> 《2021 PyTorch深度学习实战》
>
> Lesson 10：神经网络的损失函数
>
> 作者：@菜菜TsaiTsai
>
> 版本号：2021/01/30 V1.0

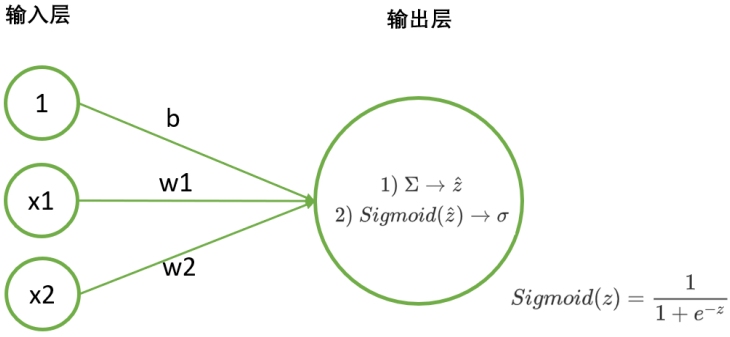
Lesson 10 神经网络的损失函数

在之前的课程中，我们已经完成了从0建立深层神经网络， 并完成正向传播的全过程。本节课开始，我们将以分类深层神经网络为例，为大家展示神经网络的学习和训练过程。在介绍PyTorch的基本工具AutoGrad库时，我们系统地介绍过数学中的优化问题和优化思想，我们介绍了最小二乘法以及梯度下降法这两个入门级优化算法的具体操作，并使用AutoGrad库实现了他们。接下来，我们将从梯度下降法向外拓展，介绍神经网络的损失函数、常用优化算法等信息，实现神经网络的学习和迭代。本节主要讲解神经网络常用的损失函数，并在PyTorch中实现这些函数。

Lesson 10 神经网络的损失函数

一、机器学习中的优化思想

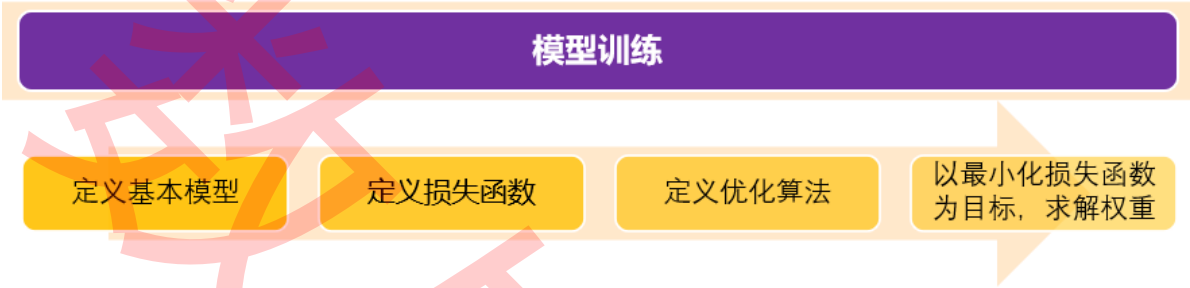
在之前的学习中，我们建立神经网络时总是先设定好w与b的值（或者由我们调用的PyTorch类帮助我们随机生成权重向量$\boldsymbol{w}$），接着通过加和求出z，再在z上嵌套sigmoid或者softmax函数，最终获得神经网络的输出。我们的代码及计算流程，总是从神经网络的左侧向右侧计算的。之前我们提到过，这是神经网络的**正向传播过程**。但很明显，这并不是神经网络算法的全流程，这个流程虽然可以输出预测结果，但却无法保证神经网络的输出结果与真实值接近。



在讲解线性回归时，我们提起过，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。构造预测函数核心就是**找出模型的权重向量$\boldsymbol{w}$，并令线性回归的输出结果与真实值相近**，也就是求解线性方程组中的w和b。对神经网络而言也是如此，我们的核心任务是**求解一组最适合的$\boldsymbol{w}$和$\boldsymbol{b}$，令神经网络的输出结果与真实值接近。**找寻这个w和b的过程就是"学习"，也叫做"训练"或者"建模"。

模型训练的目标
求解一组最适合的权重向量$\boldsymbol{w}$，令神经网络的输出结果与真实值尽量接近。

那我们如何评价w和b是否合适呢？我们又如何衡量我们的输出结果与真实值之间的差异大小呢？此时，我们就需要使用机器学习中通用的优化流程了。在讲解autograd的时候，其实我们已经提过这个优化流程，在这里我们稍微复习一下：



1）提出基本模型，明确目标

我们的基本模型就是我们自建的神经网络架构，我们需要求解的就是神经网络架构中的权重向量$\boldsymbol{w}$。

2）确定损失函数/目标函数

我们需要定义某个评估指标，用以衡量模型权重为$\boldsymbol{w}$的情况下，预测结果与真实结果的差异。当真实值与预测值差异越大时，我们就认为神经网络学习过程中丢失了许多信息，丢失的这部分被形象地称为"损失"，因此评估真实值与预测值差异的函数被我们称为"损失函数"。

关键概念：损失函数
衡量真实值与预测结果的差异，评价模型学习过程中产生的损失的函数。 **在数学上，表示为以需要求解的权重向量$\boldsymbol{w}$为自变量的函数$L(\boldsymbol{w})$。** 如果损失函数的值很小，则说明模型预测值与真实值很接近，模型在数据集上表现优异，权重优秀 如果损失函数的值大，则说明模型预测值与真实值差异很大，模型在数据集上表现差劲，权重糟糕

我们希望**损失函数越小越好**，以此，我们将问题转变为求解函数$L(\boldsymbol{w})$的最小值所对应的自变量$\boldsymbol{w}$。但是，损失函数往往不是一个简单的函数，求解复杂函数就需要复杂的数学工具。在这里，我们使用的数学工具可能有两部分：

- 将损失函数$L(\boldsymbol{w})$转变成凸函数的数学方法，常见的有拉格朗日变换等
- 在凸函数上求解$L(\boldsymbol{w})$的最小值对应的$\boldsymbol{w}$的方法，也就是以梯度下降为代表的优化算法

3）确定适合的优化算法

4）利用优化算法，最小化损失函数，求解最佳权重$\boldsymbol{w}$（训练）

之前我们在线性回归上走过这个全流程。对线性回归，我们的损失函数是SSE，优化算法是最小二乘法和梯度下降法，两者都是对机器学习来说非常重要的优化算法。但遗憾的是，最小二乘法作为入门级优化算法，有较多的假设和先决条件，不足以应对神经网络需要被应用的各种复杂环境。梯度下降法应用广泛，不过也有很多问题需要改进。接下来，我将主要以分类深层神经网络为例来介绍神经网络中所使用的入门级损失函数及优化算法。

二、回归：误差平方和SSE

对于回归类神经网络而言，最常见的损失函数是SSE（Sum of the Squared Errors），现在已经是我们第三次见到SSE的公式了：

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

其中z_i是样本i的真实值，而$\hat{z}_i$是样本i的预测值。对于全部样本的平均损失，则可以写作：

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(z_i - \hat{z}_i)^2$$

在PyTorch中，我们可以简单通过以下代码调用MSE：

```
from torch.nn import MSELoss #类

yhat = torch.randn(size=(50,),dtype=torch.float32)
y = torch.randn(size=(50,),dtype=torch.float32)

criterion =MSELoss() #实例化
loss = criterion(yhat,y)

loss #没有设置随机数种子，所以每次运行的数字都会不一致

#在MSELoss中有重要的参数，reduction
#当reduction = "mean" (默认也是mean)，则输出MSE
#当reduction = "sum"，则输出SSE

criterion = MSELoss(reduction = "mean") #实例化
criterion(yhat,y)

criterion = MSELoss(reduction = "sum")
criterion(yhat,y)
```

三、二分类交叉熵损失函数

在这一节中，我们将介绍二分类神经网络的损失函数：二分类交叉熵损失函数（Binary Cross Entropy Loss），也叫做对数损失（log loss）。这个损失函数被广泛地使用在任何输出结果是二分类的神经网络中，即不止限于单层神经网络，还可被拓展到多分类中，因此理解二分类交叉熵损失是非常重要的一环。大多数时候，除非特殊声明为二分类，否则提到交叉熵损失，我们会默认算法的分类目标是多分类。

二分类交叉熵损失函数是由极大似然估计推导出来的，对于有m个样本的数据集而言，在全部样本上的平均损失写作：

$$L(w) = -\sum_{i=1}^{m}(y_i * ln(\sigma_i) + (1-y_i) * ln(1-\sigma_i))$$

在单个样本的损失写作：

$$L(w)_i = -(y_i * ln(\sigma_i) + (1-y_i) * ln(1-\sigma_i))$$

其中，ln是以自然底数e为底的对数函数，w表示求解出来的一组权重（在等号的右侧，w在σ里），m是样本的个数，y_i是样本i上真实的标签，σ_i是样本i上，基于参数w计算出来的sigmoid函数的返回值，x_i是样本i各个特征的取值。我们的目标，就是求解出使$L(w)$最小的w取值。注意，在神经网络中，特征张量$\mathbf{X}$是自变量，权重是w。但在损失函数中，权重w是损失函数的自变量，特征x和真实标签y都是已知的数据，相当于是常数。不同的函数中，自变量和参数各有不同，因此大家需要在数学计算中，尤其是求导的时候避免混淆。

1 极大似然估计求解二分类交叉熵损失

二分类交叉熵损失函数是怎么来的呢？为什么这个函数就能够代表二分类的时候，真实值与预测值的差异呢？

在这里，我们基于极大似然法来推导交叉熵损失，这个推导过程能够帮助我们充分了解交叉熵损失的含义，以及为什么$L(w)$的最小化能够实现模型在数据集上的拟合最好。

极大似然估计（Maximum Likelihood Estimate，MLE）的感性认识

如果一个事件的发生概率很大，那这个事件应该很容易发生。相应的，如果依赖于权重w的任意事件的发生就是我们的目标，那我们只要寻找令其发生概率最大化的权重w就可以了。**寻找相应的权重w，使得目标事件的发生概率最大，就是极大似然估计的基本方法。**
其步骤如下：
1、构筑似然函数$\boldsymbol{P(w)}$，用于评估目标事件发生的概率，该函数**被设计成**目标事件发生时，概率最大
2、对整体似然函数取对数，构成对数似然函数$\boldsymbol{lnP(w)}$
3、在对数似然函数上对权重w求导，并使导数为0，对权重进行求解

在二分类的例子中，我们的“任意事件”就是**每个样本的分类都正确**，对数似然函数的负数就是我们的损失函数。接下来，我们来看看逻辑回归的对数似然函数是怎样构筑的。

- **构筑对数似然函数**

二分类神经网络的标签是[0,1]，此标签服从伯努利分布(即0-1分布)，因此可得：

样本i在由特征向量$\boldsymbol{x_i}$和权重向量$\boldsymbol{w}$组成的预测函数中，样本标签被预测为1的概率为：

$$P_1 = P(\hat{y}_i = 1|\boldsymbol{x_i}, \boldsymbol{w}) = \sigma$$

对二分类而言，σ就是sigmoid函数的结果。

样本i在由特征向量$\mathbf{x_i}$和权重向量$\mathbf{w}$组成的预测函数中，样本标签被预测为0的概率为：

$$P_0 = P(\hat{y}_i = 0|\boldsymbol{x_i}, \boldsymbol{w}) = 1 - \sigma$$

当P_1的值为1的时候，代表样本i的标签被预测为1，当P_0的值为1的时候，代表样本i的标签被预测为0。P_1与P_0相加是一定等于1的。

假设样本i的真实标签y_i为1，并且P_1也为1的话，那就说明我们将i的标签预测为1的概率很大，与真实值一致，那模型的预测就是准确的，拟合程度很高，信息损失很少。相反，如果真实标签y_i为1，我们的P_1却很接近0，这就说明我们将i的标签预测为1的概率很小，即与真实值一致的概率很小，那模型的预测就是失败的，拟合程度很低，信息损失很多。当y_i为0时，也是同样的道理。所以，当y_i为1的时候，我们希望P_1非常接近1，当y_i为0的时候，我们希望P_0非常接近1，这样，模型的效果就很好，信息损失就很少。

真实标签yi	被预测为1的概率P1	被预测为0的概率P0	样本被预测为?	与真实值一致吗?	拟合状况	信息损失
1	0	1	0	否	坏	大
1	1	0	1	是	好	小
0	0	1	0	是	好	小
0	1	0	1	否	坏	大

将两种取值的概率整合，我们可以定义如下等式：

$$P(\hat{y}_i|\boldsymbol{x_i},\boldsymbol{w}) = P_1^{y_i} * P_0^{1-y_i}$$

这个等式代表同时代表了P_1和P_0，在数学上，它被叫做逻辑回归的假设函数。

当样本i的真实标签y_i为1的时候，$1-y_i$就等于0，P_0的0次方就是1，所以$P(\hat{y}_i|\boldsymbol{x_i},\boldsymbol{w})$就等于$P_1$，这个时候，如果$P_1$为1，模型的效果就很好，损失就很小。

同理，当y_i为0的时候，$P(\hat{y}_i|\boldsymbol{x_i},\boldsymbol{w})$就等于$P_0$，此时如果$P_0$非常接近1，模型的效果就很好，损失就很小。

所以，为了达成让模型拟合好，损失小的目的，我们每时每刻都希望$P(\hat{y}_i|x_i,w)$的值等于1。而$P(\hat{y}_i|\boldsymbol{x_i},\boldsymbol{w})$的本质是样本i由特征向量$\boldsymbol{x_i}$和权重$\boldsymbol{w}$组成的预测函数中，预测出所有可能的$\hat{y}_i$的概率，因此1是它的最大值。**也就是说，每时每刻，我们都在追求$P(\hat{y}_i|\boldsymbol{x_i},\boldsymbol{w})$的最大值。**而寻找相应的参数$w$，使得每次得到的预测概率最大，正是极大似然估计的基本方法，不过$P(\hat{y}_i|\boldsymbol{x_i},\boldsymbol{w})$是对单个样本而言的，因此我们还需要将其拓展到多个样本上。

$P(\hat{y}_i|\boldsymbol{x_i},\boldsymbol{w})$是对单个样本i而言的函数，对一个训练集的m个样本来说，我们可以定义如下等式来表达所有样本在特征张量$\boldsymbol{X}$和权重向量$\boldsymbol{w}$组成的预测函数中，预测出所有可能的$\hat{y}$的概率**P**为：

$$\begin{aligned}\boldsymbol{P} &= \prod_{i=1}^{m} P(\hat{y}_i|x_i,w) \\ &= \prod_{i=1}^{m} (P_1^{y_i} * P_0^{1-y_i}) \\ &= \prod_{i=1}^{m} (\sigma_i^{y_i} * (1-\sigma_i)^{1-y_i})\end{aligned}$$

这个函数就是逻辑回归的似然函数。对该概率**P**取以e为底的对数，再由$log(A*B) = logA + logB$和$logA^B = BlogA$可得到逻辑回归的对数似然函数：

$$\begin{aligned}ln\boldsymbol{P} &= ln\prod_{i=1}^{m} (\sigma_i^{y_i} * (1-\sigma_i)^{1-y_i}) \\ &= \sum_{i=1}^{m} ln(\sigma_i^{y_i} * (1-\sigma_i)^{1-y_i}) \\ &= \sum_{i=1}^{m} (ln\,\sigma_i^{y_i} \;+\; ln(1-\sigma_i)^{1-y_i}) \\ &= \sum_{i=1}^{m} (y_i * ln(\sigma_i) \;+\; (1-y_i) * ln(1-\sigma_i))\end{aligned}$$

这就是我们的二分类交叉熵函数。为了数学上的便利以及更好地定义"损失"的含义，我们希望将极大值问题转换为极小值问题，因此我们对$ln\boldsymbol{P}$取负，并且让权重w作为函数的自变量，就得到了我们的损失函数$L(w)$：

$$L(w) = -\sum_{i=1}^{m} (y_i * ln(\sigma_i) \;+\; (1-y_i) * ln(1-\sigma_i))$$

现在，我们已经将模型拟合中的“最小化损失”问题，转换成了对函数求解极值的问题。这就是一个，基于逻辑回归的返回值σ_i的概率性质以及极大似然估计得出的损失函数。**在这个函数上，我们只要追求最小值，就能让模型在训练数据上的拟合效果最好，损失最低。**

在极大似然估计中，我们只要在对数似然函数上对权重w求导，再令导数为0，就可以求解出最合适的w，但是对于像交叉熵这样复杂的损失函数，加上神经网络中复杂的权重组合，令所有权重的导数为0并一个个求解方程的难度很大。因此我们要使用优化算法，这部分我们下一章展开来聊。

2 用tensor实现二分类交叉熵损失

现在，让我们在PyTorch中来实现二分类交叉熵损失函数。首先使用基本的tensor方法来试试看，以加深我们对二分类交叉熵损失的印象：

```
import torch
import time

N = 3*pow(10,3)
torch.random.manual_seed(420)
X = torch.rand((N,4),dtype=torch.float32)
w = torch.rand((4,1),dtype=torch.float32,requires_grad=True)
y = torch.randint(low=0,high=2,size=(N,1),dtype=torch.float32)

zhat = torch.mm(X,w)
sigma = torch.sigmoid(zhat)

Loss = -(1/N)*torch.sum((1-y)*torch.log(1-sigma)+y*torch.log(sigma))
```

注意，在写损失函数这样的复杂函数时，除了普通的加减乘除以外的全部计算，都要使用torch中的函数，因为tensor的运算速度是远远超过普通Python代码，甚至是NumPy的。你可以试着比较在样本量为300W时，以下两行代码运行的时间差异：

```
#你可以试着比较在样本量为300W时，以下两行代码运行的时间差异。这段代码不需要GPU。
#如果你的电脑内存或计算资源有限，可以试着将样本量调小为30W或3W

N = 3*pow(10,6)
torch.random.manual_seed(420)
X = torch.rand((N,4),dtype=torch.float32)
w = torch.rand((4,1),dtype=torch.float32,requires_grad=True)
y = torch.randint(low=0,high=2,size=(N,1),dtype=torch.float32)

zhat = torch.mm(X,w)
sigma = torch.sigmoid(zhat)

start = time.time()
L1 = -(1/N)*torch.sum((1-y)*torch.log(1-sigma)+y*torch.log(sigma))
now = time.time() #seconds
print(now - start)

start = time.time()
L2 = -(1/N)*sum((1-y)*torch.log(1-sigma)+y*torch.log(sigma))
now = time.time() #seconds
print(now - start)
```

从运行结果来看，除了加减乘除，我们应该尽量避免使用任何Python原生的计算方法。**如果可能的话，让PyTorch处理一切。**

3 用PyTorch中的类实现二分类交叉熵损失

在PyTorch当中，我们有多种方式可以调用二分类交叉熵损失函数。

方法1：nn模块中的类
`class` BCEWithLogitsLoss `class` BCELoss

对于二分类交叉熵损失，nn提供了两个类：*BCEWithLogitsLoss*以及*BCELoss*。虽然PyTorch官方没有直接明确，但实际上两个函数所需要输入的参数不同。

*BCEWithLogitsLoss*内置了sigmoid函数与交叉熵函数，它会自动计算输入值的sigmoid值，因此需要输入zhat与真实标签，且顺序不能变化，zhat必须在前。

相对的，*BCELoss*中只有交叉熵函数，没有sigmoid层，因此需要输入sigma与真实标签，且顺序不能变化。

同时，这两个函数都要求预测值与真实标签的**数据类型以及结构（shape）必须相同**，否则运行就会报错。

接下来，我们来看看这两个类是如何使用的：

```
import torch.nn as nn

#调用nn模块下的类
criterion = nn.BCELoss() #实例化
loss = criterion(sigma,y)
loss

criterion2 = nn.BCEWithLogitsLoss() #实例化
loss = criterion2(zhat,y)
loss
```

可以看出，两个类的结果是一致的。根据PyTorch官方的公告，他们更推荐使用BCEWithLogitsLoss这个内置了sigmoid函数的类。内置的sigmoid函数可以让精度问题被缩小（因为将指数运算包含在了内部），以维持算法运行时的稳定性，即是说当数据量变大、数据本身也变大时，BCELoss类产生的结果可能有精度问题。所以，当我们的输出层使用sigmoid函数时，我们就可以使用BCEWithLogitsLoss作为损失函数。

与MSELoss相同，二分类交叉熵的类们也有参数reduction，默认是”mean“，表示求解所有样本平均的损失，也可换为”sum”，要求输出整体的损失。以及，还可以使用选项“none”，表示不对损失结果做任何聚合运算，直接输出每个样本对应的损失矩阵。

```
criterion2 = nn.BCEWithLogitsLoss(reduction = "mean")
loss = criterion2(zhat,y)
loss

criterion2 = nn.BCEWithLogitsLoss(reduction = "sum")
loss = criterion2(zhat,y)
loss

criterion2 = nn.BCEWithLogitsLoss(reduction = "none")
loss = criterion2(zhat,y)
loss
```

第二种方法很少用，我们了解一下即可：

方法2：functional库中的计算函数

`function` F.binary_cross_entropy_with_logits
`function` F.binary_cross_entropy

和nn中的类们相似，名称中带有Logits的是内置了sigmoid功能的函数，没有带Logits的，是只包含交叉熵损失的函数。对于含有sigmoid功能的函数，我们需要的输入是zhat与标签，不含sigmoid的函数我们则需要输入sigma与标签。同样的，这两个函数对输入有严格的要求，输入的预测值必须与标签结构一致、数据类型一致。我们来看看他们的运行结果：

```
from torch.nn import functional as F

#直接调用functional库中的计算函数
F.binary_cross_entropy_with_logits(zhat,y)

F.binary_cross_entropy(sigma,y)
```

在这里，两个函数的运行结果是一致的。同样的，PyTorch官方推荐的是内置sigmoid功能的函数binary_cross_entropy_with_logits。通常来说，我们都使用类，不使用函数。虽然代码会因此变得稍稍有点复杂，但为了代码的稳定性与日后维护，使用类是更好的选择。当然，在进行代码演示和快速测算的时候，使用函数或者类都没有问题。

四、多分类交叉熵损失函数

1 由二分类推广到多分类

二分类交叉熵损失可以被推广到多分类上，但在实际处理时，二分类与多分类却有一些关键的区别。依然使用极大似然估计的推导流程，首先我们来确定单一样本概率最大化后的似然函数。

对于多分类的状况而言，标签不再服从伯努利分布（0-1分布），因此我们可以定义，样本i在由特征向量$\boldsymbol{x}_i$和权重向量$\boldsymbol{w}$组成的预测函数中，样本标签被预测为类别k的概率为：

$$P_k = P(\hat{y}_i = k|\boldsymbol{x}_i, \boldsymbol{w}) = \sigma$$

对于多分类算法而言，σ就是softmax函数返回的对应类别的值。

假设一种最简单的情况：我们现在有三分类[1, 2, 3]，则样本i被预测为三个类别的概率分别为：

$$P_1 = P(\hat{y}_i = 1|\boldsymbol{x_i}, \boldsymbol{w}) = \sigma_1$$
$$P_2 = P(\hat{y}_i = 2|\boldsymbol{x_i}, \boldsymbol{w}) = \sigma_2$$
$$P_3 = P(\hat{y}_i = 3|\boldsymbol{x_i}, \boldsymbol{w}) = \sigma_3$$

假设样本的真实标签为1，我们就希望P_1最大，同理，如果样本的真实标签为其他值，我们就希望其他值所对应的概率最大。在二分类中，我们将y和$(1-y)$作为概率P的指数，以此来融合真实标签为0和为1的两种状况。但在多分类中，我们的真实标签可能是任意整数，无法使用y和$(1-y)$这样的结构来构建似然函数。所以我们认为，如果多分类的标签也可以使用0和1来表示就好了，这样我们就可以继续使用真实标签作为指数的方式。

因此，我们对多分类的标签做出了如下变化：

样本	y
样本1	2
样本2	3
样本3	1
……	
样本N	3

样本	y=1	y=2	y=3
样本1	0	1	0
样本2	0	0	1
样本3	1	0	0
……			
样本N	0	0	1

softmax函数输出结果

样本	σ(y=1)	σ(y=2)	σ(y=3)
样本1	0.234	0.687	0.079
样本2	0.135	0.335	0.53
样本3	0.597	0.234	0.169
……			
样本N	0.246	0.112	0.642

原本的真实标签y是含有[1, 2, 3]三个分类的列向量，现在我们把它变成了标签矩阵，每个样本对应一个向量。（如果你熟悉机器学习或统计学，你能够一眼看出这其实就是独热编码one-hot）。在矩阵中，每一行依旧对应样本，但却由三分类衍生出了三个新的列，分别代表：真实标签是否等于1、等于2以及等于3。在矩阵中，我们使用“1”标注出样本的真实标签的位置，使用0表示样本的真实标签不是这个标签。不难注意到，这个标签矩阵的结构其实是和softmax函数输出的概率矩阵的结构一致，并且一一对应的。

回顾下二分类的似然函数：

$$P(\hat{y}_i|\boldsymbol{x_i}, \boldsymbol{w}) = P_1^{y_i} * P_0^{1-y_i}$$

当我们把标签整合为标签矩阵后，我们就可以将单个样本在总共K个分类情况整合为以下的似然函数：

$$P(\hat{y}_i|\boldsymbol{x_i}, \boldsymbol{w}) = P_1^{y_{i(k=1)}} * P_2^{y_{i(k=2)}} * P_3^{y_{i(k=3)}} * \cdots * P_K^{y_{i(k=K)}}$$

其中P就是样本标签被预测为某个具体值的概率，而右上角的指数就是标签矩阵中对应的值，即这个样本的真实标签是否为当前标签的判断（是就是1，否就是0）。

> **注意**
>
> 许多教材和公式中，都会把多分类似然函数概率的指数直接写作y_i，若你能够理解此处的指数其实是0和1，（0 - 不是真实标签，1 - 是真实标签），而不是真正的标签y_i，那你直接把指数写作y_i也是没问题的。为避免混淆，在我们的课程中，还是写作$y_{i(k=\text{具体标签值})}$。

更具体的，小k代表y的真实取值，K代表总共有K个分类（此处不是非常严谨，按道理说若K代表总共有K个类别，则不应该再使用K代表某个具体类别，但在这里，由于我们使用的类别编号与类别本身相同，所以为了公式的简化，使用了这样不严谨的表示方式）。虽然是连乘，但对于一个样本，除了自己所在的真实类别指数y_i会是1之外，其他类别的指数都为0，所以被分类为其他类别的概率在这个式子里就都为0。所以我们可以将式子简写为：

$$P(\hat{y}_i|\boldsymbol{x_i}, \boldsymbol{w}) = P_j^{y_{i(k=j)}}, \quad j\text{为样本}i\text{所对应的真实标签的编号}$$

对一个训练集的m个样本来说，我们可以定义如下等式来表达所有样本在特征张量$\boldsymbol{X}$和权重向量$\boldsymbol{w}$组成的预测函数中，预测出所有可能的$\hat{y}$的概率**P**为：

$$\begin{aligned}\boldsymbol{P} &= \prod_{i=1}^{m} P(\hat{y}_i|x_i,w) \\ &= \prod_{i=1}^{m} P_j^{y_{i(k=j)}} \\ &= \prod_{i=1}^{m} \sigma_j^{y_{i(k=j)}}\end{aligned}$$

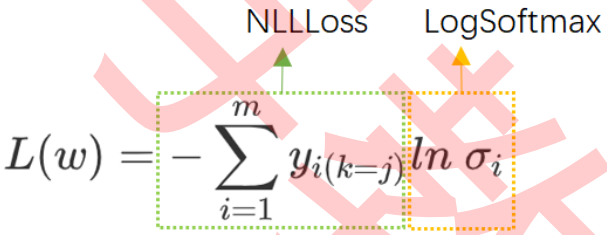
这就是多分类状况下的似然函数。与二分类一致，似然函数解出来后，我们需要对似然函数求对数：

$$\begin{aligned}ln\boldsymbol{P} &= ln\prod_{i=1}^{m} \sigma_j^{y_{i(k=j)}} \\ &= \sum_{i=1}^{m} ln(\sigma_j^{y_{i(k=j)}}) \\ &= \sum_{i=1}^{m} y_{i(k=j)} ln\ \sigma_i\end{aligned}$$

其中σ就是softmax函数返回的对应类别的值。再对整个公式取负，就得到了多分类状况下的损失函数：

$$L(w) = -\sum_{i=1}^{m} y_{i(k=j)} ln\ \sigma_i$$

这个函数就是我们之前提到过的**交叉熵函数**。不难看出，二分类的交叉熵函数其实是多分类的一种特殊情况。



交叉熵函数十分特殊，虽然我们求解过程中，取对数的操作是在确定了似然函数后才进行的，但从计算结果来看，对数操作其实只对softmax函数的结果σ起效。因此在实际操作中，我们把$ln(softmax(z))$这样的函数单独定义了一个功能做logsoftmax，PyTorch中可以直接通过nn.logsoftmax类调用这个功能。同时，我们把对数之外的，乘以标签、加和、取负等等过程打包起来，称之为负对数似然函数（Negative Log Likelihood function），在PyTorch中可以使用nn.NLLLoss来进行调用。**也就是说，在计算损失函数时，我们不再需要使用单独的softmax函数了。**

2 用PyTorch实现多分类交叉熵损失

在PyTorch中实现交叉熵函数的时候，有两种办法：

- 调用logsoftmax和NLLLoss实现

```
import torch
import torch.nn as nn

N = 3*pow(10,2)
torch.random.manual_seed(420)
X = torch.rand((N,4),dtype=torch.float32)
w = torch.rand((4,3),dtype=torch.float32,requires_grad=True)
```

```
#定义y时应该怎么做？应该设置为矩阵吗？
y = torch.randint(low=0,high=3,size=(N,),dtype=torch.float32)

zhat = torch.mm(X,w)

#从这里开始调用softmax和NLLLoss

logsm = nn.LogSoftmax(dim=1) #实例化
logsigma = logsm(zhat)

criterion = nn.NLLLoss() #实例化
#由于交叉熵损失需要将标签转化为独热形式，因此不接受浮点数作为标签的输入
#对NLLLoss而言，需要输入logsigma
criterion(logsigma,y.long())
```

更加简便的方法是：

- 直接调用CrossEntropyLoss

```
criterion = nn.CrossEntropyLoss()
#对打包好的CorssEnrtopyLoss而言，只需要输入zhat
criterion(zhat,y.long())
```

可以发现，两种输出方法得到的损失函数结果是一致的。与其他损失函数一致，CrossEntropyLoss也有参数reduction，可以设置为mean、sum以及None，大家可以自行尝试其代码并查看返回结果。

无论时二分类还是多分类，PyTorch都提供了包含输出层激活函数和不包含输出层激活函数的类两种选择。在实际神经网络建模中，类可以被放入定义好的Model类中去构建神经网络的结构，因此是否包含激活函数，就需要由用户来自行选择。

- 重视展示网络结构和灵活性，应该使用不包含输出层激活函数的类

通常在Model类中，__init__中层的数量与forward函数中对应的激活函数的数量是一致的，如果我们使用内置sigmoid/logsoftmax功能的类来计算损失函数，forward函数在定义时就会少一层（输出层），网络结构展示就不够简单明了，对于结构复杂的网络而言，结构清晰就更为重要。同时，如果激活函数是单独写的，要修改激活函数就变得很容易，如果混在损失函数中，要修改激活函数时就得改掉整个损失函数的代码，不利于维护。

- 重视稳定性和运算精度，使用包含输出层激活函数的类

如果在一个Model中，很长时间我们都不会修改输出层的激活函数，并且模型的稳定运行更为要紧，我们就使用内置了激活函数的类来计算损失函数。同时，就像之前提到的，内置激活函数可以帮助我们推升运算的精度。

因此，选择哪种损失函数的实现方式，最终还要看我们的需求。

有了损失函数，我们终于要开始进行求解了。下一部分我们来讲解神经网络的入门级优化算法：小批量随机梯度下降。