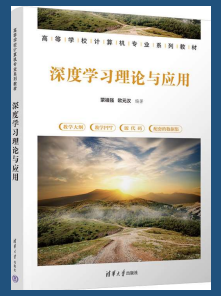
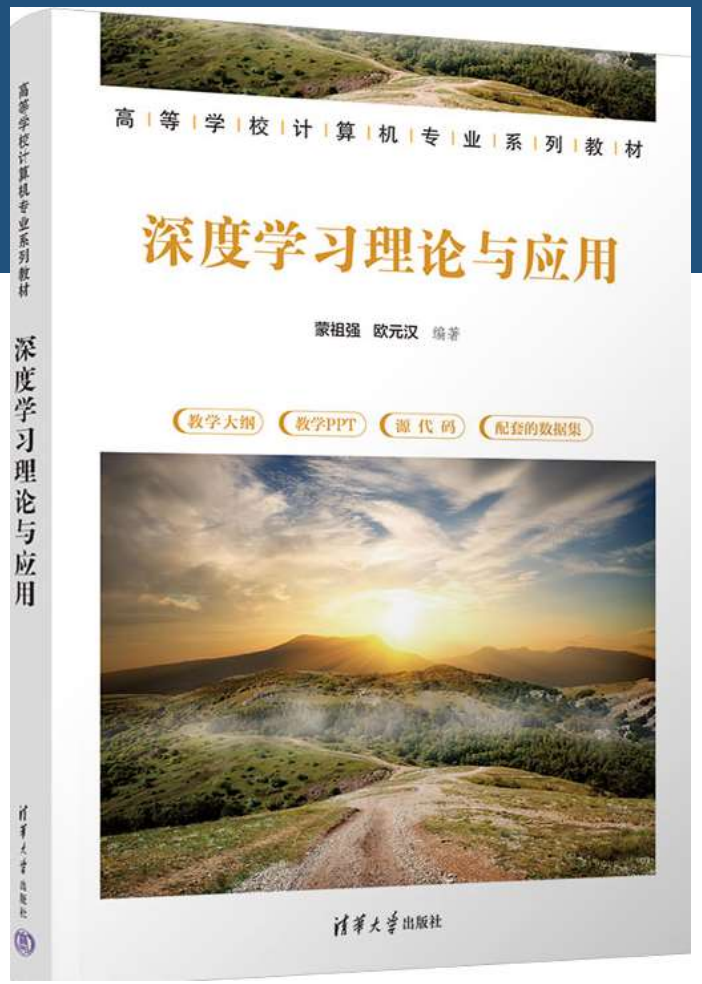


深度学习理论与应用

Deep Learning Theory and Applications

蒙祖强，欧元汉 编著



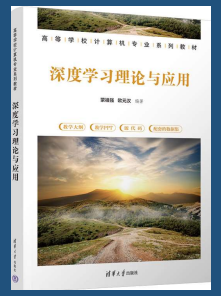
教　材

全国各大书店网店均有销售

- **教学大纲**：提供面向教育工程认证的教学大纲
- **教学PPT**：提供课堂教学用的PPT课件
- **源代码**：提供教材涉及的全部源代码
- **数据集**：提供教材示例、案例用到的全部数据集

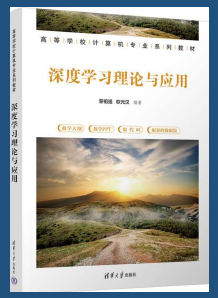
获取教学资源：
http://www.tup.tsinghua.edu.cn/booksCenter/book_09988101.html

教材： 蒙祖强，欧元汉 编著. 深度学习理论与应用. 北京: 清华大学出版社，2023年7月. (书号: 978-7-302-63508-6)



第3章 全连接神经网络

蒙祖强，欧元汉 编著. 深度学习理论与应用. 北京: 清华大学出版社，2023年7月.



本 章 内 容

contents

3.1 构建一个简单的全连接神经网络——解决二分类问题

3.2 全连接神经网络的构造方法

3.3 几种主流的损失函数

3.4 网络模型的训练与测试

3.5 正向计算和反向梯度传播的理论分析

3.1 构建一个简单的全连接神经网络——解决二分类问题

3.1.1 一个简单全连接神经网络的构建和训练

【例 3.1】构建一个全连接神经网络，并用给定的数据集对其进行训练，使其实现相应的二分类任务。

本例设计一个**三层结构**全连接神经网络，其结构如图 3-1 所示。严格说，该网络只包含**两个**网络层，因为第 0 层（即输入层）只是传递数据给后面的网络层，而它本身没有计算功能，只有后面两个网络层有计算功能。但按照习惯，还是称为“三层全连接神经网络”。

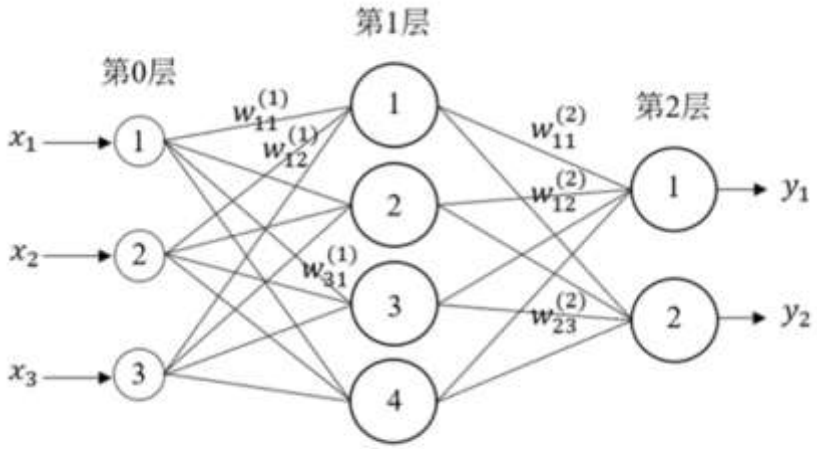


图 3-1　一个三层全连接神经网络

3.1 构建一个简单的全连接神经网络——解决二分类问题

3.1.1 一个简单全连接神经网络的构建和训练

该网络用于对一个用数学方法构造出来的**数据集**进行分类。该数据集保存在./data 目录下（“.”代表本书资源文件所在的根目录，这些资源文件都可以从清华大学出版社网站上免费下载，下同），文件名为“例 3.1 数据.txt”。在该数据集中，0 类样本和 1 类样本都分别有**1020** 条，一共有 **2040** 条数据样本。文件“例 3.1 数据.txt”中的数据格式如下：

```
1.9308,  -2.5692,  -3.5692,  0
2.6044,  -1.8956,  -2.8956,  0      } 1020条0类样本
   ……
-6.1000, -1.6000,  -0.6000,  1
-6.0949, -1.5949,  -0.5949,  1      } 1020条1类样本
   ……
```

其中，每一行表示一条数据样本，前面三个数字分别表示三维空间中的三个坐标值，它 们共同表示三维空间中的一个点；最后面的数字为 0 或 1，表示类别索引，即 0 类或 1 类。

3.1 构建一个简单的全连接神经网络——解决二分类问题

3.1.1 一个简单全连接神经网络的构建和训练

核心代码：（全部代码见教材P55）

```
class Model3_1(nn.Module):
    def __init__(self ):
        super(Model3_1, self).__init__()
        self.fc1 = nn.Linear(3, 4)      #用于构建神经网络的第 1 层（隐含层），
                                        #其中包含 4 个神经元
        self.fc2 = nn.Linear(4, 2)      #用于构建神经网络的第 2 层（输出层），
                                        #其中包含 2 个神经元，因为有 2 个类别
    def forward(self,x): #在该方法中实现网络的逻辑结构
        out = self.fc1(x)
        out = torch.tanh(out) #该激活函数可用可不用
        out = self.fc2(out)
        #此处不宜用激活函数 sigmoid，因为下面的损失函数会用到
        return out
```

3.1 构建一个简单的全连接神经网络——解决二分类问题

3.1.2 程序代码解释及网络层的构建方法

（1）**读取文件**"例 3.1 数据.txt"中的数据，并组装特征值张量 X 和标记张量 Y，它们的形状分别为 torch.Size([2040, 3])和 torch.Size([2040])。

先将数据集中的四列数据分别**保存到列表** X1,X2,X3,Y 中，然后将它们分别转换为四个一维张量，形状均为 torch.Size([2040])，

最后将这四个“竖着放”的一维张量沿水平方向**“靠拢”**在一起，形成张量 X，其形状为 torch.Size([2040, 3])。

“靠拢”操作用下面语句实现：

```
X = torch.stack((X1, X2, X3), dim=1)
```

该语句**等价**于下列语句：

```
X = torch.cat((X1.view(-1,1),X2.view(-1,1),X3.view(-1,1)),dim=1)
```

这样，整个数据集就转变为**张量** X 和 Y 了，以为送入网络模型做准备。

3.1 构建一个简单的全连接神经网络——解决二分类问题

3.1.2 程序代码解释及网络层的构建方法

（2）**定义类 Model3_1**。该类的实例即为我们要构建的神经网络模型。

在其__init__(self)

方法中用函数 **nn.Linear()**来定义了两个全连接神经网络层：

self.fc1 = nn.Linear(3, 4)

self.fc2 = nn.Linear(4, 2)

其中， nn.Linear(3, 4)**等效于**：

nn.Linear(in_features=3, out_features=4, bias=True)

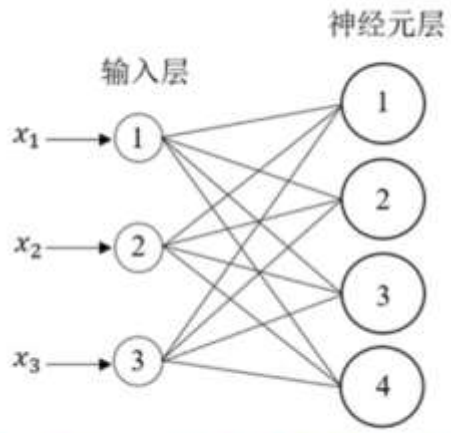


图 3-2 nn.Linear(3, 4)构造的神经网络层

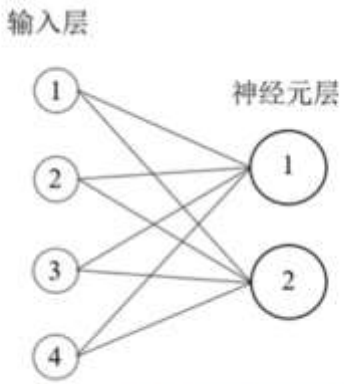


图 3-3 nn.Linear(4, 2)构造的神经网络层

3.1 构建一个简单的全连接神经网络——解决二分类问题

3.1.2 程序代码解释及网络层的构建方法

（3）网络的**训练**。神经网络的训练通过两层循环来进行。其中，每执行一次内层循环，就遍历一次数据集，而对数据集的每一次遍历通常称为**一代**或者**一轮**（epoch）。

在训练过程中，使用了下面一条语句：

```
x = x.unsqueeze(0)
```

其作用是将 x 的形状由 torch.Size([3])改为 torch.Size([1, 3])。主要原因在于，PyTorch 程序默认支持批量梯度下降算法。对于输入的张量，它的第一个维的大小一般表示批量中样本的数量。

3.1 构建一个简单的全连接神经网络——解决二分类问题

3.1.2 程序代码解释及网络层的构建方法

（4）多分类问题常使用的损失函数—nn.CrossEntropyLoss()。

训练过程中还运用到一个新的损失函数——**nn.CrossEntropyLoss()**。该函数是一种交叉熵损失函数，它经常在多分类问题用作损失函数。

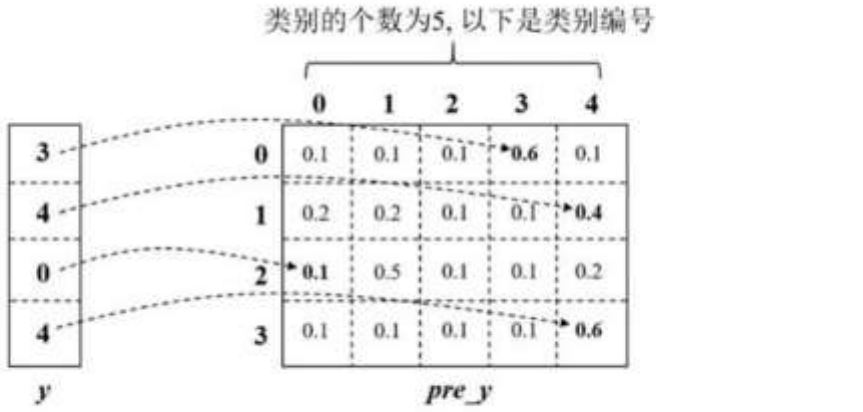


图 3-4 损失函数 nn.CrossEntropyLoss()(pre_y, y)中参数 *pre_y* 和 *y* 之间的关系

令 y_k表示 y 中第 k 个分量的值（整型），则 pre_y 和 y_k共同表示矩阵 pre_y 中第 k 行上 第 y_k列中的元素。这种对应关系可用图 3-4 来表示。

注：行和列分别都是从第 0 行和第 0 列开始编号， 本书均采用这种编号

3.1 构建一个简单的全连接神经网络——解决二分类问题

3.1.2 程序代码解释及网络层的构建方法

（4）显示训练过程中**损失**函数值的**变化趋势**。每处理 50 个样本，我们对损失函数值进行一次采样保存，最后绘制在一个坐标系，结果如图 3-5 所示。

从图中可以看出，损失函数值虽然在局部上有波动，但在总体上呈现迅速降低的趋势，以至于后面降低到 0。这说明构建的全连接神经网络对给定的数据集是有效的，网络模型是**收敛**的，在经过充分训练后可以实现对给定数据的分类。

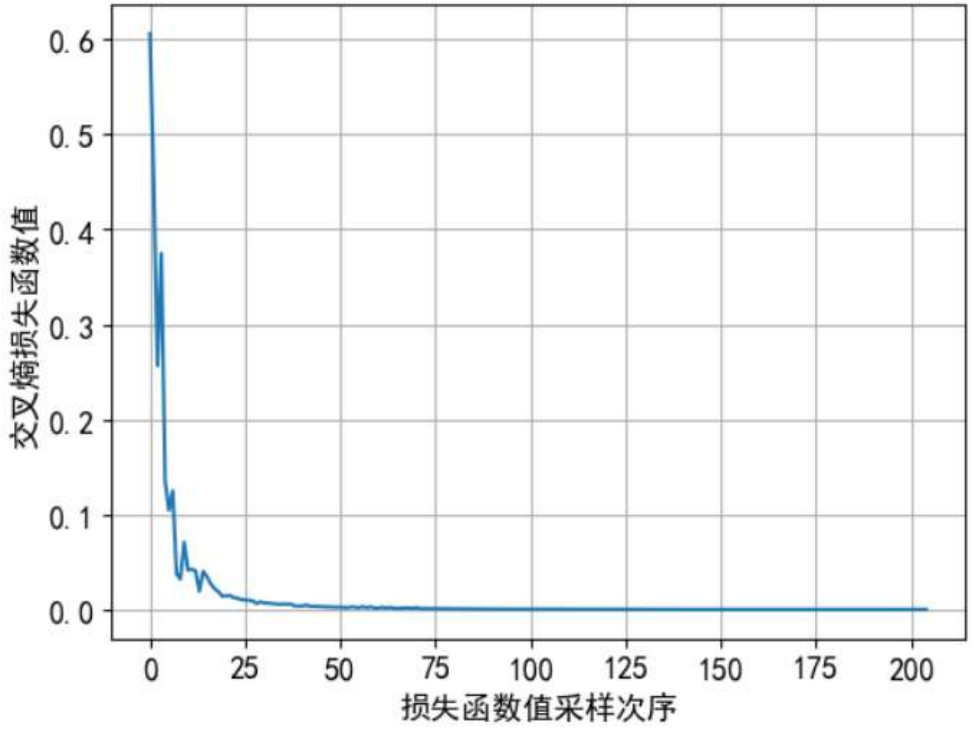
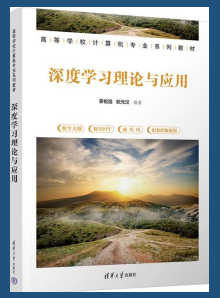


图 3-5 例 3.1 中程序损失函数值的变化趋势



本 章 内 容

contents

3.1 构建一个简单的全连接神经网络——解决二分类问题

3.2 全连接神经网络的构造方法

3.3 几种主流的损失函数

3.4 网络模型的训练与测试

3.5 正向计算和反向梯度传播的理论分析

3.2 全连接神经网络的构造方法

3.2.1 网络层的定义

一个**全连接网络层**（简称**网络层**）是由 **nn.Linear()**函数来定义的，该函数的调用格式可简化说明如下：

```
nn.Linear(in_features=m, out_features=k, bias=True/False)
```

其中：

--in_features=m：表示有 m 个**输入节点**，能够接收特征个数为 m 的特征向量的输入；

--out_features= k：表示有 k 个神经元，因而也有 k 个**输出节点**（每个神经元仅有一个输出）；

--bias=True/False：当选择 bias=True 时，表示为每个神经设置一个**偏置项**（默认设置）；当选择 bias=False 时，表示所有神经元都没有偏置项。

3.2 全连接神经网络的构造方法

3.2.1 网络层的定义

该函数用于创建这样的一个网络层：该网络层一共有 **m×k+k** 个参数（如果**bias=True**）或 **m×k** 个参数（**bias=False**）需要学习；

如果用函数 nn.Linear()的参数来表示整个网络层的参数个数，那么上述两种情况的神经元个数分别为

in_features×out_features+out_features 和 in_features×out_features。

该函数构建的**网络层的结构**如图 3-6 所示。

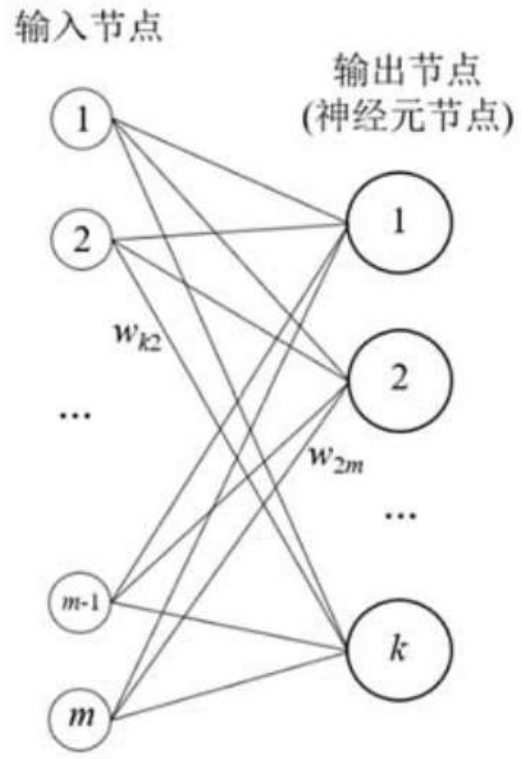


图 3-6 nn.Linear(*m*, *k*)定义的网络层

3.2 全连接神经网络的构造方法

3.2.1 网络层的定义

例如，下列代码定义了一个 50×30 的全连接神经网络层：

```
fc_layer = nn.Linear(30, 50, bias=True)
```

其中，该网络层含有 **30 个输入**节点和 **50 个输出**节点（神经元节点），同时包含 **50 个偏置项**。

右侧代码定义的**张量 x1,x2,x3** 都能被上述定义的网络层接收，而**张量 x4** 则不能被接收：

```
y4 = fc_layer(x4) #执行时产生错误
```

```
x1 = torch.randn(32,30)
y1 = fc_layer(x1) # torch.Size([32, 30]) --->
torch.Size([32, 50])
x2 = torch.randn(32,5,30)
y2 = fc_layer(x2) # torch.Size([32, 5, 30]) --->
torch.Size([32, 5, 50])
x3 = torch.randn(1,2,3,30)
y3 = fc_layer(x3) # torch.Size([1, 2, 3, 30]) --->
torch.Size([1, 2, 3, 50])
x4 = torch.randn(32,5,15) #不能被网络层 fc_layer
接收
y4 = fc_layer(x4) #执行时产生错误
```

3.2 全连接神经网络的构造方法

3.2.2 网络结构的实现

【例 3.2】利用 **nn.Linear()**函数，构建如图 3-7 所示的全连接神经网络。

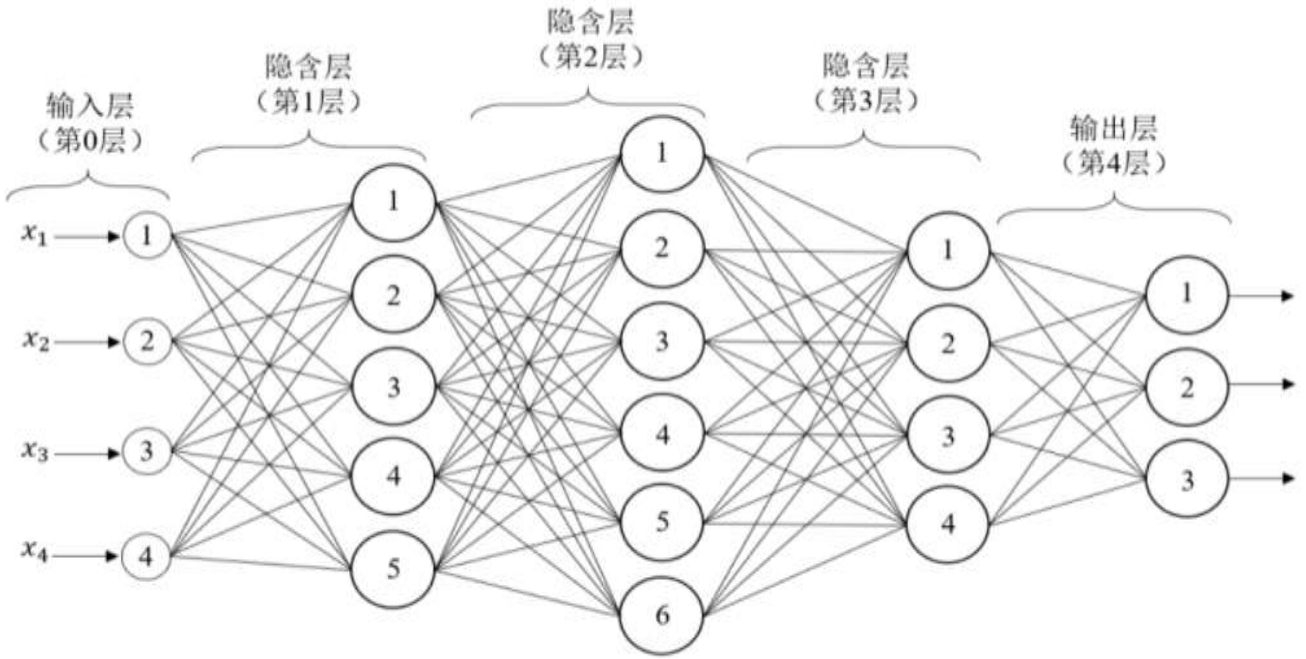


图 3-7 一个全连接神经网络

3.2 全连接神经网络的构造方法

3.2.2 网络结构的实现

显然，该网络是由 4 个**全连接网络层**组成，可通过右边的代码一来定义这些网络层。

代码一种四个语句分别用于构建图 3-7 中的第 1 层、第 2 层、第 3 层和第 4 层网络。但这些网络层是**独立**的，需要使用类似右边的代码二将它们“**连接**”起来（一般在 **forward()**方法中实现），形成逻辑上的网络：

代码一：

```
self.fc1 = nn.Linear(4, 5)
self.fc2 = nn.Linear(5, 6)
self.fc3 = nn.Linear(6, 4)
self.fc4 = nn.Linear(4, 3)
```

代码二：

```
out = self.fc1(x)
out = self.fc2(out)
out = self.fc3(out)
out = self.fc4(out)
```

3.2 全连接神经网络的构造方法

3.2.2 网络结构的实现

但上述代码并未给相应神经元启用**激活函数**。如果希望设置 **tanh()函数**为第 1 层神经元的激活函数，**rule()函数**为第 2 层神经元的激活函数，**sigmoid()函数**为第 3 层神经元的激活函数，则可以用下列代码来实现：

```
out = self.fc1(x)
out = torch.tanh(out) #以 tanh()函数为激活函数
out = self.fc2(out)
out = torch.rule(out) #以 rule()函数为激活函数
out = self.fc3(out)
out = torch.sigmoid(out) #以 sigmoid()函数为激活函数
out = self.fc4(out)
```

3.2 全连接神经网络的构造方法

3.2.2 网络结构的实现

```
class AFullNet(nn.Module):
 def __init__(self): #在该函数中定义网络层
 super().__init__()
#下面创建 4 个全连接网络层:
 self.fc1 = nn.Linear(4, 5) #如果不设置偏置项,则添加 bias=False 即可,下同
 self.fc2 = nn.Linear(5, 6)
 self.fc3 = nn.Linear(6, 4)
 self.fc4 = nn.Linear(4, 3)
 def forward(self, x): #在该方法中将各个网络层连接起来,构成一个完整的网络
 out = sef.fc1(x)
 out = self.fc2(out)
 out = self.fc3(out)
l out = self.fc4(out)
 return out
```

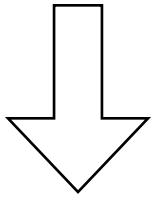
该程序的核心代码（全部代码见教材P61）

3.2 全连接神经网络的构造方法

3.2.2 网络结构的实现

按照前面介绍的参数量计算方法，该网络的参数量为：
(4*5+5)+ (5*6+6) + (6*4+4) +(4*3+3)=104，
这与运行结果是一致的。
这从一个侧面反映我们创建的全连接网络是正确的。

```
print('该网络参数总量：%d'%sum)
```

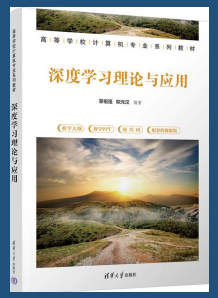


```
该网络参数总量：104
```

3.2 全连接神经网络的构造方法

3.2.3 从网络结构判断网络的功能

对于给定的一个**全连接神经网络**，不管它的结构是简单的还是复杂的，从其输入节点和输出节点的数量就可以大致判断其基本特点和功能。**输入节点**的数量越大，表示其能够处理的样本的**特征维度**越大，意味着能够处理越复杂的问题。在这种情况，其**隐含层节点**的数量一般也比较多，整个网络的**参数量**也应该**比较大**，需要的训练数据也比较多。**相反**，如果输入节点的数量比较少，那么该网络的功能一般比较弱，处理的问题相对简单。从网络**输出层节**点的数量看，我们大致能够判断该网络是用于**回归**还是**分类**。一般情况下，仅有一个输出节点的网络多用于预测，属于**回归分析**，但也有的通过逻辑回归用于解决二分类问题。有两个或两个以上输出节点的网络一般用于解决多分类问题。一般地，对于 **c 分类问题**，相应网络有 c 个输出节点：每个节点输出一个数值，在进行 **softmax 归一化**以后 得到长度为 c 的概率分布，其中最大概率值所对应的类即为预测的类。



本 章 内 容

contents

3.1 构建一个简单的全连接神经网络——解决二分类问题

3.2 全连接神经网络的构造方法

3.3 几种主流的损失函数

3.4 网络模型的训练与测试

3.5 正向计算和反向梯度传播的理论分析

3.3 几种主流的损失函数

3.3.1 nn.CrossEntropyLoss()和 nn.NLLLoss()函数

在**有监督学习**中，样本标记构成了一个固定的**数据分布**，而模型的输出又是另外一个跟模型参数相关的数据分布。显然，我们需要一个能够衡量这两种分布**相似程度的函数**，而**交叉熵**正是用来衡量两个分布之间相似性的一种度量函数。

假设 $p_1, p_2, \ldots, p_m$ 和 $q_1, q, \ldots, q_m$ 为两个分布，则二者的交叉熵可表示为：

$$-\sum_{i=1}^{m} p_i \log(q_i)$$

该交叉熵的值越小，则表示两个分布越接近。我们的目标是，通过不断更新模型参数，使得模型输出的数据分布不断接近样本标记构成的数据分布。

3.3 几种主流的损失函数

3.3.1 nn.CrossEntropyLoss()和 nn.NLLLoss()函数

在运用上述公式之前，一般p_i和q_i都应先做 **softmax 归一化**，即对两个分布进行概率归一化。就 c 分类问题（即有 c 个类别的分类问题）而言，令 x 表示样本的特征向量，y 为类别索引（整数），不妨假设 y = k。由于样本 x 只能属于一个类别，所以 y 的分布式表示为：

$$(l_1, l_2, \ldots, l_c) = (0, 0, \ldots, 0, 1, 0, \ldots, 0)$$

其中，第 k 个元素为 1，其他元素为 0。可以看到，该分布已经概率归一化了。假设输入样本 x 后，模型在这 c 个类别上的输出分别为 $v_1, v_2, \ldots, v_c$，即模型输出$\hat{\boldsymbol{y}}$的分

布式表示为：

$$(v_1, v_2, \ldots, v_c)$$

3.3 几种主流的损失函数

3.3.1 nn.CrossEntropyLoss()和 nn.NLLLoss()函数

不妨将$\hat{y}$表示为：$\hat{y} = (v_1, v_2, \ldots, v_c)$。于是，对该分布的 **softmax 归一化**公式为：

$$\frac{\exp(v_i)}{\sum_{j=1}^{c}\exp(v_j)}, i = 1,2,\ldots,c$$

显然，归一化后，该分布转变为一种概率分布，即该分布中每个分量值均为[0, 1]中的实数，它们之和为 1。这样，我们可以用上述**交叉熵**公式来表示模型关于标记 y 和输出$\hat{y}$的**损失函数：**

$$\begin{aligned}\mathcal{L}(\hat{y}, y) &= -\sum_{i=1}^{c} l_i \log\left(\frac{\exp(v_i)}{\sum_{j=1}^{c}\exp(v_j)}\right)\\ &= -\log\left(\frac{\exp(v_k)}{\sum_{j=1}^{c}\exp(v_j)}\right)\\ &= -\log(\exp(v_k)) + \log\left(\sum_{j=1}^{c}\exp(v_j)\right)\\ &= -v_k + \log\left(\sum_{j=1}^{c}\exp(v_j)\right)\end{aligned}$$

$\mathcal{L}(\hat{y}, y)$ 就是针对单条样本 x 及其标记 y 的交叉熵损失函数，其中 $\hat{y}$ 为模型的输出。

3.3 几种主流的损失函数

3.3.1 nn.CrossEntropyLoss()和 nn.NLLLoss()函数

按照这种思路，我们也可以构造针对**多条样本**的**交叉熵损失**函数。
假设一个数据批量 X 由 n 条样本构成，Y 是与其对应的标记向量，并假设 X = $(x_1, x_2, \ldots, x_n)$，Y = $(y_1, y_2, \ldots, y_n)$，其中 $y_1, y_2, \ldots, y_n$取值在[0, c-1]之间的整数，c 为类别的个数，m 为特征的个数。
假设在 X 输入模型后得到的输出是$\hat{Y}$，易知$\hat{Y}$ 是一个 n×c 的矩阵，表示如下：

$$\hat{\boldsymbol{Y}} = \begin{bmatrix} v_1^{(1)}, v_2^{(1)}, \ldots, v_c^{(1)} \\ v_1^{(2)}, v_2^{(2)}, \ldots, v_c^{(2)} \\ \ldots\ \ldots \\ v_1^{(n)}, v_2^{(n)}, \ldots, v_c^{(n)} \end{bmatrix}$$

3.3 几种主流的损失函数

3.3.1 nn.CrossEntropyLoss()和 nn.NLLLoss()函数

然后，取出$\boldsymbol{Y}$在第 1 至 n 行中分别以 $y_1, y_2, \ldots, y_n$为下标的元素，接着按照上述方法分别计算它们的**交叉熵损失**函数的**值**，最后以这些损失函数值的**平均值**作为这个数据**批量 X** 的交叉熵损失函数值：：

$$\mathcal{L}(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) = \frac{1}{n}\left(-\sum_{i=1}^{n} v_{y_i}^{(i)} + \sum_{i=1}^{n} \log\left(\sum_{j=1}^{c} \exp\left(v_j^{(i)}\right)\right)\right)$$

$\mathcal{L}(\hat{\mathbf{Y}}, \mathbf{Y})$ 就是数据批量 X 的交叉熵损失函数，其中 $\hat{Y}$ 为模型的输出。

3.3 几种主流的损失函数

3.3.1 nn.CrossEntropyLoss()和 nn.NLLLoss()函数

在 **PyTorch** 中，$\boldsymbol{Y}$可理解为二维实值**张量**，其中第一维的大小为 n，第二维的大小为 c；Y 理解为一维整型张量，其维的大小为 n。于是，我们可用 **nn.CrossEntropyLoss()**函数来计算交叉熵损失函数$\mathcal{L}(\hat{Y}, Y)$的值。

例如，给定如下两个张量 pre_y 和 y（分别对应上述公式中的$\hat{\boldsymbol{Y}}$和$\boldsymbol{Y}$）：

```
pre_y = [[5, 9, 5, 5],
         [9, 8, 7, 9],
         [7, 5, 6, 5]]
y = [0, 1, 2]                    #分别指向第 1 行中的 5、第 2 行中的 8 和第 3 行中的 6
pre_y = torch.Tensor(pre_y)      #张量化
y = torch.LongTensor(y)
```

3.3 几种主流的损失函数

3.3.1 nn.CrossEntropyLoss()和 nn.NLLLoss()函数

然后，执行下列语句：

```
loss = nn.CrossEntropyLoss()(pre_y, y)
```

结果 loss 的值为 **2.4883**。
如果按照上述$\mathcal{L}(\hat{Y},Y)$的公式，则得到如下的计算表达式：

$$\frac{1}{3}[-(5+8+6) + \log(e^5+e^9+e^5+e^5) + \log(e^9+e^8+e^7+e^9) + \log(e^7+e^5+e^6+e^5)]$$

计算结果为 **2.4883**，这与 nn.CrossEntropyLoss()计算的结果是一样的。这说明，$\mathcal{L}(\hat{Y},Y)$和 nn.CrossEntropyLoss()(pre_y, y)确实是表达相同的含义。

3.3 几种主流的损失函数

3.3.1 nn.CrossEntropyLoss()和 nn.NLLLoss()函数

从上述公式推导中也可以看出，**nn.CrossEntropyLoss(pre_y, y)**的计算过程大致分为三步：

（1）按水平方向计算矩阵 pre_y 中每一行上数值的概率分布，即按行进行 **softmax 归一化**，可用 **torch.softmax()**函数实现。
例如，对于上述张量 pre_y，对其 softmax 归一化的代码如下：

```
pre_y2 = torch.softmax(pre_y, dim=1)
```

（2）对矩阵 pre_y 中每个元素计算它们的自然对数，可用 torch.log()函数实现。
例如，对当前张量 pre_y2 运用 torch.log()函数：

```
pre_y3 = torch.log(pre_y2)
```

3.3 几种主流的损失函数

3.3.1 nn.CrossEntropyLoss()和 nn.NLLLoss()函数

（3）抽取当前矩阵 pre_y 中由 y 指定的那些元素，计算它们的平均值，然后**取反**即为**nn.CrossEntropyLoss()**的计算结果，这可由**nn.NLLLoss()函数**实现。

例如，由于 y = [0, 1, 2]，所以 y 分别指向第 1 行中的-4.0535、第 2 行中的-1.9176 和第 3 行中的-1.4938。这些元素的 平均值为(-4.0535-1.9176-1.4938)/3 = **-2.4883**，对该结果取反后得到 2.4883。

如果用 nn.NLLLoss()函数计算，相应代码如下：

```
loss = nn.NLLLoss()(pre_y3, y)
```

运行上述代码，结果可以发现 loss 的值为 **2.4883**，这与上面手工计算结果是一致的。

也就是说，nn.CrossEntropyLoss()(pre_y, y)语句等同于下列三条语句：

```
pre_y2 = torch.softmax(pre_y, dim=1)
pre_y3 = torch.log(pre_y2)
loss = nn.NLLLoss()(pre_y3, y)
```

nn.NLLLoss()也是一种损失函数，称为负对数似然损失函数(Negative Log Likelihood Loss Function)。

3.3 几种主流的损失函数

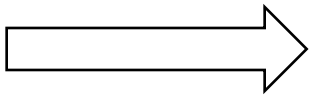
3.3.1 nn.CrossEntropyLoss()和 nn.NLLLoss()函数

对于下列调用格式:

```
loss = nn.NLLLoss()(pre_y, y)
```

其作用是，按 y 给定的下标值，取出 pre_y 中相应元素进行相加，然后除以元素个数（即求平均值），最后**取反**。例如，对于下列代码:

```
pre_y = torch.tensor([[4, 3, 4, 0],[0, 3, 3, 3],
 [4, 3, 3, 2]]).float()
y = [2, 1, 3]
y = torch.LongTensor(y)
loss = nn.NLLLoss()(pre_y, y)
```



-3.0

按照上述 **nn.NLLLoss()函数**的功能介绍，nn.NLLLoss()(pre_y, y)的值应该为-(4+3+2)/3 =-3.0，这与代码执行的结果是一致的。

3.3 几种主流的损失函数

3.3.1 nn.CrossEntropyLoss()和 nn.NLLLoss()函数

一般来说，**nn.NLLLoss()函数**甚少单独使用，往往跟**torch.softmax()函数**和**torch.log()函数**结合使用。
实际上，通常先用 torch.softmax()函数，然后再用 torch.log()函数，最后用nn.NLLLoss()函数，其效果相当于使用 nn.CrossEntropyLoss() 函数。

简而言之，**nn.CrossEntropyLoss() = torch.softmax()+torch.log()+nn.NLLLoss()**。显然，nn.CrossEntropyLoss()和 nn.NLLLoss()主要用于**解决分类问题**。

3.3 几种主流的损失函数

3.3.2 nn.MSELoss()函数

假设张量 Y= $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$,$\hat{Y} = (\hat{y}^{(1)}, \hat{y}^{(2)}, \ldots, \hat{y}^{(n)})$则 Y 和$\hat{Y}$的**均方差**可表示为:

$$\mathcal{L}(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2$$

这里假设了 Y 和$\hat{Y}$为一维张量。实际上，对于多维张量的情况，亦可依此类推:分别取 出 Y 和$\hat{Y}$中相对应的元素相减，然后平方，最后除以元素个数。 Y 和$\hat{Y}$的**形状必须相同**。

在 **PyTorch** 框架中，可用 nn.MSELoss()函数来计算 Y 和$\hat{Y}$的均方差。MSELoss 是 Mean Squared Error Loss 的缩写，对应的中文意思是平均平方误差，简称均方差。因此，nn.MSELoss() 函数称为均方差损失函数。显然，该函数用于度量两个张量的误差。

3.3 几种主流的损失函数

3.3.2 nn.MSELoss()函数

例如，下列代码先构建两个形状相同的张量 pre_y 和 y，然后利用**nn.MSELoss()函数**计算它们的**均方差**：

```
pre_y = torch.tensor([[4, 1, 0, 0],
 [4, 2, 0, 5],
 [5, 4, 5, 1]]).float()
y = torch.tensor([[0, 1, 4, 3],
 [0, 3, 1, 4],
 [4, 4, 1, 3]]).float()
loss = nn.MSELoss()(pre_y, y) #计算均方差
```

⟹ 6.75

均方差损失函数 **nn.MSELoss()**主要用于**回归分析**。

3.3 几种主流的损失函数

3.3.3 nn.BCELoss()和 nn.BCEWithLogitsLoss()函数

令 X 表示由 n 个样本构成为数据批量（张量），其对应的标记张量为 **Y**=$y^{(1)}, y^{(2)}, \ldots, y^{(n)}$其中 $y^{(i)} \in \{0, 1\}$（0 表示一个类别，1 表示另一个类别），i = 1,2,…,n，并假设在输入 X 后模型的输出为,$\hat{\boldsymbol{Y}} = \left(\hat{y}^{(1)}, \hat{y}^{(2)}, \ldots, \hat{y}^{(n)}\right)$其中$y^{(i)} \in (0,1)$, i = 1,2,…,n。这样，模型在批量 X 上的**损失函数**为：

$$\mathcal{L}(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) = -\frac{1}{n}\sum_{i=1}^{n}[y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})]$$

3.3 几种主流的损失函数

3.3.3 nn.BCELoss()和 nn.BCEWithLogitsLoss()函数

在 **PyTorch** 框架中，**nn.BCELoss()函数**来计算上述$\mathcal{L}(\hat{Y},Y)$。BCELoss 是 Binary CrossEntropyLoss 的缩写，也就是说，nn.BCELoss()也是一种交叉熵损失函数，但它只适用于**二分类**问题，因而称为二分类交叉熵损失函数。

$$\mathcal{L}(\hat{\boldsymbol{Y}},\boldsymbol{Y}) = -\frac{1}{n}\sum_{i=1}^{n}[y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})]$$

3.3 几种主流的损失函数

3.3.3 nn.BCELoss()和 nn.BCEWithLogitsLoss()函数

例如，下列代码先构造张量 pre_y 和 y（二者分别相当于上述公式中的$\hat{\boldsymbol{Y}}$和$\boldsymbol{Y}$），然后将pre_y **归一化**到(0, 1)中，最后调用 **nn.BCELoss()**计算交叉熵损失函数的值：

```
pre_y = torch.tensor([[-0.3696], [-0.2404],[-1.1969],
[ 0.2093]])
y = torch.tensor([[0],[1],[1],[0]]).float()
pre_y = torch.sigmoid(pre_y) #将 pre_y 归一化到(0, 1)中
loss = nn.BCELoss()(pre_y,y)
```

⇒ 0.9025

注意：运用激活函数 torch.sigmoid()的目的是，将 pre_y 归一化到(0, 1)中，否则可能因为 pre_y 为 0 而导致计算 pre_y 的对数时出现错误。

3.3 几种主流的损失函数

3.3.3 nn.BCELoss()和 nn.BCEWithLogitsLoss()函数

实际上，**nn.BCEWithLogitsLoss()函数**相当于先启用激活函数**torch.sigmoid()**，然后再调用**nn.BCELoss()**。也就是说，如果使用nn.BCEWithLogitsLoss()函数来计算交叉熵，则不需要再使用激活函数 torch.sigmoid()。

简而言之，nn.BCEWithLogitsLoss() = torch.sigmoid() +nn.BCELoss()。

nn.BCEWithLogitsLoss()函数是通过**逻辑回归**的方法来解决二分类问题。

3.3 几种主流的损失函数

3.3.4 nn.L1Loss()函数

有时候可能以模型输出张量和标记张量中各对应元素之差的绝对值的平均值作为刻画输出张量和标记张量之间的误差。为此，假设输出张量$\hat{\boldsymbol{Y}} = (\hat{y}^{(1)}, \hat{y}^{(2)}, \ldots, \hat{y}^{(n)})$标记张量为Y= $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$则这种误差可表示为：

$$\mathcal{L}(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) = \frac{1}{n}\sum_{i=1}^{n} |\hat{y}^{(i)} - y^{(i)}|$$

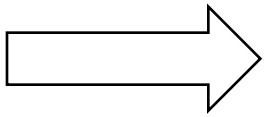
$\mathcal{L}(\hat{Y}, Y)$称为绝对值误差。

3.3 几种主流的损失函数

3.3.4 nn.L1Loss()函数

在 **PyTorch** 框架中，可用 **nn.L1Loss()函数**来实现此误差的计算。例如，下面代码先构造张量 pre_y 和 y，然后调用 nn.L1Loss()函数来计算它们之间的绝对值误差：

```
pre_y = torch.tensor([[-3, 4, -3, -5],[-5, -3, 1, 2],
 [ 4, -1, -4, -4]]).float()
y = torch.tensor([[ 1, -4, -3, 4],
 [-1, -4, -2, -5],
 [-5, 1, 0, 2]]).float()
loss = nn.L1Loss()(pre_y,y)
```

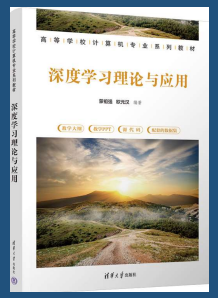


4.75

如果不想计算绝对值的平均值，而只求绝对值之和，则可用下列的 nn.L1Loss()函数来 实现：

```
loss = nn.L1Loss(reduction='sum')(pre_y,y) #默认设置为 reduction='mean’
```

其他损失函数也有类似的参数设置，请读者自行测试。 nn.L1Loss()函数也主要用于**回归分析**。



本 章 内 容

contents

3.1 构建一个简单的全连接神经网络——解决二分类问题

3.2 全连接神经网络的构造方法

3.3 几种主流的损失函数

3.4 网络模型的训练与测试

3.5 正向计算和反向梯度传播的理论分析

3.4 网络模型的训练与测试

3.4.1 数据集分割

为对构建的模型进行有效的**训练**和**评估**，一般需要将**数据集**分割为**训练集**和**测试集**。训练集用于对构造的模型进行训练，测试集则用于对训练后的模型进行评估。通常情况下，训练集和测试集的规模之比为 **7:3** 或 **8:2** 等。有很多现成的工具可以按给定的比例将一个数据集分割为训练集和测试集。然而，在 **PyTorch** 中，一般使用的数据都表示为张量。在此情况下，通过利用张量的切片操作，数据集的分割就变得**十分简单**。

3.4 网络模型的训练与测试

3.4.1 数据集分割

例如，对于**例 3.1** 中如果按 7:3 划分为训练集和测试集，则可以使用下列代码实现：

```
rate = 0.7 #定义分割的比例
X, Y = torch.randn(2040,3), torch.randn(2040) #产生模拟数据
train_len = int(len(X)*rate) #设置训练集的规模，结果是 2040*0.7=1428
trainX, trainY = X[:train_len], Y[:train_len] #取前面 70%的样本作为训练集
testX, testY = X[train_len:], Y[train_len:] #取后面 30%的样本作为测试集
```

一般来说，在进行数据集分割之前，先随机**打乱**数据集中样本的顺序，可参考下列代码：

```
index = torch.randperm(len(Y)) #效果相当于对[0, len(Y)-1]中的整数进行随机排列
X, Y = X[index], Y[index] #随机打乱 X 和 Y 中样本的顺序
```

有时候，可能需要按一定的比例将数据集分割为训练集、验证集（主要用于在模型训练过程中检验当前模型是否**过拟合**等）和测试集，这种划分方法也可以参照上述思路来解决。

3.4 网络模型的训练与测试

3.4.2 数据打包方法

批量梯度下降方法已被实践检验为可行的训练方法，也是最常用的训练方法。为使用这种方法，需要事先对训练用的数据进行打包。何为**数据打包**？实际上，就是将给定的数据集（包括训练集、验证集和测试集等）划分为若干个同等规模的数据批量（batch）的过程，而一个批量也称为一个数据包，因而划分为批量的过程也称为数据打包。当然，批量的大小是需要事先设定的，最后一个批量在规模上可能小于其他批量。

3.4 网络模型的训练与测试

3.4.2 数据打包方法

对于数据打包，可利用 **python** 语言并通过**分段切片**来实现。例如，对含有 **1428** 条样本的训练集进行打包，包的大小（batch_size）设置为 100，则可用下列代码实现：

```
batch_size = 100 #设置包的大小
train_loader = [] #放置数据包的容器
for i in range(0, len(trainX), batch_size): #分段切片，构造数据包
 t_trainX,t_trainY = trainX[i:i+batch_size],trainY[i:i+batch_size]
 t = (t_trainX,t_trainY) #将特征数据包和标记包组成元组
 train_loader.append(t) #将元组保存到容器 train_loader 中
```

输出各包的规模可以发现，前面 14 个包的规模均为 100，而最后一个包的规模为 **28**。原因在于，在总共包含 1428 条的数据样本中，前面 14 个包一共用了 1400 条样本，而最后只剩下 28 条样本，所以最后一个数据包的规模为 28。

3.4 网络模型的训练与测试

3.4.2 数据打包方法

通过上述代码，我们不难理解数据打包的基本原理。但如果在实践中，要编写这么代码才能完成数据打包，这显得比较繁琐。有没有更简便的方法呢？有！**PyTorch** 为我们提供了。
例如，为了完成上述代码相同的功能，我们仅需两条语句：

```
from torch.utils.data import DataLoader, TensorDataset
#trainX, trainY = torch.randn(1428,3), torch.randn(1428) #产生模拟数据
train_set = TensorDataset(trainX, trainY) #对 trainX 和 trainY 进行组对
train_loader = DataLoader(dataset=train_set, #调用打包函数
 batch_size=100, #包的大小
 shuffle=True) #默认 shuffle=False
```

其中，TensorDataset()函数用于对 trainX 和 trainY 进行“组对”，类似 python 中的 **zip** 功能；

3.4 网络模型的训练与测试

3.4.2 数据打包方法

DataLoader()函数则用于完成数据打包功能，其涉及的常用参数包括：

- **dataset**：用于指定加载的数据集(Dataset 对象)
- **batch_size**：用于设定包的**大小**（规模）
- **shuffle**：值为 True 表示要**打乱**样本的顺序后再打包，为 False（默认值）则表示不打乱样本的顺序
- **num_workers**：设置使用多进程加载的**进程数**，0 代表不使用多进程
- **drop_last**：当样本总数不是 batch_size 的整数倍，如果这时 drop_last 为 True，则会
- 将多出来而又不足一个数据包的样本**丢弃**，而为 False（默认值）则表示按实际剩
- 余的数据打包。

3.4 网络模型的训练与测试

3.4.2 数据打包方法

DataLoader()函数返回的结果是一个**迭代器**，可以通过循环或数组化转换来访问其中的数据包。
例如，运行下列代码：

```
for xb, yb in train_loader:
 print(xb.shape, yb.shape)
```

可以看到该迭代器包含的数据包的形状：

```
torch.Size([100, 3]) torch.Size([100])    #第一到第14组

torch.Size([28, 3]) torch.Size([28])      #最后一组
```

3.4 网络模型的训练与测试

3.4.3 模型的训练方法

当将一个已定义的 **nn.Module 类**的派生类实例化后，会得到一个网络模型。网络模型可视为由网络结构和参数组成，而实例化后得到的模型的参数大多是**随机初始化**形成的。这时模型没有任何的预测功能。训练的目的就是，将训练数据输入模型，然后正向计算模型的输出和目标之间的误差，进而反向计算误差函数在各个参数上的梯度，最后利用得到的**梯度更新参数**。反复执行这个过程，直到误差足够小的时候，停止训练过程。模型训练的一个核心工作是设计误差函数，实际上就是设计**损失函数**（在优化理论中称为目标函数）。损失函数的设计是根据问题的性质来完成的，这在 2.3 节中已经进行了介绍。
具体地，对于给定的数据批量 X 及其标记 Y，令 model 表示实例化后得到的模型，并记：

$$\hat{Y} = \text{model}(X)$$

$\boldsymbol{Y}$　表示批量 X 在输入模型 model 后产生的输出。

3.4 网络模型的训练与测试

3.4.3 模型的训练方法

令$\mathcal{L}$表示损失函数，则$\mathcal{L}(\hat{Y},Y)$表示模型输出$\hat{Y}$，和目标Y之间的误差。在 **PyTorch** 框架中，对于每个 X，利用 backward()方法，都可以自动计算$\mathcal{L}(\hat{Y},Y)$在各个参数上的**梯度**，然后调用 **step()方法**自动利用梯度更新各个参数。对所有的数据批量，轮流使用它们对模型 model 进行参数更新，每轮一遍称为**一代**或**一轮**（epoch）。一般情况下，对一个模型的训练要经过若干轮才能收敛。模型的训练过程用
伪代码表示如右图：

```
(1)  通过实例化得到模型 model
(2)  optimizer=torch.optim.Adam(model.parameters(),lr=lr)     #设置优化器
                                                              #告诉它哪些参数要优化
(3)  for epoch in range(epochs):   #epochs 为事先设定的迭代代数
(4)      for X, Y in train_loader: #train_loader 为所有数据集批量及其标记的集合
(5)          Ŷ = model(X)
(6)          loss = ℒ(Ŷ,Y)   #计算损失函数值
(7)          optimizer.zero_grad()        #对各个参数的梯度进行清零
(8)          loss.backward()              #自动反向计算梯度
(9)          optimizer.step()             #利用梯度自动更新各个参数
(10)
```

显然，模型的前向计算功能是程序员在模型 model 的类代码中定义的，而复杂的反向梯度计算和参数更新则是由**优化器 optimizer** 在后台自动完成的。

3.4 网络模型的训练与测试

3.4.4 梯度累加的训练方法

在模型训练过程中，适当增加批量的大小可以提高模型的**泛化能力**。但批量大小的增加会大量占用 **GPU 显存**资源，甚至会导致 GPU 显存溢出而无法运行程序。然而，有的模型包含大量的参数，因而模型本身就耗费大量 GPU 显存资源，因此批量大小只能设置得很低，从而会应用影响模型的泛化能力。于是，在有限 GPU 显存资源的条件下，如何提高数据批量的大小成为提升模型泛化能力的一个关键问题。

幸运的是，我们可以通过梯度累加的方法来变相提高数据批量的大小，同时不额外占用 GPU 显存资源。

3.4 网络模型的训练与测试

3.4.4 梯度累加的训练方法

该训练方法表明，在迭代过程中每做 accu_steps 次迭代（在这个过程中自动做梯度积累），才做一次参数更新（同时对梯度清零），但不增加批量的大小，因而不会增加对 GPU 显存资源的额外要求。由于做了 accu_steps 次迭代后再利用**累加的梯度**进行参数更新，因而其效果几乎相当于将批量大小由|X|改为|X|*accu_steps，可以在既定条件下大幅度提升了模型的泛化能力。

```
(1)  accu_steps = r #设定梯度积累的代数
(2)  通过实例化得到模型 model
(3)  optimizer=torch.optim.Adam(model.parameters(),lr=lr)
(4)  for epoch in range(epochs):
(5)      for k, (X, Y) in enumerate(train_loader):
(6)          Ŷ = model(X)
(7)          loss = ℒ(Ŷ, Y)   #计算损失函数值
(8)          loss = loss / accu_steps        #计算损失的平均值
(9)          loss.backward()                 #反向计算梯度并累加
(10)         if (k+1)%accu_steps == 0:       #每 accu_steps 次迭代做一次参数更新
(11)             optimizer.step()            #参数更新
(12)             optimizer.zero_grad()       #梯度清零
```

3.4 网络模型的训练与测试

3.4.5 学习率衰减在训练中的应用

在网络训练过程中，当**学习率**设置得过大时，容易造成收敛过程振荡，不易找到高精度解，但它有助于快速逼近**全局最优解**，降低陷于局部解的几率；当学习率设置得过小时，虽然有助于获得高精度解，但是收敛速度慢，容易陷于局部最优解。一种理想的做法是，训练刚开始时使用较大的学习率，使得模型快速向全局最优解逼近；随着训练过程的推进，逐步降低学习率，以找到高精度的最优解。显然，要对学习率做这样的设置，首先要找到访问学习率的方法。在 PyTorch 中，每个优化器都有 param_groups 属性，该属性是一个 list 对象，其元素param_groups[0]是一个 dict 对象。该 dict 对象含有 6 个键：Params、lr、betas、eps、weight_decay、amsgrad，其中键'lr'的值 param_groups[0]['lr']就是优化器的学习率，通过访问该键值即可以获得或修改学习率。据此，我们可以手动调整学习率。

3.4 网络模型的训练与测试

3.4.5 学习率衰减在训练中的应用

例如，作为例子，我们将例 3.1 中的学习率初始设置为 0.01，然后每迭代 50 次，让学习率自乘 **0.9**（即减少 10%），同时保证学习率的最低值不低于 0.0008；此外，为了观察衰减效果，我们仅从 X 和 Y 中选择 400 条数据样本来训练模型，并将每次迭代时的学习率保存起来，训练完后在二维平面上绘制学习率的**衰减**曲线图。
相关核心代码如下：

```
optimizer = torch.optim.Adam(model3_1.parameters(), lr=0.01)
i += 1
 if i%50==0:
 optimizer.param_groups[0]['lr'] *= 0.9 #让学习率自乘 0.9（即衰减学习率）
 #防止学习率过低：
 optimizer.param_groups[0]['lr'] =max(optimizer.param_groups[0]['lr'],0.0008)
 lr_list.append(optimizer.param_groups[0]['lr']) #保存当前的学习率
```

3.4 网络模型的训练与测试

3.4.5 学习率衰减在训练中的应用

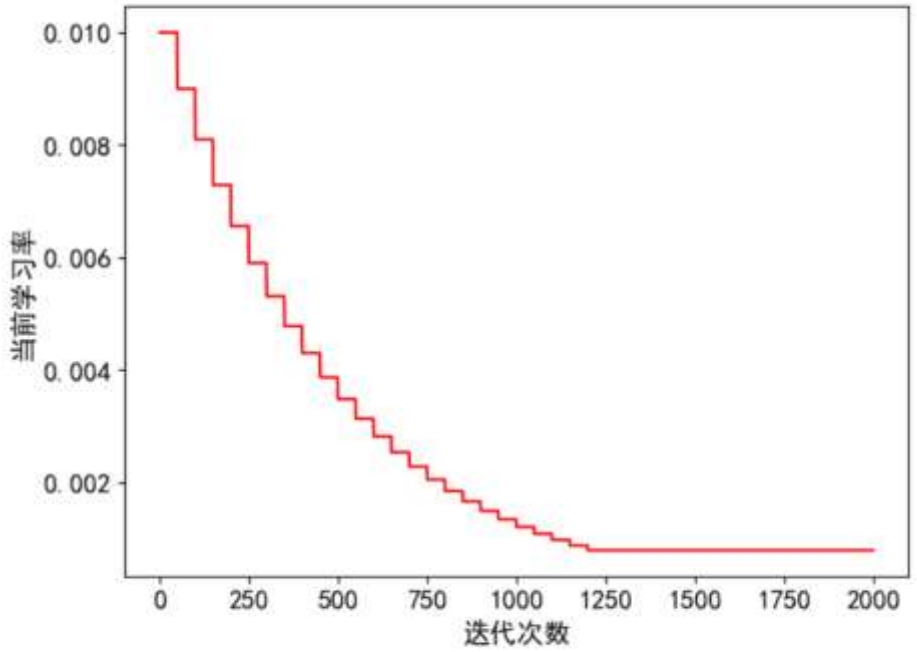


图 3-8 学习率变化曲线图

执行上述代码，结果得到如图 3-8 所示的学习率变化曲线图。从图 3-8 可以看到，学习 率确实从 0.01 开始，逐步衰减，最终降低到 0.0008。

3.4 网络模型的训练与测试

3.4.5 学习率衰减在训练中的应用

学习率衰减也可以利用 **torch.optim.lr_scheduler.StepLR()方法**来实现，书写的代码更为简洁。

该方法主要需要设置三个参数：

- **optimizer**：设置当前使用的**优化器**对象
- **step_size**：每迭代 step_size **次**后更新一次学习率
- **gamma**：每次更新时，学习率**自乘**该 gamma（学习率衰减的乘法因子，默认值为0.1）

调用 torch.optim.lr_scheduler.StepLR()方法时会产生一个对象，该对象提供了 step()方法。每执行一次 step()方法就相当于做了一次迭代，也就是说，迭代次数是按照 step()方法的执行次数来统计的。

3.4 网络模型的训练与测试

3.4.5 学习率衰减在训练中的应用

例如，为了实现跟上面有相同的**学习率衰减**效果，先用**torch.optim.lr_scheduler.StepLR() 方法**对优化器 optimizer 的学习率的更新方式进行设置：每迭代 50 次更新一个学习率，衰减 的乘法因子设置为 0.9。
相应语句如下：

```
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=50, gamma=0.9)
```

然后在循环体中用下列语句更新学习率：

```
scheduler.step()
```

当然，在学习率衰减方法中，不同例子对学习率初始值和衰减乘法因子的设置也有所不 同，需要经验积累，慢慢体会。

3.4 网络模型的训练与测试

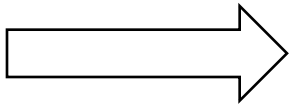
3.4.6 模型的测试

对于一个已经训练好的模型，为测试其性能，学者们研究了多种评价指标，比如**精确率**（precision, 又称查准率）、**召回率**（recall, 又称查全率）、**准确率**（accuracy）等。其中，准确率使用得最频繁，这里先介绍这个指标的测试方法。

【例 3.3】改写例 3.1 中的程序代码，将数据集分割为训练集和测试集，采用批量梯度下降方法，用训练集训练网络，用测试集测试模型的准确率。

按照前面介绍的有关数据集分割方法、数据打包方法等内容改写了例3.1中的程序代码，**具体**代码见教材P74.

```
... ...
print(s)
```



```
在测试集上的预测准确率为：100.0%
```

3.4 网络模型的训练与测试

3.4.6 模型的测试

对上述程序代码，**说明**几点：

（1）对从磁盘文件中读到的数据集进行**分割**，其中训练集占 70%，测试集占 30%。

（2）使用函数 TensorDataset()和 DataLoader()对训练集和测试集进行了**打包**，包的大小设置为 100，使得程序代码变得比较简洁。但有研究表明，包的大小设置为 **2 的幂次方**为好，如 32、64、128 等整数，这样运算效率会更高。类似地，每一网络层中神经元的个数也应该设置为 2 的幂次方。

（3）在测试阶段，用 **torch.no_grad** 设置一个上下文管理器，在此管理器中执行测试代码。其目的是，使梯度计算失效（每个计算结果的 requires_grad 属性值均为 False），即不再做梯度计算，可以减少计算所用内存消耗，**提高效率**。

3.4 网络模型的训练与测试

3.4.6 模型的测试

（4）对于训练好的模型 **model3_1**，当输入一个形状为 torch.Size([100, 3])的数据批量 x 后，会产生一个形状为 torch.Size([100, 2])的输出 pre_y，这是一个 100*2 矩阵。按照我们对网络的设计，矩阵中的每一行对应一条样本，该行上最大值所在列的索引即为模型判断该样 本所属类别的索引。这样，我们找出每一行上最大值所在列的索引即可，代码如下：

```
pre_y_index = torch.argmax(pre_y, dim=1)
```

这样的列索引保存张量 pre_y_index 中。进而，让其与给定类别索引进行对比：pre_y_index==y 对比后，返回逻辑 True 或 False。对这些逻辑值进行整数转换后，True 和 False 分别变 为 1 和 0。这样，再做一个求和即可得到被正确预测的样本数：

```
t = (pre_y_index==y).long().sum()
```

据此，经过**循环迭代**，即可计算出模型的预测准确率。 在本例中，测试集大小为 **612**，测试得到的模型准确率为 100%。

3.4 网络模型的训练与测试

3.4.7 应用案例——波士顿房价预测

【例 3.4】波士顿房屋价格预测。

在机器学习数据库（UCI，https://archive.ics.uci.edu/）中有一个关于波士顿房屋价格信息的数据集，数据文件名为 housing.data。为方便读者学习，我们将其保存在**./data** 目录下。该数据文件一共包含 14 个属性，其中前 13 个属性为可能影响房价的因素，最后一个属性为房屋价格的平均值（单位为 1 万美元）。这 14 个属性的名称、含义及其数据类型说明如表 3-1 所示。

表 3-1 波士顿房屋价格数据集（housing.data）

序号	属性名	含义	数据类型
1	CRIM	人均犯罪率	连续值
2	ZN	超过 25000 平方英尺的住宅用地比例	连续值
3	INDUS	非零售商业用地比例	连续值
4	CHAS	是否临近 Charies River	离散值（0 表示不临近，1 表示临近）
5	NOX	一氧化碳的浓度	连续值
6	RM	每栋房屋的平均房间数	连续值
7	AGE	1940 年以前建成的自住房比例	连续值
8	DIS	到波士顿 5 个就业中心的加权平均距离	连续值
9	RAD	到达高速公路的便利指数	连续值

序号10-14见**教材P77**

3.4 网络模型的训练与测试

3.4.7 应用案例——波士顿房价预测

编写实现上述任务的程序代码的主要步骤如下。

（1）构建预测模型。由于预测的属性值（房价）是连续的，因而该任务属于**回归问题**。

对于一个输入样本，有 13 个特征值输入，一个输出。为此，我们构建一个全连接网络来充当这样的预测模型。该网络的**结构如图 3-9** 所示，其中输入层有 13 个输入节点，隐含层有 512 个神经元，输出层有 1 个神经元。

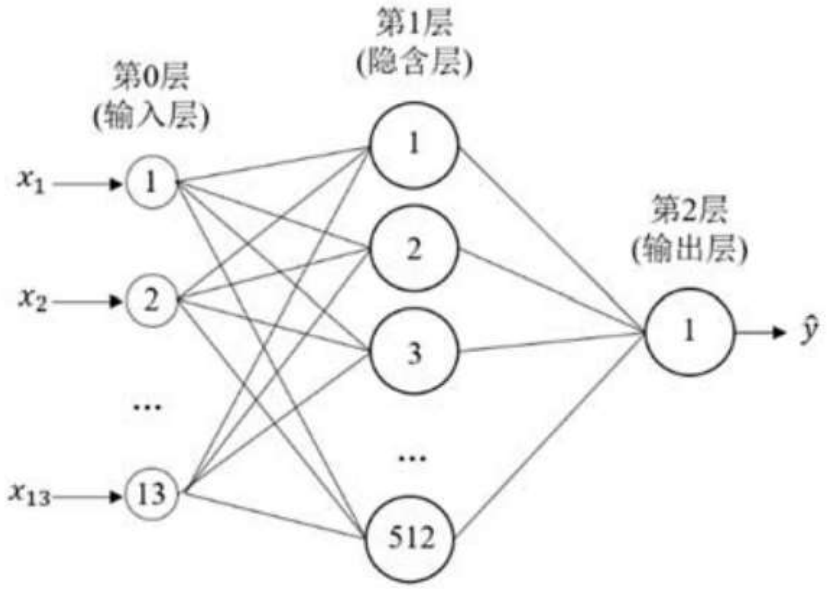


图 3-9 用于波士顿房价预测的全连接神经网络

3.4 网络模型的训练与测试

3.4.7 应用案例——波士顿房价预测

为此，先定义**两个网络层**：

```
self.fc1 = nn.Linear(13, 512)
self.fc2 = nn.Linear(512, 1)
```

然后将这两个网络层“连接”起来，形成一个完整的网络：

```
out = self.fc1(x)
out = torch.sigmoid(out) #运用激活函数 torch.sigmoid()
out = self.fc2(out)
out = torch.sigmoid(out) #运用激活函数 torch.sigmoid()
```

（2）**读取**文件“housing.data”中的数据，转换为张量 X 和 Y，并**打乱**其中样本的顺序。

（3）按 **8:2** 将数据集**分割**为训练集和测试集。

（4）按列对数据集进行**归一化**。为此，先定义归一化方法，然后分别对测试集和数据集进行归一化：

3.4 网络模型的训练与测试

3.4.7 应用案例——波士顿房价预测

```
def map_minmax(T): #归一化函数
 min,max = torch.min(T,dim=0)[0],torch.max(T,dim=0)[0] #对 T 按列归一化
 r = (1.0*T-min)/(max-min)
 return r
trainX,trainY = map_minmax(trainX),map_minmax(trainY)
testX,testY = map_minmax(testX),map_minmax(testY)
```

注意：训练集和测试集一般要分开归一化（但归一化方法和原理应一样），而 不应先对整个数据集先做归一化，然后再进行分割，否则可能导致测试集和训练集之间存在 **依赖关系**，使得测试结果不能反映实际情况。 还要注意的是，如果使用了sigmoid()函数作为激活函数，一般需要按列对数据进行归 一化，尤其需要对标记数据进行归一化，否则效果很差，甚至模型不能收敛。

3.4 网络模型的训练与测试

3.4.7 应用案例——波士顿房价预测

（5）对训练集和测试集进行打包，包的大小设置为 16。
（6）选择优化器和设置学习率：

```
optimizer = torch.optim.Adam(model2_2.parameters(), lr=0.01)
```

学习率是一个重要的超参数，需要多次调试。在本例中，几经调试，发现在学习率 lr 设置为 0.01 比较好（这也是默认值）。

（7）在训练部分中，由于该问题属于回归问题，因此使用均方差损失函数：

```
loss = nn.MSELoss()(pre_y,y)
```

（8）在测试部分中，对于回归问题，一方面通过画图来展示模型的拟合程度，代码来实现：

```
# ls 和 lsy 的初始值均为 torch.Tensor([])
ls = torch.cat((ls, pre_y)) #pre_y 为预测输出的结果，属于张量类型
lsy = torch.cat((lsy, y)) #y 为给定的结果（样本的标记）
```

3.4 网络模型的训练与测试

3.4.6 模型的测试

运行上述代码，程序输出的结果如下：

在测试集上的预测准确率为：87.3%

绘制的曲线图如图 3-10 所示。
从准确率和曲线图看，该程序取得了**较高**的拟合性能。当然，要达到实用阶段，还需更多的训练数据，并需设计更优的模型。

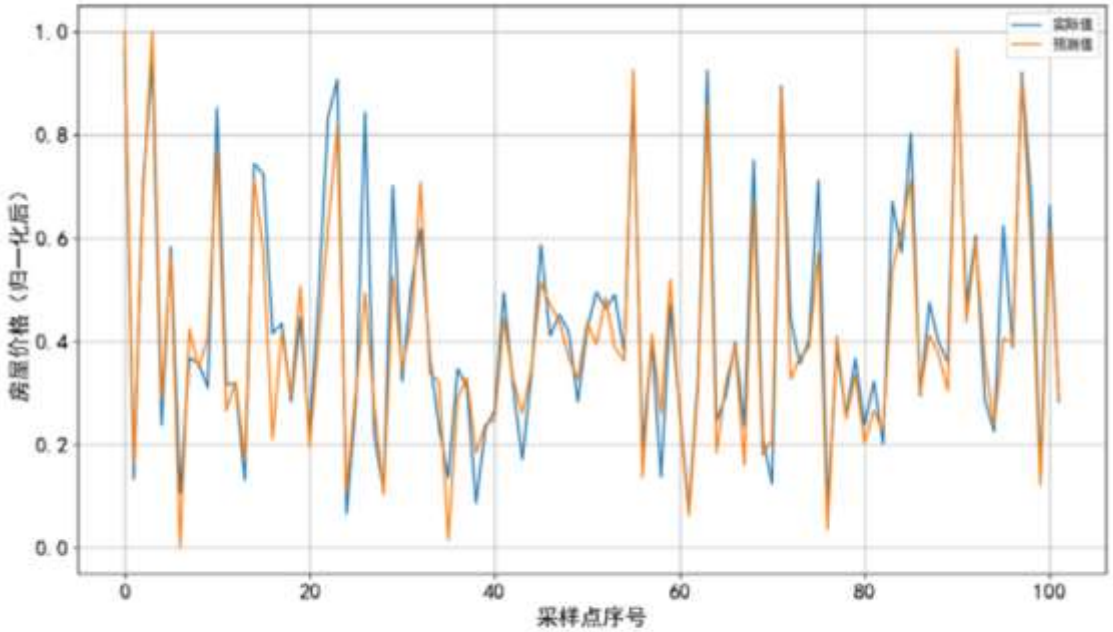
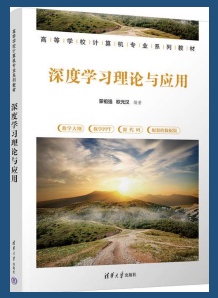


图 3-10 例 3.4 程序运行输出的曲线图



本 章 内 容

contents

3.1 构建一个简单的全连接神经网络——解决二分类问题

3.2 全连接神经网络的构造方法

3.3 几种主流的损失函数

3.4 网络模型的训练与测试

3.5 正向计算和反向梯度传播的理论分析

3.5 正向计算和反向梯度传播的理论分析

3.5.1 正向计算

正向计算是指从输入到输出的计算过程。我们仍然从简单的例子入手。
考虑图 3-11 所示的**三层全连接神经网络**。

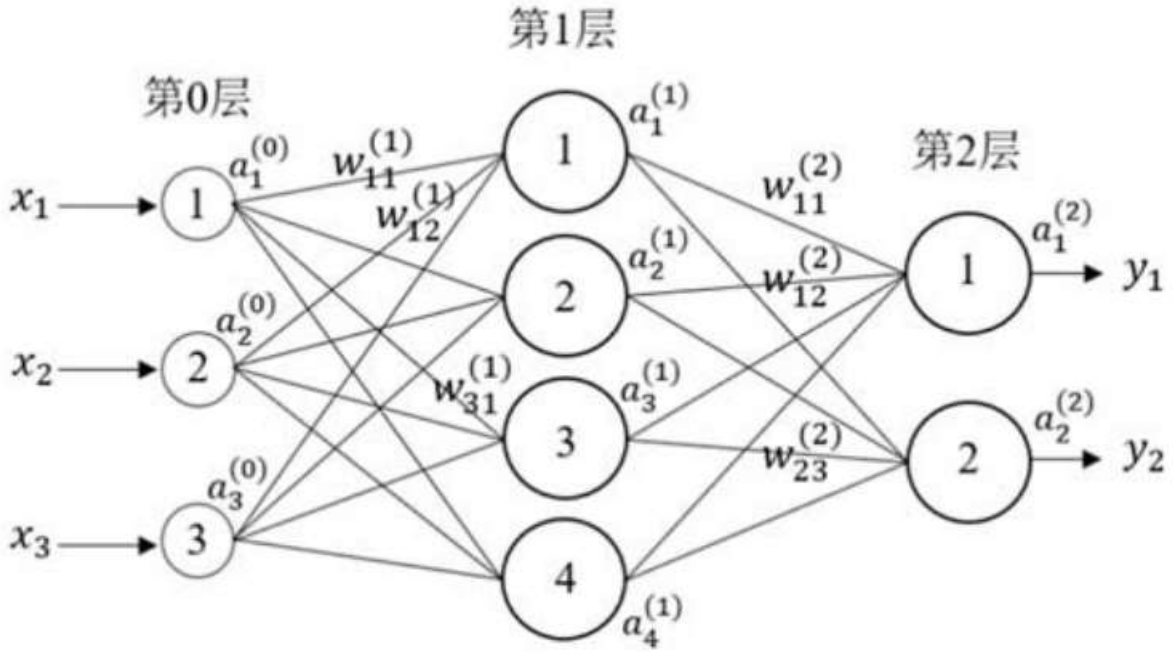


图 3-11 带输出标注的三层全连接神经网络

3.5 正向计算和反向梯度传播的理论分析

3.5.1 正向计算

为了方便表达，对于第 l 层上的神经元，我们用$a_i^{(l)}$表示其中第 i 个神经元的输出。

例如，第 1 层上各神经元的**输出**分别为$a_1^{(1)}$ $a_2^{(1)}$ $a_3^{(1)}$ $a_4^{(1)}$ 。我们把各神经元的输出标注在其**右上**方，得到如图 3-11 所示的带标注的神经网络图，其中$a_1^{(0)} = x_1, a_2^{(0)} = x_1 , a_3^{(0)} = x_1 , a_1^{(2)} = x_1 , a_2^{(2)} = x_2$ 。另外，需要说明的是，第 1 层上第 1 个神经元节点和第 0 层上第 2 个神经元节点的边的权值用$w_{12}^{(1)}$来表示，其中上标的“(1)”表示该边隶属于第 1 层，下标中的“1”和“2”分别表示第 1 层上第 1 个节点和第 0 层上第 2 个节点，其他表示依此类推。

3.5 正向计算和反向梯度传播的理论分析

3.5.1 正向计算

我们先考虑第 1 层上各神经元节点。根据第二章有关感知器的内容，其输出的计算表达式如下：

$$a_1^{(1)} = \sigma(w_{11}^{(1)} a_1^{(0)} + w_{12}^{(1)} a_2^{(0)} + w_{13}^{(1)} a_3^{(0)} + b_1^{(1)})$$

$$a_2^{(1)} = \sigma(w_{21}^{(1)} a_1^{(0)} + w_{22}^{(1)} a_2^{(0)} + w_{23}^{(1)} a_3^{(0)} + b_2^{(1)})$$

$$a_3^{(1)} = \sigma(w_{31}^{(1)} a_1^{(0)} + w_{32}^{(1)} a_2^{(0)} + w_{33}^{(1)} a_3^{(0)} + b_3^{(1)})$$

$$a_4^{(1)} = \sigma(w_{41}^{(1)} a_1^{(0)} + w_{42}^{(1)} a_2^{(0)} + w_{43}^{(1)} a_3^{(0)} + b_4^{(1)})$$

其中，$b_1^{(0)}$ 为第 1 层上第 1 个神经元的偏置项。如果每一层的输出、参数和偏置项都用向量或矩阵来表示：

3.5 正向计算和反向梯度传播的理论分析

3.5.1 正向计算

$$\boldsymbol{A}^{(0)}=\begin{bmatrix}a_1^{(0)}\\a_2^{(0)}\\a_3^{(0)}\end{bmatrix}=\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix},\quad \boldsymbol{A}^{(1)}=\begin{bmatrix}a_1^{(1)}\\a_2^{(1)}\\a_3^{(1)}\\a_4^{(1)}\end{bmatrix},\quad \boldsymbol{W}^{(1)}=\begin{bmatrix}w_{11}^{(1)},w_{12}^{(1)},w_{13}^{(1)}\\w_{21}^{(1)},w_{22}^{(1)},w_{23}^{(1)}\\w_{31}^{(1)},w_{32}^{(1)},w_{33}^{(1)}\\w_{41}^{(1)},w_{42}^{(1)},w_{43}^{(1)}\end{bmatrix},\quad \boldsymbol{b}^{(1)}=\begin{bmatrix}b_1^{(1)}\\b_2^{(1)}\\b_3^{(1)}\\b_4^{(1)}\end{bmatrix}$$

那么，第 1 层的输出就可以表示为：

$$\boldsymbol{A}^{(1)}=\sigma(\boldsymbol{W}^{(1)}\boldsymbol{A}^{(0)}+\boldsymbol{b}^{(1)})$$

那么，第 1 层的输出就可以表示为：

$$\boldsymbol{A}^{(2)}=\sigma(\boldsymbol{W}^{(2)}\boldsymbol{A}^{(1)}+\boldsymbol{b}^{(2)})$$

3.5 正向计算和反向梯度传播的理论分析

3.5.1 正向计算

其中，**σ为激活函数**，另外：

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} w_{11}^{(2)}, w_{12}^{(2)}, w_{13}^{(2)}, w_{14}^{(2)} \\ w_{21}^{(2)}, w_{22}^{(2)}, w_{23}^{(2)}, w_{24}^{(2)} \end{bmatrix}, \quad \boldsymbol{b}^{(1)} = \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \end{bmatrix}$$

从上述参数向量表达式的构造可以看出，

（1）前一层神经元的输出是当前网络层的输入；

（2）一个网络层的边的权值构成了一个权重参数矩阵，其中每一个神经元（当前层的输出节点）对应着矩阵的一行，每一个输入节点对应着矩阵的一列；

（3）每个神经元一般都有一个偏置项。

3.5 正向计算和反向梯度传播的理论分析

3.5.1 正向计算

一般地，假设一个网络一共有 o+1 个网络层：第 0 层（输入层）、第 1 层、…、第 o 层 （输出层），网络的输入为 X，则该网络的正向计算过程可表示如下：

$$
\begin{aligned}
&\boldsymbol{A}^{(0)} = \boldsymbol{X} \\
&\boldsymbol{A}^{(1)} = \sigma(\boldsymbol{W}^{(1)}\boldsymbol{A}^{(0)} + \boldsymbol{b}^{(1)}) \\
&\boldsymbol{A}^{(2)} = \sigma(\boldsymbol{W}^{(2)}\boldsymbol{A}^{(1)} + \boldsymbol{b}^{(2)}) \\
&\qquad \cdots\ \cdots \\
&\boldsymbol{A}^{(o)} = \sigma(\boldsymbol{W}^{(o)}\boldsymbol{A}^{(o-1)} + \boldsymbol{b}^{(o)}) \\
&\hat{\boldsymbol{Y}} = \boldsymbol{A}^{(o)}
\end{aligned}
$$

其中，矩阵$\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \ldots, \boldsymbol{W}^{(o)}$以及向量$\boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}, \ldots, \boldsymbol{b}^{(o)}$包含的参数即为所有待优化和**学习的参数**，$\hat{\boldsymbol{Y}}$表示网络的最终输出向量。

3.5 正向计算和反向梯度传播的理论分析

3.5.1 正向计算

【例 3.5】假设有一个全连接神经网络，其结构和各边权值如图 3-12 所示，其中各神经元节点下方标注的是相应的偏置项，并假设激活函数**σ是恒等映射**（相当于没有激活函数）。请给出每一层的输出结果和计算出网络中待优化参数的数量。

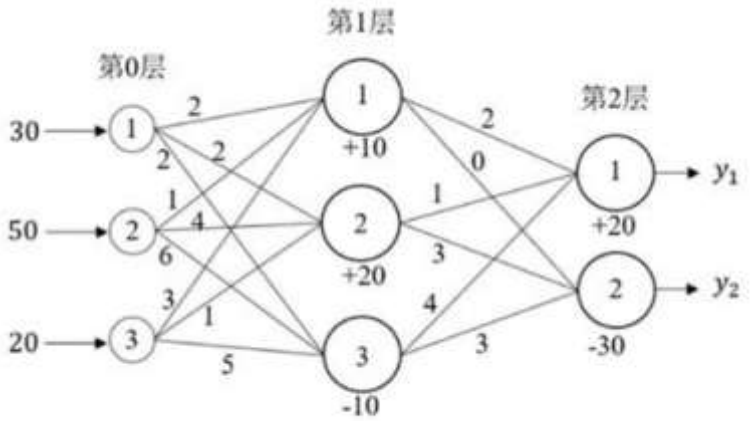


图 3-12　带参数标注的三层全连接神经网络

3.5 正向计算和反向梯度传播的理论分析

3.5.1 正向计算

下面按上述公式计算网络每一层的输出和最终的输出结果。

（1）根据网络结构及其标注，我们得到：

$$\boldsymbol{X}=\begin{bmatrix}30\\50\\20\end{bmatrix}$$

$$\boldsymbol{W}^{(1)}=\begin{bmatrix}2&1&3\\2&4&1\\2&6&5\end{bmatrix},\quad \boldsymbol{b}^{(1)}=\begin{bmatrix}10\\20\\-10\end{bmatrix}$$

$$\boldsymbol{W}^{(2)}=\begin{bmatrix}2&1&4\\0&3&3\end{bmatrix},\quad \boldsymbol{b}^{(2)}=\begin{bmatrix}20\\-30\end{bmatrix}$$

$$\boldsymbol{A}^{(0)}=\boldsymbol{X}=\begin{bmatrix}30\\50\\20\end{bmatrix}$$

$$\boldsymbol{A}^{(1)}=\boldsymbol{W}^{(1)}\boldsymbol{A}^{(0)}+\boldsymbol{b}^{(1)}$$

（注：激活函数σ是恒等映射）

$$=\begin{bmatrix}2&1&3\\2&4&1\\2&6&5\end{bmatrix}\begin{bmatrix}30\\50\\20\end{bmatrix}+\begin{bmatrix}10\\20\\-10\end{bmatrix}$$

$$=\begin{bmatrix}180\\300\\450\end{bmatrix}$$

3.5 正向计算和反向梯度传播的理论分析

3.5.1 正向计算

$$A^{(2)} = W^{(2)}A^{(1)} + b^{(2)}$$

$$= \begin{bmatrix} 2 & 1 & 4 \\ 0 & 3 & 3 \end{bmatrix} \begin{bmatrix} 180 \\ 300 \\ 450 \end{bmatrix} + \begin{bmatrix} 20 \\ -30 \end{bmatrix}$$

$$= \begin{bmatrix} 2480 \\ 2220 \end{bmatrix}$$

$$\hat{Y} = A^{(2)} = \begin{bmatrix} 2480 \\ 2220 \end{bmatrix}$$

$$A^{(1)} = \sigma(W^{(1)}A^{(0)} + b^{(1)}) = \sigma(\begin{bmatrix} 180 \\ 300 \\ 450 \end{bmatrix}) = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$ （注：约等于 1.0）

$$A^{(2)} = \sigma(W^{(2)}A^{(1)} + b^{2}) = \sigma(\begin{bmatrix} 27 \\ -24 \end{bmatrix}) = \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix}$$ （注：分别约等于 1.0 和 0.0）

$$\hat{Y} = A^{(2)} = \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix}$$

这个例子展示了该网络的正向计算过程。同时也说明，当输入数值比较大时，如 180、 300、450，这些数值被 sigmoid 函数视为几乎一样大小，从而失去它们对事物的区分能力。 因此，当使用 **sigmoid** 作为激活函数时，最好先对**数据进行归一化**。

3.5 正向计算和反向梯度传播的理论分析

3.5.1 正向计算

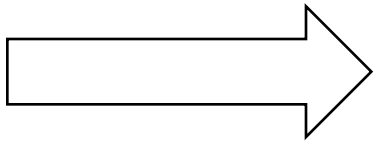
对图 3-12 所示的神经网络，我们也可以在 **PyTorch** 中用手工构建出来。其基本思路是，先定义包含对应两个全连接网络层的 **nn.Module 子类**，然后对其**实例化**，建立网络的模型对象，接着用手工构建相应的权重矩阵，并以之更新模型中相应的网络层参数，最后手工构造输入张量 X 并送入模型进行计算。相关代码如下：

```
fcNet.layer1.state_dict()['weight'].copy_(W1) #将第 1 层上的参数设置为 W1 中的参数
fcNet.layer2.state_dict()['weight'].copy_(W2) #将第 2 层上的参数设置为 W2 中的参数
fcNet.layer1.state_dict()['bias'].copy_(b1) #将第 1 层上的偏置项设置为 b1 中的参数
fcNet.layer2.state_dict()['bias'].copy_(b2) #将第 2 层上的偏置项设置为 b2 中的参数
```

3.5 正向计算和反向梯度传播的理论分析

3.5.1 正向计算

```
print(pre_Y.data.long())
```



```
tensor([30, 50, 20])
tensor([2480, 2220])
```

可以看到，这个结果和我们上面手工算出来的结果是一样的。这里例子同时也告诉我们 如何利用已有的参数去更新模型中的相应参数。 另外，如果想使用sigmoid激活函数，只需在**forward()方法**中放开相应的注释代码即可。运算后，输出的结果是：tensor([1, 0])，这与上面分析的结果也是一致的。

3.5 正向计算和反向梯度传播的理论分析

3.5.1 正向计算

在网络中，待优化的参数包含在矩阵$\boldsymbol{W}^{(1)}$和$\boldsymbol{b}^{(1)}$以及$\boldsymbol{W}^{(2)}$和$\boldsymbol{b}^{(2)}$中。实际上，每一个网络层中所有边的权重构成了一个参数矩阵，矩阵的行数等于该层中神经元节点的个数，列数等于输入节点的个数，因此该矩阵包含的参数的数量为：输入节点数×神经元节点数；除此外，每个神经元都包含一个偏置项（除非没有设置偏置项，即 bias=False）。这样，一个网络层中**参数的个数**为：

输入节点数×神经元节点数+神经元个数

在 **PyTorch** 中，可用下列代码求模型 fcNet 包含的参数的数量：

```
param_num = 0
for param in fcNet.parameters():
 param_num += torch.numel(param) #统计模型参数总量
print('该网络参数的总量为：', param_num)
```

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

1. 面向回归问题的全连接神经网络

在**回归问题**中，网络的输出一般是连续类型的数值，被预测的对象也是连续型数值。这 种网络一般只有一个输出，也就是说，网络的**输出层只有一个**神经元，这时通常设计如下的目标函数:

$$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

其中，$\hat{y}$为网络模型的实际输出，y是样本的实际标记，也是网络的期望输出。$\mathcal{L}$和$\hat{y}$都是关于网络 f 中所有参数的函数。我们的目标是，通过更新这些参数，使得$\mathcal{L}$达到最小值，这时的参数即为我们需要的参数。

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \lambda \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}}$$

$$b_i^{(l)} \leftarrow b_i^{(l)} - \lambda \frac{\partial \mathcal{L}}{\partial b_i^{(l)}}$$

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

具体来说，对任意给定的第 l 层，该层上各参数的更新公式如下：

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \lambda \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}}$$

$$b_i^{(l)} \leftarrow b_i^{(l)} - \lambda \frac{\partial \mathcal{L}}{\partial b_i^{(l)}}$$

其中，W_{ij}^{l}表示第 l 层中神经元节点 i 和输入节点 j（前一层上的节点 j，因为当前层的 输入实际上就是前一层神经元节点的输出）之间的边的权值，b_i^l为该层中神经元节点 i 的 偏置项，σ为学习率。

显然，问题的关键在于如何计算 $\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}}$和$\frac{\partial \mathcal{L}}{\partial b_i^{(l)}}$

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

假设网络 f 一共有 **o+1** 个网络层：第 0 层（输入层）、第 1 层、…、第 o 层（输出层），并假设激活函数为 sigmoid 函数，简记为σ。先定义一个符号：$I_i^{(l)}$令表示第 l 层上节点 i 的加权输入，即：

$$I_i^{(l)} = w_{i1}^{(l)} a_1^{(l-1)} + w_{i2}^{(l)} a_2^{(l-1)} + \ldots + w_{in}^{(l)} a_n^{(l-1)} + b_i^{(l)}$$

其中，此处 n 表示第 **l-1** 层中节点的数量。前面已经指出，$a_i^{(l)}$是对$I_i^{(l)}$第 l 层中节点 i 的输出，因此$a_i^{(l)}$是对$I_i^{(l)}$运用激活函数σ后的结果，即$a_i^{(l)} = \sigma(I_i^{(l)})$。两者之间的关系可用图 3-13 表示。

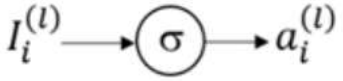


图 3-13　$I_i^{(l)}$和$a_i^{(l)}$之间的关系

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

由于σ为 sigmoid 函数，所以有：

$$\frac{\partial a_i^{(l)}}{\partial I_i^{(l)}} = a_i^{(l)}(1-a_i^{(l)})$$

考虑任意一个参数W_{ij}^{l}, 其中 0<l≤o。令$\delta_i^{(l)}$,表示目标函数$\mathcal{L}$关于加权输入$I_i^{(l)}$的导数，即：

$$\delta_i^{(l)} = \frac{\partial \mathcal{L}}{\partial I_i^{(l)}}$$

$\delta_i^{(l)}$称为第 l 层中节点 i 的误差项。

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

因为$\mathcal{L}$是加权输入$I_i^{(l)}$的函数，而$I_i^{(l)}$是$W_{ij}^{(l)}$,的函数，所以我们可以得到：：

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{L}}{\partial I_i^{(l)}} \frac{\partial I_i^{(l)}}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} a_j^{(l-1)}$$

即

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} a_j^{(l-1)}$$

考虑任意一个参数W_{ij}^{l}，其中 0<l≤o。令$\delta_i^{(l)}$,表示目标函数$\mathcal{L}$关于加权输入$I_i^{(l)}$的导数，即：

上式中，$a_j^{(l-1)}$是第 l-1 层中节点 j 的激活输出，在反向传播中是已知的。这样，如果**误差项$\boldsymbol{\delta_i^{(l)}}$**能确定下来，那么$\frac{\partial \mathcal{L}}{\partial W_{ij}^{l}}$就确定了。令 l=o，即考虑最后一层——**输出层**。

$$\delta_1^{(o)} = \frac{\partial \mathcal{L}}{\partial I_1^{(o)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(o)}} \frac{\partial a_1^{(o)}}{\partial I_1^{(o)}} = (a_1^{(o)} - y) \frac{\partial a_1^{(o)}}{\partial I_1^{(o)}} = (a_1^{(o)} - y)\, a_1^{(o)} (1 - a_1^{(o)})$$

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

在**反向传播**时，$a_1^{(o)}$等于$\hat{y}$，y是已知的样本标记值，因此$\delta_1^{(o)}$是确定的。这样，根据数学**归纳法**的思想，如果$\delta_i^{(l)}$可以用$\delta_k^{(l+1)}$来表达，那么$\delta_i^{(l)}$就可以确定了，其中 1≤l≤o-1，下同。

我们注意到，$\delta_i^{(l)}$是$\mathcal{L}$关于$I_i^{(l)}$的导数，而$I_i^{(l)}$对$\mathcal{L}$的影响是通过第 l+1 层中的节点来实现的。同时也注意到，第 l+1 层中任意节点 k 的加权输入$I_k^{(l+1)}$都是的$a_i^{(l)}$的函数，而$a_i^{(l)}$又是$I_i^{(l)}$的函数。令E^{l+1}表示第 l+1 层中节点的集合，则根据**全导数公式**可得到：

$$\delta_i^{(l)} = \frac{\partial \mathcal{L}}{\partial I_i^{(l)}} = \sum_{k \in E^{(l+1)}} \frac{\partial \mathcal{L}}{\partial I_k^{(l+1)}} \frac{\partial I_k^{(l+1)}}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial I_i^{(l)}}$$

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

其中，

$$\frac{\partial \mathcal{L}}{\partial I_k^{(l+1)}} = \delta_k^{(l+1)}$$

$$\frac{\partial I_k^{(l+1)}}{\partial a_i^{(l)}} = w_{ki}^{(l+1)}$$

$$\frac{\partial a_i^{(l)}}{\partial I_i^{(l)}} = a_i^{(l)}(1 - a_i^{(l)})$$

于是，

$$\delta_i^{(l)} = \sum_{k \in E^{(l+1)}} \delta_k^{(l+1)} w_{ki}^{(l+1)} a_i^{(l)}(1 - a_i^{(l)}) = a_i^{(l)}(1 - a_i^{(l)}) \sum_{k \in E^{(l+1)}} \delta_k^{(l+1)} w_{ki}^{(l+1)}$$

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

即，

$$\delta_i^{(l)} = a_i^{(l)}(1 - a_i^{(l)}) \sum_{k \in E^{(l+1)}} \delta_k^{(l+1)} w_{ki}^{(l+1)}，\text{其中 } 1 \le l \le o\text{-}1$$

在反向传播过程中，$a_i^{(l)}$和$w_{ki}^{(l+1)}$都是已知的，因此$\delta_i^{(l)}$确实可以用$\delta_k^{(l+1)}$来表达。这样，$\delta_i^{(l)}$也是确定的。根据上述推导，对于网络 f 及其输入 x 和标记 y，我们可以得出如下的**反向**梯度计算与参数更新过程：

首先执行正向计算：

$$\boldsymbol{A}^{(0)} = \boldsymbol{x}$$

$$\boldsymbol{A}^{(l)} = \sigma(\boldsymbol{W}^{(1)}\boldsymbol{A}^{(0)} + \boldsymbol{b}^{(1)})$$

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

$$A^{(2)} = \sigma(W^{(2)}A^{(1)} + b^{(2)})$$

$$\cdots\cdots$$

$$A^{(o)} = \sigma(W^{(o)}A^{(o-1)} + b^{(o)})$$

$$\hat{Y} = A^{(o)}$$

其中，$A^{(l)}$表示由第 l 层中各神经元节点输出$a_i^{(l)}$构成的向量，$A^{(o)} = (a_1^{(o)})$，即$a_1^{(o)} = \hat{y}$。

然后利用上述计算结果，执行后向梯度计算和参数更新：

(1) $\delta_1^{(o)} = (a_1^{(o)} - y)\, a_1^{(o)}(1 - a_1^{(o)})$

(2) $\delta_i^{(o-1)} = a_i^{(o-1)}(1 - a_i^{(o-1)}) \sum_{k \in E^{(o)}} \delta_k^{(o)} w_{ki}^{(o)}$

$= a_i^{(o-1)}(1 - a_i^{(o-1)})\, \delta_1^{(o)} w_{1i}^{(o)}$ $(i \in E^{(o-1)})$

(3) $\delta_i^{(o-2)} = a_i^{(o-2)}(1 - a_i^{(o-2)}) \sum_{k \in E^{(o-1)}} \delta_k^{(o-1)} w_{ki}^{(o-1)}$ $(i \in E^{(o-2)})$

$\cdots\cdots$

(4) $\delta_i^{(2)} = a_i^{(2)}(1 - a_i^{(2)}) \sum_{k \in E^{(3)}} \delta_k^{(3)} w_{ki}^{(3)}$ $(i \in E^{(2)})$

(5) $\delta_i^{(1)} = a_i^{(1)}(1 - a_i^{(1)}) \sum_{k \in E^{(2)}} \delta_k^{(2)} w_{ki}^{(2)}$ $(i \in E^{(1)})$

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

在上述计算过程中，每计算完一个误差项$\delta_i^{(l)}$后即可用下式计算$\mathcal{L}$关于W_{ij}^l的导数：

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} a_j^{(l-1)}$$

进而执行下列操作对参数W_{ij}^l进行更新：

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \lambda \delta_i^{(l)} a_j^{(l-1)}$$

对于偏置项$\delta_i^{(l)}$的更新：

$$b_i^{(l)} \leftarrow b_i^{(l)} - \lambda \delta_i^{(l)}$$

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

也就是说，根据上述分析和推导，网络权重参数和偏置项的更新公式如下：

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \lambda \delta_i^{(l)} a_j^{(l-1)}$$

$$b_i^{(l)} \leftarrow b_i^{(l)} - \lambda \delta_i^{(l)}$$

依照这个两个公式，从输出层开始，可**反向逐层**对网络中的参数进行更新，从而实现**参数的学习**。

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

2. 面向分类问题的全连接神经网络

对神经网络而言，**回归问题**和**分类问题**的主要不同是目标函数的不同。但这并未在本质上对后向传播中的梯度计算和参数更新造成本质区别。仔细分析后向传播过程可以发现，目标函数只是在最初的误差项计算中使用到，也就是下面这个式子：

$$\delta_i^{(o)} = \frac{\partial \mathcal{L}}{\partial I_i^{(o)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(o)}} \frac{\partial a_i^{(o)}}{\partial I_i^{(o)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(o)}} a_i^{(o)}(1 - a_i^{(o)}), i \in E^{(o)}$$

其中，$a_i^{(o)} = \hat{y}$，因而 $\frac{\partial \mathcal{L}}{\partial a_i^{(o)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}}$，这是计算目标函数$\mathcal{L}$关于网络输出$\hat{y}$的导数。因此，不同的目标函数会得到不同的导数，从而得到不同的最初误差项$\delta_i^{(o)}$，而其他误差项$\delta_i^{(o-1)}$，$\delta_i^{(o-2)}, \ldots, \delta_i^{(1)}$的计算方法都不变，与目标函数**没有**关系。

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

对于**二分类问题**而言，常用下面的目标函数：

$$\mathcal{L}(\hat{y}, y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$$

这时，网络只有一个输出节点 1 和一个输出$\hat{y}$，$\mathcal{L}$关于$\hat{y}$的导数$\frac{\partial \mathcal{L}}{\partial \hat{y}}$ = [(1-y) $\hat{y}$ -(1- $\hat{y}$)y],因而得到下面**最初的误差项**：

$$\delta_1^{(o)} = \frac{\partial \mathcal{L}}{\partial I_1^{(o)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(o)}} \frac{\partial a_i^{(o)}}{\partial I_1^{(o)}} = [(1-y)\hat{y} - (1-\hat{y})y]\hat{y}(1-\hat{y})$$

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

对于**多分类问题**，神经网络的输出层一般有多个输出节点，其数量一般跟类别的个数相等，即如果类别个数为 m，则| $E^{(o)}$ | = m，其中$E^{(o)}$表示输出层中节点的集合。这样，便有多个最初的误差项：$\delta_1^{(o)}$, $\delta_2^{(o)}$,..., $\delta_m^{(o)}$ （前面介绍的网络都只有一个输出节点，因而也都只有一个**最初的误差项**$\delta_1^{(o)}$ 。

在多分类问题中，预测输出$\hat{y}$一般为分布式表示，多采用**交叉熵损失**函数。

$\delta_1^{(o)}$,i=1,2,...,m, $\frac{\partial \mathcal{L}}{\partial a_i^{(o)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}}$ 。$\mathcal{L}$关于$\hat{y}$的求导过程比较复杂，纯属于数学问题，我们在这里不展开介绍了，有兴趣的读者可自行参考相关资料。

3.5 正向计算和反向梯度传播的理论分析

3.5.2 梯度反向传播与参数更新

上面介绍的有关梯度计算和参数更新的方法都是基于单个样本进行的，实际上也很容易 扩展到多个样本的情况，参数更新采用平均梯度来完成即可。

需要**注意**的是，在全连接神经网络中，由于采用 sigmoid 函数作为激活函数，其输出在 (0, 1)范围内，因而最初误差项$\delta_1^{(o)} = (a_1^{(o)} - y)\, a_1^{(o)}(1- a_1^{(o)})$显然小于 1，而在后面的迭代中 也不断乘以网络的输出值以及在此之前形成的误差项的加权和，如果各个乘法项都小于 1 （或长时间小于 1），则在反向传播到后面时梯度会变得小，甚至等于 0，这样就容易导致低层网络的参数无法得到更新。这种问题就是所谓的**梯度消失**问题。网络的层数越多，这种问题就越严重，因此全连接神经网络的网络层不会太多，一般不超过 3-4 层。

3.6 本章小结

本章内容：

- 全连接神经网络的构建和训练方法
- 回归、分类问题的损失函数的基本原理和设计方法
- 数据集的分割方法、数据打包方法、模型训练方法
- 网络的反向传播过程的理论基础