第 3 章 全连接神经网络

本章主要介绍 4 方面的内容，一是全连接神经网络的设计、构造和训练方法；二是几种主流损失函数的使用方法；三是神经网络的前向计算方法；四是神经网络的反向梯度计算和参数更新的原理，同时结合具体的示例进行介绍。

为了表述简洁，本章一般将全连接神经网络（Fully Connected Neural Network，FCNN）简称为神经网络。

3.1 构建一个简单的全连接神经网络——解决二分类问题

本节构建一个简单的全连接神经网络，用于解决一个二分类问题，并对程序代码进行解释，以让读者快速地对全连接神经网络的构建和训练方法有一个初步的认知。

3.1.1 一个简单全连接神经网络的构建和训练

本章从一个非常简单的示例入手，让读者逐步了解和掌握全连接神经网络的基本原理和使用方法。

【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。

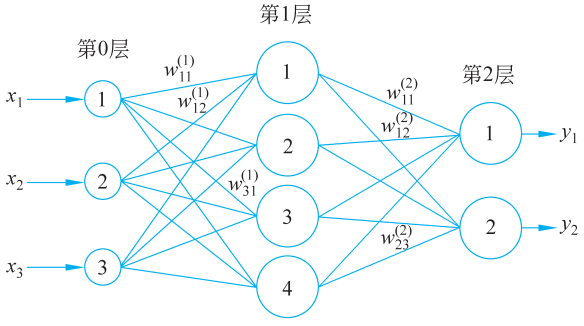

图 3-1 一个三层全连接神经网络

严格地说，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，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”。

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(这里“.”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:

$$\left.\begin{matrix}1.9308,-2.5692,-3.5692,0\\2.6044,-1.8956,-2.8956,0\end{matrix}\right]\text{—1020 条 0 类样本}$$

$$\left.\begin{matrix}-6.1000,-1.6000,-0.6000,1\\-6.0949,-1.5949,-0.5949,1\end{matrix}\right]\text{—1020 条 1 类样本}$$

其中,每一行表示一条数据样本,前面 3 个数字分别表示三维空间中的 3 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。

在产生这些数据时,通过数学方法使得它们能够被一个平面隔开,在平面的一边是属于 0 类的点,在另一边是属于 1 类的点,即确保它们是线性可分的。

本例构造如图 3-1 所示的三层全连接神经网络来实现对该数据集中的数据进行分类,全部代码及其说明如下:

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
torch.manual_seed(123)
#--------------------------------
#读取文件"例 3.1 数据.txt"中的数据:
path = r'.\\data'
fg=open(path+'\\'+"例 3.1 数据.txt","r",encoding='utf-8')
s=list(fg)
X1,X2,X3,Y = [],[],[],[]
for i,v in enumerate(s):
    v = v.replace('\n','')
    v = v.split(',')
    X1.append(float(v[0]))
    X2.append(float(v[1]))
    X3.append(float(v[2]))
    Y.append(int(v[3]))
fg.close()
#张量化
X1,X2,X3,Y = torch.Tensor(X1),torch.Tensor(X2),torch.Tensor(X3),torch.
LongTensor(Y)
#按列“组装”特征值,形成特征张量,形状为 torch.Size([2040, 3])
X = torch.stack((X1,X2,X3),dim=1)
del X1,X2,X3
index = torch.randperm(len(Y))          #随机打乱样本的顺序,注意保持 X 和 Y 的一致性
X,Y = X[index],Y[index]                 #Y 的形状为 torch.Size([2040])

#定义类 Model3_1
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
#------------------------------------------------------------
model3_1 = Model3_1()
optimizer = torch.optim.Adam(model3_1.parameters(), lr=0.01)
LS = []
for epoch in range(5):
    i = 0
    for x,y in zip(X,Y):
    #增加在 x 的第一个维上插入一个长度为 1 的维,这是
    #nn.CrossEntropyLoss()的需要。这个 1 可理解为由 1 个样本构成的批量
        x = x.unsqueeze(0)
        pre_y = model3_1(x)                           #默认调用 forward()方法
        #nn.CrossEntropyLoss()函数要求 y 的数据类型为 long
        y = torch.LongTensor([y])
        loss = nn.CrossEntropyLoss()(pre_y,y)  #交叉熵损失函数
        #print(loss.item())
        if i%50==0:
            LS.append(loss.item())                    #采样损失函数值,用于画图
        i += 1
        optimizer.zero_grad()                         #梯度清零
        loss.backward()                               #反向计算梯度
        optimizer.step()                              #参数更新
#绘制损失函数值的变化趋势图
plt.plot(LS)
plt.tick_params(labelsize=13)
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False
plt.grid()
plt.xlabel("损失函数值采样次序",fontsize=13)
plt.ylabel("交叉熵损失函数值",fontsize=13)
plt.show()
```

3.1.2 程序代码解释及网络层的构建方法

上述代码主要是做了如下的工作。

(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])。在 PyTorch 环境中,建议将数值数据全部表示为张量,因为张量的计算效率比较高,而且 PyTorch 的许多函数只接受张量输入,其输出亦为张量。

在本程序中，读取数据时，先将数据集中的 4 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 4 个一维张量，形状均为 torch.Size([2040])，最后将这 4 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”操作由如下语句实现：

```
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，为送入网络模型做准备。

(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)
```

或者说前者省略了参数 in_features 和 out_features，同时使参数 bias 的默认值 True。今后，我们将更多使用这种省略式的写法。

nn.Linear(3,4)的作用是构造这样的一个神经网络层：该网络层上一共有 4 个神经元，每个神经元都有一个偏置项，且每个神经元都有共同的 3 个输入，或者说都有 3 个输入节点，如图 3-2 所示。类似地，nn.Linear(4,2)构造了由 2 个神经元节点和 4 个输入节点构成的神经网络层，如图 3-3 所示。

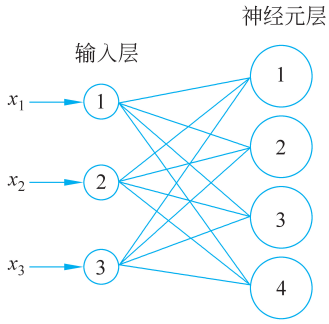

图 3-2　nn.Linear(3,4)构造的神经网络层

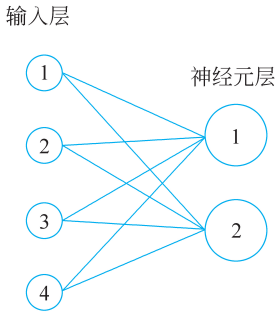

图 3-3　nn.Linear(4,2)构造的神经网络层

这两个网络层“拼接”在一起时就得到如图 3-1 所示的神经网络。实际上，从 forward()方法的代码中可以看出，最初输入的数据进入第一个网络层(图 3-2)，经过计算后产生了输出；而这个输出是作为第二个网络层(图 3-3) 的输入，经计算又产生输出。也就是说，输入和输出是相对网络层而言的，实际上在物理上并没有将任何两个网络层连接在一起，只是在逻辑上看似乎形成了一张网络。因此，在今后谈到神经络时，上一层的输出都可以看成当前层的输入，而当前层的输出又是下一层的输入。

(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)。显然，外层

循环用于控制训练的代数。

在训练过程中，使用了如下一条语句：

```
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。但现在我们每输入一个样本就进行一次梯度计算和一次参数更新，实际上相当于使用随机梯度下降算法，可以不需要这个维。但为了符合 PyTorch 程序的输入要求(严格说，是损失函数 nn.CrossEntropyLoss()的需要)，需要添加这个大小为 1 的维，故使用上面的语句。

(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中被用作损失函数。

对于其输入的两个参数 pre_y 和 y，可这样理解：pre_y 是一个数字矩阵(张量)，一行对应一个样本，一列对应一个类别；第 i 行和第 j 列上的元素 m_{ij} 表示当前批量中第 i 个样本被判定为第 j 类的程度，而 y 包含了当前批量中每个样本的类别标记(整型)，是类别标记构成的一维张量。从形状上看，如果 pre_y 的形状为 torch.Size([2,3])，则 y 的形状必须为 torch.Size([2])，其取值范围是[0,3-1]的整数(有 3 个类别)；如果 pre_y 的形状为 torch.Size([500,10])，则 y 的形状必为 torch.Size([500])，其取值范围是[0,10-1]的整数(有 10 个类别)。

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。例如，y 中第 1 个分量的值 $y_1=4$ 对应矩阵 pre_y 中第 1 行上第 4 列中的值 0.4(注意：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)。

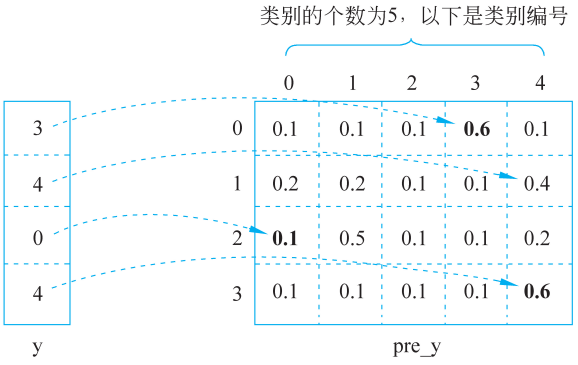

图 3-4 损失函数 nn.CrossEntropyLoss()(pre_y,y)中参数 pre_y 和 y 之间的关系

在本例中，由于 pre_y 的形状为 torch.Size([1,2])，而原来的 y 为标量，故 y 的形状必须改为 torch.Size([1])。为此，用下列语句改造 y 的形状：

```
y = torch.LongTensor([y])  #执行后,y的形状变为 torch.Size([1])
```

(5) 显示训练过程中损失函数值的变化趋势。每处理 50 个样本，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。

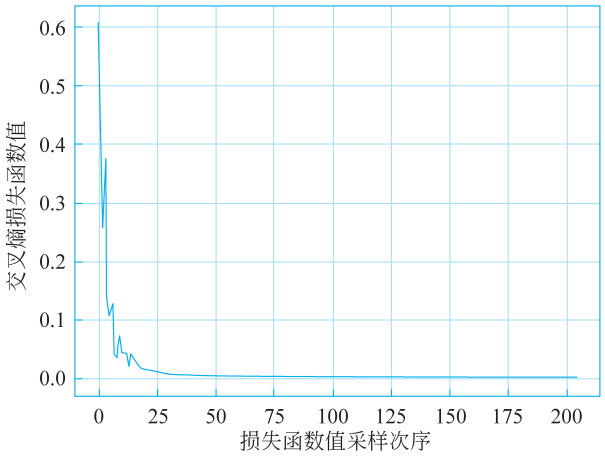

图 3-5　例 3.1 中程序损失函数值的变化趋势

从图 3-5 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。

3.2　全连接神经网络的构造方法

在 PyTorch 框架中，构建全连接网络的方法是，先定义所需要的网络层，然后将这些网络层“连接”起来，在逻辑上形成具有特定结构的神经网络。下面将以此为线索，介绍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。

该函数的作用是，建立一个全连接神经网络层，该网络层中有 m 个输入节点和 k 个输出节点（神经元节点）。从逻辑上看，每个输入节点和每个神经元节点都有一条边相连，每条边都有一个权值与之相对应。所有这些权值即是该网络层要学习的参数，总数为 m×k 个；此外，如果 bias=True，则每个神经都带有一个偏置项，这也是需要学习的参数。

总之，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

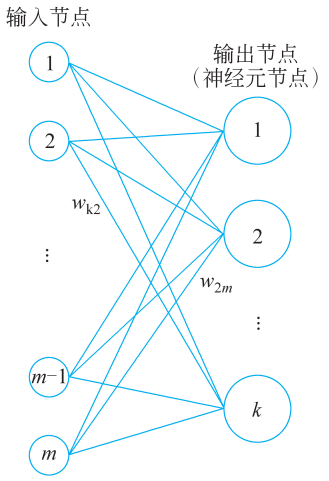

图 3-6　nn.Linear(m,k)定义的网络层

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。

在 PyTorch 框架中，一个全连接神经网络层并非按照图的存储结构来保存，而是被定义为一个参数矩阵以及由偏置项构成的偏置项向量。例如，nn.Linear(m,k,bias=False)定义了一个 k×m 的参数矩阵，nn.Linear(m,k,bias=True)则定义了一个 k×m 的参数矩阵和一个长度为 k 的偏置项向量。因此，一个全连接神经网络层可以视为一个参数矩阵和偏置项向量。当然，不管是矩阵还是向量，在 PyTorch 框架中实际上都是张量。

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。

对于任何张量，只要它的最后一个维的大小为 30，它都能被该网络层接收。在经过该网络层以后，大小为 30 的维就变成了大小为 50 的维(其他维的大小保持不变)。例如，下列代码定义的张量 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：

```
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)                  #执行时将产生错误
```

3.2.2　网络结构的实现

由 nn.Linear()函数定义而产生的网络层是一个孤立的网络层，只有将各个孤立的网络层“连接”起来才能形成一个完整的神经网络。观察下面的例子。

【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。

显然，该网络是由 4 个全连接网络层组成，可通过如下代码来定义这些网络层：

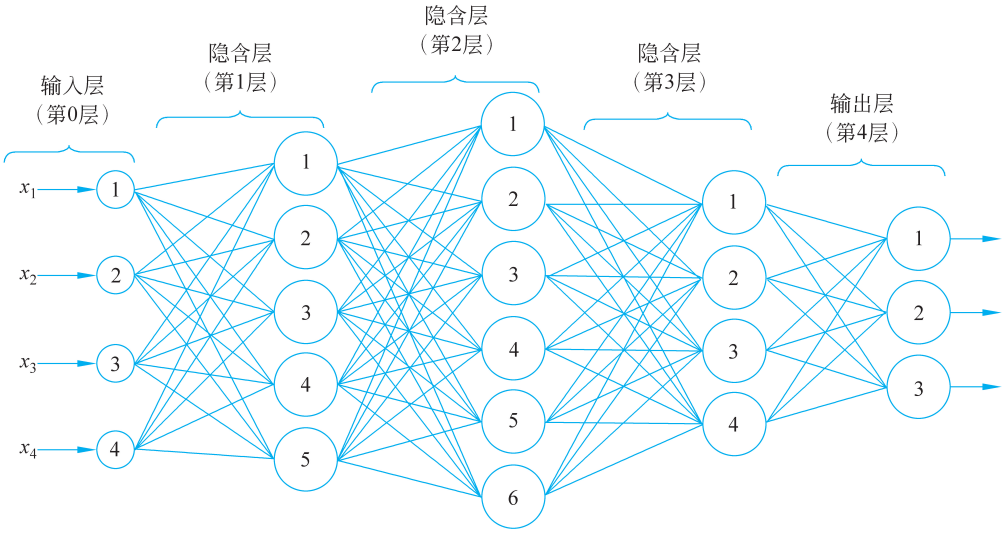

图 3-7　一个全连接神经网络

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

这 4 条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，需要使用如下代码将它们“连接”起来（一般在 forward()方法中实现），形成逻辑上的网络：

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

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，可以用如下代码实现：

```
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)
```

下面是本例的完整代码。

```
import torch
import torch.nn as nn
#定义全连接神经网络
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 = self.fc1(x)
        out = self.fc2(out)
        out = self.fc3(out)
        out = self.fc4(out)
        return out
anet = AFullNet()
sum = 0
for param in anet.parameters():    #计算整个网络的参数量
    sum += torch.numel(param)
print('该网络参数总量: %d'%sum)
```

执行该代码,输出结果如下:

```
该网络参数总量: 104
```

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。这也从一个侧面反映了我们创建的全连接网络是正确的。

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都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,能够处理的问题越复杂。在这种情况下,其隐含层节点的数量一般比较多,整个网络的参数量也比较大,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。

3.3 几种主流的损失函数

损失函数大致可以分为两种类型,一种是用于解决回归问题的损失函数,另一种是用于解决分类问题的损失函数。当然,这种分类并非是严格的。下面从理论上介绍几种主流的损失函数的基本原理和使用方法。

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与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 $p_1,p_2,\cdots,p_m$ 和 $q_1,q_2,\cdots,q_m$ 为两个分布，则两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，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。

在运用上述公式之前，p_i 和 q_i 一般都应先做 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 $\boldsymbol{x}$ 只能属于一个类别，所以 y 的分布式表示为：

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

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 $\boldsymbol{x}$ 后，模型在这 c 个类别上的输出分别为 $v_1,v_2,\cdots,v_c$，即模型输出 $\hat{y}$ 的分布式表示为：

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

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

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

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]上的实数，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 $\boldsymbol{x}$ 及其标记 y 的交叉熵损失函数，其中 $\hat{y}$ 为模型的输出。

从上述公式可以看出，作为类别索引的整数 y，它实际上指向分布 $(v_1,v_2,\cdots,v_c)$ 中某一元素的下标值(索引)。被指向的元素将被用于构造损失函数值，其他元素则被“丢弃”。从上式可以看出，对被指向的元素(如 v_k)，依次进行 softmax 归一化操作、自然对数运算和

取反操作，最后得到样本 $\boldsymbol{x}$ 的实际输出$\hat{y}$和期望输出 y 之间的误差(交叉熵损失函数的值)。

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 $\boldsymbol{X}$ 由 n 条样本构成，$\boldsymbol{Y}$ 是与其对应的标记向量，并假设 $\boldsymbol{X}=(\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_m)$，$\boldsymbol{Y}=(y_1,y_2,\cdots,y_n)$，其中 $y_1,y_2,\cdots,y_n$ 取值为$[0,c-1]$的整数，c 为类别的个数，m 为特征的个数。假设在 $\boldsymbol{X}$ 输入模型后得到的输出是$\hat{\boldsymbol{Y}}$，易知$\hat{\boldsymbol{Y}}$是一个 $n\times c$ 的矩阵，表示如下：

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

然后，取出$\hat{\boldsymbol{Y}}$在第 1 至 n 行中分别以 $y_1,y_2,\cdots,y_n$ 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 $\boldsymbol{X}$ 的交叉熵损失函数值：

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

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

在 PyTorch 中，$\hat{\boldsymbol{Y}}$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；$\boldsymbol{Y}$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$\mathcal{L}(\hat{\boldsymbol{Y}},\boldsymbol{Y})$的值。

nn.CrossEntropyLoss()函数是类的形式封装，所以需要先对其进行实例化，然后调用(本节介绍的损失函数都是一样，不再赘述)。

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)
```

然后，执行如下语句：

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

结果 loss 的值为 2.4883。

如果按照上述$\mathcal{L}(\hat{\boldsymbol{Y}},\boldsymbol{Y})$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，$\mathcal{L}(\hat{\boldsymbol{Y}},\boldsymbol{Y})$和 nn.CrossEntropyLoss()(pre_y,y)确实是表达相同的含义。

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)
```

结果 pre_y2 的内容如下：

```
tensor([[0.0174, 0.9479, 0.0174, 0.0174],
        [0.3995, 0.1470, 0.0541, 0.3995],
        [0.6103, 0.0826, 0.2245, 0.0826]])
```

(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)
```

结果 pre_y3 的内容变为：

```
tensor([[-4.0535, -0.0535, -4.0535, -4.0535],
        [-0.9176, -1.9176, -2.9176, -0.9176],
        [-0.4938, -2.4938, -1.4938, -2.4938]])
```

经过简单的计算可以知道，这个结果 pre_y3 确实是对 pre_y2 中的每个元素计算自然对数后得到的。

(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，这与上面手工计算的结果是一致的。

也就是说，nn.CrossEntropyLoss()(pre_y,y)语句等同于如下 3 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)。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，对于如下代码：

```
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)
```

执行后，loss 的值为－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。

一般来说，nn.NLLLoss()函数甚少单独使用，往往与 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。

3.3.2 nn.MSELoss()函数

假设张量 $\boldsymbol{Y}=(y^{(1)},y^{(2)},\cdots,y^{(n)})$，$\hat{\boldsymbol{Y}}=(\hat{y}^{(1)},\hat{y}^{(2)},\cdots,\hat{y}^{(n)})$，则 $\boldsymbol{Y}$ 和 $\hat{\boldsymbol{Y}}$ 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 $\boldsymbol{Y}$ 和 $\hat{\boldsymbol{Y}}$ 为一维张量。实际上，对于多维张量的情况，亦可以此类推：分别取出 $\boldsymbol{Y}$ 和 $\hat{\boldsymbol{Y}}$ 中相对应的元素相减，然后平方，最后除以元素个数。显然，要求 $\boldsymbol{Y}$ 和 $\hat{\boldsymbol{Y}}$ 的形状必须相同。

在 PyTorch 框架中，可用 nn.MSELoss()函数来计算 $\boldsymbol{Y}$ 和 $\hat{\boldsymbol{Y}}$ 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。

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
```

执行这些代码后可以发现，loss 的值为 6.75。

均方差损失函数 nn.MSELoss()主要用于回归分析。

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

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

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

观察该公式可以发现，对于 $\boldsymbol{X}$ 中所有属于 0 类的样本（$y^{(i)}=0$），它们的模型输出（$\hat{y}^{(i)}$）越接近 0，$\mathcal{L}(\hat{\boldsymbol{Y}},\boldsymbol{Y})$的值越小；对于所有属于 1 类的样本（$y^{(i)}=1$），它们的模型输出（$\hat{y}^{(i)}$）越接近 1，$\mathcal{L}(\hat{\boldsymbol{Y}},\boldsymbol{Y})$的值越小。因此，通过最小化$\mathcal{L}(\hat{\boldsymbol{Y}},\boldsymbol{Y})$，更新模型中的参数，可以使得模型的输出越来越接近目标值。

在 PyTorch 框架中，nn.BCELoss()函数用来计算上述$\mathcal{L}(\hat{\boldsymbol{Y}},\boldsymbol{Y})$。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。

例如，下列代码先构造张量 pre_y 和 y（两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)
```

执行后，loss 为 0.9025。

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。

实际上，nn.BCEWithLogitsLoss()函数相当于先启用激活函数 torch.sigmoid()，然后调用 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。

3.3.4　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$\hat{\boldsymbol{Y}}=(\hat{y}^{(1)},\hat{y}^{(2)},\cdots,\hat{y}^{(n)})$，标记张量为$\boldsymbol{Y}=(y^{(1)},y^{(2)},\cdots,y^{(n)})$，则这种误差可表示为：

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

$\mathcal{L}(\hat{\boldsymbol{Y}},\boldsymbol{Y})$称为绝对值误差。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)
```

执行后，loss 的值为 4.75。显然，pre_y 和 y 必须有相同的形状，否则无法计算。

如果不想计算绝对值的平均值，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。

显然，nn.L1Loss()函数也主要用于回归分析。

3.4 网络模型的训练与测试

一般情况下，程序员的工作是定义 nn.Module 类的派生类并将神经网络的功能封装在其中。因此，需要实例化派生类后才形成网络模型，进而将数据输入网络模型，以对其进行训练，训练完毕后才能测试，这是一个基本的流程。为此，本节先介绍数据集分割、数据打包的方法，然后再介绍网络模型的训练和测试方法。

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。

例如，对于将例 3.1 中已经读取并已放在张量 $\boldsymbol{X}$ 和 $\boldsymbol{Y}$ 中的数据集，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，可参考如下代码：

```
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。

3.4.2　数据打包

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。

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，则可用如下代码来实现：

```
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。

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，要编写这么多代码才能完成数据打包，就会显得比较烦琐。有没有更简便的方法呢？有！PyTorch 为我们提供了更简便的方法。

例如，为了完成上述代码相同的功能，我们仅需如下两条语句：

```
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；DataLoader()函数则用于完成数据打包功能，其涉及的常用参数如下。

- 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 的整数倍时，如果 drop_last 为 True，则会将多出来而又不足一个数据包的样本丢弃；如果为 False（默认值）则表示按实际剩余的数据打包。

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，运行如下代码：

```
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])
torch.Size([100, 3]) torch.Size([100])
torch.Size([100, 3]) torch.Size([100])
torch.Size([100, 3]) torch.Size([100])
torch.Size([100, 3]) torch.Size([100])
torch.Size([100, 3]) torch.Size([100])
torch.Size([100, 3]) torch.Size([100])
torch.Size([100, 3]) torch.Size([100])
torch.Size([100, 3]) torch.Size([100])
torch.Size([100, 3]) torch.Size([100])
torch.Size([100, 3]) torch.Size([100])
torch.Size([100, 3]) torch.Size([100])
torch.Size([100, 3]) torch.Size([100])
torch.Size([100, 3]) torch.Size([100])
torch.Size([28, 3]) torch.Size([28])
```

注意，DataLoader()函数往往结合 Dataset 类来实现数据打包，这在后面将会逐步接触到。

3.4.3 网络模型的训练方法

把一个已定义的 nn.Module 类的派生类实例化，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，而实例化后得到的网络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，直到误差足够小时，停止训练过程。

网络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 $\boldsymbol{X}$ 及其标记 $\boldsymbol{Y}$，令 model 表示实例化后得到的模型，并记为：

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

$\hat{\boldsymbol{Y}}$表示批量 $\boldsymbol{X}$ 在输入模型 model 后产生的输出。令$\mathcal{L}$表示损失函数，则$\mathcal{L}(\hat{\boldsymbol{Y}},\boldsymbol{Y})$表示模型输出$\hat{\boldsymbol{Y}}$和目标 $\boldsymbol{Y}$ 之间的误差。在 PyTorch 框架中，对于每个 $\boldsymbol{X}$，利用 backward()方法，都可以自动计算$\mathcal{L}(\hat{\boldsymbol{Y}},\boldsymbol{Y})$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。一般情况下，对一个模型的训练要经过若干轮才能收敛。模型的训练过程可用伪代码表示如下：

```
(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
```

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。

3.4.4　梯度累加的训练方法

在网络模型训练过程中，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。

所谓梯度累加方法，就是在训练过程中每次迭代一般只计算梯度并对梯度进行累加，而不是每次都做参数更新；当累加到既定次数后再做参数更新，并对梯度清零。这种训练方法可用伪代码描述如下：

```
(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, X) 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
```

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，因此其效果几乎相当于将批量大小由 $|\boldsymbol{X}|$ 改为 $|\boldsymbol{X}| *$ accu_steps，可以在既定条件下大幅度提升模型的泛化能力。

注意，在运用大批量梯度下降方法时，应适当增加学习率。梯度累加方法是一种变相的大批量梯度下降方法，因此在运用该方法时也应适当增加学习率。

这种基于梯度累加的训练方法的具体例子可参考例 8.6。

3.4.5 学习率衰减在训练中的应用

通过第 2 章的学习我们知道，在网络模型训练过程中，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，降低陷于局部解的概率；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，使得网络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。

例如，我们把例 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。相关代码如下：

```
model3_1 = Model3_1()
optimizer = torch.optim.Adam(model3_1.parameters(), lr=0.01)
lr_list = []                              #保存每次迭代时的学习率
X,Y = X[:400],Y[:400]                     #仅取 400 条数据样本
i = 0
for epoch in range(5):
    for x,y in zip(X,Y):
        x = x.unsqueeze(0)
        pre_y = model3_1(x)
        y = torch.LongTensor([y])
        loss = nn.CrossEntropyLoss()(pre_y,y)
        optimizer.zero_grad()             #梯度清零
        loss.backward()                   #反向计算梯度
        optimizer.step()                  #参数更新

        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
#训练完毕,下面代码用于绘制学习率的变化曲线图
plt.rcParams['font.sans-serif'] = ['SimHei']             #用来正常显示中文标签
plt.plot(range(len(lr_list)),lr_list,c='r')
```

```
plt.xlabel("迭代次数",fontsize=14)          #X 轴标签
plt.ylabel("当前学习率",fontsize=14)        #Y 轴标签
plt.tick_params(labelsize=14)
plt.show()
```

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，最终降到 0.0008。

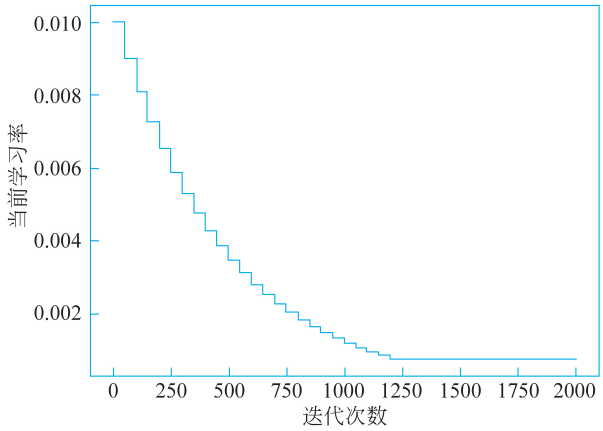

图 3-8　学习率变化曲线图

学习率衰减也可以利用 torch.optim.lr_scheduler.StepLR()方法来实现，编写的代码更为简洁。该方法主要需要设置如下 3 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。

例如，为了实现与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()
```

更改后的完整代码如下：

```
model3_1 = Model3_1()
optimizer = torch.optim.Adam(model3_1.parameters(), lr=0.01)
#设置学习率的衰减方式:
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=50, gamma=0.9)
lr_list = []                                #保存每次迭代时的学习率
X,Y = X[:400],Y[:400]                       #仅取 400 条数据样本
```