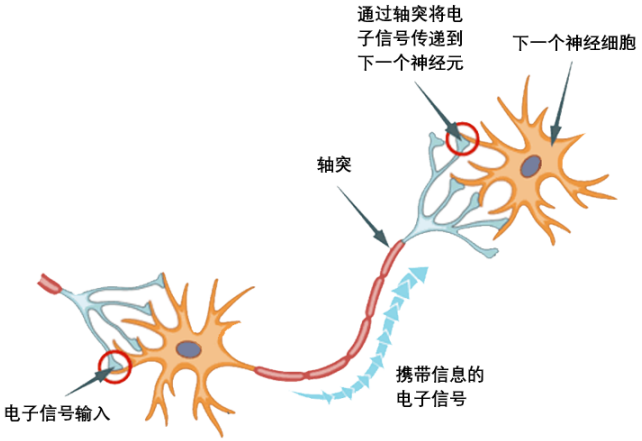
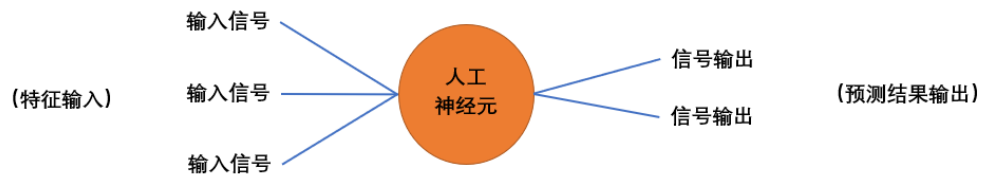
> 《PyTorch深度学习实战》
>
> Lesson 8：单层神经网络
>
> 作者：@菜菜TsaiTsai
>
> 版本号：V2.0

Lesson 8 单层神经网络



在之前的介绍中，我们已经了解了神经网络是模仿人类大脑结构所构建的算法，在人脑里，我们有轴突连接神经元，在算法中，我们用圆表示神经元，用线表示神经元之间的连接，数据从神经网络的左侧输入，让神经元处理之后，从右侧输出结果。



上图是一个最简单的神经元的结构。从这里开始，我们正式开始认识神经网络。

Lesson 8 单层神经网络

一、单层回归网络：线性回归

1 单层回归网络的理论基础

许多人都以为神经网络是一个非常复杂的算法，其实它的基本原理其实并不难理解。还记得我们在讲解GPU那一节曾提到过的，深度学习中的计算是"简单大量"，而不是"复杂的单一问题"吗？神经网络的原理很多时候都比经典机器学习算法简单。了解神经网络，可以从**线性回归**算法开始。在之前的课程中，我们讲解过PyTorch基本数据结构Tensor和基本库Autograd，在给autograd举例时，我们对线性回归其实已经有简单的说明，在这里，给大家复习一下，并且明确一些数学符号。

线性回归算法是机器学习中最简单的回归类算法，多元线性回归指的就是一个样本对应多个特征的线性回归问题。假设我们的数据现在就是二维表，对于**一个**有n个特征的样本i而言，它的预测结果可以写作一个几乎人人熟悉的方程：

$$\hat{z}_i = b + w_1 x_{i1} + w_2 x_{i2} + \ldots + w_n x_{in} \tag{1}$$

w和b被统称为模型的权重，其中b被称为截距(intercept)，也叫做偏差(bias)，w_1~w_n被称为回归系数(regression coefficient)，也叫作权重(weights)，x_{i1}~x_{in}是样本i上的不同特征。这个表达式，其实就和我们小学时就无比熟悉的$y = ax + b$是同样的性质。其中y被我们称为因变量，在线性回归中表示为z，在机器学习中也就表现为我们的标签。如果写作z，则代表真实标签。如果写作$\hat{z}$（读作z帽或者zhat），则代表预测出的标签。模型得出的结果，一定是预测的标签。

> **符号规范**
>
> 在我们学习autograd的时候，我们说线性回归的方程是$\hat{y}_i = b + w_1 x_{i1} + w_2 x_{i2} + \ldots + w_n x_{in}$，但在这里，为什么写做$z$呢？
> 首先，无论是回归问题还是分类问题，**y永远表示标签（labels）**。在回归问题中，y是连续型数字，在分类问题中，y是离散型的整数。
> 对于线性回归来说，线性方程的输出结果就是最终的标签。但对于整个深度学习体系而言，复杂神经网络的输出才是最后的标签。在我们单独对线性回归进行说明的时候，行业惯例就是使用z来表示线性回归的结果。

如果考虑我们有m个样本，则回归结果可以被写作：

$$\hat{\boldsymbol{z}} = b + w_1 \boldsymbol{x_1} + w_2 \boldsymbol{x_2} + \ldots + w_n \boldsymbol{x_n} \tag{2}$$

其中$\hat{\boldsymbol{z}}$是包含了m个全部的样本的预测结果的列向量。注意，我们通常使用粗体的小写字母来表示列向量，粗体的大写字母表示矩阵或者行列式。**并且在机器学习中，我们默认所有的一维向量都是列向量。**我们可以使用矩阵来表示上面多个样本的回归结果的方程，其中$\boldsymbol{w}$可以被看做是一个结构为(n+1,1)的列矩阵（这里的n加上的1是我们的截距b），$\boldsymbol{X}$是一个结构为(m,n+1)的特征矩阵（这里的n加上的1是为了与截距b相乘而留下的一列1，这列1有时也被称作x_0），则有：

$$\begin{bmatrix} \hat{z}_1 \\ \hat{z}_2 \\ \hat{z}_3 \\ \cdots \\ \hat{z}_m \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & x_{13} & \cdots & x_{1n} \\ 1 & x_{21} & x_{22} & x_{23} & \cdots & x_{2n} \\ 1 & x_{31} & x_{32} & x_{33} & \cdots & x_{3n} \\ & & \cdots & & & \\ 1 & x_{m1} & x_{m2} & x_{m3} & \cdots & x_{mn} \end{bmatrix} * \begin{bmatrix} b \\ w_1 \\ w_2 \\ \cdots \\ w_n \end{bmatrix}$$

$$\hat{\boldsymbol{z}} = \boldsymbol{X}\boldsymbol{w}$$

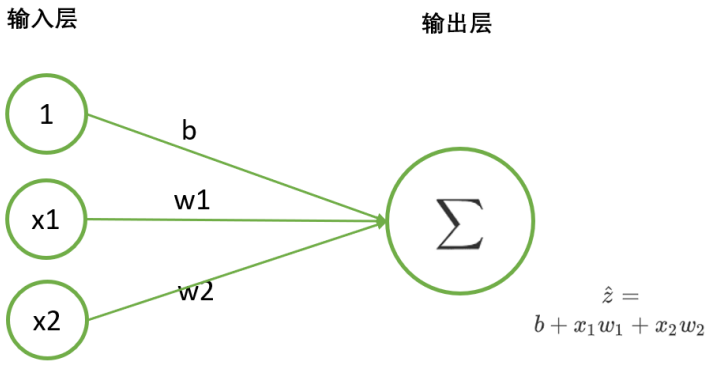
如果在我们的方程里没有常量b，我们则可以不写$\mathbf{X}$中的第一列以及$\mathbf{w}$中的第一行。

线性回归的任务，就是构造一个预测函数来映射输入的特征矩阵$\boldsymbol{X}$和标签值$\boldsymbol{y}$的**线性关系**。这个预测函数的图像是一条直线，所以线性回归的求解就是对直线的拟合过程。预测函数的符号在不同的教材上写法不同，可能写作$f(x)$，$y_w(x)$，或者$h(x)$等等形式，但无论如何，**这个预测函数的本质就是我们需要构建的模型，而构造预测函数的核心就是找出模型的权重向量$\boldsymbol{w}$**，也就是求解线性方程组的参数（相当于求解$y = ax + b$里的a与b）。

现在假设，我们的数据只有2个特征，则线性回归方程可以写作如下结构：

$$\hat{z} = b + x_1 w_1 + x_2 w_2 \tag{3}$$

此时，我们只要对模型输入特征x_1, x_2的取值，就可以得出对应的预测值$\hat{z}$。神经网络的预测过程是从神经元左侧输入特征，让神经元处理数据，并从右侧输出预测结果。这个过程和我们刚才说到的线性回归输出预测值的过程是一致的。如果我们使用一个神经网络来表达线性回归上的过程，则可以有：



这就是一个最简单的**单层回归神经网络**的表示图。

在神经网络中，竖着排列在一起的一组神经元叫做“一层网络”，所以线性回归的网络直观看起来有两层，两层神经网络通过写有参数的线条相连。我们从左侧输入常数1和特征取值x_1, x_2，再让它们与相对应的参数**相乘**，就可以得到b，$x_1 w_1$，$x_2 w_2$三个结果。这三个结果通过连接到下一层神经元的直线，被输入下一层神经元。我们在第二层的神经元中将三个乘积进行**加和**（使用符号$\sum$表示），就可以得到加和结果$\hat{z}$，即$b + x_1 w_1 + x_2 w_2$，这个值正是我们的预测值。**可见，线性回归方程与上面的神经网络图达到的效果是一模一样的。**

在上述过程中，左侧的是神经网络的**输入层**（input layer）。输入层由众多承载数据用的神经元组成，数据从这里输入，并流入处理数据的神经元中。在所有神经网络中，输入层永远只有一层，且每个神经元上只能承载一个特征(一个x)或一个常量（通常都是1）。现在的二元线性回归只有两个特征，所以输入层上只需要三个神经元，包括两个特征和一个常量，其中这里的常量仅仅是被用来乘以偏差b用的。对于没有偏差的线性回归来说，我们可以不设置常量1。

右侧的是**输出层**（output layer）。输出层由大于等于一个神经元组成，我们总是从这一层来获取预测结果。输出层的每个神经元上都承载着单个或多个功能，可以处理被输入神经元的数据。在线性回归中，这个功能就是“加和”，当我们把加和替换成其他的功能，就能够形成各种不同的神经网络。

在神经元之间相互连接的线表示了数据流动的方向，就像人脑神经细胞之间相互联系的“轴突”。在人脑神经细胞中，轴突控制电子信号流过的强度，在人工神经网络中，神经元之间的连接线上的权重也代表了“信息可通过的强度。最简单的例子是，当w_1为0.5时，在特征x_1上的信息就只有0.5倍能够传递到下一层神经元中，因为被输入到下层神经元中去进行计算的实际值是$0.5x_1$。相对的，如果w_1是2.5，则会传递2.5倍的x_1上的信息。因此，有的深度学习课程会将权重w比喻成是电路中的“电压“，电压越大，则电信号越强烈，电压越小，信号也越弱，这都是在描述权重w会如何影响传入下一层神经元的信息/数据量的大小。

到此，我们已经了解了线性回归的网络是怎么一回事，它是最简单的回归神经网络，同时也是最简单的神经网络。类似于线性回归这样的神经网络，被称为**单层神经网络**。

单层神经网络

从直观来看，线性回归的网络结构明明有两层，为什么线性回归被叫做“单层神经网络”呢？
业内通识是，在描述神经网络的层数的时候，**我们不考虑输入层。**
输入层是每个神经网络都必须存在的一层，当使用相同的输入数据时，任意两个神经网络之间的不同之处就在输入层之后的所有层。所以，我们把输入层之后只有一层的神经网络称为单层神经网络。
在非常非常少见的情况下，有的深度学习课程或教材中也会直接将所有层都算入其中，将上述网络称为“两层神经网络”，这种做法虽然不太规范，但也不能称之为“错误的”。因此，当出现“N层神经网络”的描述时，一定要注意原作者是否将输入层考虑进去了。

2 tensor实现单层神经网络的正向传播

让我们使用一组非常简单（简直是简单过头了）的代码来实现一下回归神经网络求解$\hat{z}$的过程，在神经网络中，这个过程是从左向右进行的，被称为神经网络的正向传播（forward spread）（当然，这是正向传播中非常简单的一种情况）。来看下面这组数据：

x0	x1	x2	z
1	0	0	-0.2
1	1	0	-0.05
1	0	1	-0.05
1	1	1	0.1

我们将构造能够拟合出以上数据的单层回归神经网络：

```
import torch
X = torch.tensor([[1,0,0],[1,1,0],[1,0,1],[1,1,1]], dtype = torch.float32)
z = torch.tensor([-0.2, -0.05, -0.05, 0.1])
w = torch.tensor([-0.2,0.15,0.15])

def LinearR(X,w):
    zhat = torch.mv(X,w)
    return zhat

zhat = LinearR(X,w)
```

3 tensor计算中的新手陷阱

接下来，我们对这段代码进行详细的说明：

```
#导入库
import torch

#首先生成特征张量
X = torch.tensor([[1,0,0],[1,1,0],[1,0,1],[1,1,1]])

#我们输入的是整数，默认生成的是int64的类型
```

```
X.dtype

#查看一下特征张量是什么样子
X

X.shape

#生成标签
z = torch.tensor([-0.2, -0.05, -0.05, 0.1])

#标签我们输入的是浮点数，默认生成的则是float32的类型
z.dtype

#查看标签
z

#定义常量b和权重w
#注意，常量b所在的位置必须与特征张量X中全为1的那一列所在的位置相对应，在这里，-0.2是b，0.15是两个w
w = torch.tensor([-0.2,0.15,0.15])

w.shape
```

- **tensor计算中的第一大坑：PyTorch的静态性**

```
#定义线性回归计算的函数
def LinearR(X,w):
    zhat = torch.mv(X,w)
    #矩阵与向量相乘，使用函数torch.mv。注意，矩阵与向量相乘时，向量必须写作第二个参数。
    #float32与int64相乘会出现什么问题？看似是个问题，其实这是PyTorch计算快速的秘诀之一。
    return zhat

#因此之后需要修改成：
def LinearR(X,w):
    zhat = torch.mv(X.float(),w)
    return zhat

#你还可以使用大写的Tensor来解决这个问题，但这个方法并不推荐：
X_ = torch.Tensor([[1,0,0],[1,1,0],[1,0,1],[1,1,1]])

X_.dtype

#或者，你可以直接养成好习惯：
X = torch.tensor([[1,0,0],[1,1,0],[1,0,1],[1,1,1]], dtype = torch.float32)

'''
PyTorch中的许多函数都不接受浮点型的分类标签，但也有许多函数要求真实标签的类型必须与预测值的类型一致，因此标签的类型定义总是一个容易踩坑的地方。通常来说，我们还是回将标签定义为float32，如果在函数运行时报错，要求整形，我们再使用.long()方法将其转换为整型。

另一个非常容易踩坑的地方是，PyTorch中许多函数不接受一维张量
但同时也有许多函数不接受二维标签(￣_￣|||)
因此我们在生成标签时，可以默认生成二维标签，若函数报错说不能接受二维标签，我们再使用view()函数将其调整为一维。
'''
```

```
#进行计算，查看返回的结果
zhat = LinearR(X,w)

zhat

z_reg
```

可以看到，只要能够给到适合的w和b，回归神经网络其实非常容易实现。在这里，我们需要提出PyTorch中另一个比较严肃的问题。

- **tensor计算中的第二大坑：精度问题**

来看这段代码。

```
zhat == z_reg

zhat.dtype

z_reg.dtype

#为什么会出现这种现象呢？还记得如何衡量线性回归的结果优劣吗？
```

在多元线性回归中，我们的使用SSE（误差平方和，有时候也叫做RSS残差平方和）来衡量回归的结果优劣：

$$SSE = \sum_{i=1}^{m}(z_i - \hat{z}_i)^2 \tag{4}$$

如果预测值$\hat{z}_i$与真实值z_i完全相等，那SSE的结果应该为0。在这里，SSE虽然非常接近0，但的确是不为0的。

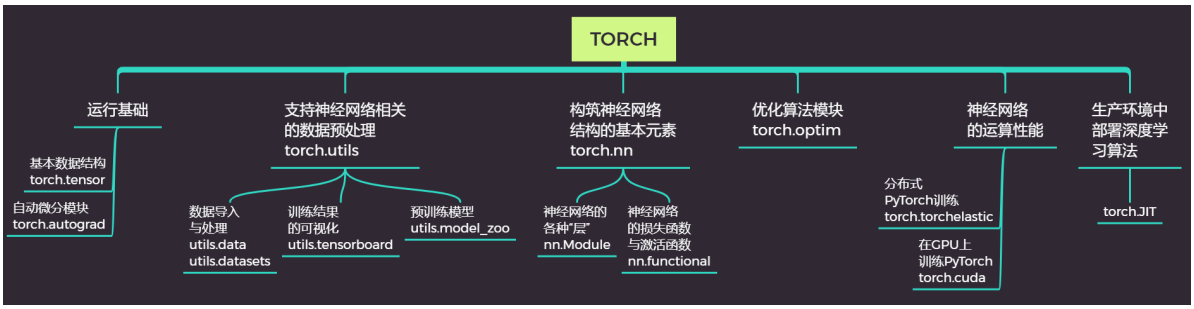
```
SSE = sum((zhat - z_reg)**2)

#设置显示精度，再来看yhat与y_reg
torch.set_printoptions(precision=30)

zhat

z_reg

#精度问题在tensor维度很高，数字很大时，也会变得更大

preds = torch.ones(300, 68, 64, 64) * 0.1
#preds.shape #运行查看preds的结构
preds.sum() * 10

preds = torch.ones(300, 68, 64, 64)
preds.sum()

#怎么解决这个问题呢？
#与Python中存在Decimal库不同，PyTorch设置了64位浮点数来尽量减轻精度问题
preds = torch.ones(300, 68, 64, 64, dtype=torch.float64) * 0.1
preds.sum() * 10
```

```
#但即便如此，我们也不能完全消除掉精度问题所带来的区别
#如果你希望能够无视掉非常小的区别，而让两个张量的比较结果展示为True，你可以使用下面的代码：
torch.allclose(zhat,z_reg)
```

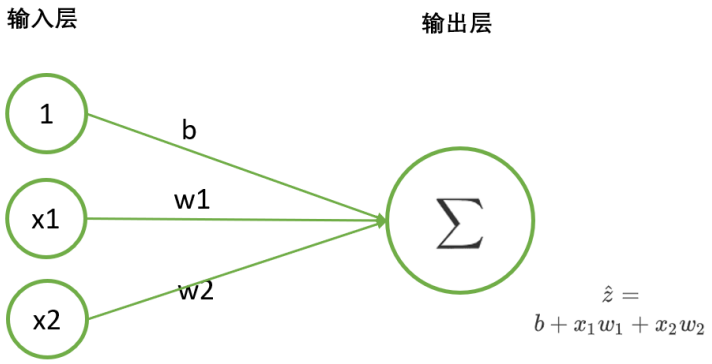
4 torch.nn.Linear实现单层回归神经网络的正向传播

p.s. 从本节开始，要使用类的相关知识。我假设你已经了解类，如果你不熟悉，可以去查看九天老师在B站放出的免费Python课程：https://www.bilibili.com/video/BV1U54y1W7jw

在PyTorch中，我们使用类torch.nn.Linear类来实现单层回归神经网络，不过需要注意的是，**它可不是代表单层回归神经网络这个算法。**还记得之前我们的架构图吗？



从这张架构图中我们可以看到，torch.nn是包含了构筑神经网络结构基本元素的包，在这个包中，你可以找到任意的神经网络层。这些神经网络层都是nn.Module这个大类的子类。我们的torch.nn.Linear就是神经网络中的"线性层"，它可以实现形如$\hat{\boldsymbol{z}} = \boldsymbol{X}\boldsymbol{w}$的加和功能。在我们的单层回归神经网络结构图中，torch.nn.Linear类表示了我们的**输出层**。现在我们就来看看它是如何使用的。



回顾一下我们的数据：

x0	x1	x2	z
1	0	0	-0.2
1	1	0	-0.05
1	0	1	-0.05
1	1	1	0.1

接下来，我们使用nn.Linear来实现单层回归神经网络：

```
import torch

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

output = torch.nn.Linear(2,1)
zhat = output(X)
```

怎么样，代码是不是异常简单？但在这段代码中，却有许多细节需要声明：

- nn.Linear是一个类，在这里代表了输出层，所以我使用output作为变量名，output = 的这一行相当于是类的实例化过程
- 实例化的时候，nn.Linear需要输入两个参数，分别是（上一层的神经元个数，这一层的神经元个数）。上一层是输入层，因此神经元个数由特征的个数决定（2个）。这一层是输出层，作为回归神经网络，输出层只有一个神经元。因此nn.Linear中输入的是（2，1）。
- 我只定义了X，没有定义w和b。所有nn.Module的子类，形如nn.XXX的层，都会在实例化的同时随机生成w和b的初始值。所以实例化之后，我们就可以调用以下属性来查看生成的w和b：

```
output.weight #查看生成的w

output.bias #查看生成的b
```

- 其中，w是必然会生成的，b是我们可以控制是否要生成的。在nn.Linear类中，有参数bias，默认bias = True。如果我们希望不拟合常量b，在实例化时将参数bias设置为False即可：

```
output = torch.nn.Linear(2,1,bias=False)

#再次调用属性weight和bias

output.weight

output.bias
```

- 由于w和b是随机生成的，所以同样的代码多次运行后的结果是不一致的。如果我们希望控制随机性，则可以使用torch中的random类。如下所示：

```
torch.random.manual_seed(420) #人为设置随机数种子
output = torch.nn.Linear(2,1)
zhat = output(X)

zhat
```

- 由于不需要定义常量b，因此在特征张量中，也不需要留出与常数项相乘的x0那一列。在输入数据时，我们只输入了两个特征x1与x2
- 输入层只有一层，并且输入层的结构（神经元的个数）由输入的特征张量$\mathbf{X}$决定，因此在PyTorch中构筑神经网络时，不需要定义输入层
- 实例化之后，将特征张量输入到实例化后的类中，即可得出输出层的输出结果。

让我们来看看输出结果的形状：

```
zhat.shape
```

这个形状与我们自己定义的z是一致的。但就数字大小上来说，由于我们没有自己定义w和b，所以我们无法让nn.Linear输出的zhat与我们真实的z接近——让真实值与预测值差异更小的部分，我们会在之后进行讲解。现在，让我们继续了解神经网络。在下一节中，我们会将单层回归神经网络的例子推广到分类问题上。

二、二分类神经网络：逻辑回归

1 二分类神经网络的理论基础

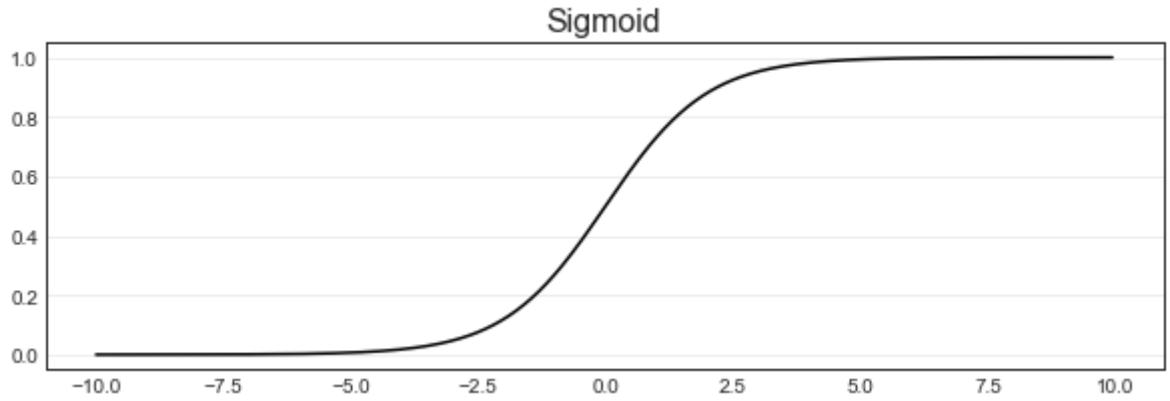
线性回归是统计学经典算法，它能够拟合出一条直线来描述变量之间的**线性关系。**但在实际中，变量之间的关系通常都不是一条直线，而是呈现出某种曲线关系。在统计学的历史中，为了让统计学模型能够更好地拟合曲线，统计学家们在线性回归的方程两边引入了联系函数（link function），对线性回归的方程做出了各种样的变化，并将这些变化后的方程称为“广义线性回归”。其中比较著名的有等式两边同时取对数的对数函数回归、同时取指数的S形函数回归等。

$$y = ax + b \quad \rightarrow \quad lny = ln(ax + b)$$
$$y = ax + b \quad \rightarrow \quad e^{y} = e^{ax+b}$$

在探索的过程中，一种奇特的变化吸引了统计学家们的注意，这个变化就是sigmoid函数带来的变化。Sigmoid函数的公式如下：

$$\sigma = Sigmoid(z) = \frac{1}{1+e^{-z}} \tag{5}$$

其中e为自然常数（约为2.71828），其中z是它的自变量，σ是因变量，z的值常常是线性模型的取值（比如，线性回归的结果z）。Sigmoid函数是一个S型的函数，它的图像如下：



从图像上就可以看出，这个函数的性质相当特别。当自变量z趋近正无穷时，因变量σ趋近于1，而当z趋近负无穷时，σ趋近于0，这使得sigmoid函数能够将任何实数映射到(0,1)区间。同时，Sigmoid的导数在$z=0$点时最大（这一点的斜率最大），所以它可以快速将数据从$z=0$的附近排开，让数据点到远离自变量取0的地方去。这样的性质，让sigmoid函数拥有**将连续性变量z转化为离散型变量σ的力量，这也就是化回归算法为分类算法的力量。**

具体怎么操作呢？只要将线性回归方程的结果作为自变量带入sigmoid函数，得出的数据就一定是(0,1)之间的值。此时，只要我们设定一个阈值（比如0.5），规定σ大于0.5时，预测结果为1类，σ小于0.5时，预测结果为0类，则可以顺利将回归算法转化为分类算法。此时，我们的标签就是类别0和1了。这个阈值可以自己调整，在没有调整之前，一般默认0.5。

$$\sigma = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-\boldsymbol{X}\boldsymbol{w}}}$$

更神奇的是，当我们对线性回归的结果取sigmoid函数之后，只要再进行以下操作：

1）将结果σ以几率（$\frac{\sigma}{1-\sigma}$）的形式展现

2）在几率上求以e为底的对数

就很容易得到：

$$
\begin{aligned}
ln\frac{\sigma}{1-\sigma} &= ln(\frac{\frac{1}{1+e^{-\boldsymbol{X}\boldsymbol{w}}}}{1-\frac{1}{1+e^{-\boldsymbol{X}\boldsymbol{w}}}}) \\
&= ln(\frac{\frac{1}{1+e^{-\boldsymbol{X}\boldsymbol{w}}}}{\frac{e^{-\boldsymbol{X}\boldsymbol{w}}}{1+e^{-\boldsymbol{X}\boldsymbol{w}}}}) \\
&= ln(\frac{1}{e^{-\boldsymbol{X}\boldsymbol{w}}}) \\
&= ln(e^{\boldsymbol{X}\boldsymbol{w}}) \\
&= \boldsymbol{X}\boldsymbol{w}
\end{aligned}
\tag{6}
$$

不难发现，让σ取对数几率后所得到的值就是我们线性回归的$\mathbf{z}$！因为这个性质，在等号两边加sigmoid的算法被称为“对数几率回归”，在英文中就是Logistic Regression，就是**逻辑回归**。逻辑回归可能是广义线性回归中最广为人知的算法，它是一个叫做“回归“实际上却总是被用来做分类的算法，对机器学习和深度学习都有重大的意义。在面试中，如果我们希望了解一个人对机器学习的理解程度，第一个问题可能就会从sigmoid函数以及逻辑回归是如何来的开始。

加餐：σ值代表了样本为某一类标签的概率

$ln\frac{\sigma}{1-\sigma}$是形似对数几率的一种变化。而几率odds的本质其实是$\frac{p}{1-p}$，其中p是事件A发生的概率，而1-p是事件A不会发生的概率，并且p+(1-p)=1。因此，很多人在理解逻辑回归时，都对σ做出如下的解释：

我们让线性回归结果逼近0和1，此时σ和$1-\sigma$之和为1，因此它们可以被我们看作是一对**正反例发生的概率**，即σ是某样本i的标签被预测为1的概率，而$1-\sigma$是i的标签被预测为0的概率，$\frac{\sigma}{1-\sigma}$就是样本i的标签被预测为1的相对概率。基于这种理解，逻辑回归、即单层二分类神经网络返回的结果被当成是概率来看待和使用（如果直接说它就是概率，或许不太严谨）。每当我们希望求解“样本i的标签是1或是0的概率”时，我们就使用逻辑回归。

因此，当一个样本对应的σ_i越接近1或0，我们就认为逻辑回归对这个样本的预测结果越肯定，样本被分类正确的可能性也越高。如果σ_i非常接近阈值（比如0.5），就说明逻辑回归其实对这个样本究竟应该是哪一类别，不是非常肯定。

2 tensor实现二分类神经网络的正向传播

我们可以在PyTorch中非常简单地实现逻辑回归的预测过程，让我们来看下面这一组数据。很容易注意到，这组数据和上面的回归数据的特征（x_1, x_2）是完全一致的，只不过标签y由连续型结果转变为了分类型的0和1。这一组分类的规律是这样的：当两个特征都为1的时候标签就为1，否则标签就为0。这一组特殊的数据被我们称之为**“与门”（AND GATE）**，这里的“与”正是表示“特征一与特征二都是1”的含义。

x0	x1	x2	andgate
1	0	0	0
1	1	0	0
1	0	1	0
1	1	1	1

要拟合这组数据，只需要在刚才我们写好的代码后面加上sigmoid函数以及阈值处理后的变化。

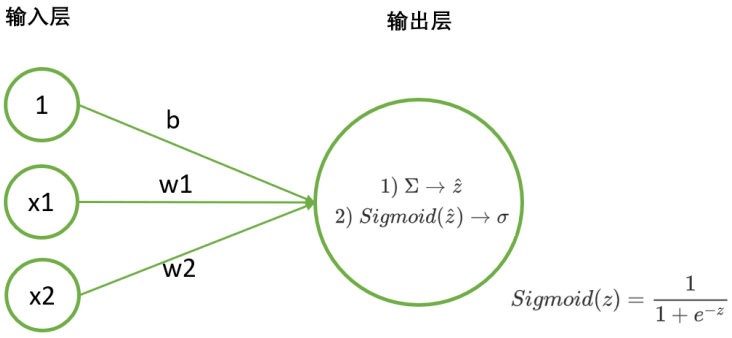
```
import torch
X = torch.tensor([[1,0,0],[1,1,0],[1,0,1],[1,1,1]], dtype = torch.float32)
andgate = torch.tensor([[0],[0],[0],[1]], dtype = torch.float32)
#保险起见，生成二维的、float32类型的标签
w = torch.tensor([-0.2,0.15,0.15], dtype = torch.float32)

def LogisticR(X,w):
    zhat = torch.mv(X,w)
    sigma = torch.sigmoid(zhat)
    #sigma = 1/(1+torch.exp(-zhat))
    andhat = torch.tensor([int(x) for x in sigma >= 0.5], dtype = torch.float32)
    return sigma, andhat

sigma, andhat = LogisticR(X,w)
```

接下来，我们对这段代码进行详细的说明：

```
#导入torch库
import torch

#特征张量，养成良好习惯，上来就定义数据类型
X = torch.tensor([[1,0,0],[1,1,0],[1,0,1],[1,1,1]], dtype = torch.float32)

#标签，分类问题的标签是整型
andgate = torch.tensor([0,0,0,1], dtype = torch.float32)

#定义w，注意这一组w与之前在回归中使用的完全一样
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
    #sigma = 1/(1+torch.exp(-zhat))
    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

andgate
```

```
andgate == andhat
#最后得到的都是0和1，虽然是andhat的数据格式是float32，但本质上数还是整数，不存在精度问题
```

可见，这里得到了与我们期待的结果一致的结果，这就将回归算法转变为了二分类。这个过程在神经网络中的表示图如下：

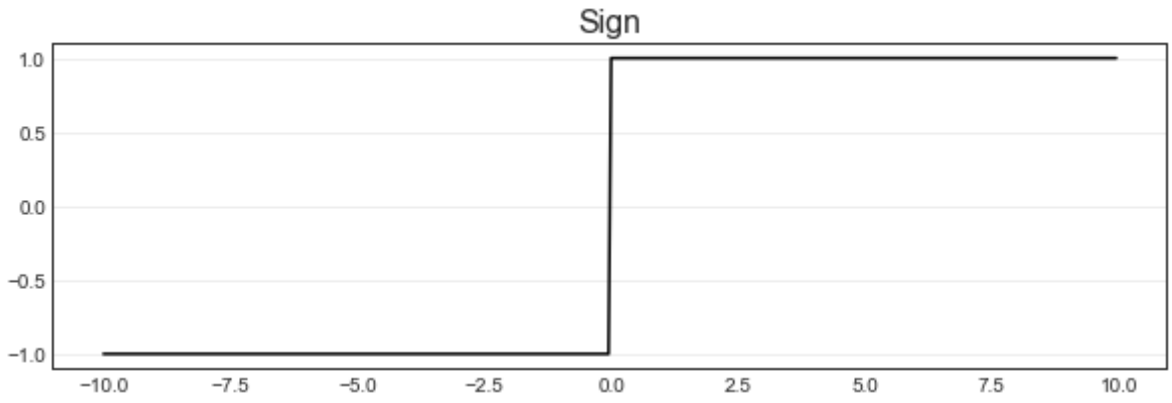


可以看出，这个结构与线性回归的神经网络唯一不同的就是**输出层中多出了一个Sigmoid(z)**。当有了Sigmoid函数的结果σ之后，只要了解阈值是0.5（或者任意我们自己设定的数值），就可以轻松地判断任意样本的预测标签$\hat{y}$。在二分类神经网络中，Sigmoid实现了将连续型数值转换为分类型数值的作用，在现代神经网络架构中，除了Sigmoid函数之外，还有许多其他的函数可以被用来将连续型数据分割为离散型数据，接下来，我们就介绍一下这些函数。

3 符号函数sign，ReLU，Tanh

- 符号函数sign

符号函数是图像如下所示的函数。



我们可以使用以下表达式来表示它：

$$y = \begin{cases} 1 & if\ z > 0 \\ 0 & if\ z = 0 \\ -1 & if\ z < 0 \end{cases} \tag{7}$$

由于函数的取值是间断的，**符号函数也被称为“阶跃函数”**，表示在0的两端，函数的结果y是从-1直接阶跃到了1。在这里，我们使用y而不是σ来表示输出的结果，是因为输出结果直接是0、1、-1这样的类别，就相当于标签了。对于sigmoid函数而言，σ返回的是0~1之间的概率值，如果我们希望获取最终预测出的类别，还需要将概率转变成0或1这样的数字才可以。但符号函数可以直接返回类别，因此我们可以认为符号函数输出的结果就是最终的预测结果y。在二分类中，符号函数也可以忽略中间$z = 0$的时候，直接分为0和1两类，用如下式子表示：

$$y = \begin{cases} 1 & if\ z > 0 \\ 0 & if\ z \leq 0 \end{cases} \tag{8}$$

等号被并在上方或下方都可以。这个式子可以很容易被转化为下面的式子：

$$\because z = w_1x_1 + w_2x_2 + b \tag{9}$$

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

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

此时，$-b$就是一个阈值，我们可以使用任意字母来替代它，比较常见的是字母θ。当然，不把它当做阈值，依然保留$w_1x_1 + w_2x_2 + b$与0进行比较的关系也没有任何问题。和sigmoid一样，我们也可以使用阶跃函数来处理"与门"的数据：

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

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

zhat, andhat = LinearRwithsign(X,w)

zhat

andhat
```

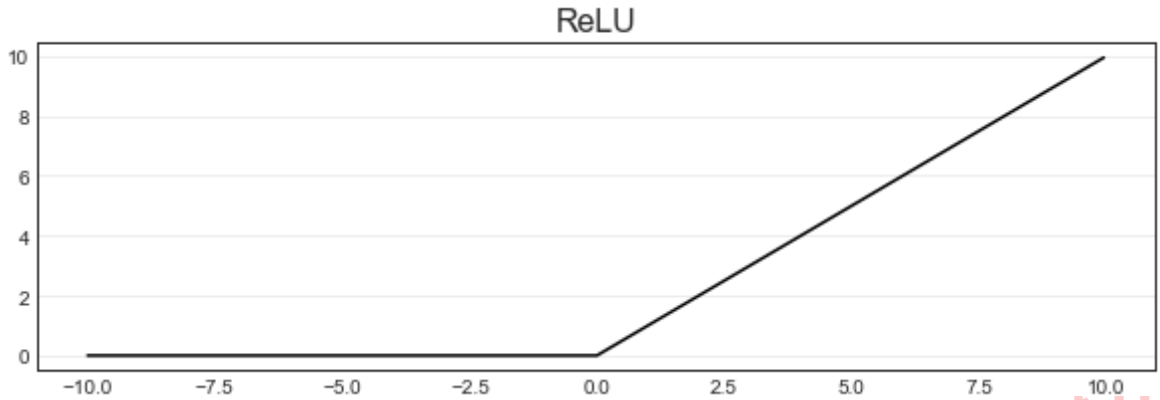
阶跃函数和sigmoid都可以完成二分类的任务。在神经网络的二分类中，σ的默认取值一般都是sigmoid函数，少用阶跃函数，这是由神经网络的解法决定的，我们将在第三部分来详细讲解。

- ReLU

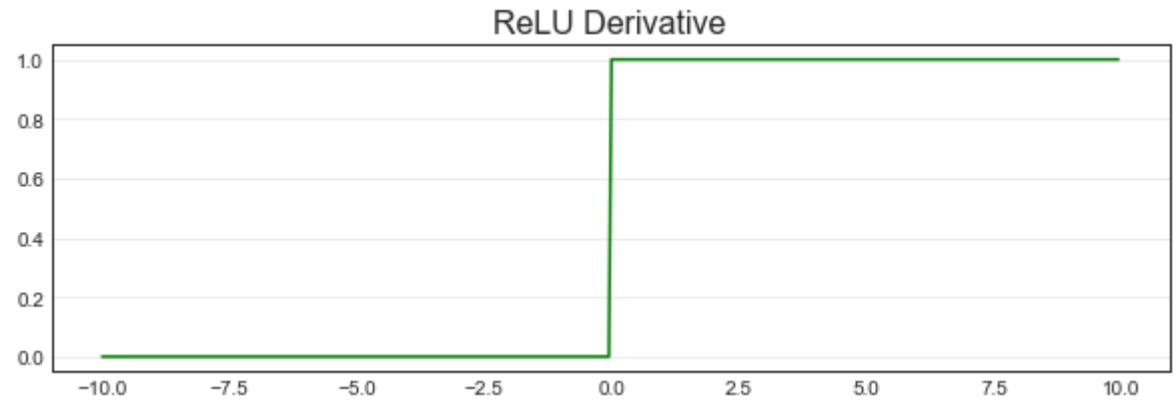
ReLU(Rectified Linear Unit)函数又名整流线型单元函数，英文发音为/rel-you/，是现在神经网络领域中的宠儿，应用甚至比sigmoid更广泛。ReLU提供了一个很简单的非线性变换：当输入的自变量大于0时，直接输出该值，当输入的自变量小于等于0时，输出0。这个过程可以用以下公式表示出来：

$$ReLU: \sigma = \begin{cases} z & (z > 0) \\ 0 & (z \leq 0) \end{cases}$$

ReLU函数是一个非常简单的函数，本质就是max(0,z)。max函数会从输入的数值中选择较大的那个值进行输出，以达到保留正数元素，将负元素清零的作用。ReLU的图像如下所示：



相对的，ReLU函数导数的图像如下：



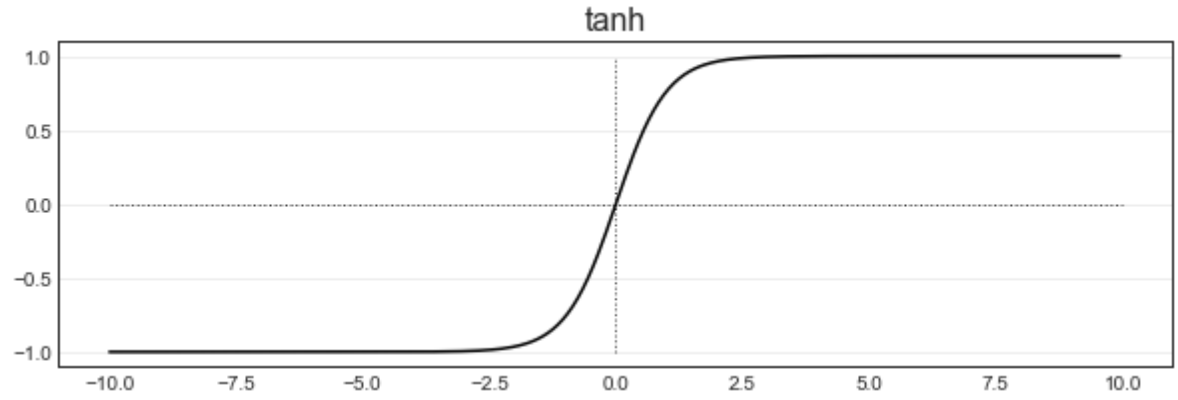
当输入z为正数时，ReLU函数的导数为1，当z为负数时，ReLU函数的导数为0，当输入为0时，ReLU函数不可导。因此，ReLU函数的导数图像看起来就是阶跃函数，这是一个美好的巧合。

- **tanh**

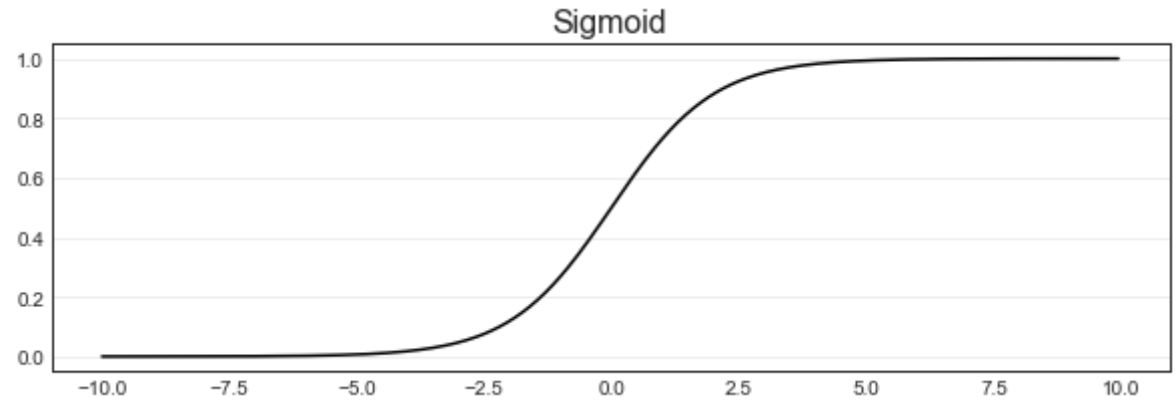
tanh（hyperbolic tangent）是双曲正切函数，英文发音/tæntʃ/或者/θæn/（两者均来源于柯林斯简明词典，p1520）。双曲正切函数的性质与sigmoid相似，它能够将数值压缩到(-1,1)区间内。

$$tanh : \sigma = \frac{e^{2z} - 1}{e^{2z} + 1} \tag{10}$$

而双曲正切函数的图像如下：

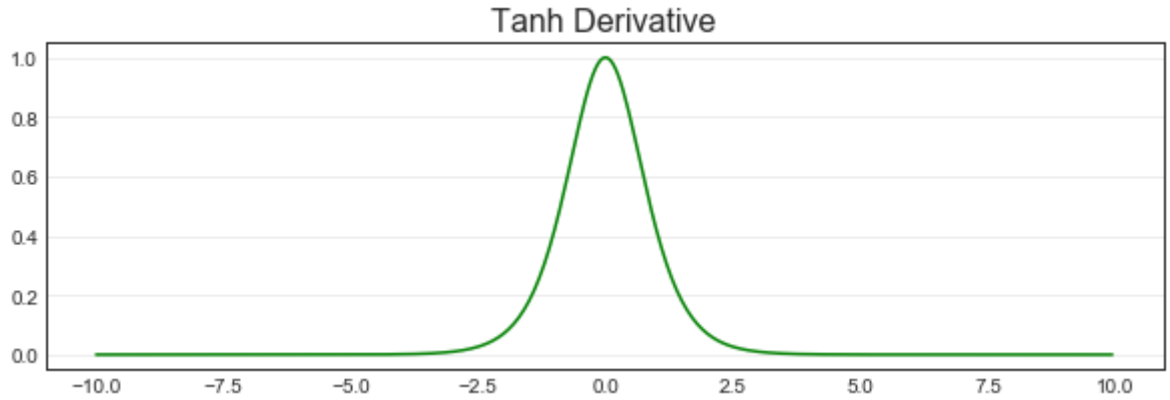


可以看出，tanh的图像和sigmoid函数很像，不过sigmoid函数的范围是在(0,1)之间，tanh却是在坐标系的原点(0,0)点上中心对称。



对tanh求导后可以得到如下公式和导数图像：

$$tanh'(z) = 1 - tanh^2(z) \tag{11}$$

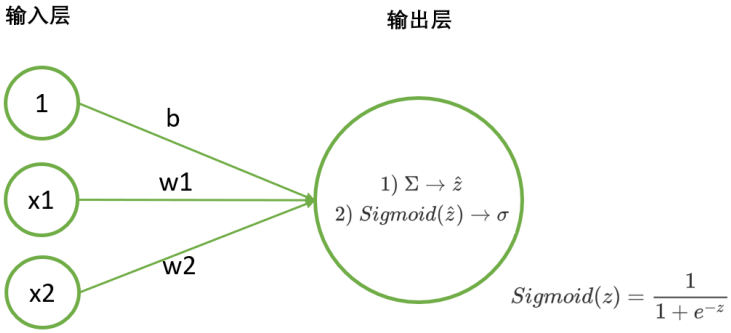


可以看出，当输入的z约接近于0，tanh函数导数也越接近最大值1，当输入越偏离0时，tanh函数的导数越接近于0。**这些函数是最常见的二分类转化函数，他们在神经网络的结构中有着不可替代的作用。**在单层神经网络中，这种作用是无法被体现的，因此关于这一点，我们可以之后再进行说明。到这里，我们只需要知道这些函数都可以将连续型数据转化为二分类就足够了。

4 torch.functional实现单层二分类神经网络的正向传播

之前我们使用torch.nn.Linear类实现了单层回归神经网络，现在我们试着来实现单层二分类神经网络，也就是逻辑回归。逻辑回归与线性回归的唯一区别，就是在线性回归的结果之后套上了sigmoid函数。不难想象，只要让nn.Linear的输出结果再经过sigmoid函数，就可以实现逻辑回归的正向传播了。

在PyTorch中，我们几乎总是从nn.functional中调用相关函数。



回顾一下我们的数据和网络架构：

x0	x1	x2	andgate
1	0	0	0
1	1	0	0
1	0	1	0
1	1	1	1

接下来，我们在之前线性回归代码的基础上，加上nn.functional来实现单层二分类神经网络：

```
import torch
from torch.nn import functional as F

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

torch.random.manual_seed(420) #人为设置随机数种子
dense = torch.nn.Linear(2,1)
zhat = dense(X)
sigma = F.sigmoid(zhat)
y = [int(x) for x in sigma > 0.5]
```

在这里，nn.Linear虽然依然是输出层，但却没有担任最终输出值的角色，因此这里我们使用dense作为变量名。dense表示紧密链接的层，即上一层的大部分神经元都与这一层的大部分神经元相连，在许多神经网络中我们都会用到密集链接的层，因此dense是我们经常会用到的一个变量名。我们将数据从nn.Linear传入，得到zhat，然后再将zhat的结果传入sigmoid函数，得到sigma，之后在设置阈值为0.5，得到最后的y。

在PyTorch中，我们可以从functional模块里找出大部分之前我们提到的函数，来看看ReLU、Tanh以及Sign函数都是如何使用PyTorch实现的吧：

```
torch.random.manual_seed(420) #人为设置随机数种子
dense = torch.nn.Linear(2,1)
zhat = dense(X)

#符号函数sign
torch.sign(zhat)

#ReLU
F.relu(zhat)

#tanh
torch.tanh(zhat)
```

在PyTorch的安排中，符号函数sign与双曲正切函数tanh更多时候只是被用作数学计算工具，而ReLU和Sigmoid却作为神经网络的组成部分被放在库functional中，这其实反映出实际使用时大部分人的选择。ReLU与Sigmoid还是主流的、位于nn.Linear后的函数。

三、多分类神经网络：Softmax回归

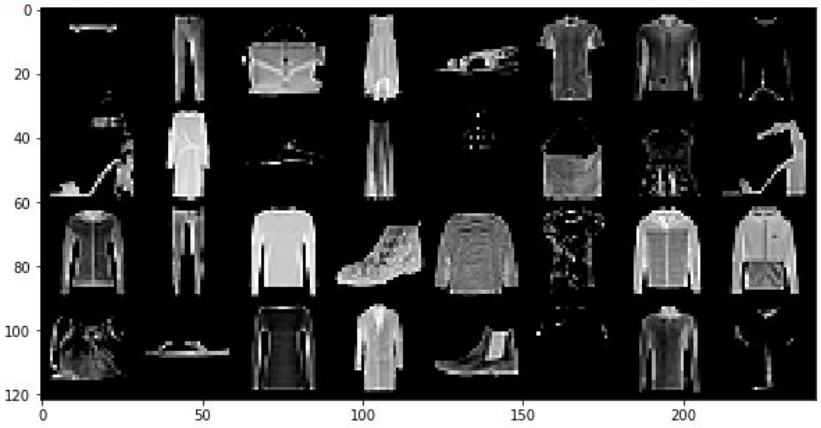
1 认识softmax函数

之前介绍分类神经网络时，我们只说明了二分类问题，即标签只有两种类别的问题（0和1，猫和狗）。虽然在实际应用中，许多分类问题都可以用二分类的思维解决，但依然存在很多多分类的情况，最典型的就是手写数字的识别问题。计算机在识别手写数字时，需要对每一位数字进行判断，而个位数字总共有10个（0_9），所以手写数字的分类是十分类问题，一般分别用0^9表示。

在之前展示深度学习中的数据时，我们也展示过另一个数据集的标签：

```
[90]: imshow(torchvision.utils.make_grid(image))
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```




```
[91]: image.shape
```

```
[91]: torch.Size([32, 1, 28, 28])
```

```
[92]: labels
```

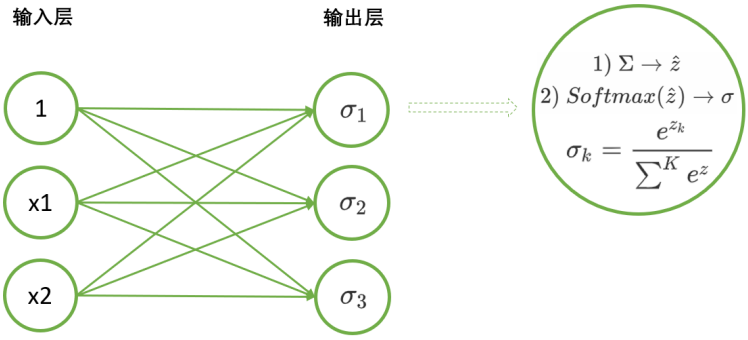
```
[92]: tensor([3, 1, 8, 3, 5, 6, 4, 2, 5, 3, 7, 3, 0, 8, 6, 5, 4, 1, 2, 9, 2, 0, 4, 4,
              8, 5, 2, 4, 9, 0, 6, 6])
```

这个数据集是对服装样式进行分类，从它标签也可以看出来，这也是一个多分类问题。正常来说，我们都对多分类的类别按整数[1,+∞]的数字进行编号，比较少有使用0类的情况。

在机器学习中，我们会使用二分类算法的Many-vs-Many（多对多）和One-vs-Rest（一对多）模式来进行多分类。其中，OvR是指将多个标签类别中的一类作为类别1，其他所有类别作为类别0，分别建立多个二分类模型，综合得出多分类结果的方法。MvM是指把好几个标签类作为1，剩下的几个标签类别作为0，同样分别建立多个二分类模型来得出多分类结果的方法。这两种方法非常有效，尤其是在逻辑回归做多分类的问题上能够解决很多问题，但是在深度学习世界却完全不奏效。理由非常简单：

1. 逻辑回归是一个单层神经网络，计算非常快速，在使用OvR和MvM这样需要同时建立多个模型的方法时，运算速度不会成为太大的问题。但真实使用的神经网络往往是一个庞大的算法，建立一个模型就会耗费很多时间，因此必须建立很多个模型来求解的方法对神经网络来说就不够高效。
2. 我们有更好的方法来解决这个问题，那就是softmax回归。

Softmax函数是深度学习基础中的基础，它是神经网络进行多分类时，默认放在输出层中处理数据的函数。假设现在神经网络是用于三分类数据，且三个分类分别是苹果，柠檬和百香果，序号则分别是分类1、分类2和分类3。则使用softmax函数的神经网络的模型会如下所示：



与二分类一样，我们从网络左侧输入特征，从右侧输出概率，且概率是通过线性回归的结果z外嵌套softmax函数来进行计算。在二分类时，输出层只有一个神经元，只输出样本对于正类别的概率（通常是标签为1的概率），而softmax的输出层有三个神经元，分别输出该样本的真实标签是苹果、柠檬或百香果的概率$\sigma_1, \sigma_2, \sigma_3$。**在多分类中，神经元的个数与标签类别的个数是一致的**，如果是十分类，在输出层上就会存在十个神经元，分别输出十个不同的概率。此时，**样本的预测标签就是所有输出的概率$\sigma_1, \sigma_2, \sigma_3$中最大的概率对应的标签类别。**

那每个概率是如何计算出来的呢？来看Softmax函数的公式：

$$\sigma_k = Softmax(z_k) = \frac{e^{z_k}}{\sum^K e^z} \tag{12}$$

其中e为自然常数（约为2.71828），z与sigmoid函数中的z一样，表示回归类算法（如线性回归）的结果。K表示该数据的标签中总共有K个标签类别，如三分类时$K=3$，四分类时$K=4$。k表示标签类别k类。很容易可以看出，Softmax函数的分子是多分类状况下某**一个**标签类别的回归结果的指数函数，分母是多分类状况下**所有**标签类别的回归结果的指数函数之和，因此**Softmax函数的结果代表了样本的结果为类别k的概率。**

举个例子，当我们有三个分类，分别是苹果，梨，百香果的时候，样本i被分类为百香果的概率$\sigma_{百香果}$为：

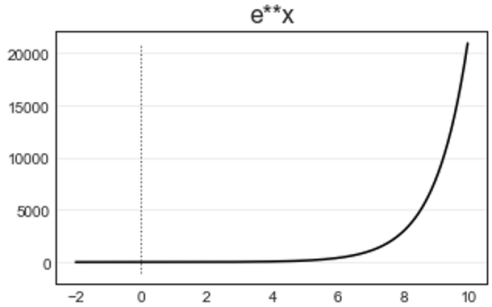
$$\sigma_{百香果} = \frac{e^{z_{百香果}}}{e^{z_{苹果}} + e^{z_{梨}} + e^{z_{百香果}}} \tag{13}$$

思考

为什么二分类时我们只需要输出一个概率，而多分类却需要输出多个概率？
当输入的特征张量结构为（4,2），且分类为三分类时，softmax回归输出的结构应该是怎样的？

2 PyTorch中的softmax函数

我们曾经提到过，神经网络是模型效果很好，但运算速度非常缓慢的算法。softmax函数也存在相同的问题——它可以将多分类的结果转变为概率（这是一个极大的优势），但它需要的计算量非常巨大。由于softmax的分子和分母中都带有e为底的指数函数，所以在计算中非常容易出现极大的数值。



如上图所示，e^{10}就已经等于20000了，而回归结果z完全可能是成千上万的数字。事实上e^{100}会变成一个后面有40多个0的超大值，e^{1000}则会直接返回无限大inf，这意味着这些数字已经超出了计算机处理数时要求的有限数据宽度，超大数值无法被计算机运算和表示。这种现象叫做“溢出”，当计算机返回”内存不足”或Python服务器直接断开连接的时候，可能就是发生了这个问题。来看看这个问题实际发生时的状况：

```
#对于单一样本，假定一组巨大的z
z = torch.tensor([1010,1000,990], dtype=torch.float32)
torch.exp(z) / torch.sum(torch.exp(z)) # softmax函数的运算
```

```
[77]: tensor([nan, nan, nan])
```

因此，我们一般不会亲自使用tensor来手写softmax函数。在PyTorch中，我们往往使用内置好的softmax函数来计算softmax的结果，我们可以使用torch.softmax来轻松的调用它，具体代码如下：

```
z = torch.tensor([1010,1000,990], dtype=torch.float32)
torch.softmax(z,0)

#你也可以使用F.softmax，它返回的结果与torch.softmax是完全一致的
```

可以看出，同样的数据在PyTorch的softmax计算下，不会发生溢出问题，这是因为PyTorch使用了一些巧妙地手段来解决溢出问题，感兴趣地小伙伴可以自行阅读本节最后的加餐内容。如果我们将z调整到较小的数值，可以发现，手写的softmax函数与torch库自带的softmax返回的结果是一模一样的。

```
#假设三个输出层神经元得到的z分别是10，9，5
z = torch.tensor([10,9,5], dtype=torch.float32)
torch.exp(z) / torch.sum(torch.exp(z)) # softmax函数的运算

z = torch.tensor([10,9,5], dtype=torch.float32)
torch.softmax(z,0)
```

从上面的结果可以看出，softmax函数输出的是从0到1.0之间的实数，而且多个输出值的总和是1。因为有了这个性质，我们可以把softmax函数的输出解释为“概率”，这和我们使用sigmoid函数之后认为函数返回的结果是概率异曲同工。从结果来看，我们可以认为返回了我们设定的z（[10,9,5]）的这个样本的结果应该是第一个类别（也就是z=10的类别），因为类别1的概率是最大的（实际上几乎达到了99.9%）。

需要注意的是，使用了softmax函数之后，各个z_k之间的大小关系并不会随之改变，这是因为指数函数e^z是单调递增函数，也就是说，使用softmax之前的z如果比较大，那使用softmax之后返回的概率也依然比较大。这是说，**无论是否使用softmax，我们都可以判断出样本被预测为哪一类，我们只需要看z最大的那一类就可以了。**所以，在神经网络进行分类的时候，如果不需要了解具体分类问题每一类的概率是多少，而只需要知道最终的分类结果，我们可以省略输出层上的softmax函数。

但如果，你确实遇见了需要自己计算的情况，你就可以使用torch中的softmax函数。大家可能有注意到，torch中的softmax函数有两个参数，第一个参数是我们输入的用来进行计算的张量z，另一个参数我输入了0。这其实代表了**你希望运行softmax计算的维度的索引。**

softmax函数只能对单一维度进行计算，它只能够识别单一维度上的不同类别，但我们输入softmax的张量z却可能是一个很高维的张量。比如：

```
s = torch.tensor([[[1,2],[3,4],[5,6]],[[5,6],[7,8],[9,10]]],
dtype=torch.float32)

s.ndim

s.shape
```

对于s而言，我们现在有三个维度——最外层代表了“2个二维张量”，3则代表每个二维张量中有3行，最后的2则代表每个二维张量中有2列。此时，我们可以从外向内索引我们的维度，索引0对应的就是最外层，索引2对应的就是最里层，相似的，我们也可以反向索引，-1对应的就是最里层，-3对应的就是最外层。所以softmax函数中需要我们输入的，就是我们希望在哪个维度上进行softmax运算。

```
torch.softmax(s,dim=0)
#在整个张量中，有2个张量，一个二维张量就是一类

torch.softmax(s,dim=1)
#在一个二维张量中，有3行数据，每一行是一种类别

torch.softmax(s,dim=2)
#在每一行中，有4个数据，每个数据是一种类别
```

看到计算的结果，你明白这个维度的含义了吗？实际上softmax函数在实际中应用不多，但是只要用到，新手就一定会被绕进dim参数的选项中去，因此我们在这里特此说明。

在实际中，训练神经网络时往往会使用softmax函数，但在预测时就不再使用softmax函数，而是直接读取结果最大的z对应的类别了。但无论如何，了解softmax是必要的，也是非常有用的。

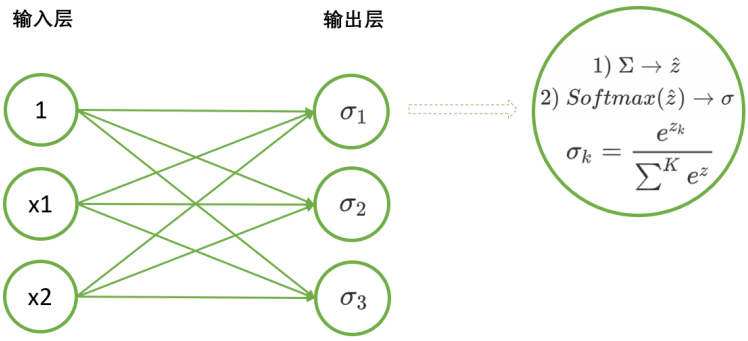
3 使用nn.Linear与functional实现多分类神经网络的正向传播

学到这里，我相信你应该能够写出softmax回归的正向传播代码。请使用如下的数据：

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

我强烈建议你先进行思考，对照下面的神经网络结构，尝试自己写一写这段代码，然后再查看答案。在这里你需要弄明白这些问题：

1. nn.Linear中需要输入什么结构？
2. softmax中的dim应该填写什么值？
3. 最终输出的sigma的shape应该是几乘几才对？



答案如下：

```
import torch
from torch.nn import functional as F

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

torch.random.manual_seed(420)
dense = torch.nn.Linear(2,3) #此时，输出层上的神经元个数是3个，因此应该是（2，3）
zhat = dense(X)
sigma = F.softmax(zhat,dim=1) #此时需要进行加和的维度是1

sigma.shape #这里应当返回的正确结构是(4,3)，4个样本，每个样本都有自己的3个类别对应的3个概率
```

4【加餐】解决Softmax的溢出问题

为了解决softmax函数的溢出问题，我们需要对softmax的公式进行如下更改。首先，在分子和分母上都乘上常数C。因为同时对分母和分子乘以相同的常数，所以计算结果不变。然后，把这个C移动到指数函数中，计作e^{logC}，在这里log指的是以自然底数e为底的对数函数。只要C大于0，logC就必然存在，此时我们可以把logC替换为另一个符号C'，它同样代表了一个常数。**即是说，其实我们只需要在softmax函数的两个指数函数自变量上都加上一个常数就可以了，这样做不会改变计算结果。**

$$
\begin{aligned}
o_k &= \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} \\
&= \frac{Ce^{z_k}}{C\sum_{i=1}^{K} e^{z_i}} \\
&= \frac{e^{logC} \cdot e^{z_k}}{\sum_{i=1}^{K} e^{logC} \cdot e^{z_i}} \\
&= \frac{e^{(z_k+logC)}}{\sum_{i=1}^{K} e^{(z_i+logC)}} \\
&= \frac{e^{(z_k+C')}}{\sum_{i=1}^{K} e^{(z_i+C')}}
\end{aligned}
$$

现在来思考，加入一个怎样的C'才会对抑制计算溢出最有效呢？我们是希望避免十分巨大的z，因此我们只要让C'的符号为负号，在让其大小接近z_k中的最大值就可以了。来看看在python中我们如何操作：

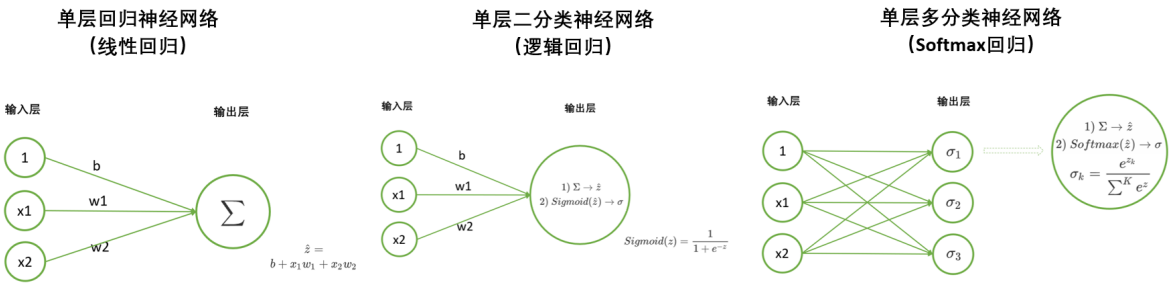
```
import numpy as np

z = z = np.array([1010,1000,990])

#定义softmax函数
def softmax(z):
    c = np.max(z)
    exp_z = np.exp(z - c) #溢出对策
    sum_exp_z = np.sum(exp_z)
    o = exp_z / sum_exp_z
    return o

#导入刚才定义的z
softmax(z)
```

来看看刚才求出的softmax函数的结果加和之后会显示怎样的效果：

```
sum(softmax(z))
```

四、回归vs二分类vs多分类

到这里，我们已经见过了三个不同的神经网络：



注意到有什么相似和不同了吗？

首先可能会注意到的是，这三个神经网络都是单层神经网络，除了输入层，他们都有且只有一层网络。实际上，现实中使用的神经网络几乎99%都是多层的，但我们的网络也能够顺利进行预测，这说明单层神经网络其实已经能够实现基本的预测功能。同时，这也说明了一个问题，无论处理的是回归还是分类，神经网络的处理原理是一致的。实际上，就连算法的限制、优化方法和求解方法也都是一致的。回归和分类神经网络唯一的不同只有输出层上的σ。

虽然线性回归看起来并没有σ的存在，但实际上我们可以认为线性回归中的σ是一个恒等函数（identity function），即是说$\sigma(z) = z$（相当于$y = x$，或$f(x) = x$）。而多分类的时候也可以不采用任何函数，只观察z的大小，所以多分类也可以被认为是利用了恒等函数作为σ。总结来说，回归和分类对应的σ分别如下：

输出类型	σ
回归	恒等函数
二分类	sigmoid或任意可以实现二分类的函数（通常都是sigmoid）
多分类	softmax或恒等函数

第二个很容易发现的现象是，只有多分类的情况在输出层出现了超过一个神经元。实际上，当处理单标签问题时（即只有一个y的问题），回归神经网络和二分类神经网络的输出层永远只有一个神经元，而只有多分类的情况才会让输出层上超过一个神经元。为了方便计算和演示，在之后的课程中我们都会使用**二分类神经网络**作为例子。