Lesson 11 神经网络的学习

在之前的课程中，我们已经完成了从0建立深层神经网络，并介绍了各类神经网络所使用的损失函数。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。在本节课中，我们将从梯度下降法向外拓展，介绍更常用的优化算法，实现神经网络的学习和迭代。在本节课结束的时候，你将能够完整地实现一个神经网络训练的全流程。

Lesson 11 神经网络的学习

在我们的优化流程之中，我们使用损失函数定义预测值与真实值之间的差异，也就是模型的优劣。当损失函数越小，就说明模型的效果越好，我们要追求的时损失函数最小时所对应的权重向量$\boldsymbol{w}$。对于凸函数而言，导数为0的点就是极小值点，因此在数学中，我们常常先对权重w求导，再令导数为0来求解极值和对应的w。但是对于像神经网络这样的复杂模型，可能会有数百个w的存在，同时如果我们使用的是像交叉熵这样复杂的损失函数（机器学习中还有更多更加复杂的函数），令所有权重的导数为0并一个个求解方程的难度很大、工作量也很大。因此我们转换思路，不追求一步到位，而使用迭代的方式逐渐接近损失函数的最小值。这就是优化算法的具体工作，优化算法的相关知识也都是关于“逐步迭代到损失函数最小值”的具体操作。

在讲解AutoGrad的时候，九天老师给大家详细阐述了梯度下降的细节，因此本篇章会假设你对梯度下降是熟悉的。在这里，你可以问自己以下的问题来进行自查：

1、梯度下降的基本流程是什么，它是怎么找到损失函数的最小值的（写出流程）？

2、梯度下降中是如何迭代权重w的（写出公式）？

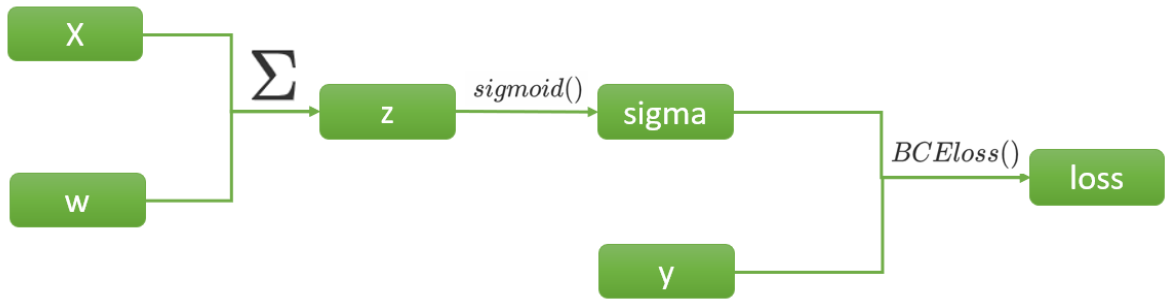
3、什么是梯度？什么是步长？

如果你还不能回答这些问题，那我建议你回到Lesson 6中仔细复习一下梯度下降的细节，因为上面这些问题对于之后使用PyTorch实现神经网络的优化至关重要。

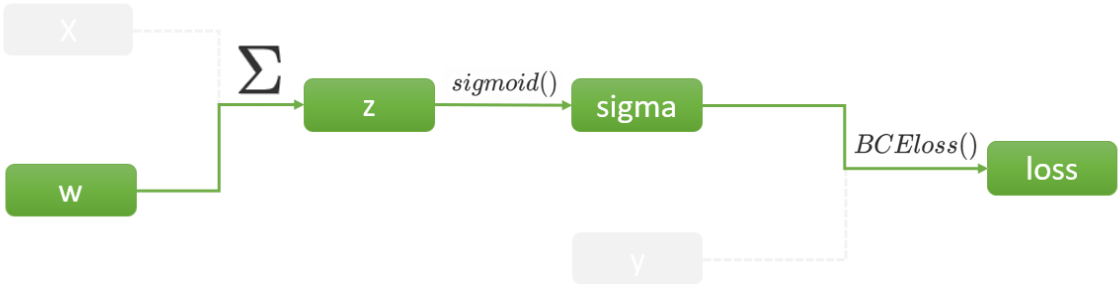
二、找出距离和方向：反向传播

1 反向传播的定义与价值

在梯度下降的最初，我们需要先找出坐标点对应的梯度向量。梯度向量是各个自变量求偏导后的表达式再带入坐标点计算出来的，在这一步骤中，最大的难点在于如何获得梯度向量的表达式——也就是损失函数对各个自变量求偏导后的表达式。在单层神经网络，例如逻辑回归（二分类单层神经网络）中，我们有如下计算：



其中BCEloss是二分类交叉熵损失函数。在这个计算图中，从左向右计算以此的过程就是正向传播，因此进行以此计算后，我们会获得所有节点上的张量的值（z、sigma以及loss）。根据梯度向量的定义，在这个计算过程中我们要求的是损失函数对w的导数，所以求导过程需要涉及到的链路如下：



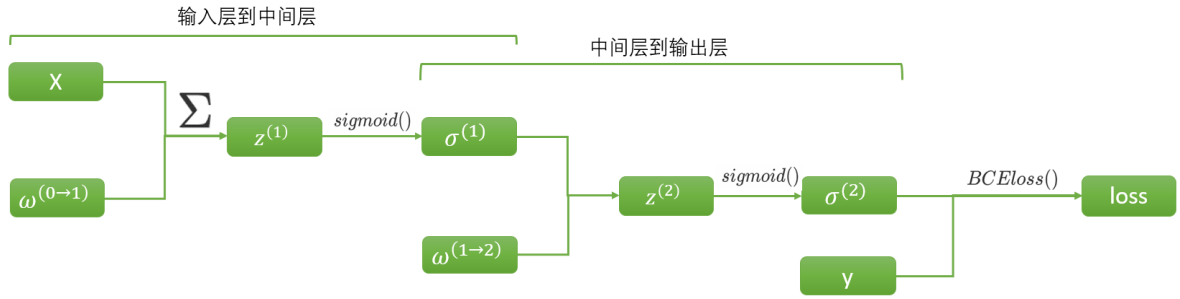
用公式来表示则为在以下式子上求解对w的导数：

$$\frac{\partial Loss}{\partial w}, 其中$$

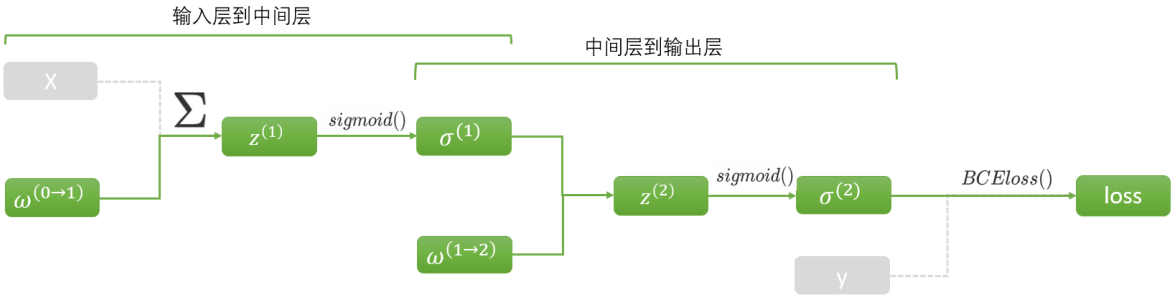
$$\begin{aligned} Loss &= -\sum_{i=1}^{m}(y_i * ln(\sigma_i) \ + \ (1-y_i) * ln(1-\sigma_i)) \\ &= -\sum_{i=1}^{m}(y_i * ln(\frac{1}{1+e^{-\boldsymbol{X_i w}}}) \ + \ (1-y_i) * ln(1-\frac{1}{1+e^{-\boldsymbol{X_i w}}})) \end{aligned}$$

可以看出，已经很复杂了。

更夸张的是，在双层的、各层激活函数都是sigmoid的二分类神经网络上，我们有如下计算流程：



同样的，进行从左到右的正向传播之后，我们会获得所有节点上的张量。其中涉及到的求导链路如下：



用公式来表示，对$w^{(1\to2)}$我们有：

$$\frac{\partial Loss}{\partial w^{(1\to2)}}, 其中$$

$$\begin{aligned} Loss &= -\sum_{i=1}^{m}(y_i * ln(\sigma_i^{(2)}) \ + \ (1-y_i) * ln(1-\sigma_i^{(2)})) \\ &= -\sum_{i=1}^{m}(y_i * ln(\frac{1}{1+e^{-\boldsymbol{\sigma_i^{(1)} w^{(1\to2)}}}}) \ + \ (1-y_i) * ln(1-\frac{1}{1+e^{-\boldsymbol{\sigma_i^{(1)} w^{(1\to2)}}}})) \end{aligned}$$

对$w^{(0\to1)}$我们有：

$$\frac{\partial Loss}{\partial w^{(0\to1)}}, 其中$$

$$\begin{aligned} Loss &= -\sum_{i=1}^{m}(y_i * ln(\sigma_i^{(2)}) \ + \ (1-y_i) * ln(1-\sigma_i^{(2)})) \\ &= -\sum_{i=1}^{m}(y_i * ln(\frac{1}{1+e^{-\frac{1}{1+e^{-X_i w^{(0\to1)}}} w^{(1\to2)}}}) \ + \ (1-y_i) * ln(1-\frac{1}{1+e^{-\frac{1}{1+e^{-X_i w^{(0\to1)}}} w^{(1\to2)}}})) \end{aligned}$$

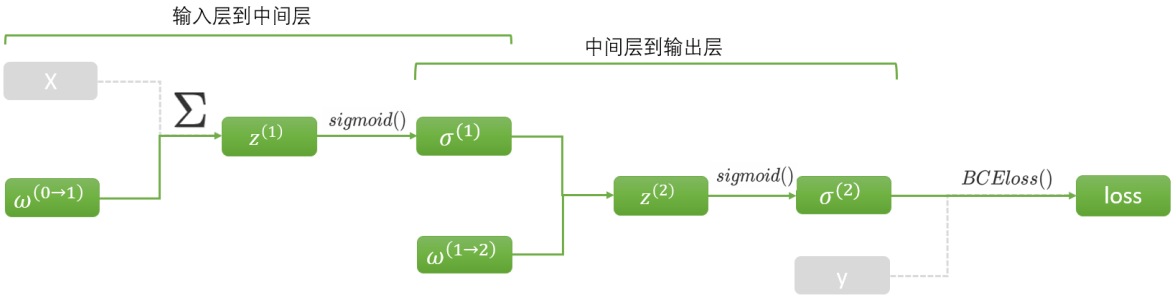
对于需要对这个式子求导，大家感受如何？而这只是一个两层的二分类神经网络，对于复杂神经网络来说，所需要做得求导工作是无法想象的。求导过程的复杂是神经网络历史上的一大难题，这个难题直到1986年才真正被解决。1986年，Rumelhart、Hinton和Williams提出了反向传播算法（Backpropagation algorithm，又叫做Delta法则），利用链式法则成功实现了复杂网络求导过程的简单化。（值得一提的是，多层神经网络解决XOR异或门问题是在1985年被提出的）。接下来，我们就来看看反向传播是怎么解决复杂求导问题的。

在高等数学中，存在着如下规则：

假设有函数$u=h(z)$，$z=f(w)$，且两个函数在各自自变量的定义域上都可导，则有：(1)

$$\frac{\partial u}{\partial w}=\frac{\partial u}{\partial z}*\frac{\partial z}{\partial w}$$

感性（但不严谨）地来说，**当一个函数是由多个函数嵌套而成，最外层函数向最内层自变量求导的值，等于外层函数对外层自变量求导的值 * 内层函数对内层自变量求导的值。**这就是链式法则。当函数之间存在复杂的嵌套关系，并且我们需要从最外层的函数向最内层的自变量求导时，链式法则可以让求导过程变得异常简单。



以双层二分类网络为例，对$w^{(1\to2)}$我们本来需要求解：

$\frac{\partial Loss}{\partial w^{(1\to2)}}$，其中

$$\begin{aligned}Loss&=-\sum_{i=1}^{m}(y_i*ln(\sigma_i^2)\ +\ (1-y_i)*ln(1-\sigma_i^2))\\&=-\sum_{i=1}^{m}(y_i*ln(\frac{1}{1+e^{-\sigma_i^1w^{(1\to2)}}})\ +\ (1-y_i)*ln(1-\frac{1}{1+e^{-\sigma_i^1w^{(1\to2)}}}))\end{aligned}$$

现在，因为Loss是一个内部嵌套了很多函数的函数，我们可以用链式法则将$\frac{\partial Loss}{\partial w^{(1\to2)}}$拆解为如下结构：

$$\frac{\partial Loss}{\partial w^{(1\to2)}}=\frac{\partial L(\sigma)}{\partial\sigma}*\frac{\partial\sigma(z)}{\partial z}*\frac{\partial z(w)}{\partial w}$$

其中，

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

$$=\sum_{i=1}^{m}\frac{\partial(-(y_i*ln(\sigma_i)\ +\ (1-y_i)*ln(1-\sigma_i)))}{\partial\sigma}$$

求导不影响加和符号，因此暂时不看加和符号：

$$\begin{aligned}&=-(y*\frac{1}{\sigma}+(1-y)*\frac{1}{1-\sigma}*(-1))\\&=-(\frac{y}{\sigma}+\frac{y-1}{1-\sigma})\\&=-(\frac{y(1-\sigma)+(y-1)\sigma}{\sigma(1-\sigma)})\\&=-(\frac{y-y\sigma+y\sigma-\sigma}{\sigma(1-\sigma)})\\&=\frac{\sigma-y}{\sigma(1-\sigma)}\end{aligned}$$

假设我们已经进行过以此正向传播，那此时的σ就是$\sigma^{(2)}$，y就是真实标签，我们可以很容易计算出$\frac{\sigma-y}{\sigma(1-\sigma)}$的数值。

再来看剩下的两部分：

$$\begin{aligned}\frac{\partial\sigma(z)}{\partial z}&=\frac{\partial\frac{1}{1+e^{-z}}}{\partial z}\\&=\frac{\partial(1+e^{-z})^{-1}}{\partial z}\\&=-1*(1+e^{-z})^{-2}*e^{-z}*(-1)\\&=\frac{e^{-z}}{(1+e^{-z})^2}\\&=\frac{1+e^{-z}-1}{(1+e^{-z})^2}\\&=\frac{1+e^{-z}}{(1+e^{-z})^2}-\frac{1}{(1+e^{-z})^2}\\&=\frac{1}{(1+e^{-z})}-\frac{1}{(1+e^{-z})^2}\\&=\frac{1}{(1+e^{-z})}(1-\frac{1}{(1+e^{-z})})\\&=\sigma(1-\sigma)\end{aligned}$$

此时的σ还是$\sigma^{(2)}$。接着：

$$\begin{aligned}\frac{\partial z(w)}{\partial w}&=\frac{\partial\boldsymbol{\sigma^{(1)}w}}{\partial w}\\&=\boldsymbol{\sigma^{(1)}}\end{aligned}$$

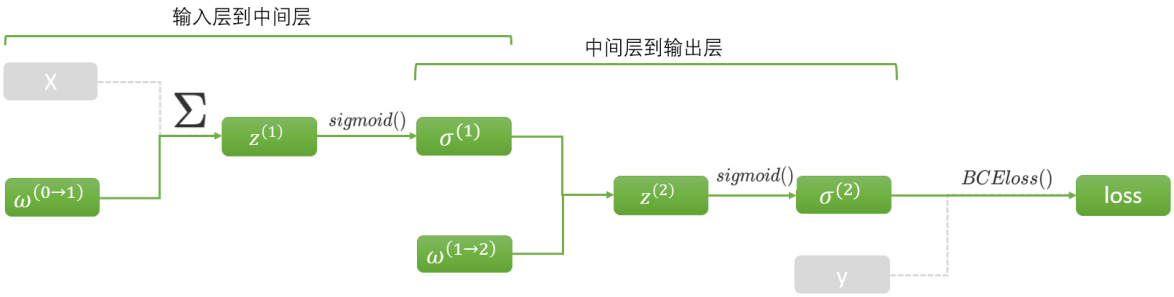
对任意一个特征权杖w而言，$\frac{\partial z(w)}{\partial w}$的值就等于其对应的输入值，所以如果是对于单层逻辑回归而言，这里的求导结果应该是x。不过现在我们是对于双层神经网络的输出层而言，所以这个输入就是从中间层传过来的$\boldsymbol{\sigma^1}$。现在将三个导数公式整合：

$$\begin{aligned}\frac{\partial Loss}{\partial w^{(1\to2)}}&=\frac{\partial L(\sigma)}{\partial\sigma}*\frac{\partial\sigma(z)}{\partial z}*\frac{\partial z(w)}{\partial w}\\&=\frac{\sigma^{(2)}-y}{\sigma^2(1-\sigma^{(2)})}*\sigma^{(2)}(1-\sigma^{(2)})*\sigma^{(1)}\\&=\sigma^{(1)}(\sigma^{(2)}-y)\end{aligned}$$

可以发现，将三个偏导数相乘之后，得到的最终的表达式其实非常简单。并且，其中所需要的数据都是我们在正向传播过程中已经计算出来的节点上的张量。同理，我们也可以得到对$w^{(0\to1)}$的导数。本来我们需要求解：

$\frac{\partial Loss}{\partial w^{(0\to1)}}$，其中

$$\begin{aligned}Loss&=-\sum_{i=1}^{m}(y_i*ln(\sigma_i^{(2)})\ +\ (1-y_i)*ln(1-\sigma_i^{(2)}))\\&=-\sum_{i=1}^{m}(y_i*ln(\frac{1}{1+e^{-\frac{1}{1+e^{-\boldsymbol{X_i}\boldsymbol{w}^{(0\to1)}}}\boldsymbol{w}^{(1\to2)}}})\ +\ (1-y_i)*ln(1-\frac{1}{1+e^{-\frac{1}{1+e^{-\boldsymbol{X_i}\boldsymbol{w}^{(0\to1)}}}\boldsymbol{w}^{(1\to2)}}}))\end{aligned}$$



现在根据链式法则，就有：

$$\frac{\partial Loss}{\partial w^{(0\to1)}} = \frac{\partial L(\sigma)}{\partial \sigma^{(2)}} * \frac{\partial \sigma(z)}{\partial z^{(2)}} * \frac{\partial z(\sigma)}{\partial \sigma^{(1)}} * \frac{\partial \sigma(z)}{\partial z^{(1)}} * \frac{\partial z(w)}{\partial w^{(0\to1)}}$$

其中前两项是在求解$w^{(1\to2)}$时求解过的，而后三项的求解结果都显而易见：

$$= (\sigma^{(2)} - y) * \frac{\partial z(\sigma)}{\partial \sigma^{(1)}} * \frac{\partial \sigma(z)}{\partial z^{(1)}} * \frac{\partial z(w)}{\partial w^{(0\to1)}}$$

$$= (\sigma^{(2)} - y) * w^{1\to2} * (\sigma^{(1)}(1 - \sigma^{(1)})) * X$$

同样，这个表达式现在变得非常简单，并且，这个表达式中所需要的全部张量，都是我们在正向传播中已经计算出来储存好的，或者再模型建立之初就设置好的，因此在计算$w^{(0\to1)}$的导数时，无需再重新计算如$\sigma^{(2)}$这样的张量，这就为神经网络计算导数节省了时间。**你是否注意到，我们是从左向右，从输出向输入，逐渐往前求解导数的表达式，并且我们所使用的节点上的张量，也是从后向前逐渐用到，这和我们正向传播的过程完全相反。**这种从左到右，不断使用正向传播中的元素对梯度向量进行计算的方式，就是反向传播。

2 PyTorch实现反向传播

在梯度下降中，每走一步都需要更新梯度，所以计算量是巨大的。幸运的是，PyTorch可以帮助我们自动计算梯度，我们只需要提取梯度向量的值来进行迭代就可以了。在PyTorch中，我们有两种方式实现梯度计算。一种是使用我们之前已经学过的AutoGrad。在使用AutoGrad时，我们可以使用torch.autograd.grad()函数计算出损失函数上具体某个点/某个变量的导数，当我们需要了解具体某个点的导数值时autograd会非常关键，比如：

```
import torch

x = torch.tensor(1.,requires_grad = True) #requires_grad，表示允许对X进行梯度计算
y = x ** 2

torch.autograd.grad(y, x) #这里返回的是在函数y=x**2上，x=1时的导数值。
```

对于单层神经网络，autograd.grad会非常有效。但深层神经网络就不太适合使用grad函数了。对于深层神经网络，我们需要一次性计算大量权重对应的导数值，并且这些权重是以层为单位阻止成一个个权重的矩阵，要一个个放入autograd来进行计算有点麻烦。所以我们会直接使PyTorch提供的基于autograd的反向传播功能，lossfunction.backward()来进行计算。

注意，在实现反向传播之前，首先要完成模型的正向传播，并且要定义损失函数，因此我们会借助之前的课程中我们完成的三层神经网络的类和数据（500行，20个特征的随机数据）来进行正向传播。

我们来看具体的代码：

```
#导入库、数据、定义神经网络类，完成正向传播
```

```
#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) * 100
y = torch.randint(low=0,high=3,size=(500,1),dtype=torch.float32)

#定义神经网路的架构
"""
注意：这是一个三分类的神经网络，因此我们需要调用的损失函数多分类交叉熵函数CEL
CEL类已经内置了sigmoid功能，因此我们需要修改一下网络架构，删除forward函数中输出层上的sigmoid
函数，并将最终的输出修改为zhat
"""
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 z3

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_)

#前向传播
zhat = net.forward(X)

#定义损失函数
criterion = nn.CrossEntropyLoss()
#对打包好的CorssEnrtopyLoss而言，只需要输入zhat
loss = criterion(zhat,y.reshape(500).long())

loss

net.linear1.weight.grad #不会返回任何值

#反向传播，backward是任意损失函数类都可以调用的方法，对任意损失函数，backward都会求解其中全部
w的梯度
loss.backward()
```

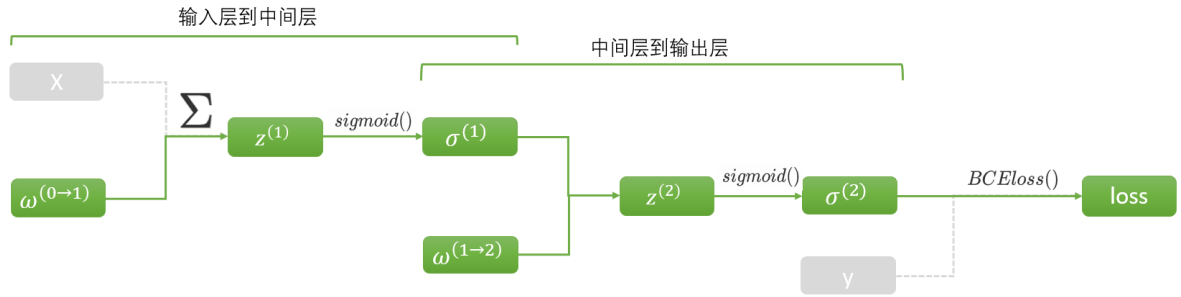
```
net.linear1.weight.grad #返回相应的梯度

#与可以重复进行的正向传播不同，一次正向传播后，反向传播只能进行一次
#如果希望能够重复进行反向传播，可以在进行第一次反向传播的时候加上参数retain_graph
loss.backward(retain_graph=True)

loss.backward()
```

backward求解出的结果的结构与对应的权重矩阵的结构一模一样，因为一个权重就对应了一个偏导数。

这几行代码非常简单，短到几乎不需要去记忆。咋这里，唯一需要说明的点是，在使用autograd的时候，我们强调了requires_grad的用法，但在定义打包好的类以及使用loss.backward()的时候，我们却没有给任何数据定义requires_grad=True。这是因为：

1、当使用nn.Module继承后的类进行正向传播时，我们的权重w是自动生成的，在生成时就被自动设置为允许计算梯度（requires_grad=True），所以不需要我们自己去设置

2、同时，观察我们的反向传播过程：



不难发现，我们的特征张量X与真实标签y都不在反向传播的过程当中，但是X与y其实都是损失函数计算需要用的值，在计算图上，这些值都位于叶子节点上，我们在定义损失函数时，并没有告诉损失函数哪些值是自变量，哪些是常数，那backward函数是怎么判断具体求解哪个对象的梯度的呢？

其实就是靠requires_grad。首先backward值会识别叶子节点，不在叶子上的变量是不会被backward考虑的。对于全部叶子节点来说，只有属性requires_grad=True的节点，才会被计算。在设置X与y时，我们都没有写requires_grad参数，也就是默认让“允许求解梯度”这个选项为False，所以backward在计算的时候就只会计算关于w的部分。

当然，我们也可以将X和y或者任何除了权重以及截距的量的requires_grad打开，一旦我们设置为True，backward就会在帮助我们计算w的导数的同时，也棒我们计算以X或y为自变量的导数。在正常的梯度下降和反向传播过程中，我们是不需要这些导数的，因此我们一律不去管requires_grad的设置，就让它默认为False，以节约计算资源。当然，如果你的w是自己设置的，千万记得一定要设置requires_grad=True。