} \sigma_1^1 \\ \sigma_2^1 \end{bmatrix} = \begin{bmatrix} z_1^1 \\ z_2^1 \end{bmatrix}$$

从第1层到输出层：

$$\mathbf{Z}^2 = \mathbf{W^2} \cdot \mathbf{\Sigma} + \mathbf{B^2}$$

$$\begin{bmatrix} z_1^2 \end{bmatrix} = \begin{bmatrix} w_{1\to1}^{1\to2} & w_{1\to2}^{1\to2} \end{bmatrix} * \begin{bmatrix} \sigma_1^1 \\ \sigma_2^1 \end{bmatrix} + \begin{bmatrix} b_{\to1}^{\to2} \end{bmatrix}$$

由于公式太长，不再给w写箭头角标，而是直接写普通数字角标：

$$\begin{aligned} \begin{bmatrix} z_1^2 \end{bmatrix} &= \sigma_1^1 w_{11}^2 + \sigma_2^1 w_{12}^2 + b_1^2 \\ &= z_1^1 w_{11}^2 + z_2^1 w_{12}^2 + b_1^2 \\ &= (x_1 w_{11}^1 + x_2 w_{12}^1 + b_1^1) w_{11}^2 + (x_1 w_{21}^1 + x_2 w_{22}^1 + b_2^1) w_{12}^2 + b_1^2 \\ &= x_1 (w_{11}^1 w_{11}^2 + w_{21}^1 w_{12}^2) + x_2 (w_{12}^1 w_{11}^2 + w_{22}^1 w_{12}^2) + b_1^1 w_{11}^2 + b_2^1 w_{12}^2 + b_1^2 \\ &= x_1 W_1 + x_2 W_2 + B, \text{其中} W_1, W_2 \text{和} B \text{都是常数} \end{aligned}$$

不难发现，最终从输出层输出的结果和第一层的输出结果$x_1 w_{11}^1 + x_2 w_{12}^1 + b_1^1$是类似的，只不过是乘以特征$x_1, x_2$的具体数值不同。在没有$h(z)$时，在层中流动的数据被做了仿射变换（affine transformation），仿射变换后得到的依然是一个线性方程，而这样的方程不能解决非线性问题。**可见，“层”本身不是神经网络解决非线性问题的关键，层上的$\mathbf{h(z)}$才是。**从上面的例子和数学公式中可以看出，**如果$h(z)$是线性函数，或不存在，那增加再多的层也没有用。**

那是不是任意非线性函数作为$h(z)$都可以解决问题呢？让我们来试试看，在XOR例子中如果不使用阶跃函数，而使用sigmoid函数作为$h(z)$，会发生什么。

```
def AND(X):
    w = torch.tensor([-0.2,0.15, 0.15], dtype = torch.float32)
```

```
    zhat = torch.mv(X,w)
    #下面这一行就是阶跃函数的表达式，注意AND函数是在输出层，所以保留输出层的阶跃函数g(z)
    andhat = torch.tensor([int(x) for x in zhat >= 0],dtype=torch.float32)
    return andhat

def OR(X):
    w = torch.tensor([-0.08,0.15,0.15], dtype = torch.float32) #在这里我修改了b的数
值
    zhat = torch.mv(X,w)
    #h(z)，使用sigmoid函数
    sigma = torch.sigmoid(zhat)
    return sigma

def NAND(X):
    w = torch.tensor([0.23,-0.15,-0.15], dtype = torch.float32)
    zhat = torch.mv(X,w)
    #h(z)，使用sigmoid函数
    sigma = torch.sigmoid(zhat)
    return sigma

def XOR(X):
    #输入值：
    input_1 = X
    #中间层：
    sigma_nand = NAND(input_1)
    sigma_or = OR(input_1)
    x0 = torch.tensor([[1],[1],[1],[1]],dtype=torch.float32)
    #输出层：
    input_2 = torch.cat((x0,sigma_nand.view(4,1),sigma_or.view(4,1)),dim=1)
    y_and = AND(input_2)
    #print("NANE:",y_nand)
    #print("OR:",y_or)
    return y_and

XOR(X)
```

可以发现，如果将$h(z)$换成sigmoid函数，XOR结构的神经网络同样会失效！可见，**即便是使用了$h(z)$，也不一定能够解决曲线分类的问题。在不适合的非线性函数加持下，神经网络的层数再多也无法起效。**所以，$h(z)$是真正能够让神经网络算法"活起来"的关键，没有搭配合适$h(z)$的神经网络结构是无用的，而$h(z)$正是神经网络中最关键的概念之一**激活函数（activation function）**。

四、激活函数

关键概念：激活函数

在人工神经网络的神经元上，根据一组输入定义该神经元的输出结果的函数，就是激活函数。激活函数一般都是非线性函数，它出现在神经网络中除了输入层以外的每层的每个神经元上。

经过前面的介绍与铺垫，到这里相信大家已经充分理解激活函数的作用了。神经网络中可用的激活函数多达数十种（详情可以在激活函数的维基百科中找到：https://en.wikipedia.org/wiki/Activation_function），但机器学习中常用的激活函数只有恒等函数（identity function），阶跃函数（sign），sigmoid函数，ReLU，tanh，softmax这六种，其中Softmax与恒等函数几乎不会出现在隐藏层上，Sign、Tanh几乎不会出现在输出层上，ReLU与Sigmoid则是两种层都会出现，并且应用广泛。幸运的是，这6种函

数我们在之前的课程中已经全部给大家介绍完毕。在这里，我们将总结性声明一下输出层的g(z)与隐藏层的h(z)之间的区别，以帮助大家获得更深的理解：

1. 虽然都是激活函数，但隐藏层和输出层上的激活函数作用是完全不一样的。输出层的激活函数$g(z)$是为了让神经网络能够输出不同类型的标签而存在的。其中恒等函数用于回归，sigmoid函数用于二分类，softmax用于多分类。换句说，**g(z)仅仅与输出结果的表现形式有关，与神经网络的效果无关**，也因此它可以使用线性的恒等函数。但隐藏层的激活函数就不同了，如我们之前尝试的XOR，隐藏层上的**激活函数$h(z)$的选择会影响神经网络的效果**，而线性的$h(z)$是会让神经网络的结构失效的。
2. **在同一个神经网络中，$g(z)$与$h(z)$可以是不同的，并且在大多数运行回归和多分类的神经网络时，他们也的确是不同的。**每层上的$h(z)$可以是不同的，但是同一层上的激活函数必须一致。

我们可以通过下面的这段代码来实际体会一下，$h(z)$影响模型效果，而$g(z)$只影响模型输出结果的形式的事实。之前我们曾经尝试过以下几种情况：

代码位置	隐藏层h(z)	输出层g(z)	XOR网络是否有效
3.1 异或门问题	阶跃函数	阶跃函数	**有效**
3.3 探索多层神经网络	恒等函数	阶跃函数	无效
3.3 探索多层神经网络	sigmoid函数	阶跃函数	无效
3.4 激活函数	阶跃函数	sigmoid函数	**?**

现在我们来试试看，隐藏层上的$h(z)$是阶跃函数，而输出层的$g(z)$是sigmoid的情况。如果XOR网络依然有效，就证明了$g(z)$的变化对神经网络结果输出无影响。反之，则说明$g(z)$也影响神经网络输出结果。

```
#如果g(z)是sigmoid函数，而h(z)是阶跃函数

#输出层，以0.5为sigmoid的阈值
def AND(X):
    w = torch.tensor([-0.2,0.15, 0.15], dtype = torch.float32)
    zhat = torch.mv(X,w)
    sigma = torch.sigmoid(zhat)
    andhat = torch.tensor([int(x) for x in sigma >= 0.5],dtype=torch.float32)
    return andhat

#隐藏层，OR与NAND都使用阶跃函数作为h(z)
def OR(X):
    w = torch.tensor([-0.08,0.15,0.15], dtype = torch.float32) #在这里我修改了b的数值
    zhat = torch.mv(X,w)
    yhat = torch.tensor([int(x) for x in zhat >= 0],dtype=torch.float32)
    return yhat

def NAND(X):
    w = torch.tensor([0.23,-0.15,-0.15], dtype = torch.float32)
    zhat = torch.mv(X,w)
    yhat = torch.tensor([int(x) for x in zhat >= 0],dtype=torch.float32)
    return yhat

def XOR(X):
    #输入值：
    input_1 = X
    #中间层：
```

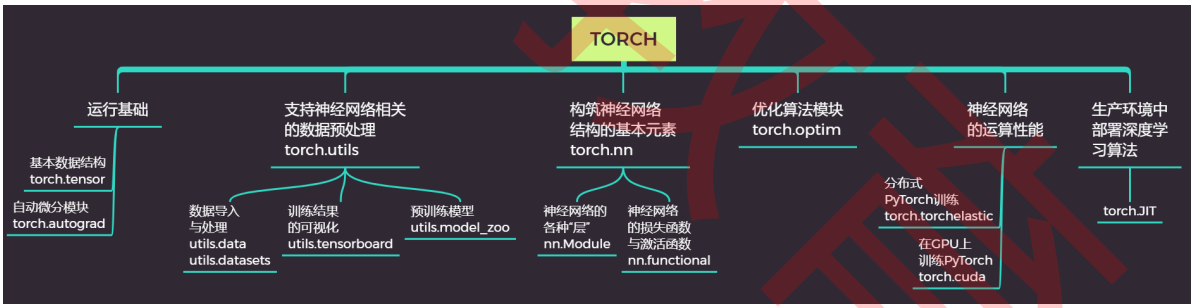
```
    sigma_nand = NAND(input_1)
    sigma_or = OR(input_1)
    x0 = torch.tensor([[1],[1],[1],[1]],dtype=torch.float32)
    #输出层:
    input_2 = torch.cat((x0,sigma_nand.view(4,1),sigma_or.view(4,1)),dim=1)
    y_and = AND(input_2)
    #print("NANE:",y_nand)
    #print("OR:",y_or)
    return y_and

XOR(X)
```

从结果可以看出，只要隐藏层的$h(z)$是阶跃函数，XOR网络就一定能有效，这与输出层$g(z)$是什么函数完全无关。从这里开始，若没有特别说明，当我们提到"激活函数"时，特指隐藏层上的激活函数$h(z)$。当需要表达输出层上的激活函数时，我们需称其为"输出层激活函数"（out_activation）。

代码位置	隐藏层h(z)	输出层g(z)	XOR网络是否有效
3.1 异或门问题	阶跃函数	阶跃函数	**有效**
3.3 探索多层神经网络	恒等函数	阶跃函数	无效
3.3 探索多层神经网络	sigmoid函数	阶跃函数	无效
3.4 激活函数	阶跃函数	sigmoid函数	**有效**

到这里，我们已经对深度神经网络有了深入的了解。下一节，我们将首次完成深度神经网络的从0实现。

五、从0实现深度神经网络的正向传播



学到这里，我们已经学完了一个普通深度神经网络全部的基本元素——用来构筑神经网络的结构的层与激活函数，输入神经网络的数据（特征、权重、截距），并且我们了解**从左向右的过程是神经网络的正向传播**（也叫做前向传播，或者向前传播）。还记得我们的架构图吗？在过去的课程中我们所学习的内容都是在torch.nn这个模块下，现在我们就使用封装好的torch.nn模块来实现一个完整、多层的神经网络的正向传播。

假设我们有500条数据，20个特征，标签为3分类。我们现在要实现一个三层神经网络，这个神经网络的架构如下：第一层有13个神经元，第二层有8个神经元，第三层是输出层。其中，第一层的激活函数是relu，第二层是sigmoid。我们要如何实现它呢？来看代码：

```
#继承nn.Module类完成正向传播
import torch
import torch.nn as nn
from torch.nn import functional as F

#确定数据
```

```
torch.manual_seed(420)
X = torch.rand((500,20),dtype=torch.float32)
y = torch.randint(low=0,high=3,size=(500,1),dtype=torch.float32)

#继承nn.Modules类来定义神经网路的架构
class Model(nn.Module):
    #init：定义类本身，__init__函数是在类被实例化的瞬间就会执行的函数
    def __init__(self,in_features=10,out_features=2):
        super(Model,self).__init__() #super(请查找这个类的父类，请使用找到的父类替换现在的类)
        self.linear1 = nn.Linear(in_features,13,bias=True) #输入层不用写，这里是隐藏层的第一层
        self.linear2 = nn.Linear(13,8,bias=True)
        self.output = nn.Linear(8,out_features,bias=True)

    #__init__之外的函数，是在__init__被执行完毕后，就可以被调用的函数
    def forward(self, x):
        z1 = self.linear1(x)
        sigma1 = torch.relu(z1)
        z2 = self.linear2(sigma1)
        sigma2 = torch.sigmoid(z2)
        z3 = self.output(sigma2)
        sigma3 = F.softmax(z3,dim=1)
        return sigma3

input_ = X.shape[1] #特征的数目
output_ = len(y.unique()) #分类的数目

#实例化神经网络类
torch.manual_seed(420)
net = Model(in_features=input_, out_features=output_)
#在这一瞬间，所有的层就已经被实例化了，所有随机的w和b也都被建立好了

#前向传播
net(X)

net.forward(X)

#查看输出的标签
sigma = net.forward(X)
sigma.max(axis=1)

#查看每一层上的权重w和截距b
net.linear1.weight

net.linear1.bias
```

如果你是初次接触“类”，那这段代码中新内容可能会有点多。但如果你对python基础比较熟悉，你就会发现这个类其实非常简单。在神经网络的类中，我们以线性的顺序从左向右描绘神经网络的计算过程，并且无需考虑在这之间w的结构是如何，矩阵之间如何进行相互运算。所以只要你对自己要建立的神经网络的架构是熟悉的，pytorch代码就非常容易。在这里，特别需要强调一下的可能是super函数的用法。

super函数用于调用父类的一个函数，在这里我们使用super函数来帮助子类（我们建立的神经网络模型）继承一些通过**类名调用**无法被继承的属性和方法。谨防小伙伴们不熟悉super的用法，在这里我们来说明一下：

```
#建立一个父类
class FooParent(object):
    def __init__(self):
        self.parent = 'PARENT!!'
        print ('Running __init, I am parent')

    def bar(self,message):
        self.bar = "This is bar"
        print ("%s from Parent" % message)

FooParent() #父类实例化的瞬间，运行自己的__init__

FooParent().parent #父类运行自己的__init__中定义的属性

#建立一个子类，并通过类名调用让子类继承父类的方法与属性
class FooChild(FooParent):
    def __init__(self):
        print ('Running __init, I am child')

#查看子类是否继承了方法
FooChild().bar("HAHAHA")

FooChild().parent #子类没有继承到父类的__init__中定义的属性

#为了让子类能够继承到父类的__init__函数中的内容，我们使用super函数

#新建一个子类，并使用super函数
class FooChild(FooParent):
    def __init__(self):
        super(FooChild,self).__init__()
        print ('Child')
        print ("I am running the __init__")

#再次调用parent属性
FooChild() #执行自己的init功能的同时，也执行了父类的init函数定义的功能

FooChild().parent
```

通过使用super函数，我们的神经网络模型从nn.Module那里继承了哪些有用的属性和方法呢？首先，如果不加super函数，神经网络的向前传播是无法运行的：

```
class Model(nn.Module):
    def __init__(self,in_features=10,out_features=2):
        #super(Model,self).__init__() #super(请查找这个类的父类，请使用找到的父类替换现在的类)
        self.linear1 = nn.Linear(in_features,13,bias=True) #输入层不用写，这里是隐藏层的第一层
        self.linear2 = nn.Linear(13,8,bias=True)
        self.output = nn.Linear(8,out_features,bias=True)

    def forward(self, x):
        z1 = self.linear1(x)
        sigma1 = torch.relu(z1)
        z2 = self.linear2(sigma1)
        sigma2 = torch.sigmoid(z2)
        z3 = self.output(sigma2)
        sigma3 = F.softmax(z3,dim=1)
```

```
        return sigma3

net=Model(in_features=input_, out_features=output_)
```

nn.Module类的定义代码有一千多行，其__init__函数中有许多复杂的内容需要被继承，因此super函数一定不能漏下。

```
def __init__(self):
    """
    Initializes internal Module state, shared by both nn.Module and ScriptModule.
    """
    torch._C._log_api_usage_once("python.nn_module")

    self.training = True
    self._parameters = OrderedDict()
    self._buffers = OrderedDict()
    self._non_persistent_buffers_set = set()
    self._backward_hooks = OrderedDict()
    self._forward_hooks = OrderedDict()
    self._forward_pre_hooks = OrderedDict()
    self._state_dict_hooks = OrderedDict()
    self._load_state_dict_pre_hooks = OrderedDict()
    self._modules = OrderedDict()

forward: Callable[..., Any] = _forward_unimplemented
```

我们从nn.Module中继承了这些有用的方法：

```
#属性的继承
net.training #是否用于训练

#方法的继承
net.cuda()

net.cpu()

net.apply() #对__init__中的所有对象（全部层）都执行同样的操作

#比如，令所有线性层的初始权重w都为0
def initial_0(m):
    print(m)
    if type(m) == nn.Linear:
        m.weight.data.fill_(0)
        print(m.weight)

net.apply(initial_0)

net.linear1.weight

#一个特殊的方法
net.parameters() #一个迭代器，我们可以通过循环的方式查看里面究竟是什么内容

for param in net.parameters():
    print(param)
```

当然，nn.Module中的方法完全不止这几个。之后我们用到更多方法时，我们会一一向大家进行介绍。相信你现在已经对从0建立自己的神经网络有些感觉了，之后我们会在这个代码的基础上，不断加入新知识，并让我们的代码变得越来越丰富。试试看自己编写一个架构，进行一些探索，并执行完整的向前传播流程吧。