第 3 章 深度学习基础

前两章详细介绍了深度学习的发展历史及深度学习环境的安装与配置。本章将正式开始深度学习基础内容的学习。首先详尽介绍深度学习入门的基础知识，涵盖了线性回归和逻辑回归的由来与建模，什么是深度学习及为什么需要深度学习等相关理念，然后介绍梯度下降和反向传播的原理及实现方法。本章将全面介绍如何从零开始构建回归和分类模型，包括逻辑回归到 Softmax 回归的转换，以及常用的评估指标和应对过拟合的方法。此外，本章内容还将深入探讨超参数选择的重要性、交叉验证的实践意义、激活函数的作用，以及在多标签分类场景下的损失函数与模型评估方法。

3.1 线性回归

经过前面预备知识的介绍，现在终于正式进入了深度学习的内容介绍中。那什么是深度学习呢？为什么需要深度学习呢？要想弄清楚这两个问题，还得先从机器学习中的线性回归说起。

3.1.1 理解线性回归模型

通常来讲，我们所学的每个算法都是为了解决某类问题而诞生的。换句话说，也就是在实际情况中的确存在一些问题能够通过线性回归来解决，例如对房价的预测，但是有人可能会问，为什么对于房价的预测就应该用线性回归，而不是用其他算法模型呢？其原因就在于常识告诉我们房价是随着面积的增长而增长的，并且总体上呈线性增长的趋势。那有没有当面积大到一定程度后价格反而降低，而不符合线性增长的呢？这当然也可能存在，但在实际处理中肯定会优先选择线性回归模型，当效果不佳时才会尝试其他算法，因此，当学习过多个算法模型后，在得到某个具体的问题时，可能就需要考虑哪种模型更适合。

例如某市的房价走势如图 3-1 所示，其中横坐标为面积，纵坐标为价格，并且房价整体上呈线性增长的趋势。假如现在随意告诉你一个房屋的面积，要怎样才能预测（或者叫计算）出其对应的价格呢？

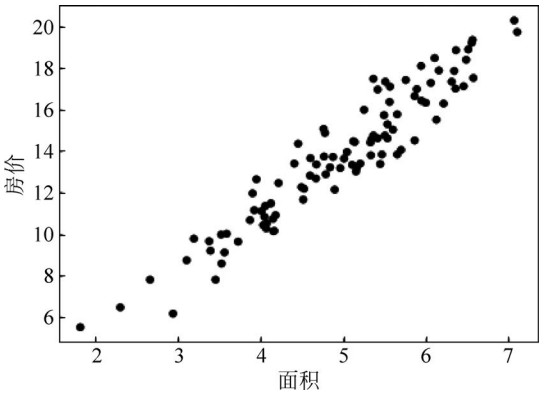


图 3-1　某市的房价走势图

3.1.2　建立线性回归模型

一般来讲，当得到一个实际问题时，首先会根据问题的背景结合常识选择一个合适的模型。同时，现在常识告诉我们房价的增长更优先符合线性回归这类模型，因此可以考虑建立一个如下所示的线性回归模型(Linear Regression)。

$$\hat{y} = h(\boldsymbol{x}) = \boldsymbol{w}\boldsymbol{x} + b \tag{3-1}$$

其中，$\boldsymbol{w}$ 为权重(Weight)，b 为偏置(Bias)或者截距(Intercept)，两者都称为模型参数(Parameter)。当通过某种方法求解得到未知参数 $\boldsymbol{w}$ 和 b 之后，也就意味着得到了这个预测模型，即给定一个房屋面积 $\boldsymbol{x}$，就能够预测出其对应的房价 $\hat{y}$。

注意：在机器学习中所谓的模型，可以简单地理解为一个复合函数。

当然，尽管影响房价的主要因素是面积，但是其他因素同样也可能影响房屋的价格。例如房屋到学校的距离、到医院的距离和到大型商场的距离等，只是各个维度对应的权重大小不同而已。虽然现实生活中一般不这么量化，但是开发商总会拿学区房做卖点，所以这时便有了影响房价的 4 个因素，而在机器学习中将其称为特征(Feature)或者属性(Attribute)，因此，包含多个特征的线性回归就叫作多变量线性回归(Multiple Linear Regression)。

此时，便可以得到如下所示的线性回归模型。

$$\hat{y} = h(\boldsymbol{x}) = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + b = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b \tag{3-2}$$

其中，x_1, x_2, x_3, x_4 表示输入的 4 项房屋信息特征；w_1，w_2, w_3, w_4 表示每个特征对应的权重参数；b 为偏置。

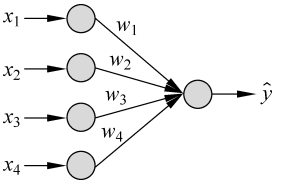


图 3-2　房价预测线性回归结构图(偏置未画出)

并且还可以通过示意图来对式(3-2)中的模型进行表示，如图 3-2 所示。

3.1.3 求解线性回归模型

当建立好一个模型后，自然而然想到的就是如何通过给定的数据，也叫训练集（Training Data），来对模型 $h(\boldsymbol{x})$ 进行求解。在中学时期我们学过如何通过两个坐标点来求解过这两点的直线，可在上述的场景中这种做法显然是行不通的（因为求解线性回归模型所有的点并不在一条直线上），那么有没有什么好的解决办法呢？

此时就需要转换一下思路了，既然不能直接进行求解，那就换一种间接的方式。现在来想象一下，当 $h(\boldsymbol{x})$ 满足一个什么样的条件时，它才能称得上是一个好的 $h(\boldsymbol{x})$？回想一下求解 $h(\boldsymbol{x})$ 的目的是什么，不就是希望输入面积 x 后能够输出“准确”的房价 $\hat{y}$ 吗？既然直接求解 $h(\boldsymbol{x})$ 不好入手，那么就从“准确”来入手。

可又怎样来定义准确呢？在这里，可以通过计算每个样本的真实房价与预测房价之间的均方误差来对“准确”进行刻画。

$$\begin{cases} J(\boldsymbol{w},b)=\dfrac{1}{2m}\displaystyle\sum_{i=1}^{m}(y^{(i)}-\hat{y}^{(i)})^2 \\ \hat{y}^{(i)}=h(\boldsymbol{x}^{(i)})=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}^{(i)}+b \end{cases} \tag{3-3}$$

其中，m 表示样本数量；$\boldsymbol{x}^{(i)}$ 表示第 i 个样本为一个列向量；$\boldsymbol{w}$ 表示模型对应的参数，也为一个列向量；$y^{(i)}$ 表示第 i 个房屋的真实价格；$y^{(i)}$ 表示第 i 个房屋的预测价格。

由式(3-3)可知，当函数 $J(\boldsymbol{w},b)$ 取最小值时的参数 $\hat{\boldsymbol{w}}$ 和 $\hat{b}$ 就是要求的目标参数。为什么？因为当 $J(\boldsymbol{w},b)$ 取最小值时就意味着此时所有样本的预测值与真实值之间的误差（Error）最小。如果极端一点，就是所有预测值都等同于真实值，那么此时的 $J(\boldsymbol{w},b)$ 就是 0 了，因此，对于如何求解模型 $h(\boldsymbol{x})$ 的问题就转换成了如何最小化函数 $J(\boldsymbol{w},b)$ 的问题，而 $J(\boldsymbol{w},b)$ 也有一个专门的术语叫作目标函数（Objective Function）或者代价函数（Cost Function）抑或损失函数（Loss Function）。关于目标函数的求解问题将在 3.2 节内容中进行介绍。

3.1.4 多项式回归建模

3.1.2 节分别介绍了单变量线性回归和多变量线性回归，接下来将开始介绍多项式回归。那么什么是多项式回归呢？现在假定已知矩形的面积公式，而不知道求解梯形的面积公式，并且同时手上有若干类似图 3-3 所示的梯形。已知梯形的上底和下底，并且上底均等于高。现在需要建立一个模型，当任意给定一个类似图 3-3 中的梯形时能近似地算出其面积。面对这样的问题该如何进行建模呢？

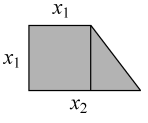


图 3-3 梯形

首先需要明确的是，即使直接建模成类似于 3.1.2 节中的多变量线性回归模型 $h(\boldsymbol{x})=w_1x_1+w_2x_2+b$ 也是可以的，只是效果可能不会太好。现在来分析一下，对于这个梯形，左边可以看成正方形，所以可以人为地构造第 3 个特征 $(x_1)^2$，而整体也可以看成长方形的一部分，则又可以人为地构造出 x_1x_2 这

个特征，最后，整体还可以看成大正方形的一部分，因此还可以构造出$(x_2)^2$这个特征。

根据上述内容可知，建模时除了将x_1，x_2作为特征外，还人为地构造了x_1x_2，x_1^2，x_2^2这3个特征，并且后3个特征也存在着一定意义上的可解释性，因此，对于这个模型也可以通过类似图3-4所示的方式表示。

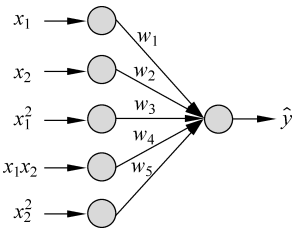


图3-4　梯形面积预测线性回归结构图(偏置未画出)

此时，便可以建立一个如式(3-4)所示的模型

$$h(\boldsymbol{x})=x_1w_1+x_2w_2+(x_1)^2w_3+x_1x_2w_4+(x_2)^2w_5+b \tag{3-4}$$

此时有读者可能会问，式(3-4)中有的部分重复累加了，计算出来的面积岂不大于实际面积吗？这当然不会，因为每项前面都有一个权重参数w_i作为系数，只要这些权重有正有负，就不会出现大于实际面积的情况。同时，可以发现$h(\boldsymbol{x})$中包含了x_1x_2、$(x_1)^2$、$(x_2)^2$这些项，因此将其称为多项式回归(Polynomial Regression)。

但是，只要进行如下替换，便可回到普通的线性回归：

$$h(\boldsymbol{x})=x_1w_1+x_2w_2+x_3w_3+x_4w_4+x_5w_5+b \tag{3-5}$$

其中，$x_3=(x_1)^2$、$x_4=x_1x_2$、$x_5=(x_2)^2$，只是在实际建模时先要将原始两个特征的数据转换为5个特征的数据，同时在正式进行预测时，向模型$h(\boldsymbol{x})$输入的也将是包含5个特征的数据。

3.1.5　从特征输入到特征提取

从图3-4可以看出，尽管使用了5个特征作为线性回归的特征输入进行建模，但是其原始特征依旧只有x_1，x_2这两个，而其余的3个只是人为构造的，其本质就相当于首先以人为的方式对原始输入进行了一次特征提取，然后以提取后的特征来建模。那么既然如此，可不可以通过图3-5所示的结构图来进行建模呢？

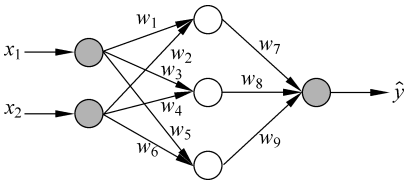


图3-5　梯形面积预测结构图(偏置未画出)

其中，左边的圆圈表示原始的输入特征，中间的圆圈表示对原始特征提取后的特征，右边的圆圈表示最终的预测输出。

通过图3-5可以知道，该结构首先以x_1，x_2为基础进行特征提取得到3个不同的特征，然后以这3个特征来建立了一个线性回归模型进行预测输出，因此，$\hat{y}$可以表示为

$$\begin{aligned}a&=w_1x_1+w_2x_2+b_1\\b&=w_3x_1+w_4x_2+b_2\\c&=w_5x_1+w_6x_2+b_3\\\hat{y}&=w_7a+w_8b+w_9c+b_4\end{aligned} \tag{3-6}$$

以图3-5所示的方式进行建模和以图3-4所示的方式进行建模的差别在哪儿呢？差别有很多，但最大的差别在于构造特征的可解释性。也就是说，人为构造的特征一般具有一定的可解释性，知道每个特征的含义(例如上面的x_1x_2，x_1^2，x_2^2)，而以图3-5中的方式得到的

特征在我们直观看来并不具有可解释性(例如上面 a,b,c 这 3 个特征),因此,在传统的机器学习中还专门有一个分支叫作特征工程,即人为地根据原始特征来构造一系列可解释性的新特征。

说一千道一万,到底能不能用式(3-6)的描述来进行建模呢?很遗憾,并不能。为什么呢?

根据式(3-6)可得

$$\begin{aligned}\hat{y} &= w_7(w_1x_1+w_2x_2+b_1)+w_8(w_3x_1+w_4x_2+b_2)+w_9(w_5x_1+w_6x_2+b_3)+b_4\\ &= (w_1w_7+w_3w_8+w_5w_9)x_1+(w_2w_7+w_4w_8+w_6w_9)x_2+w_7b_1+w_8b_2+w_9b_3+b_4\\ &= \alpha x_1+\beta x_2+\gamma\end{aligned} \tag{3-7}$$

由此可知,根据式(3-7)的描述,建模得到的仍旧只是一个以原始特征 x_1,x_2 为输入的线性回归模型。那么图 3-5 这么好的结构设想难道就这么放弃了?当然不会,图 3-5 的结构并没错,错的是式子(3-6)中的描述。

3.1.6 从线性输入到非线性变换

上文说到,如果以式(3-6)中的描述进行建模,则最终得到的仍旧只是一个简单的线性回归模型,其原因在于,通过式(3-6)得到的 3 个特征 a,b,c 仅是 x_1,x_2 之间在不同权重下的线性组合。也就是说 a,b,c 都是 3 个线性的特征,如果再将其进行线性组合作为输出,则整个模型仍旧只是原始特征的线性组合,并没有增加模型的复杂度。那么该怎么办呢?既然一切都是“线性”的错,那么唯有抛弃“线性”引入非线性才是解决问题的正道,而所谓非线性是指通过一个非线性函数对原始输出进行一次变换。

如式(3-8)所示,只需对 a,b,c 这 3 个特征再进行一次非线性变换,那么整个模型就不可能再被化简为线性了,因此所有问题也将迎刃而解。

$$\begin{aligned}a &= g(w_1x_1+w_2x_2+b_1)\\ b &= g(w_3x_1+w_4x_2+b_2)\\ c &= g(w_5x_1+w_6x_2+b_3)\\ \hat{y} &= w_7a+w_8b+w_9c+b_4\end{aligned} \tag{3-8}$$

其中,$g(\cdot)$为非线性的变换操作,称为激活函数,例如常见的 Sigmoid 函数,这部分内容将在 3.12 节进行介绍。

3.1.7 单层神经网络

经过以上内容的介绍其实已经在不知不觉中将大家带到了深度学习(Deep Learning)的领域中。所谓深度学习是指构建一个多层神经网络(Neural Network),然后进行参数学习的过程,而“深度”只是多层神经网络的一个别称而已,因此,还可以将深度学习称作多层神经网络学习。

线性回归模型就是一个简单的神经网络结构图,如图 3-6 所示,其中每个圆圈表示一个神经元(Neuron),输入层神经元的个数表示数据集的特征维度,输出层神经元的个数表示

输出维度，并且尽管这里有输入层和输出层两层，但是在一般情况下只将含有权重参数的层称为一个网络层，所以要视情况而定，因此线性回归模型是一个单层的神经网络。

同时，需要注意的是在图 3-6 所示的网络结构中，输出层的所有神经元和输入层的所有神经元都是完全连接的。在深度学习中，如果某一层每个神经元的输入都完全依赖于上一层所有神经元的输出，就将该层称作一个全连接层（Fully-connected Layer）或者稠密层（Dense Layer）。例如图 3-6 中的输出层就是一个全连接层。

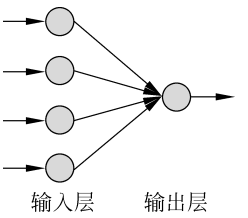


图 3-6　单层神经网络结构图（偏置未画出）

3.1.8　深度神经网络

所谓深度神经网络（Deep Neural Network，DNN）也叫作深度前馈神经网络（Deep Forward Neural Network），一般是指网络层数大于 2 的神经网络，一个简单的深度神经网络如图 3-7 所示，其包含 3 个全连接层。同时，将输入层与输出层之间的所有层都称为隐藏层或隐含层（Hidden Layer）。

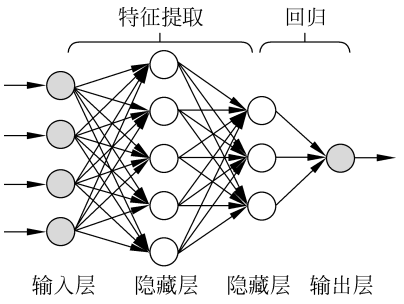


图 3-7　深度神经网络结构图（偏置未画出）

这里值得注意的是，通过上面房价预测和梯形面积预测这两个例子的介绍可以知道，对于输出层之前的所有层都可以将其看成一个特征提取的过程，而且越靠近输出层的隐藏层也就意味着提取的特征越抽象。当原始输入经过多层网络的特征提取后，就可以将提取的特征输入最后一层进行相应操作（分类或者回归等）。

到此，对于什么是深度学习及深度学习的理念就介绍完了，这一点在 3.5 节后半部分内容中还会再次提及。

3.1.9　小结

本节首先以房价预测为例引入了单变量线性回归及如何转换模型的求解思路，然后通过梯形面积预测的实例引入了什么是多项式回归，并进一步引出了抽象特征提取的概念；最后顺理成章地引出了深度学习的概念，即所谓深度学习就是指将原始特征通过多层神经网络进行抽象特征提取，然后将提取的特征输入最后一层进行回归或者分类的处理过程。

3.2　线性回归的简捷实现

经过 3.1 节的介绍对于深度学习的基本概念已经有了一定的了解，接下来将开始介绍如何借助 PyTorch 框架来快速实现上面介绍的房价预测和梯形面积预测这两个实际示例。

3.2.1 PyTorch 使用介绍

在正式介绍模型实现前，先来了解即将用到的 PyTorch 中的相关模型接口的使用方法。

1. nn.Linear()的使用

根据图 3-7 可知，对于每个网络层来讲均是一个全连接层，并且都可以看成由多个线性组合构成。例如对于第 1 个全连接层来讲，其输入维度为原始样本的特征维度数 4，输出维度为 5，即由 5 个线性组合构成了该全连接层。此时，可以通过以下方式来定义该全连接层，示例代码如下：

```
1  import torch.nn as nn
2  layer = nn.Linear(4, 5)
```

在上述代码中，第 1 行表示导入 torch 中的 nn 模块。第 2 行表示定义一个全连接层，并且该全连接层的输入特征(神经元)的数量为 4，输出特征的数量为 5，并且 nn.Linear()内部已经自动随机初始化了网络层对应的权重参数。同理，对于第 2 个全连接层来讲，其定义方式为 nn.Linear(5,3)，因此对于式(3-1)中的单变量线性回归来讲，其定义方式为 nn.Linear(1,1)。

接着，便可以通过以下方式来完成一次全连接层的计算，示例代码如下：

```
1  def test_linear():
2      x = torch.tensor([[1., 2, 3, 4], [4, 5, 6, 7]])
3      layer = nn.Linear(4, 5)
4      y = layer(x)
```

在上述代码中，第 2 行表示定义输入样本，形状为[2,4]列，即样本数量为 2，特征数量为 4。第 4 行则用于计算该全连接层对应的结果，输出形状为[2,5]。

2. nn.Sequential()的使用

此时已经知道了如何定义一个全连接层并完成对应的计算过程，但现在出现的一个问题是图 3-7 中有多个全连接网络层，该如何定义并完成整个计算过程呢？一种最直接的办法就是逐层单独定义并完成相应的计算过程，示例代码如下：

```
1  def multi_layers():
2      x = torch.tensor([[1., 2, 3, 4], [4, 5, 6, 7]])
3      layer1 = nn.Linear(4, 5)
4      layer2 = nn.Linear(5, 3)
5      layer3 = nn.Linear(3, 1)
6      y1 = layer1(x)
7      y2 = layer2(y1)
8      y3 = layer3(y2)
9      print(y3)
```

但这样的写法会略显冗余，因为对于整个计算过程来讲，几乎很少会用到中间结果，因此可以采用省略的写法。在 PyTorch 中，可以将所有的网络层放入一个有序的容器中，然后一次完成整个计算过程，示例代码如下：

```
1  def multi_layers_sequential():
2      x = torch.tensor([[1., 2, 3, 4], [4, 5, 6, 7]],)
3      net = nn.Sequential(nn.Linear(4, 5),
```

```
4                       nn.Linear(5, 3), nn.Linear(3, 1))
5       y = net(x)
6       print(y)
```

在上述代码中，第 3 行中 nn.Sequential()便是这个有序容器，通过它便可以完成整个 3 层网络的计算过程。

3. nn.MSELoss()的使用

根据 3.1.3 节内容可知，在定义好一个模型之后便需要通过最小化对应的损失函数来求解模型对应的权重参数。在此处，可以通过计算预测值与真实值之间的均方误差来构造损失函数。在 PyTorch 中，可以借助 nn.MSELoss()来达到这一目的，示例代码如下：

```
1  def test_loss():
2       y = torch.tensor([1., 2, 3])
3       y_hat = torch.tensor([2., 2, 1])
4       l1 = 0.5 * torch.mean((y - y_hat) ** 2)
5       loss = nn.MSELoss(reduction = 'mean')
6       l2 = loss(y, y_hat)
7       print(l1,l2)
```

在上述代码中，第 2～3 行表示定义真实值和预测值这两个张量。第 4 行表示自行实现式(3-3)中的损失计算。第 5～6 行表示借助 PyTorch 中的 nn.MSELoss()来实现，其中 reduction＝'mean'表示返回均值，而 reduction＝'sum'表示返回和。

在上述代码运行结束后得到的结果如下：

```
tensor(0.8333) tensor(1.6667)
```

可以发现两者并不相等，其原因在于 nn.MSELoss()在计算损失时并没有乘以 0.5 这个系数，不过两者本质上并没有区别。至于式(3-3)中为什么需要乘以 0.5 这个系数将在 3.3.6 节中进行介绍。上述示例代码可参见 Code/Chapter03/C01_OP/main.py 文件。

3.2.2　房价预测实现

在熟悉了 nn.Linear()和 nn.MSELoss()这两个模块的基本使用方法后，再来看如何借助 PyTorch 快速实现房价预测这个线性回归模型。完整示例代码可参见 Code/Chapter03/C02_HousePrice/main.py 文件。

1. 构建数据集

首先需要构造后续用到的数据集，实现代码如下：

```
1  def make_house_data():
2       np.random.seed(20)
3       x = np.random.randn(100, 1) + 5 #面积
4       noise = np.random.randn(100, 1)
5       y = x * 2.8 - noise #价格
6       x = torch.tensor(x, dtype = torch.float32)
7       y = torch.tensor(y, dtype = torch.float32)
8       return x, y
```

在上述代码中，第 2 行为设置一个固定的随机种子，以此来使每次产生的样本保持一

样。第3～5行表示随机生成100个样本点并加入相应的噪声，其中x表示房屋面积，y表示房屋价格。第6～7行表示将其转换为PyTorch中的张量，并且将类型指定为浮点型。第8行表示返回测试数据，两者的形状均为[100,1]。

2. 构建模型

在构建完成数据集之后便需要构造整个单变量线性回归模型，实现代码如下：

```
1  def train(x, y):
2      input_node = x.shape[1]
3      output_node = 1
4      net = nn.Sequential(nn.Linear(input_node, output_node))
5      loss = nn.MSELoss() #定义损失函数
6      optimizer = torch.optim.SGD(net.parameters(), lr=0.003)
7      for epoch in range(40):
8          logits = net(x)
9          l = loss(logits, y)
10         optimizer.zero_grad()
11         l.backward()
12         optimizer.step() #执行梯度下降
13         print("Epoch: {}, loss: {}".format(epoch, l))
14     logits = net(x)
15     return logits.detach().NumPy()
```

在上述代码中，第2～3行表示分别指定模型的输入、输出特征维度，其中x.shape[1]表示数据集的列数(特征维度数)，这里得到的结果也是1。第4行则是先定义一个全连接层，然后将其放入序列容器中。第5行是定义网络的损失函数。第6行用于定义随机梯度下降优化器，以此来求解模型的权重参数，其中net.parameters()表示得到容器中所有网络层对应的参数，lr表示执行梯度下降时的学习率，关于梯度下降算法的具体原理将在3.3节的内容中进行介绍。第7～13行则用于迭代求解网络中的权重参数，并且整个迭代过程在深度学习中将其称为训练(Training)，其中第8～12行也是今后所有模型训练的固定写法，各行代码的具体含义将在3.3节中逐一进行介绍。第14～15行则根据训练完成的模型来对房价进行预测，并同时返回预测后的结果。

3. 可视化结果

在得到模型的预测结果后，便可以借助Matplotlib中的plot模块对其进行可视化，代码如下：

```
1  def visualization(x, y, y_pred=None):
2      plt.xlabel('面积', fontsize=15)
3      plt.ylabel('房价', fontsize=15)
4      plt.scatter(x, y, c='black')
5      plt.plot(x, y_pred)
6      plt.show()
7
8  if __name__ == '__main__':
9      x, y = make_house_data()
10     y_pred = train(x, y)
11     visualization(x, y, y_pred)
```

在上述代码中，第2～3行用于指定横纵坐标的显示标签。第4行用于对原始样本点进

行可视化。第 5 行则用于对预测结果进行可视化。第 6 行表示展示所有的绘制结果。最终可视化的结果如图 3-8 所示。

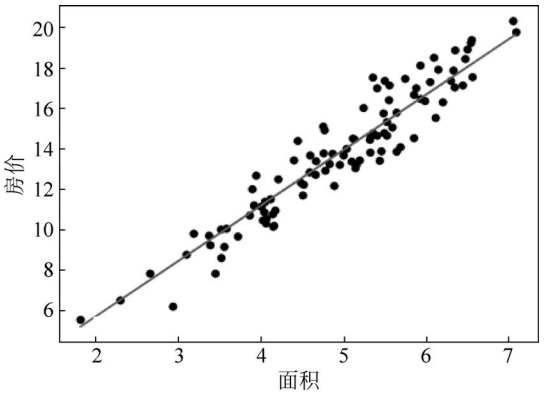


图 3-8　线性回归预测结果图

在图 3-8 中，圆点表示原始样本，直线表示模型根据输入面积预测得到的结果。

3.2.3　梯形面积预测实现

在介绍完上面的线性回归的简洁实现示例后，对于 3.1.4 节中梯形面积预测的实现过程就容易多了。完整示例代码可参见 Code/Chapter03/C03_Trapezoid/main.py 文件。

1. 构建数据集

首先依旧需要构建相应的梯形样本数据集，代码如下：

```
1  def make_trapezoid_data():
2      x1 = np.random.uniform(0.5, 1.5, [50, 1])
3      x2 = np.random.uniform(0.5, 1.5, [50, 1])
4      x = np.hstack((x1, x2))
5      y = 0.5 * (x1 + x2) * x1
6      x = torch.tensor(x, dtype = torch.float32)
7      y = torch.tensor(y, dtype = torch.float32)
8      return x, y
```

在上述代码中，第 2～3 行为根据均匀分布在区间[0.5，1.5]随机生成梯形的上底和下底，并且形状为 50 行 1 列向量。第 4 行表示将两列向量拼接在一起，从而得到一个 50 行 2 列的矩阵。第 5 行表示计算梯形真实的面积。第 6～8 行分别将 x 和 y 转换为 PyTorch 中的张量并返回。

2. 构建模型

在构建完数据集之后便需要图 3-5 中的网络结构来构造整个多层神经网络模型，代码如下：

```
1  def train(x, y):
2      input_node = x.shape[1]
3      losses = []
4      net = nn.Sequential(nn.Linear(input_node,80),
                           nn.Sigmoid(),nn.Linear(80, 1))
5      loss = nn.MSELoss()
```

```
6       optimizer = torch.optim.Adam(net.parameters(), lr = 0.003)
7       for epoch in range(1000):
8           logits = net(x)
9           l = loss(logits, y)
10          optimizer.zero_grad()
11          l.backward()
12          optimizer.step()
13          losses.append(l.item())
14      logits = net(x)
15      l = loss(logits, y)
16      print("真实值:", y[:5].detach().NumPy().reshape(-1))
17      print("预测值:", logits[:5].detach().NumPy().reshape(-1))
18      return losses
```

在上述代码中，第 4 行表示定义整个两层的网络模型，并且同时将隐藏层神经元的个数设定为 80 并加入了 Sigmoid 非线性变换。第 6 行用于定义一个优化器，以此来求解模型参数，关于 Adam 优化器将在 6.9 节内容中进行介绍，简单来讲它就是梯度下降算法的改进版。第 13 行用于对每次迭代后模型的损失值进行保存，其中 item()方法表示将 PyTorch 中的一个标量转换为 Python 中的标量，如果是向量，则需要使用.detach().NumPy()方法进行转换。第 16～17 行用于分别输出前 5 个真实值和预测值。

上述代码运行结束后得到的结果如下：

```
1  真实值: [1.263554 1.611813 1.857847 1.723620 0.488184]
2  预测值: [1.262243 1.620144 1.855083 1.728053 0.503922]
```

从输出结果可以看出，模型的预测结果和真实值已经非常接近了。

最后，还可以对网络在训练过程中保存的损失值进行可视化，如图 3-9 所示。

从图 3-9 可以看出，模型大约在迭代 800 次之后便进行入了收敛阶段。

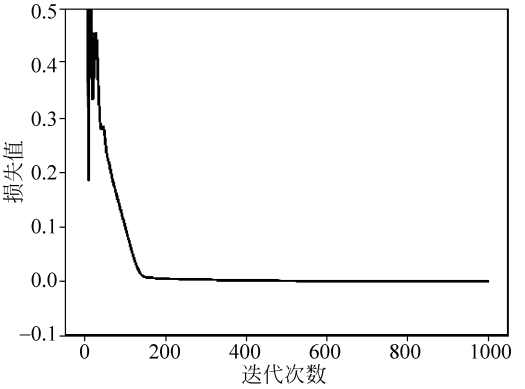


图 3-9　梯形面积预测损失图

3.2.4　小结

本节首先介绍了 PyTorch 框架中 nn.Linear()、nn.Sequential()和 nn.MSELoss()这 3 个模块的原理与应用，然后介绍了如何借助 PyTorch 来快速实现单变量线性回归模型及可

视化最终的预测结果；最后介绍了多项式回归的简单实现过程，并对训练过程中模型的损失变化进行可视化展示。

3.3　梯度下降与反向传播

根据 3.1.3 节可知，求解网络模型参数的过程便等价于最小化目标函数 $J(\boldsymbol{w},b)$ 的过程。同时，经过 3.2 节的介绍已经知道了如何借助 PyTorch 中的优化器来求解网络模型对应的权重参数，不过对于整个求解过程的具体原理并没有介绍。在 3.2 节中，当定义好损失函数后直接通过两行代码便完成了模型权重参数的优化求解过程，一句是 l.backward()，而另一句则是 optimizer.step()。那这两句代码又是什么意思呢？

本节将会详细介绍如何通过梯度下降算法来最小化目标函数 $J(\boldsymbol{w},b)$，以及深度学习中求解网络参数梯度的利器——反向传播(Back Propagation)算法。

3.3.1　梯度下降引例

根据上面的介绍可以知道，梯度下降算法的目的是用来最小化目标函数，也就是说梯度下降算法是一个求解的工具。当目标函数取到(或接近)全局最小值时，也就得到了模型所对应的参数。不过什么是梯度下降(Gradient Descent)呢？如图 3-10 所示，假设有一个山谷，并且你此时处于位置 A 处，那么请问以什么样的方向(角度)往前跳，才能最快地到达谷底 B 处呢？

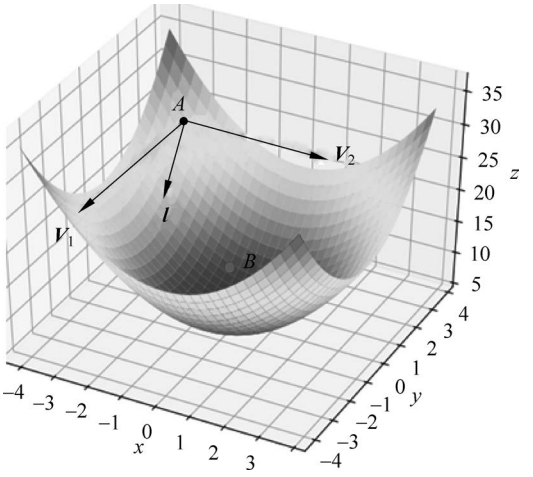


图 3-10　跳跃方向

现在大致有 3 个方向可以选择，沿着 y 轴的 $\boldsymbol{V}_1$ 方向，沿着 x 轴的 $\boldsymbol{V}_2$ 方向及沿着两者间的 $\boldsymbol{l}$ 方向，其实不用问，各位读者一定都会选择 $\boldsymbol{l}$ 所在的方向往前跳第 1 步，然后接着选类似的方向往前跳第 2 步，直到谷底。可为什么都应该这样选呢？答：这还用问，一看就知，不信自己试一试。

3.3.2 方向导数与梯度

由一元函数导数的相关知识可知，函数 $f(x)$ 在 x_0 处的导数反映的是 $f(x)$ 在 $x=x_0$ 处时的变化率；$|f'(x_0)|$ 越大，也就意味着 $f(x)$ 在该处的变化率越大，即移动 Δx 后产生的函数增量 Δy 越大。同理，在二元函数 $z=f(x,y)$ 中，为了寻找 z 在 A 处的最大变化率，就应该计算函数 z 在该点的方向导数

$$\frac{\partial f}{\partial \boldsymbol{l}}=\left\{\frac{\partial f}{\partial x},\frac{\partial f}{\partial y}\right\}\cdot\{\cos\alpha,\cos\beta\}=|\ \mathrm{grad}f\ |\cdot|\ \boldsymbol{l}\ |\cdot\cos\theta \tag{3-9}$$

其中，$\boldsymbol{l}$ 为单位向量；α 和 β 分别为 $\boldsymbol{l}$ 与 x 轴和 y 轴的夹角；θ 为梯度方向与 $\boldsymbol{l}$ 的夹角。

根据式(3-9)可知，要想方向导数取得最大值，那么 θ 必须为 0。由此可知，只有当某点方向导数的方向与梯度的方向一致时，方向导数在该点才会取得最大的变化率。

在图 3-10 中，已知 $z=x^2+y^2+5$，A 的坐标为(−3,3,23)，则 $\partial z/\partial x=2x$，$\partial z/\partial y=2y$。由此可知，此时在点 A 处梯度的方向为(−6,6)，所以当站在 A 点并沿各个方向往前跳跃同样大小的距离时，只有沿着($\sqrt{2}/2$,$-\sqrt{2}/2$)这个方向(进行了单位化，并且同时取了相反方向，因为这里需要的是负增量)才会产生最大的函数增量 Δz。

如果想每次都能以最快的速度下降，则每次都必须向着梯度的反方向向前跳跃，如图 3-11 所示。

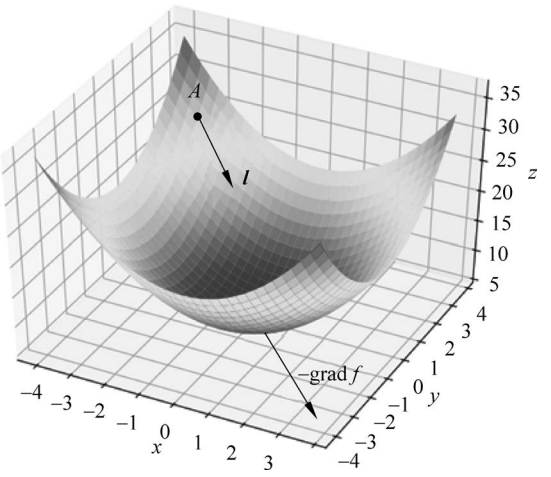


图 3-11 负梯度方向

3.3.3 梯度下降原理

介绍这么多总算把梯度的概念讲清楚了，那么如何用具体的数学表达式进行描述呢？为了方便后面的表述及将读者带入一个真实求解的过程中，这里先将图 3-10 中的字母替换成模型中的参数进行表述。现在有一个模型的目标函数 $J(w_1,w_2)=w_1^2+w_2^2+2w_2+5$(为了方便可视化，此处省略了参数 b，原理都一样)，其中 w_1 和 w_2 为待求解的权重参数，

并且随机将点 A 初始化为初始权重值。下面就一步步地通过梯度下降算法进行求解。

设初始点 $A=(w_1,w_2)=(-2,3)$，则此时 $J(-2,3)=24$，并且点 A 第 1 次往前跳的方向为 $-\text{grad}J=-(2w_1,2w_2+2)=(1,-2)$，即$(1,-2)$这个方向，如图 3-12 所示。

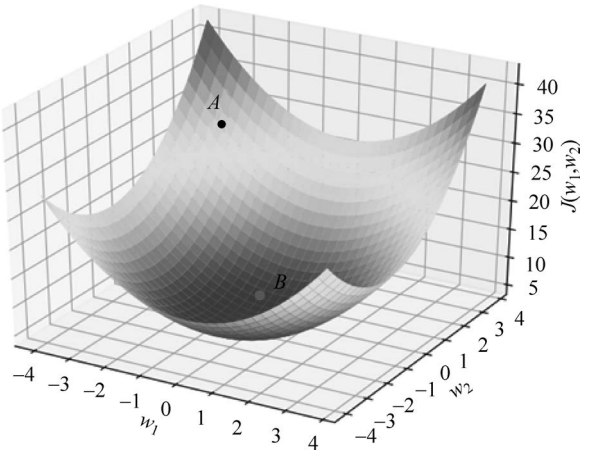


图 3-12　梯度下降

OQ 为平面上梯度的反方向，AP 为其平移后的方向，但是长度为之前的 α 倍，如图 3-13 所示，因此，根据梯度下降的原则，此时曲面上的 A 点就该沿着其梯度的反方向跳跃，而投影到平面则为 A 应该沿着 AP 的方向移动。假定曲面上从 A 点跳跃到了 P 点，那么对应在投影平面上就是图 3-13 中的 AP 部分，同时权重参数也从 A 的位置更新到了 P 点的位置。

从图 3-13 可以看出，向量 $\boldsymbol{AP}$、$\boldsymbol{OA}$ 和 $\boldsymbol{OP}$ 三者的关系为

$$\boldsymbol{OP}=\boldsymbol{OA}-\boldsymbol{PA} \tag{3-10}$$

可以将式(3-10)改写成

$$\boldsymbol{OP}=\boldsymbol{OA}-\alpha\cdot\text{grad}J \tag{3-11}$$

又由于 $\boldsymbol{OP}$ 和 $\boldsymbol{OA}$ 本质上就是权重参数 w_1 和 w_2 更新后与更新前的值，所以便可以得出梯度下降的更新公式为

$$\boldsymbol{w}=\boldsymbol{w}-\alpha\cdot\frac{\partial J}{\partial \boldsymbol{w}} \tag{3-12}$$

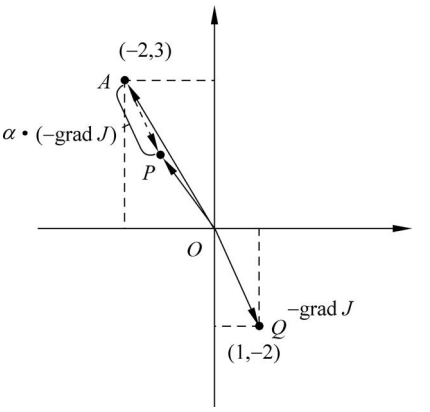


图 3-13　梯度计算

其中，$\boldsymbol{w}=(w_1,w_2)$，$\partial J/\partial \boldsymbol{w}$ 为权重的梯度方向；α 为步长，用来放缩每次向前跳跃的距离，即优化器中的学习率(Learning Rate)参数。

根据式(3-12)可以得出，对于待优化求解的目标函数 $J(\boldsymbol{w})$来讲，如果需要通过梯度下降算法来进行求解，则首先需要做的便是得到目标函数关于未知参数的梯度，即$\partial J/\partial \boldsymbol{w}$。各位读者一定要记住这一点，在 3.7 节内容中也将会再次提及。

将式(3-12)代入具体数值后可以得出曲面上的点 A 在第 1 次跳跃后的着落点为

$$w_1 = w_1 - 0.1 \times 2 \times w_1 = -2 - 0.1 \times 2 \times (-2) = -1.6$$
$$w_2 = w_2 - 0.1 \times (2 \times w_2 + 2) = 3 - 0.1 \times (2 \times 3 + 2) = 2.2 \quad (3\text{-}13)$$

此时，权重参数便从(−2,3)更新为(−1.6,2.2)。当然其目标函数 $J(w_1, w_2)$ 也从 24 更新为 16.8。至此，我们便详细地完成了 1 轮梯度下降的计算。当 A 跳跃到 P 之后，又可以再次利用梯度下降算法进行跳跃，直到跳到谷底(或附近)为止，如图 3-14 所示。

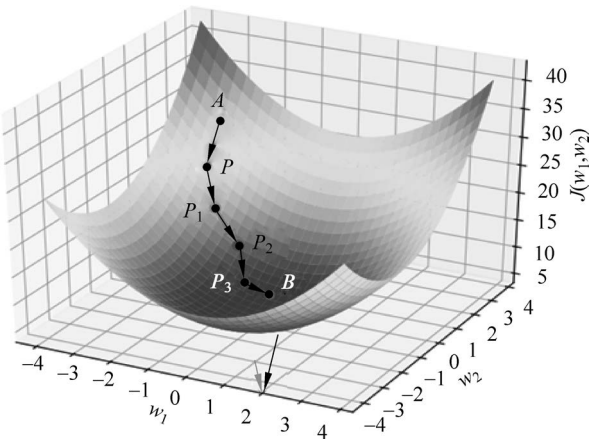


图 3-14 梯度下降

到此可以发现，利用梯度下降算法来最小化目标函数是一个循环迭代的过程。

最后，根据上述原理，还可以通过实际的代码将整个过程展示出来，完整的代码见 Code/Chapter03/C04_GradientDescent/main.py 文件，梯度下降的核心代码如下：

```
1  def compute_gradient(w1, w2):
2      return [2 * w1, 2 * w2 + 2]
3
4  def gradient_descent():
5      w1, w2 = -2, 3
6      jump_points = [[w1, w2]]
7      costs,step = [cost_function(w1, w2)],0.1
8      print("P:({},{})".format(w1, w2), end = ' ')
9      for i in range(20):
10         gradients = compute_gradient(w1, w2)
11         w1 = w1 - step * gradients[0]
12         w2 = w2 - step * gradients[1]
13         jump_points.append([w1, w2])
14         costs.append(cost_function(w1, w2))
15         print("P{}:({},{})".format(i + 1,round(w1,3),round(w2,3)),end = ' ')
16     return jump_points, costs
```

在上述代码中，第 1～2 行用于返回目标函数关于参数的梯度。第 5～6 行用于初始化起点。第 7 行用于计算初始损失值并且将学习率定义为 0.1，它决定了每次向前跳跃时的缩放尺度。第 9～15 行则用于迭代整个梯度下降过程，迭代次数为 20 次，其中第 11～12 行用于执行式(3-12)中的计算过程。第 16 行则用于返回最后得到的计算结果。

上述代码运行结束后便可以得到如下所示的结果：

```
P:(-2,3) P1:(-1.6,2.2) P2:(-1.28,1.56) P3:(-1.024,1.048) P4:(-0.819,0.638) P5:(-0.655,
0.311) P6:(-0.524,0.049) P7:(-0.419,-0.161) P8:(-0.336,-0.329) P9:(-0.268,-0.463)
P10:(-0.215,-0.571).....
```

通过上述代码便可以详细地展示跳向谷底时每次的落脚点，并且可以看到谷底的位置就在(－0.023，－0.954)附近，如图 3-15 所示。

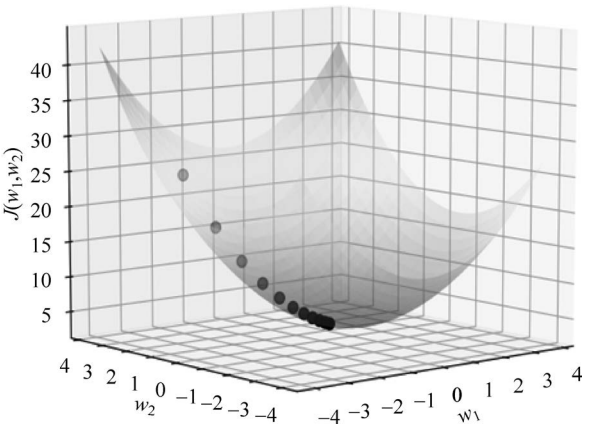


图 3-15　梯度下降可视化

至此，我们就介绍完了如何通过编码实现梯度下降算法的求解过程，等后续再来自己编码，从而从零完成网络模型的参数求解过程。

3.3.4　前向传播过程

在具体介绍网络的训练过程前，先来介绍网络训练结束后的整个预测过程。假定现在有如图 3-16 所示的一个网络结构图。

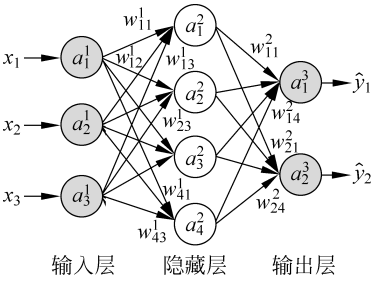


图 3-16　网络结构图

此时定义：L 表示神经网络总共包含的层数，S_l 表示第 l 层的神经元数目，K 表示输出层的神经元数目，w_{ij}^{l} 表示第 l 层第 j 个神经元与第 $l+1$ 层第 i 个神经元之间的权重值。

此时对于图 3-16 所示的网络结构来讲，$L=3$，$S_1=3$，$S_2=4$，$S_3=K=2$，a_i^l 表示第 l 层第 i 个神经元的激活值(输入层 $a_i^1=x_i$，输出层 $a_i^L=\hat{y}_i$)，b_i^l 表示第 l 层的第 i 个偏置(未画出)。

根据图 3-16 所示的网络结构图，当输入 1 个样本对其进行预测时，网络第 1 层的计算过程可以表示成如下形式。

$$z_1^2=a_1^1w_{11}^1+a_2^1w_{12}^1+a_3^1w_{13}^1+b_1^1$$

$$z_2^2=a_1^1w_{21}^1+a_2^1w_{22}^1+a_3^1w_{23}^1+b_2^1$$

$$z_3^2=a_1^1w_{31}^1+a_2^1w_{32}^1+a_3^1w_{33}^1+b_3^1$$

$$z_4^2 = a_1^1 w_{41}^1 + a_2^1 w_{42}^1 + a_3^1 w_{43}^1 + b_4^1 \tag{3-14}$$

如果是将其以矩阵的形式进行表示，则式(3-14)可以改写为

$$\begin{bmatrix} z_1^2 \\ z_2^2 \\ z_3^2 \\ z_4^2 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} a_1^1 & a_2^1 & a_3^1 \end{bmatrix}_{1\times 3} \times \begin{bmatrix} w_{11}^1 & w_{21}^1 & w_{31}^1 & w_{41}^1 \\ w_{12}^1 & w_{22}^1 & w_{32}^1 & w_{42}^1 \\ w_{13}^1 & w_{23}^1 & w_{33}^1 & w_{43}^1 \end{bmatrix}_{3\times 4} + \begin{bmatrix} b_1^1 \\ b_2^1 \\ b_3^1 \\ b_4^1 \end{bmatrix}^{\mathrm{T}} \tag{3-15}$$

将式(3-15)中的形式进行简化可以得出

$$\boldsymbol{z}^2 = \boldsymbol{a}^1 \boldsymbol{w}^1 + \boldsymbol{b}^1 \Rightarrow \boldsymbol{a}^2 = f(\boldsymbol{z}^2) \tag{3-16}$$

其中，$f(\cdot)$表示激活函数，如 Sigmoid 函数等。

同理对于第 2 层来讲有

$$\boldsymbol{z}^3 = \boldsymbol{a}^2 \boldsymbol{w}^2 + \boldsymbol{b}^2 \Rightarrow \boldsymbol{a}^3 = f(\boldsymbol{z}^3) \tag{3-17}$$

现在如果用一个通式对(3-17)进行表示，则为

$$\begin{aligned} z_i^{l+1} &= a_1^l w_{i1}^l + a_2^l w_{i2}^l + \cdots + a_{S_l}^l w_{iS_l}^l + b^l \\ \boldsymbol{z}^{l+1} &= \boldsymbol{a}^l \boldsymbol{w}^l + b^l \\ \boldsymbol{a}^{l+1} &= f(\boldsymbol{z}^{l+1}) \end{aligned} \tag{3-18}$$

由此可以发现，上述整个计算过程，输入输出是根据从左到右按序计算而得到的，因此，整个计算过程又被形象地叫作正向传播(Forward Propagation)或者前向传播。

现在已经知道了什么是正向传播过程，即当训练得到权重参数 $\boldsymbol{w}$ 之后便可以使用正向传播来进行预测了。进一步，再来看如何求解目标函数关于权重参数的梯度，以便通过梯度下降算法求解网络参数。

3.3.5 传统方式梯度求解

根据 3.3.3 节可知，使用梯度下降求解模型参数的前提便是需要知道损失函数 J 关于权重的梯度，也就是要求得 J 关于模型中每个参数的偏导数。以图 3-16 所示的网络结构为例，假设网络的目标函数为均方误差损失，并且同时只考虑一个样本，即

$$J(\boldsymbol{w}, b) = \frac{1}{2}(y - \hat{y})^2 \tag{3-19}$$

其中，$\boldsymbol{w}$ 表示整个网络中的所有权重参数；b 表示所有的偏置；$\hat{y} = \boldsymbol{a}^3$。

由此根据图 3-16 可以发现，如果 J 对 w_{11}^1 求导，则 J 是关于 $\boldsymbol{a}^3$ 的函数，$\boldsymbol{a}^3$ 是关于 $\boldsymbol{z}^3$ 的函数，$\boldsymbol{z}^3$ 是关于 $\boldsymbol{a}^2$ 的函数，$\boldsymbol{a}^2$ 是关于 $\boldsymbol{z}^2$ 的函数，$\boldsymbol{z}^2$ 是关于 $\boldsymbol{w}_{11}^1$ 的函数，所以根据链式求导法则有

$$\frac{\partial J}{\partial w_{11}^1} = \frac{\partial J}{\partial a_1^3}\frac{\partial a_1^3}{\partial z_1^3}\frac{\partial z_1^3}{\partial a_1^2}\frac{\partial a^2}{\partial z_1^2}\frac{\partial z_1^2}{\partial w_{11}^1} + \frac{\partial J}{\partial a_2^3}\frac{\partial a_2^3}{\partial z_2^3}\frac{\partial z_2^3}{\partial a_1^2}\frac{\partial a^2}{\partial z_1^2}\frac{\partial z_1^2}{\partial w_{11}^1}$$

$$\frac{\partial J}{\partial w_{12}^{1}}=\frac{\partial J}{\partial a_{1}^{3}}\frac{\partial a_{1}^{3}}{\partial z_{1}^{3}}\frac{\partial z_{1}^{3}}{\partial a_{1}^{2}}\frac{\partial a^{2}}{\partial z_{1}^{2}}\frac{\partial z_{1}^{2}}{\partial w_{12}^{1}}+\frac{\partial J}{\partial a_{2}^{3}}\frac{\partial a_{2}^{3}}{\partial z_{2}^{3}}\frac{\partial z_{2}^{3}}{\partial a_{1}^{2}}\frac{\partial a^{2}}{\partial z_{1}^{2}}\frac{\partial z_{1}^{2}}{\partial w_{12}^{1}}$$

$$\vdots$$

$$\frac{\partial J}{\partial w_{22}^{2}}=\frac{\partial J}{\partial a_{2}^{3}}\frac{\partial a_{2}^{3}}{\partial z_{2}^{3}}\frac{\partial z_{2}^{3}}{\partial w_{22}^{2}} \tag{3-20}$$

根据式(3-20)可以发现，当目标函数 J 对第 2 层的参数(如 w_{22}^{2})求导还相对不太麻烦，但当 J 对第 1 层的参数进行求导时，就做了很多重复计算，并且这还是网络相对简单的时候，对于深度学习中动辄几十，甚至上百层的网络参数，这个过程便无从下手。显然这种求解梯度的方式非常低效，是不可取的，这也是神经网络在经过一段时间后发展缓慢的原因。

3.3.6　反向传播过程

由式(3-20)中的第 1 行可知，可以将其整理成如下形式

$$\frac{\partial J}{\partial w_{11}^{1}}=\left(\frac{\partial J}{\partial a_{1}^{3}}\frac{\partial a_{1}^{3}}{\partial z_{1}^{3}}\frac{\partial z_{1}^{3}}{\partial a_{1}^{2}}\frac{\partial a^{2}}{\partial z_{1}^{2}}\right)\frac{\partial z_{1}^{2}}{\partial w_{11}^{1}}+\left(\frac{\partial J}{\partial a_{2}^{3}}\frac{\partial a_{2}^{3}}{\partial z_{2}^{3}}\frac{\partial z_{2}^{3}}{\partial a_{1}^{2}}\frac{\partial a^{2}}{\partial z_{1}^{2}}\right)\frac{\partial z_{1}^{2}}{\partial w_{11}^{1}} \tag{3-21}$$

从式(3-21)可以看出，不管是从哪一条路径过来，在对 w_{11}^{1} 求导之前都会先到达 z_{1}^{2}，即先对 z_{1}^{2} 求导之后，才会有$\partial z_{1}^{2}/\partial w_{11}^{1}$。由此可以得出，不管之前经过什么样的路径到达 w_{ij}^{l}，在对连接第 l 层第 j 个神经元与第 $l+1$ 层第 i 个神经元的参数 w_{ij}^{l} 求导之前，肯定会先对 z_{i}^{l+1} 求导，因此，对任意参数的求导过程可以改写为

$$\frac{\partial J}{\partial w_{ij}^{l}}=\frac{\partial J}{\partial z_{i}^{l+1}}\frac{\partial z_{i}^{l+1}}{\partial w_{ij}^{l}}=\frac{\partial J}{\partial z_{i}^{l+1}}a_{j}^{l} \tag{3-22}$$

例如

$$\frac{\partial J}{\partial w_{11}^{1}}=\frac{\partial J}{\partial z_{1}^{1+1}}\frac{\partial z_{1}^{1+1}}{\partial w_{11}^{1}}=\frac{\partial J}{\partial z_{1}^{2}}\frac{\partial z_{1}^{2}}{\partial w_{11}^{1}}=\frac{\partial J}{\partial z_{1}^{2}}a_{1}^{1} \tag{3-23}$$

所以，现在的问题变成了如何快速求解式(3-22)中的$\partial J/\partial z_{i}^{l+1}$ 部分。

从图 3-16 所示的网络结构可以看出，目标函数 J 对任意 z_{i}^{l} 求导时，求导路径必定会经过第 $l+1$ 层的所有神经元，于是结合式(3-18)有

$$\begin{aligned}\frac{\partial J}{\partial z_{i}^{l}}&=\frac{\partial J}{\partial z_{1}^{l+1}}\frac{\partial z_{1}^{l+1}}{\partial z_{i}^{l}}+\frac{\partial J}{\partial z_{2}^{l+1}}\frac{\partial z_{2}^{l+1}}{\partial z_{i}^{l}}+\cdots+\frac{\partial J}{\partial z_{S_{l+1}}^{l+1}}\frac{\partial z_{S_{l+1}}^{l+1}}{\partial z_{i}^{l}}\\&=\sum_{k=1}^{S_{l+1}}\frac{\partial J}{\partial z_{k}^{l+1}}\frac{\partial z_{k}^{l+1}}{\partial z_{i}^{l}}\\&=\sum_{k=1}^{S_{l+1}}\frac{\partial J}{\partial z_{k}^{l+1}}\frac{\partial}{\partial z_{i}^{l}}(a_{1}^{l}w_{k1}^{l}+a_{2}^{l}w_{k2}^{l}+\cdots+a_{S_{l}}^{l}w_{kS_{l}}^{l}+b^{l})\\&=\sum_{k=1}^{S_{l+1}}\frac{\partial J}{\partial z_{k}^{l+1}}\frac{\partial}{\partial z_{i}^{l}}\sum_{j=1}^{S_{l}}a_{j}^{l}w_{kj}^{l}\end{aligned}$$

$$=\sum_{k=1}^{S_{l+1}}\frac{\partial J}{\partial z_k^{l+1}}\frac{\partial}{\partial z_i^l}\sum_{j=1}^{S_l}f(z_j^l)w_{kj}^l$$

$$=\sum_{k=1}^{S_{l+1}}\frac{\partial J}{\partial z_k^{l+1}}f'(z_i^l)w_{ki}^l \tag{3-24}$$

于是此时有

$$\frac{\partial J}{\partial z_i^l}=\sum_{k=1}^{S_{l+1}}\frac{\partial J}{\partial z_k^{l+1}}\cdot f'(z_i^l)w_{ki}^l \tag{3-25}$$

根据式(3-25)可以推导得出

$$\frac{\partial J}{\partial z_i^{l+1}}=\sum_{k=1}^{S_{l+2}}\frac{\partial J}{\partial z_k^{l+2}}\cdot f'(z_i^{l+1})w_{ki}^{l+1} \tag{3-26}$$

为了便于书写和观察规律，引入一个中间变量 $\delta_i^l=\frac{\partial J}{\partial z_i^l}$，则式(3-24)可以重新写为

$$\delta_i^l=\frac{\partial J}{\partial z_i^l}=\sum_{k=1}^{S_{l+1}}\delta_k^{l+1}f'(z_i^l)w_{ki}^l,(l\leqslant L-1) \tag{3-27}$$

需要注意的是，之所以要 $l\leqslant L-1$，是因为由式(3-24)的推导过程可知，l 最大只能取 $L-1$，因为第 L 层后面没有网络层了。

所以，当以均方误差为损失函数时有

$$\delta_i^L=\frac{\partial J}{\partial z_i^L}=\frac{\partial}{\partial z_i^L}\frac{1}{2}\sum_{k=1}^{S_L}(\hat{y}_k-y_k)^2$$

$$=\frac{\partial}{\partial z_i^L}\frac{1}{2}\sum_{k=1}^{S_L}(f(z_k^L)-y_k)^2$$

$$=[f(z_i^L)-y_i]f'(z_i^L)$$

$$=[a_i^L-y_i]f'(z_i^L) \tag{3-28}$$

根据式(3-28)可以看出，均方误差损失函数前面乘以 0.5 的目的便是在求导时能消除平方项，使整个式子看起来更简洁。

同时将式(3-27)代入式(3-22)可得

$$\frac{\partial J}{\partial w_{ij}^l}=\delta_i^{l+1}a_j^l \tag{3-29}$$

通过上面的所有推导，由此可以得到如下 4 个迭代公式

$$\frac{\partial J}{\partial w_{ij}^l}=\delta_i^{l+1}a_j^l \tag{3-30}$$

$$\frac{\partial J}{\partial b_i^l}=\delta_i^{l+1} \tag{3-31}$$

$$\delta_i^l=\frac{\partial J}{\partial z_i^l}=\sum_{k=1}^{S_{l+1}}\delta_k^{l+1}f'(z_i^l)w_{ki}^l,(0<l\leqslant L-1) \tag{3-32}$$

$$\delta_i^L = [a_i^L - y_i] f'(z_i^L) \tag{3-33}$$

这里 δ_i^L 的结果只是针对损失函数为均方误差时的情况，如果采用其他损失函数，则需根据类似式(3-28)的形式重新推导。

且式(3-30)～式(3-33)经过向量化后的形式为

$$\frac{\partial J}{\partial \boldsymbol{w}^l} = (\boldsymbol{a}^l)^{\mathrm{T}} \otimes \boldsymbol{\delta}^{l+1} \tag{3-34}$$

$$\frac{\partial J}{\partial \boldsymbol{b}^l} = \boldsymbol{\delta}^{l+1} \tag{3-35}$$

$$\boldsymbol{\delta}^l = \boldsymbol{\delta}^{l+1} \otimes (\boldsymbol{w}^l)^{\mathrm{T}} \odot f'(\boldsymbol{z}^l) \tag{3-36}$$

$$\boldsymbol{\delta}^L = [\boldsymbol{a}^L - \boldsymbol{y}] \odot f'(\boldsymbol{z}^L) \tag{3-37}$$

其中，⊗表示矩阵乘法，⊙表示按位乘操作。

由式(3-34)～式(3-37)分析可知，欲求 J 对 w^l 的导数，必先知道$\boldsymbol{\delta}^{l+1}$，而欲知$\boldsymbol{\delta}^{l+1}$，必先求$\boldsymbol{\delta}^{l+2}$，以此类推。由此可知，对于整个求导过程，一定是先求$\boldsymbol{\delta}^L$，再求$\boldsymbol{\delta}^{L-1}$，一直到$\boldsymbol{\delta}^2$，因此，对于图 3-16 这样一个网络结构，整个梯度求解过程为先根据式(3-37)求解得到$\boldsymbol{\delta}^3$，然后根据式(3-34)和式(3-35)分别求得$\partial J/\partial \boldsymbol{w}^2$ 和$\partial J/\partial \boldsymbol{b}^2$ 的结果；接着根据式(3-36)并依赖 $\boldsymbol{\delta}^3$ 求解得到$\boldsymbol{\delta}^2$ 的结果；最后根据式(3-34)和式(3-35)分别求得$\partial J/\partial \boldsymbol{w}^1$ 和$\partial J/\partial \boldsymbol{b}^1$ 的结果。

此时，终于发现了这么一个不争的事实：①最先求解出偏导数的参数一定位于第 $L-1$ 层(如此处的 $\boldsymbol{w}^2$)；②要想求解第 l 层参数的偏导数，一定会用到第 $l+1$ 层的中间变量$\boldsymbol{\delta}^{l+1}$(如此处求解 $\boldsymbol{w}^1$ 的导数，用到了$\boldsymbol{\delta}^2$)；③整个过程是从右往左依次进行的，所以整个从右到左的计算过程又被形象地称为反向传播(Back Propagation)，并且$\boldsymbol{\delta}^l$ 被称为第 l 层的残差(Residual)。

在通过整个反向传播过程计算得到所有权重参数的梯度后，便可以根据式(3-12)中的梯度下降算法对参数进行更新，而这两个计算过程对应的便是本节内容一开始所提到的 l.backward()和 optimizer.step()这两个操作。

3.3.7　梯度消失和梯度爆炸

当然，也正是由于反向传播这一叠加累乘的计算特性为深度神经网络的训练过程埋下了两个潜在的隐患——梯度爆炸(Gradient Exploding)和梯度消失(Gradient Vanishing)。

对于梯度爆炸来讲通常是指模型在训练过程中网络的某一层或几层的梯度值过大，使梯度在反向传播时由于累乘的作用使越是靠近输入层的梯度越大，甚至超过了计算机能够处理的范围，从而导致模型的参数得不到更新。梯度爆炸通常是由于神经网络中存在的数值计算问题所导致的，例如网络的参数初始化不当、学习率设置过大等。为了避免产生梯度爆炸问题，常见的方法有使用合适的参数初始化方法、调整学习率大小、使用梯度裁剪(参见 6.2 节)等。

对于梯度消失来讲则恰好与梯度爆炸相反，它是由于网络中的某一层或几层的梯度值过小，在梯度连续累乘的作用下将会得到一个非常小的梯度值，从而导致模型的参数无法得到有效更新。出现梯度消失的原因一般有参数初始化不当、使用不合适的激活函数及网络结构设计不合理等，常见的处理方法有选择合适的激活函数(参见 3.12 节)、使用批量归一化(参见 6.3 节)或者参数初始化方法(参见 6.10 节)等。

3.3.8 小结

本节首先通过一个跳跃的例子详细地向大家介绍了什么是梯度，以及为什么要沿着梯度的反方向进行跳跃才能最快地到达谷底，然后通过图示导出了梯度下降的更新迭代公式；接着详细介绍了网络模型的前向传播过程和反向传播过程，并推导了整个梯度的求解过程；最后，还介绍了梯度消失和梯度爆炸这两种深度学习模型训练时常见的问题，并列出了几种可行的解决方案。

这里，需要再次强调的是，梯度下降算法是用来最小化目标函数求解网络参数，但使用梯度下降算法的前提是要知道目标函数关于所有参数相应的梯度，而反向传播算法正是一种高效求解梯度的工具，千万不要把两者混为一谈。

3.4 从零实现回归模型

经过 3.3 节的介绍，已经清楚了神经网络训练的基本流程，即先进行正向传播并计算预测值，然后进行反向传播并计算梯度，接着根据梯度下降算法对网络中的权重参数进行更新，最后循环迭代这 3 个步骤，直到损失函数收敛为止。在接下来的内容中，将会详细介绍如何从零实现 3.2.3 节中的梯形面积预测实例，即一个简单的两层神经网络。

3.4.1 网络结构

在正式介绍实现部分之前，先来看整个模型的网络结构及整理出前向传播和反向传播各自的计算过程。

整个网络一共包含两层，其中输入层有两个神经元，即梯形的上底（等同于高）和下底；隐藏层有 80 个神经元；输出层有一个神经元。由此可以得出，在第 1 层中 $\boldsymbol{a}^1$ 的形状为 $[m,2]$（m 为样本个数），权重 $\boldsymbol{w}^1$ 的形状为 $[2,80]$，$\boldsymbol{b}^1$ 的形状为 $[80]$；在第 2 层中 $\boldsymbol{a}^2$ 的形状为 $[m,80]$，权重 $\boldsymbol{w}^2$ 的形状为 $[80,1]$，$\boldsymbol{b}^2$ 的形状为 $[1]$；最终预测输出 $\boldsymbol{a}^3$ 的形状为 $[m,1]$，如图 3-17 所示。

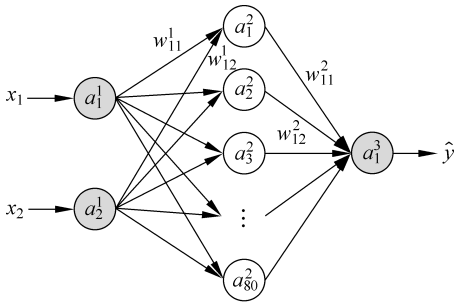


图 3-17 梯形面积预测网络结果图（偏置未画出）

可以得到模型的前向传播计算过程为

$$\boldsymbol{z}^2=\boldsymbol{a}^1\boldsymbol{w}^1+\boldsymbol{b}^1\Rightarrow\boldsymbol{a}^2=f(\boldsymbol{z}^2) \tag{3-38}$$

$$\boldsymbol{z}^3=\boldsymbol{a}^2\boldsymbol{w}^2+\boldsymbol{b}^2\Rightarrow\boldsymbol{a}^3=\boldsymbol{z}^3 \tag{3-39}$$

这里需要注意的是，式(3-39)中最后一层的输出并没有经过非线性变换处理。

同时，模型的损失函数为

$$J(\boldsymbol{w},\boldsymbol{b})=\frac{1}{2m}\sum_{i=1}^{m}(y_i-\hat{y}_i)^2 \tag{3-40}$$

最后，根据式(3-28)可得

$$\boldsymbol{\delta}^3=[\boldsymbol{a}^3-\boldsymbol{y}]\odot 1 \tag{3-41}$$

根据式(3-34)、式(3-35)和式(3-41)可得

$$\frac{\partial J}{\partial \boldsymbol{w}^2}=(\boldsymbol{a}^2)^{\mathrm{T}}\otimes\boldsymbol{\delta}^3$$

$$\frac{\partial J}{\partial \boldsymbol{b}^2}=\boldsymbol{\delta}^3 \tag{3-42}$$

根据式(3-36)可得

$$\boldsymbol{\delta}^2=\boldsymbol{\delta}^3\otimes(\boldsymbol{w}^2)^{\mathrm{T}}\odot f'(\boldsymbol{z}^2) \tag{3-43}$$

进一步根据式(3-34)、式(3-35)和式(3-43)可得

$$\frac{\partial J}{\partial \boldsymbol{w}^1}=(\boldsymbol{a}^1)^{\mathrm{T}}\otimes\boldsymbol{\delta}^2$$

$$\frac{\partial J}{\partial \boldsymbol{b}^1}=\boldsymbol{\delta}^2 \tag{3-44}$$

3.4.2　模型实现

在完成相关迭代公式的梳理后，下面开始介绍如何从零实现这个两层神经网络模型。首先需要实现相关辅助函数，以下完整示例代码可参见 Code/Chapter03/C05_MultiLayerReg/main.py 文件。

1. Sigmoid 实现

对于 Sigmoid 函数的具体介绍可参见 3.12 节，这里先直接进行使用，代码如下：

```
1  def sigmoid(z):
2      return 1 / (1 + np.exp(-z))
```

同时，后续需要用到其对应的导数，因此也要进行实现，代码如下：

```
1  def sigmoid_grad(z):
2      return sigmoid(z) * (1 - sigmoid(z))
```

2. 损失函数实现

这里采用均方误差作为损失函数，根据式(3-40)可知，代码如下：

```
1  def loss(y, y_hat):
2      y_hat = y_hat.reshape(y.shape)
3      return 0.5 * np.mean((y - y_hat) ** 2)
```

在上述代码中，第 2 行用于将 y 和 y_hat 转换为同一个形状，否则容易出错且不易排查。第 3 行则用于计算损失并返回结果。

3. 前向传播实现

需要实现整个网络模型的前向传播过程，实现代码如下：

```
1  def forward(x, w1, b1, w2, b2): #预测
2      a1 = x
3      z2 = np.matmul(a1, w1) + b1
4      a2 = sigmoid(z2)
5      z3 = np.matmul(a2, w2) + b2
```

```
    a3 = z3
    return a3, a2
```

在上述代码中，第 1 行中各个变量的信息在 3.4.1 节已经介绍过，这里就不再赘述了。第 3～4 行用于进行第 1 个全连接层的计算，对应式(3-38)中的计算过程。第 5 行则用于对输出层进行计算，对应式(3-39)中的计算过程。第 7 行用于返回最后的预测结果，但由于 a2 在反向传播的计算过程中需要用到，所以也进行了返回。

4. 反向传播实现

接着实现反向传播，用于计算参数梯度，实现代码如下：

```
def backward(a3, w2, a2, a1, y):
    m = a3.shape[0]
    delta3 = (a3 - y) * 1. #[m,output_node]
    grad_w2 = (1 / m) * np.matmul(a2.T, delta3)
    grad_b2 = (1 / m) * np.sum(delta3, axis = 0)
    delta2 = np.matmul(delta3, w2.T) * sigmoid_grad(a2)
    grad_w1 = (1 / m) * np.matmul(a1.T, delta2)
    grad_b1 = (1 / m) * np.sum(delta2, axis = 0)
    return [grad_w2, grad_b2, grad_w1, grad_b1]
```

在上述代码中，第 2 行表示获取样本个数。第 3 行则根据式(3-41)来计算 delta3，形状为[m,1]。第 4～5 行根据式(3-42)分别计算输出层参数的梯度 grad_w2 和 grad_b2，形状分别为[80,1]和[1]，同时因为有 m 个样本，所以需要取均值。第 6 行根据式(3-43)来计算 delta2，形状为[m,80]。第 7～8 行根据式(3-44)分别计算隐藏层参数的梯度 grad_w1 和 grad_b1，形状分别为[2,80]和[80]。第 9 行则用于返回最后所有权重参数对应的梯度。

5. 梯度下降实现

接着实现梯度下降算法，用于根据梯度更新网络中的权重参数，代码如下：

```
def gradient_descent(grads, params, lr):
    for i in range(len(grads)):
        params[i] -= lr * grads[i]
    return params
```

在上述代码中，第 1 行中的 grads 和 params 均为一个列表，分别表示所有权重参数对应的梯度及权重参数本身，lr 则表示学习率。第 2～3 行取列表中对应的参数和梯度，根据梯度下降来更新参数，这里需要注意的是传入各个参数的梯度 grads 要和 params 中参数的顺序一一对应。

6. 模型训练实现

在实现上述所有过程后便可以实现整个模型的训练过程，代码如下：

```
def train(x, y):
    epochs,lr = 1000, 0.05
    input_node,hidden_node = 2, 80
    output_node = 1
    losses = []
    w1 = np.random.normal(size = [input_node, hidden_node])
    b1 = np.random.normal(size = hidden_node)
    w2 = np.random.normal(size = [hidden_node, output_node])
```

```
9       b2 = np.random.normal(size = output_node)
10      for i in range(epochs):
11          logits, a2 = forward(x, w1, b1, w2, b2)
12          l = loss(y, logits)
13          grads = backward(logits, w2, a2, x, y)
14          w2, b2, w1, b1 = gradient_descent(grads,[w2, b2, w1, b1], lr = lr)
15          if i % 10 == 0:
16              print("Epoch: {}, loss: {}".format(i, l))
17          losses.append(l)
18      logits, _ = forward(x, w1, b1, w2, b2)
19      print("真实值:", y[:5].reshape(-1))
20      print("预测值:", logits[:5].reshape(-1))
21      return losses, w1, b1, w2, b2
```

在上述代码中，第 2 行表示定义梯度下降迭代的轮数和学习率。第 3～4 行用于定义网络模型的结构。第 5 行用于定义一个列表，以此来保存每次迭代后模型当前的损失值。第 6～9 行根据正态分布来生成权重参数的初始化结果。第 10～17 行则用于完成整个梯度下降的迭代过程，其中第 11 行为前向传播过程，第 12 行用于计算模型当前的损失值，第 13 行表示通过反向传播来计算梯度，第 14 行执行梯度下降过程，以此来更新权重参数。第 15～16 行表示每迭代 10 次输出一次损失值。第 18 行表示用训练好的模型对 x 进行预测。第 19～20 行用于输出前 5 个样本的预测值和真实值。第 21 行用于返回训练好的模型参数和整个在训练过程中保存的损失值。

上述代码运行结束后得到的输出结果如下：

```
1   真实值: [1.26355453 1.61181353 1.85784564 1.7236208 0.48818497]
2   预测值: [1.25302678 1.60291594 1.85990525 1.72523891 0.50386205]
```

同时，还可以对网络模型在训练过程中保存的损失值进行可视化，如图 3-18 所示。

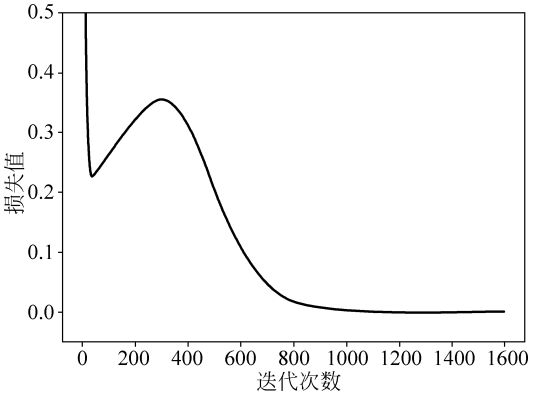


图 3-18　梯形面积预测损失图

从图 3-18 可以看出，模型大约在迭代 1400 次后便进行入了收敛阶段。

7. 模型预测实现

在完成模型训练之后，便可将其运用在新样本上，以此来预测其对应的结果，代码如下：

```
def prediction(x, w1, b1, w2, b2):
    x = np.reshape(x, [-1, 2])
    logits, _ = forward(x, w1, b1, w2, b2)
    print(f"预测结果为\n{logits}")
    return logits
```

在上述代码中，第 1 行用于传入带预测的样本点及网络模型前向传播时所依赖的 4 个权重参数。第 2 行用于确保带预测样本为 m 行两列的形式。第 3 行使模型进行前向传播并返回预测的结果。

最后，可以通过如下代码来完成模型的训练与预测过程：

```
if __name__ == '__main__':
    x, y = make_trapezoid_data()
    losses, w1, b1, w2, b2 = train(x, y)
    x = np.array([[0.6, 0.8], [0.7, 1.5]])
    prediction(x, w1, b1, w2, b2)
```

在上述代码中，第 2～3 行用于生成模拟数据并完成模型的训练过程。第 3～4 行用于制作带预测的新样本。第 5 行根据已训练好的网络模型来对新样本进行预测。

上述代码运行后便可以得到如下所示的结果：

```
预测结果为[[0.40299857] [0.82788597]]
```

到此，对于如何从零实现一个简单的多层神经网络就介绍完了。

3.4.3 小结

本节首先通过一个两层的神经网络来回顾和梳理了前向传播的详细计算过程，然后根据 3.4.1 节中介绍的内容导出了模型在反向传播过程中权重参数的梯度计算公式；最后，一步一步详细地介绍了如何从零开始实现这个两层神经网络，包括模型的正向传播和反向传播过程，以及如何对新样本进行预测等。

3.5 从逻辑回归到 Softmax 回归

前面几节详细地介绍了线性回归模型的原理及其实现，本节将继续介绍一个经典的机器学习算法——逻辑回归（Logistic Regression）及其变种 Softmax 回归，同时也将再次介绍深度学习中抽象特征的意义。

3.5.1 理解逻辑回归模型

通常来讲，一个新算法的诞生要么用来改善已有的算法模型，要么用来解决一类新的问题，而逻辑回归模型恰恰属于后者，它是用来解决一类新的问题——分类（Classification）。什么是分类问题呢？

现在有两堆样本点，需要建立一个模型来对新输入的样本进行预测，判断其应该属于哪个类别，即二分类问题（Binary Classification），如图 3-19 所示。对于这个问题的描述用线性回归来解决肯定是不行的，因为两者本就属于不同类型的问题。退一步讲，即使用线性回

归来建模得到的估计就是一条向右倾斜的直线，而这里需要的却是一条向左倾斜的且位于两堆样本点之间的直线。同时，回归模型的预测值都位于预测曲线附近，而无法做到区分直线两边的东西。既然用已有的线性回归解决不了，那么可不可以在此基础上进行改进以实现分类的目的呢？答案是当然可以。

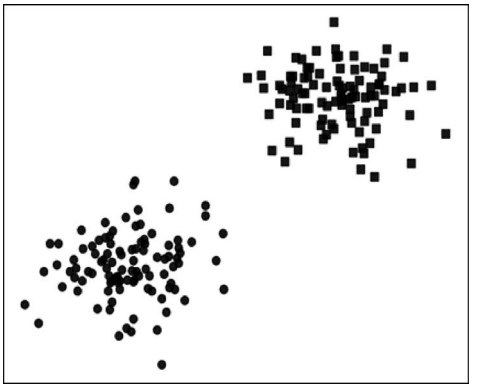


图 3-19　分类任务

3.5.2　建立逻辑回归模型

既然是解决分类问题，那么完全可以通过建立一个模型来预测每个样本点(x_1, y_2)属于其中一个类别的概率 p，如果 $p>0.5$，就可以认为该样本点属于这个类别，这样就能解决上述的二分类问题了。该怎样建立这个模型呢？

在前面的线性回归中，通过建模 $h(x)=wx+b$ 来对新样本进行预测，其输出值为可能的任意实数，但此处既然要得到一个样本所属类别的概率，那最直接的办法就是通过一个函数 $g(z)$，将 x_1 和 x_2 这两个特征的线性组合映射至$[0,1]$的范围。由此，便得到了逻辑回归中的预测模型

$$\hat{y}=h(x)=g(w_1x_1+w_2x_2+b) \tag{3-45}$$

其中，$g(x)$同样为 Sigmoid 函数；w_1、w_2 和 b 为未知参数；$h(x)$称为假设函数(Hypothesis)，当 $h(x)$大于某个值(通常设为 0.5)时，便可以认为样本 x 属于正类，反之则认为其属于负类。同时，也将 $w_1x_1+w_2x_2+b=0$ 称为两个类别间的决策边界(Decision Boundary)。当求解得到 w_1、w_2 和 b 后，也就意味着得到了这个分类模型。

当然，如果该数据集有 n 个特征维度，则同样只需将所有特征的线性组合映射至区间 $[0,1]$

$$\hat{y}=h(x)=g(w_1x_1+w_2x_2+\cdots+w_nx_n+b) \tag{3-46}$$

可以看出，逻辑回归本质上也是一个单层的神经网络。

同时，有了前面几节关于神经网络内容的介绍，还可以通过示意图来表示式(3-46)中的模型，如图 3-20 所示。

其中，输出层的曲线就表示这个映射函数 $g(x)$。

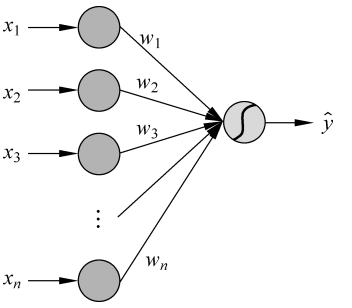


图 3-20　逻辑回归模型结构图
(偏置未画出)

3.5.3 求解逻辑回归模型

当建立好模型之后就需要找到一种方法来求解模型中的未知参数。同线性回归一样，此时也需要通过一种间接的方式，即通过目标函数来刻画预测标签(Label)与真实标签之间的差距。当最小化目标函数后，便可以得到需要求解的参数 $\boldsymbol{w}$ 和 b。

对于逻辑回归来讲，可以通过最小化式(3-47)中的目标函数来求解模型参数

$$J(\boldsymbol{w},b)=-\frac{1}{m}\left[\sum_{i=1}^{m}y^{(i)}\log h(\boldsymbol{x}^{(i)})+(1-y^{(i)})\log(1-h(\boldsymbol{x}^{(i)}))\right]$$

$$h(\boldsymbol{x}^{(i)})=g(\boldsymbol{w}\boldsymbol{x}^{(i)}+b) \tag{3-47}$$

其中，m 表示样本总数；$\boldsymbol{x}^{(i)}$ 表示第 i 个样本；$y^{(i)}$ 表示第 i 个样本的真实标签，取值为 0 或 1；$h(\boldsymbol{x}^{(i)})$ 表示第 i 个样本为正类的预测概率。

由式(3-47)可知，当函数 $J(\boldsymbol{w},b)$ 取得最小值的参数 $\hat{w}$ 和 $\hat{b}$ 时，也就是要求的目标参数。原因在于，当 $J(\boldsymbol{w},b)$ 取得最小值时就意味着此时所有样本的预测标签与真实标签之间的差距最小，这同时也是最小化目标函数的意义，因此，对于如何求解模型 $h(\boldsymbol{x})$ 的问题就转换为如何最小化目标函数 $J(\boldsymbol{w},b)$ 的问题。

3.5.4 从二分类到多分类

在讲完逻辑回归这个二分类模型后自然而然就会想到如何完成多分类任务，因为在实际情况中，绝大多数任务场景不会是一个简单的二分类任务。通常情况下在用逻辑回归处理多分类任务时会采取一种称为 One-vs-all(也叫作 One-vs-rest)的方法。

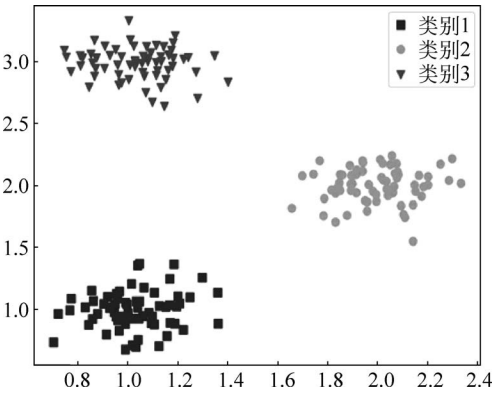


图 3-21 三分类示例数据集

图 3-21 为一个三分类的数据集，One-vs-all 策略的核心思想是每次将其中一个类别的样本和剩余其他类的所有样本看作一个二分类任务进行模型训练，如图 3-22 所示，最后在预测过程中选择输出概率值最大的那个模型对应的类别作为该样本点的所属类别。

因此，对于图 3-21 中所示的数据集来讲，便可以建立 3 个二分类模型 $h_1(\boldsymbol{x})$、$h_2(\boldsymbol{x})$ 和 $h_3(\boldsymbol{x})$，以此来完成整个三分类任务。

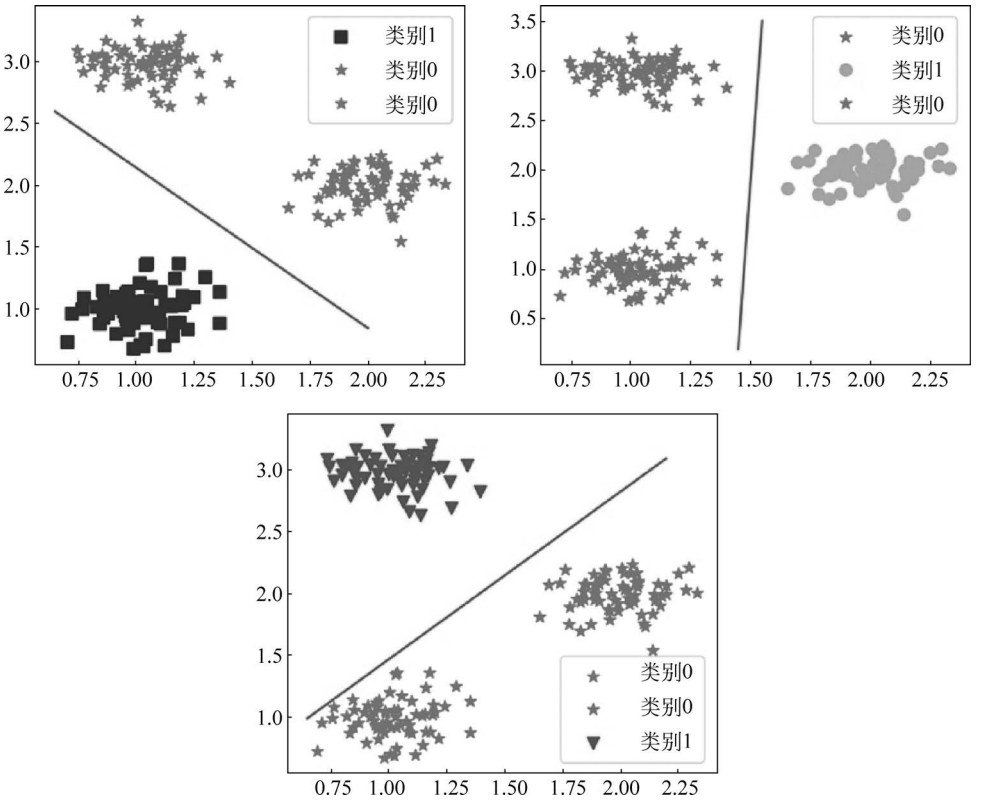


图 3-22　One-vs-all 示意图

3 个逻辑回归的结构图如图 3-23 所示，并且在训练模型时需要对每个样本的类标重新进行编码。例如有 5 个样本的原始标签为[0,0,1,2,1]，那么在训练 $h_1(\boldsymbol{x})$这个模型时这 5 个标签将会变为[1,1,0,0,0]。同理，在训练 $h_2(\boldsymbol{x})$和 $h_3(\boldsymbol{x})$时，样本标签将会重新编码为[0,0,1,0,1]和[0,0,0,1,0]。最后，对于每个新样本来讲，其预测结果为 $h_1(\boldsymbol{x})$、$h_2(\boldsymbol{x})$和 $h_3(\boldsymbol{x})$这 3 个模型中概率值最大的模型对应的类标。

当然，对于图 3-23 所示的这种表示方法来讲，当分类类别较多时表示起来就不那么简洁了。由于图 3-23 中每个模型的输入均相同，因此可以简化为如图 3-24 所示的形式。

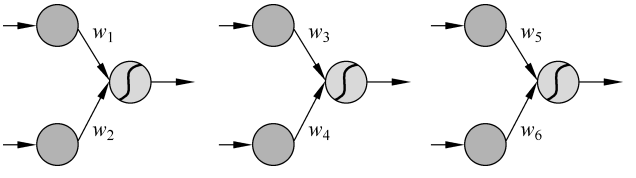


图 3-23　三分类模型结构示意图(偏置未画出)

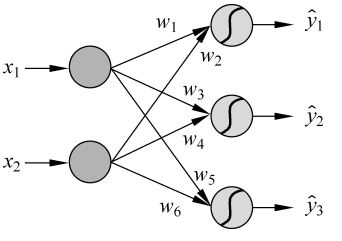


图 3-24　三分类模型结构示意图(偏置未画出)

从图 3-24 可以看出，此时图 3-23 里所示的 3 个模型已经被简化到了一个结构中，并且除了简化整个模型结构之外，图 3-24 所示的 3 个模型还能同时进行训练并输出 3 个结果，其中每个输出值表示当前样本属于该类别对应的概率，因此，在这种条件下，模型训练时的样本标签将会被重新编码为另外一种形式。仍旧以上面的 5 个样本的标签为例，第 1 个样本的标签将被编码为[1,0,0]，第 2 个样本的标签将被编码为[1,0,0]，后续 3 个依次为[0,1,0]、[0,0,1]和[0,1,0]，以此来分别与图 3-24 中模型的 3 个输出进行损失计算。同时，这种形式的编码在深度学习中被称为独热(One-Hot)编码。

3.5.5 Softmax 回归

在介绍逻辑回归时我们讲过，经过激活函数 $g(\cdot)$ 作用后可以将原本取值为 $(-\infty,+\infty)$ 的输出映射到范围[0,1]中，进而可以看作输入样本被预测为正样本的概率，因此，如果是通过图 3-24 所示的结构进行预测，则某个输入样本的预测值可能为[0.8,0.7,0.9]。虽然根据前面的规则该样本应该被认为属于概率值 0.9 对应的第 2 个类别，但这样的结果并不具有直观上的意义。

如果能有一种方法对这 3 个置信值进行归一化，使 3 者的大小关系仍旧不变，但是 3 者相加等于 1，则可将整个输出结果视为该样本属于各个类别的概率分布，然后依旧选择最大的值即可。例如将上面的[0.8,0.7,0.9]归一化成[0.33,0.30,0.37]。那有没有这样的方法呢？答案是当然有，而 Softmax 操作就是其中之一。

同上面介绍的逻辑回归一样，对于图 3-24 中的三分类模型来讲，Softmax 回归首先进行各个特征之间的线性组合，即

$$\begin{aligned} o_1 &= x_1 w_1 + x_2 w_2 + b_1 \\ o_2 &= x_1 w_3 + x_2 w_4 + b_2 \\ o_3 &= x_1 w_5 + x_2 w_6 + b_3 \end{aligned} \tag{3-48}$$

接着，再对得到的结果 o_1, o_2, o_3 进行归一化处理

$$\hat{y}_1, \hat{y}_2, \hat{y}_3 = \mathrm{Softmax}(o_1, o_2, o_3) \tag{3-49}$$

其中

$$\hat{y}_1 = \frac{\exp(o_1)}{\sum_{i=1}^{3}\exp(o_i)}, \quad \hat{y}_2 = \frac{\exp(o_2)}{\sum_{i=1}^{3}\exp(o_i)}, \quad \hat{y}_3 = \frac{\exp(o_3)}{\sum_{i=1}^{3}\exp(o_i)} \tag{3-50}$$

在经过式(3-50)的归一化过程后，不难看出 $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$ 并且 $0 \leqslant \hat{y}_1, \hat{y}_2, \hat{y}_3 \leqslant 1$，即 y_1, y_2, y_3 是一个合法的概率分布。最后通过不同类别输出概率值的大小便能够判断每个样本的所属类别。

同时，对于多分类任务来讲可以通过衡量两个概率分布之间的相似性，即交叉熵(Cross Entropy)来构建目标函数，并通过梯度下降算法对其进行最小化，从而求解模型对应的权重参数[1]，即

$$H(\boldsymbol{y}^{(i)},\hat{\boldsymbol{y}}^{(i)})=-\sum_{j=1}^{c}y_j^{(i)}\log\hat{y}_j^{(i)} \tag{3-51}$$

其中，$\boldsymbol{y}^{(i)}$和$\hat{\boldsymbol{y}}^{(i)}$分别表示第 i 个样本的真实概率分布和预测概率分布，$y_j^{(i)}$ 和 $\hat{y}_j^{(i)}$ 分别表示第 i 个样本第 j 个类别对应的概率值，c 表示分类类别数，log 表示取自然对数。

例如真实概率分布 $y=[0,0,1]$，预测概率分布 $p=[0.3,0.1,0.6]$，$q=[0.7,0.2,0.1]$，则两种情况下的交叉熵分别为

$$\begin{aligned}H(\boldsymbol{y},\boldsymbol{p})&=-(0\cdot\log0.3+0\cdot\log0.1+1\cdot\log0.6)=0.51\\H(\boldsymbol{y},\boldsymbol{q})&=-(0\cdot\log0.7+0\cdot\log0.2+1\cdot\log0.1)=2.3\end{aligned} \tag{3-52}$$

从式(3-52)中的计算结果可以看出，$\boldsymbol{y}$ 与 $\boldsymbol{p}$ 之间的概率分布最相似，并且从直观上也能发现这一点。

因此，对于包含 m 个样本的训练集来讲，其损失函数为

$$J(\boldsymbol{w},b)=\frac{1}{m}\sum_{i=1}^{m}H(\boldsymbol{y}^{(i)},\hat{\boldsymbol{y}}^{(i)}) \tag{3-53}$$

其中，$\boldsymbol{w}$ 和 b 表示整个模型的所有参数，同时将式(3-53)称为交叉熵损失函数。

最后，这里有两点值得注意：①回归模型一般来讲是指对连续值进行预测的一类模型，而分类模型则是指对离散值(类标)进行预测的一类模型，但逻辑回归和 Softmax 回归例外；②Softmax 回归也是一个单层神经网络且直接对各个原始特征的线性组合进行归一化操作，但是 Softmax 这一操作却可以运用到每个神经网络的最后一层，而这也是深度学习中分类模型的标准操作。

3.5.6　特征的意义

3.1 节从线性回归里的房价预测到梯形块面积介绍了输入特征对于模型预测结果的重要性，接着又从特征提取及非线性变换的角度介绍了特征提取对于模型的重要性，最后从单层神经网络(线性回归模型)顺利地过渡到了多层神经网络，也就是过渡到深度学习的概念中，当然这样的理念同样体现在分类模型中。

与传统的机器学习相比，深度学习最大的不同点便在于特征的可解释性。在机器学习中，我们会尽可能地要求每个特征(包括不同特征之间组合后得到的新特征)都具有一定的含义。例如在 3.1.4 节介绍的梯形面积预测示例中，每个特征 x_1 和 x_2 及手工构造出来的新特征 x_1x_2、x_1^2 和 x_2^2 都具有极强的可解释性，因此，在机器学习中基本上不存在所谓的“抽象特征”的概念，但是，当用机器学习算法来完成某些分类任务时却又不得不用这些不知道什么意思的特征。

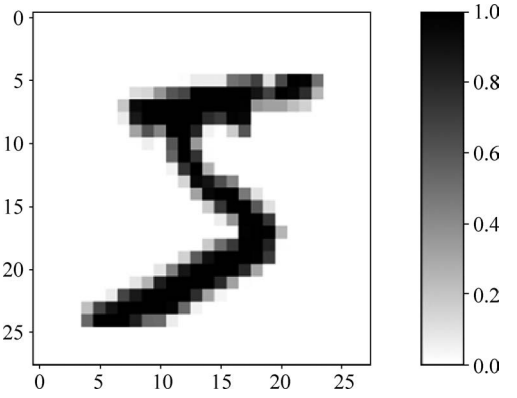


图 3-25　MNIST 手写体示意图

例如通过 Softmax 回归来对图 3-25 所示的 MNIST 手写体数字进行分类时，通常的做

法就是将整张图片展开，从而形成一列维的向量(像素值)，然后输入模型中进行分类。

例如对于图 3-25 所示的数字 5 来讲，其展开后的向量表示如下：

```
[0. 0. 0. 0. ... 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0.012 0.071 0.071 0.071
 0.494 0.533 0.686 0.102 0.651 1. 0.969 0.498 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0.118 0.141 0.369 0.604
 0.667 0.992 0.992 0.992 0.992 0.992 0.882 0.675 0.992 0.949 0.765 0.251
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.192
 0.933 0.992 0.992 0.992 0.992 0.992 0.992 0.992 0.992 0.984 0.365 0.322
 0.322 0.22 0.153 0. 0. 0. 0. 0. 0. 0. 0. 0.
...]
```

现在让你说出上述 784 个特征的每个特征维度的含义，你能说清楚吗？显然不能，不过这依然不影响模型最后的分类结果，但这又是为什么呢？想一想，狗主要是靠什么来辨识事物的？对，主要靠味道，但人主要通过“味道”这个特征来辨识事物吗？类似的还有蝙蝠能够通过声波这个特征来辨识事物等。那既然是这样，为什么不可以认为是模型具备了这种人所不具备的特征识别能力呢？

3.5.7 从具体到抽象

在 3.1.4 节梯形块面积预测部分的内容中讲到，为了使模型能有一个更好的预测效果，在原始特征 x_1，x_2 的基础上还人为地构造了 3 个依旧可解释的特征 x_1x_2，x_1^2，x_2^2。由此得到的是，如果仅依靠原始特征来建模，则最后的效果往往不尽人意，因此，在机器学习中通常会在原始特征的基础上再人为地构造一部分特征进行建模，但问题在于，当以手工的方式来构造特征时，我们的大脑会潜意识地去寻求一个具备解释性的结果，也就是新特征要具有明确的含义，而在实际中这几乎难以进行。

利用深度卷积神经网络对图片进行特征提取后的可视化结果如图 3-26 所示，其中左边是靠近输入层的特征图，右边是靠近输出层的特征图，而以人类的视角根本无法说出上述特征图的实际意义。

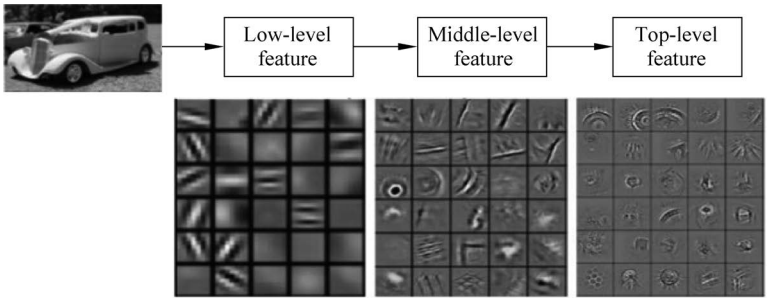


图 3-26 深层特征提取示意图

因此，人工构造特征的方法通常会带来两个问题：①即使是在知道原始特征含义的情况下也只会构造出极少的特征，而这对深度学习来讲可谓杯水车薪；②若是在不知道原始特征含义的情况下(例如素)，则几乎不可能再构造出新的特征。那么该怎么办呢？既然如

此，何不把这个过程交给模型自己去完成呢？

因此，对于图 3-24 所示的模型来讲，可以将现有的输出（并多加几个神经元）作为原始特征经组合后得到的新特征，然后将这部分特征作为输入进行分类，如图 3-27 所示。

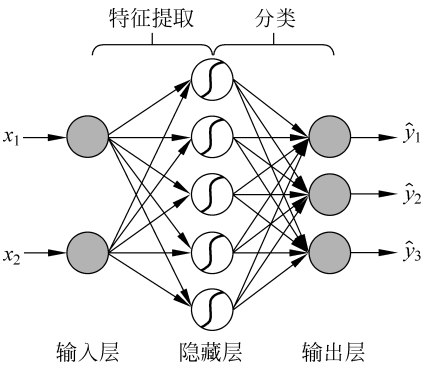


图 3-27　特征提取示意图

隐藏层的 5 个神经元便是原始特征输入 x_1 和 x_2 经过多次线性组合和非线性变换后所构成的新特征，而 $\hat{y}_1$、$\hat{y}_2$ 和 $\hat{y}_3$ 则是通过新特征进行三分类后的结果，当然这里并不会知道特征 $a_1 \sim a_5$ 具有什么样的实际意义。

到此，对于如何在原始特征上进行抽象特征提取的工作似乎就完成了，但此时突然从远处传来了两个声音：①图 3-27 在进行特征提取时能否组合得到更多的特征，例如 10 个或者 20 个？②图 3-27 中的示例仅仅进行了一次特征提取，那么能不能在现有的基础上，再进行几次非线性特征提取，然后完成最后的分类任务呢？

3.5.8　从浅层到深层

虽然看起来这是两个问题，但其实背后都有着同样的初衷，那就是为了得到更为丰富的特征表示，以此提高下游任务的精度。那么到底哪种做法会更好呢？大量的实验研究表明，第 2 种方式所取得的效果要远远好于第 1 种方法，这也是深度学习中网络层数动辄几十，甚至上百层的缘故。由此可以知道，通过深层次的特征提取能够有效地提高模型的特征表达能力。

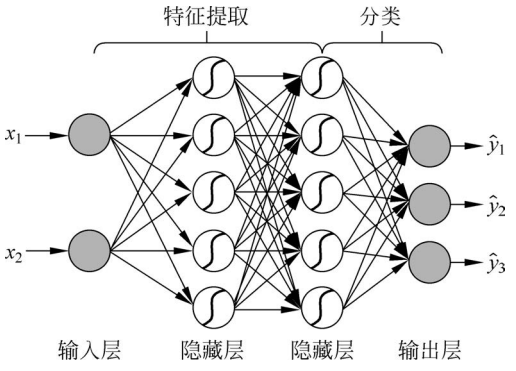


图 3-28　深层特征提取网络结构图

在图 3-27 的结构上又加入了一个新的非线性特征提取层，然后用提取的特征完成最后的分类任务，如图 3-28 所示。此时可以发现，对于输出层之前的所有层都可以将其看成一个特征提取的过程，并且越靠后的隐藏层意味着提取的特征越抽象。当原始输入经过多层网络的特征提取后便可将其输入最后一层进行相应操作（分类或者回归等），因此，总结为一句话，深度学习最核心的目的就是 4 个字——特征提取。

3.5.9　小结

本节首先通过一个例子引入了什么是分类任务，介绍了为什么不能用线性回归模型进行建模，然后通过对线性回归的改进得到了逻辑回归模型，并直接给出了逻辑回归模型的目

标函数；接着介绍了如何通过多个逻辑回归模型来构建多分类任务的模型并引入 Softmax 回归；最后介绍了深度学习中特征的意义及可以通过深层特征提取的方式来获得更为抽象和丰富的特征，以此来提高模型在下游任务中的精度。

因此可以再次得出，所谓深度学习，其实就是将原始特征通过多层神经网络进行抽象特征提取，然后将提取得到的特征输入最后一层进行回归或者分类处理的过程。

3.6 Softmax 回归的简捷实现

经过 3.5 节的介绍，对于分类模型已经有了一定的了解，接下来将开始介绍如何借助 PyTorch 框架来快速实现基于 Softmax 回归的手写体分类任务。

3.6.1 PyTorch 使用介绍

3.2.1 节已经介绍过了 PyTorch 中 nn.Linear()和 nn.Sequential()的用法，接下来再介绍数据集迭代器 DataLoader 和分类任务中需要用到的 nn.CrossEntropyLoss()模块的使用方式。

1. DataLoader 的使用

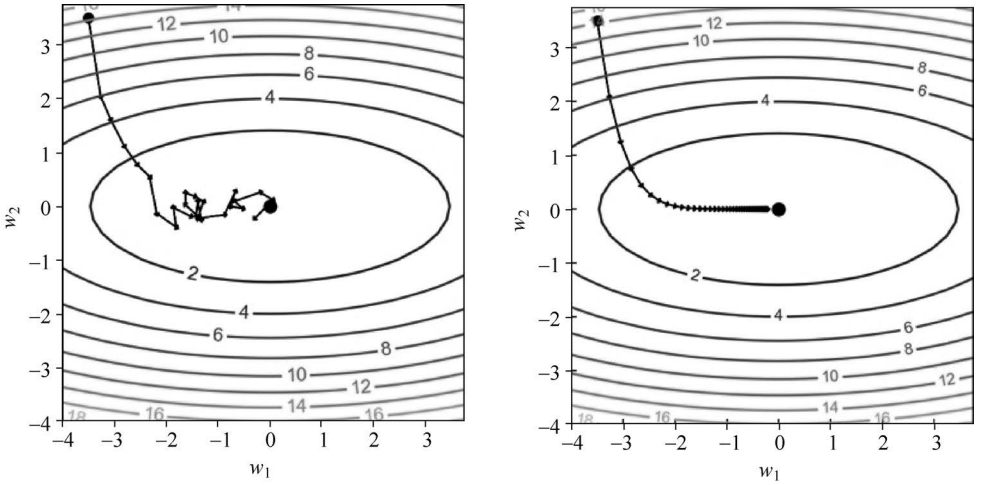
根据 3.3 节介绍的内容可知，在构造完成模型的目标函数之后便可以通过梯度下降算法来求解模型对应的权重参数。同时，由于在深度学习中训练集的数量巨大，所以很难一次同时计算所有权重参数在所有样本上的梯度，因此可以采用随机梯度下降(Stochastic Gradient Descent)或者小批量梯度下降(Mini-batch Gradient Descent)来解决这个问题[1]。

相比于小批量梯度下降算法在所有样本上计算得到目标函数关于参数的梯度，然后进行平均，随机梯度下降算法的做法是每次迭代时只取一个样本来计算权重参数对应的梯度[2]。由于随机梯度下降是基于每个样本进行梯度计算的，所以在迭代过程中每次计算得到的梯度值抖动很大，因此在实际情况中每次会选择一小批量的样本来计算权重参数的梯度，而这个批量的大小在深度学习中就被称为批大小(Batch Size)。

环形曲线表示目标函数对应的等高线，左右两边分别为随机梯度下降算法和小批量梯度下降算法求解参数 w_1 和 w_2 的模拟过程，其中箭头方向表示负梯度方向，中间的原点表示目标函数对应的最优解，如图 3-29 所示。从左侧的优化过程可以看出，尽管随机梯度下降算法最终也能近似求解最优解，但是在整个迭代优化过程中梯度却不稳定，极有可能导致陷入局部最优解中，但是对于小批量梯度下降算法来讲，由于其梯度是取多个样本的均值，因此在每次迭代过程中计算得到的梯度会相对更稳定，从而有更大的概率得到全局最优解。上述可视化代码可参见 Code/Chapter03/C07_DigitClassification/main.py 文件。

在 PyTorch 中，可以借助 DataLoader 模块来快速完成小批量数据样本的迭代生成，示例代码如下：

```
1  import torchvision.transforms as transforms
2  from torch.utils.data import DataLoader
```



(a) 随机梯度下降　　(b) 小批量梯度下降

图 3-29　随机梯度下降与小批量梯度下降模拟结果图

```
3  from torchvision.datasets import MNIST
4  def DataLoader1():
5      data_loader = MNIST(root = '~/Datasets/MNIST', download = True,
6                          transform = transforms.ToTensor())
7      data_iter = DataLoader(data_loader, batch_size = 32)
8      for (x, y) in data_iter:
9          print(x.shape,y.shape)
10 #输出结果
11 torch.Size([32, 1, 28, 28]) torch.Size([32])
12 torch.Size([32, 1, 28, 28]) torch.Size([32])
13 ......
```

在上述代码中，第 5～6 行表示载入 PyTorch 中内置的 MNIST 手写体图片（见图 3-25）数据集，root 参数为指定数据集所在的目录，当 download 为 True 时表示当指定目录不存在时通过网络下载，transform 用于指定对原始数据进行变化（这里仅仅是将原始的浮点数转换成 PyTorch 中的张量）。第 7 行便是通过 DataLoader 来根据上面载入的原始数据构造一个批大小为 32 的迭代器。第 8～9 行则用于遍历这个迭代器。第 11～12 行用于遍历迭代器所输出的结果，其中[32,1,28,28]的含义便是该张量中有 32 个样本（32 张图片），每张图片的通道数为 1（黑白），长和宽均为 28 像素。

当然，此时可能有的读者会问，如果载入本地的数据样本，则又该怎么来构造这个迭代器呢？对于这种非 PyTorch 内置数据集的情况，同样可以通过 DataLoader 来完成迭代器的构建，只是前面多了一个步骤，示例代码如下：

```
1  from torch.utils.data import TensorDataset
2  import torch
3  import numpy as np
4  def DataLoader2():
5      x = torch.tensor(np.random.random([100, 3, 16, 16]))
```

```
    y = torch.tensor(np.random.randint(0, 10, 100))
    dataset = TensorDataset(x, y)
    data_iter = DataLoader(dataset, batch_size = 32)
    for (x, y) in data_iter:
        print(x.shape, y.shape)

torch.Size([32, 3, 16, 16]) torch.Size([32])
torch.Size([32, 3, 16, 16]) torch.Size([32])
......
```

在上述代码中，第 5～6 行用于生成原始的样本数据，并转换成张量。第 7 行则根据原始数据得到实例化的 TensorDataset(继承自类 Dataset)，因为 FashionMNIST 本质上也继承自类 Dataset。第 8 行则同样用于生成对应的迭代器，并将批大小指定为 32。第 12～13 行用于最终遍历迭代器所输出的结果，含义同上，不再赘述。上述示例代码可参见 Code/Chapter03/C09_DataLoader/main.py 文件。

2. nn.CrossEntropyLoss()的使用

根据 3.5.5 节可知，在分类任务中通常会使用交叉熵来作为目标函数，并且在计算交叉熵损失之前需要对预测概率进行 Softmax 归一化操作。在 PyTorch 中，可以借助 nn.CrossEntropyLoss()模块来一次性地完成这两步计算过程，示例代码如下：

```
if __name__ == '__main__':
    logits = torch.tensor([[0.5, 0.3, 0.6], [0.5, 0.4, 0.3]])
    y = torch.LongTensor([2, 0])
    loss = nn.CrossEntropyLoss(reduction = 'mean')
    l = loss(logits, y)
    print(l)
#tensor(0.9874)
```

在上述代码中，第 2 行是模拟的模型输出结果，包含两个样本和 3 个类别。第 3 行表示两个样本的正确类标，需要注意的是 nn.CrossEntropyLoss()在计算交叉熵损失时接受的正确标签是非 One-Hot 的编码形式。第 4 行则用于实例化 CrossEntropyLoss 类对象，其中 reduction='mean'表示返回所有样本损失的均值，如果 reduction='sum'，则表示返回所有样本的损失和。第 5～6 行表示计算交叉熵并输出计算后的结果。

上述示例代码可参见 Code/Chapter03/C10_CrossEntropy/main.py 文件。

3.6.2 手写体分类实现

在熟悉了 DataLoader 和 nn.CrossEntropyLoss()这两个模块的基本使用方法后，再来看如何借助 PyTorch 快速地实现基于 Softmax 回归的手写体分类任务。完整示例代码可参见 Code/Chapter03/C11_DigitClassification/main.py 文件。

1. 构建数据集

首先需要构造后续用到的数据集，实现代码如下：

```
def load_dataset():
    data = MNIST(root = '~/Datasets/MNIST', download = True,
                 transform = transforms.ToTensor())
    return data
```

在上述代码中，ToTensor()的作用是将载入的原始图片由[0,255]取值范围缩放至[0.0,1.0]取值范围。

2. 构建模型

在完成数据集的构建后，便需要构造整个 Softmax 回归模型，实现代码如下：

```
1  def train(data):
2      epochs,lr = 2,0.001
3      batch_size = 128
4      input_node,output_node = 28 * 28, 10
5      losses = []
6      data_iter = DataLoader(data, batch_size = batch_size, shuffle = True)
7      net = nn.Sequential(nn.Flatten(),nn.Linear(input_node, output_node))
8      loss = nn.CrossEntropyLoss()                                    #定义损失函数
9      optimizer = torch.optim.SGD(net.parameters(), lr = lr)          #定义优化器
10     for epoch in range(epochs):
11         for i, (x, y) in enumerate(data_iter):
12             logits = net(x)
13             l = loss(logits, y)
14             optimizer.zero_grad()
15             l.backward()
16             optimizer.step()                                        #执行梯度下降
17             acc = (logits.argmax(1) == y).float().mean().item()
18             print(f"Epos[{epoch + 1}/{epochs}]batch[{i}/{len(data_iter)}]"
19                 f" -- Acc: {round(acc, 4)} -- loss: {round(l.item(), 4)}")
20             losses.append(l.item())
21     return losses
```

在上述代码中，第 2 行中 epochs 表示在整个数据集上迭代训练多少轮。第 3 行中 batch_size 便是 3.6.1 节介绍的样本批大小。第 4 行中 input_node 和 output_node 分别用于指定网络输入层神经元(特征)的个数和输出层神经元(分类)的个数。第 6 行是用来构造返回小批量样本的迭代器。第 7 行用于定义整个网络模型，其中 nn.Flatten()表示将原始的图片拉伸成一个向量。第 8 行用于定义损失函数，在默认情况下返回的是每个小批量样本损失的均值。第 9 行用于实例化优化器。第 11～20 行则每次通过小批量样本来迭代更新网络中的权重参数，其中第 12～13 行分别是前向传播及损失计算，第 14 行是将前一次迭代中每个参数计算得到的梯度置零，第 15～16 行则用于进行反向传播和梯度下降。第 17 行用于计算在每个小批量样本上预测结果应对的准确率，关于准确率将在 3.9 节中进行介绍，简单来讲就是预测正确的样本数除以总的样本数。第 20～21 行则用于保存每次前向传播时网络的损失值并返回。

在完成上述代码后，便可以通过以下方式来完成整个模型的训练过程：

```
1  if __name__ == '__main__':
2      data = load_dataset()
3      losses = train(data)
4      visualization_loss(losses)
```

在上述代码运行结束后，便可以得到类似的输出结果：

```
1  Epochs[1/2] -- batch[0/469] -- Acc: 0.1172 -- loss: 2.3273
2  Epochs[1/2] -- batch[1/469] -- Acc: 0.1328 -- loss: 2.2881
```

```
...
Epochs[2/2] -- batch[466/469] -- Acc: 0.9141 -- loss: 0.4141
Epochs[2/2] -- batch[467/469] -- Acc: 0.8438 -- loss: 0.5982
Epochs[2/2] -- batch[468/469] -- Acc: 0.8958 -- loss: 0.3955
```

最后，还可以对网络在训练过程中保存的损失值进行可视化，如图 3-30 所示。

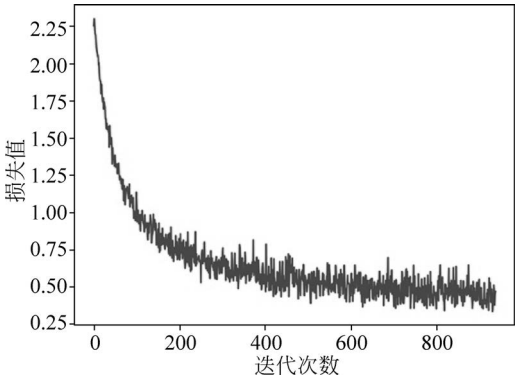


图 3-30　手写体分类模型训练损失图

从图 3-30 可以看出，模型大约在迭代 800 次后便逐步进行入了收敛阶段。

3.6.3　小结

本节首先介绍了什么是随机梯度下降和小批量梯度下降，并顺利地引出了 PyTorch 框架中的 DataLoader 模块，然后介绍了 PyTorch 中用于计算分类任务模型损失的 nn.CrossEntropyLoss()模块及其使用示例；最后详细介绍了如何借助 PyTorch 来快速实现基于 Softmax 回归的手写体分类模型。

3.7　从零实现分类模型

经过 3.5 节的介绍，已经清楚了深度学习中分类模型的基本原理，同时也掌握了如何快速地通过 PyTorch 来实现 Softmax 回归模型。在接下来的这节内容中，将会详细介绍如何从零实现基于多层神经网络的手写体分类模型。

3.7.1　网络结构

在正式介绍实现部分之前，先来看一下整个模型的网络结构及整理出前向传播和反向传播各自的计算过程。

整个网络一共包含 3 层(含有权重参数的层)，其中输入层有 784 个神经元，即长和宽均为 28 的图片展开后的向量维度；两个隐藏层均有 1024 个神经元；输出层有 10 个神经元，即分类类别数量，如图 3-31 所示。由此可以得出，在第 1 层中 $\boldsymbol{a}^1$ 的形状为$[m,784]$(m 为样本个数)，权重 $\boldsymbol{w}^1$ 的形状为$[784,1024]$，$\boldsymbol{b}^1$ 的形状为$[1024]$；在第 2 层中 $\boldsymbol{a}^2$ 的形状为$[m,1024]$，权重 $\boldsymbol{w}^2$ 的形状为$[1024,1024]$，$\boldsymbol{b}^2$ 的形状为$[1024]$；在第 3 层中 $\boldsymbol{a}^3$ 的形状为

$[m,1024]$，权重 $\boldsymbol{w}^3$ 的形状为$[1024,10]$，$\boldsymbol{b}^3$ 的形状为$[10]$；最终预测输出 $\boldsymbol{a}^4$ 的形状为$[m,10]$。

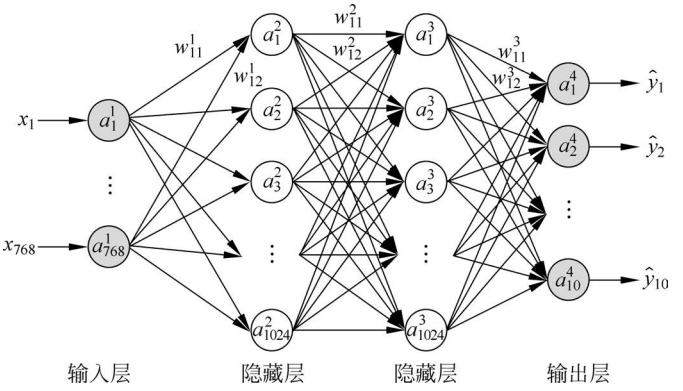


图 3-31　手写体识别网络结构图(偏置未画出)

可以得到模型的前向传播计算过程为

$$\boldsymbol{z}^2=\boldsymbol{a}^1\boldsymbol{w}^1+\boldsymbol{b}^1\Rightarrow\boldsymbol{a}^2=f(\boldsymbol{z}^2) \tag{3-54}$$

$$\boldsymbol{z}^3=\boldsymbol{a}^2\boldsymbol{w}^2+\boldsymbol{b}^2\Rightarrow\boldsymbol{a}^3=f(\boldsymbol{z}^3) \tag{3-55}$$

$$\boldsymbol{z}^4=\boldsymbol{a}^3\boldsymbol{w}^3+\boldsymbol{b}^3\Rightarrow\boldsymbol{a}^4=\mathrm{Softmax}(\boldsymbol{z}^4) \tag{3-56}$$

其中，$f(\cdot)$表示非线性变换，Softmax 的计算公式为

$$a_k^L=\frac{\mathrm{e}^{z_k^L}}{\sum_{i=1}^{S_L}\mathrm{e}^{z_i^L}} \tag{3-57}$$

同时，模型的损失函数为式(3-53)中所示的交叉熵损失函数，并且如果假设此时仅考虑一个样本，则对应的目标函数为

$$J(\boldsymbol{w},\boldsymbol{b})=-\sum_{k=1}^{S_L}y_k\cdot\log a_k^L \tag{3-58}$$

其中，$S_L=10$ 表示输出层对应神经元的个数，$L=4$ 表示输出层的层数。

根据式(3-58)可得目标函数 J 关于 z_i^L 的梯度，即

$$\delta_i^L=\frac{\partial J}{\partial z_i^L}=\frac{\partial J}{\partial a_1^L}\frac{\partial a_1^L}{\partial z_i^L}+\frac{\partial J}{\partial a_2^L}\frac{\partial a_2^L}{\partial z_i^L}+\cdots+\frac{\partial J}{\partial a_{S_L}^L}\frac{\partial a_{S_L}^L}{\partial z_i^L}=\sum_{k=1}^{S_L}\frac{\partial J}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_i^L} \tag{3-59}$$

由图 3-31 可知，J 关于任何一个输出值 a_i^L 的梯度均只有一条路径上的依赖关系，其计算过程相对简单，即

$$\frac{\partial J}{\partial a_i^L}=\frac{\partial}{\partial a_i^L}\Big[-\sum_{k=1}^{S_L}y_k\cdot\log a_k^L\Big]=-y_i\frac{1}{a_i^L} \tag{3-60}$$

接下来，需要求解的便是 a_i^L 关于 z_j^L 的梯度(此处不要被各种下标所迷惑，一定要结合式(3-57)进行理解)。例如在求解 a_1^4 关于 z_2^4 的梯度时，根据式(3-57)可知，此时的分子与

$e^{z_1^4}$ 是没关系的(分子 $e^{z_1^4}$ 可看作常数),但是在求解 a_1^4 关于 z_1^4 的梯度时,此时的分子就不能看作常数了。具体有

当 $i \neq j$ 时:

$$\frac{\partial a_i^L}{\partial z_j^L} = \frac{\partial}{\partial z_j^L} \frac{e^{z_i^L}}{\sum_{k=1}^{S_L} e^{z_k^L}} = \frac{0 - e^{z_i^L} e^{z_j^L}}{\left(\sum_{k=1}^{S_L} e^{z_k^L}\right)^2} = -a_i^L a_j^L \tag{3-61}$$

当 $i = j$ 时:

$$\frac{\partial a_i^L}{\partial z_j^L} = \frac{\partial}{\partial z_j^L} \frac{e^{z_i^L}}{\sum_{k=1}^{S_L} e^{z_k^L}} = \frac{e^{z_i^L} \sum_{k=1}^{S_l} e^{z_k^L} - (e^{z_i^L})^2}{\left(\sum_{k=1}^{S_L} e^{z_k^L}\right)^2} = \frac{e^{z_i^L}}{\sum_{k=1}^{S_L} e^{z_k^L}} \left(1 - \frac{e^{z_i^L}}{\sum_{k=1}^{S_L} e^{z_k^L}}\right) = a_i^L (1 - a_i^L) \tag{3-62}$$

进一步,由式(3-59)~式(3-62)便可以得到

$$\begin{aligned} \delta_i^L &= \frac{\partial J}{\partial z_j^L} = \sum_{i=1}^{S_L} \frac{\partial J}{\partial a_i^L} \frac{\partial a_i^L}{\partial z_j^L} = \sum_{i \neq j} \frac{\partial J}{\partial a_i^L} \frac{\partial a_i^L}{\partial z_j^L} + \frac{\partial J}{\partial a_i^L} \frac{\partial a_i^L}{\partial z_j^L} \\ &= \sum_{i \neq j} y_i \frac{1}{a_i^L} a_i^L a_j^L + \left[-y_j \frac{1}{a_j^L} a_j^L (1 - a_j^L)\right] \\ &= \sum_{i \neq j} y_i a_j^L + y_j a_j^L - y_j = a_j^L \sum_{i=1}^{S_L} y_i - y_j = a_j^L - y_j \end{aligned} \tag{3-63}$$

对式(3-63)进行向量化表示有

$$\boldsymbol{\delta}^L = \boldsymbol{a}^L - \boldsymbol{y} \tag{3-64}$$

由此,便得到了在 Softmax 作用下,利用反向传播算法对交叉熵损失函数关于所有参数进行梯度求解的计算公式。

对于第 3 层中的参数来讲有

$$\begin{aligned} \boldsymbol{\delta}^4 &= \boldsymbol{a}^4 - \boldsymbol{y} \\ \frac{\partial J}{\partial \boldsymbol{w}^3} &= (\boldsymbol{a}^3)^{\mathrm{T}} \otimes \boldsymbol{\delta}^4 \\ \frac{\partial J}{\partial \boldsymbol{b}^3} &= \boldsymbol{\delta}^4 \end{aligned} \tag{3-65}$$

对于第 2 层中的参数来讲有

$$\begin{aligned} \boldsymbol{\delta}^3 &= \boldsymbol{\delta}^4 \otimes (\boldsymbol{w}^3)^{\mathrm{T}} \odot f'(\boldsymbol{z}^3) \\ \frac{\partial J}{\partial \boldsymbol{w}^2} &= (\boldsymbol{a}^2)^{\mathrm{T}} \otimes \boldsymbol{\delta}^3 \\ \frac{\partial J}{\partial \boldsymbol{b}^2} &= \boldsymbol{\delta}^3 \end{aligned} \tag{3-66}$$

对于第 1 层中的参数来讲有

$$\boldsymbol{\delta}^2 = \boldsymbol{\delta}^3 \otimes (\boldsymbol{w}^2)^{\mathrm{T}} \odot f'(\boldsymbol{z}^2)$$
$$\frac{\partial J}{\partial \boldsymbol{w}^1} = (\boldsymbol{a}^1)^{\mathrm{T}} \otimes \boldsymbol{\delta}^2$$
$$\frac{\partial J}{\partial \boldsymbol{b}^1} = \boldsymbol{\delta}^2 \tag{3-67}$$

最后，值得一提的是，如果在式(3-56)中先对 $\boldsymbol{z}^4$ 进行 Sigmoid 映射操作，再进行 Softmax 归一化操作，则式(3-64)的结果将会变成

$$\boldsymbol{\delta}^L = (\boldsymbol{a}^L - \boldsymbol{y})(1 - g(\boldsymbol{z}^L))g(\boldsymbol{z}^L) \tag{3-68}$$

其中，$g(\cdot)$表示 Sigmoid 函数。

由于 $g(\cdot) \in (0,1)$，而这将使第 L 层的残差急剧变小，进而使模型在训练过程中出现梯度消失问题，导致模型难以收敛，因此，在分类模型中通常在最后一层线性组合的基础上直接进行 Softmax 运算。

3.7.2　模型实现

在完成相关迭代公式的梳理后，下面开始介绍如何从零实现这个 3 层网络的分类模型。首先需要实现相关辅助函数，完整示例代码可参见 Code/Chapter03/C12_MultiLayerCla/main.py 文件。

1. 数据集构建实现

这里，依旧使用之前的 MNIST 数据集来建模，同时由于不再借助 PyTorch 框架，所以需要将原始数据转换为 NumPy 中的 arry 类型，实现代码如下：

```
1  def load_dataset():
2      data = MNIST(root = '~/Datasets/MNIST', download = True,
3                   transform = transforms.ToTensor())
4      x, y = [], []
5      for img in data:
6          x.append(np.array(img[0]).reshape(1, -1))
7          y.append(img[1])
8      x = np.vstack(x)
9      y = np.array(y)
10     return x, y
```

在上述代码中，第 5～7 行表示遍历原始数据中的每个样本，从而得到输入和标签，并直接将图片拉伸成一个 784 维的向量。第 8～9 行分别将输入和标签转换成 np.array 类型。最终，x 和 y 的形状分别为(60000,784)和(60000,)。

2. 迭代器实现

由于不再借助 PyTorch 中的 DataLoader 模块，所以需要自己实现一个迭代器，实现代码如下：

```
1  def gen_batch(x, y, batch_size = 64):
2      s_index, e_index = 0, 0 + batch_size
3      batches = len(y) //batch_size
```

```
    if batches * batch_size < len(y):
        batches += 1
    for i in range(batches):
        if e_index > len(y):
            e_index = len(y)
        batch_x = x[s_index:e_index]
        batch_y = y[s_index: e_index]
        s_index, e_index = e_index, e_index + batch_size
        yield batch_x, batch_y
```

在上述代码中，第 1 行 x 和 y 分别表示上面构建的数据和标签。第 2 行用来标识取每个 batch 样本时的开始和结束索引。第 3～5 行用来判断，当样本数不能被 batch_size 整除时的特殊情况。第 6～11 行用于按索引依次取每个 batch 对应的样本。第 12 行用于返回对应一个 batch 的样本，这里需要注意的是，Python 中 yield 在函数中的功能类似于 return，不同的是 yield 每次返回结果之后函数并没有退出，而是每次遇到 yield 关键字后返回相应结果，并保留函数当前的运行状态，等待下一次的调用。

3. 交叉熵与 Softmax 实现

根据式(3-58)可得，对于预测结果和真实结果的交叉熵实现代码如下：

```
def crossEntropy(y_true, logits):
    loss = y_true * np.log(logits) #[m,n]
    return -np.sum(loss) / len(y_true)
```

在上述代码中，第 1 行 y_true 和 logits 分别表示每个样本的真实标签和预测概率，其形状均为$[m,c]$，即 y_true 为 One-Hot 的编码形式。第 2 行表示同时计算所有样本的损失值。第 3 行用于计算所有样本的损失的均值。

同时，根据式(3-57)可得，对于预测结果的 Softmax 运算实现代码如下：

```
def softmax(x):
    s = np.exp(x)
    return s / np.sum(s, axis = 1, keepdims = True)
```

在上述代码中，第 1 行 x 表示模型最后一层的线性组合结果，形状为$[m,c]$。第 2 行表示取所有值对应的指数。第 3 行用于计算 Softmax 的输出结果。这里值得注意的是，因为 np.sum(s,axis=1)操作后变量的维度会减 1，为了保证广播机制正常，所以设置 keepdims=True 以保持维度不变。

4. 前向传播实现

进一步地，需要实现整个网络模型的前向传播过程，实现代码如下：

```
def forward(x, w1, b1, w2, b2, w3, b3):
    z2 = np.matmul(x, w1) + b1
    a2 = sigmoid(z2)
    z3 = np.matmul(a2, w2) + b2
    a3 = sigmoid(z3)
    z4 = np.matmul(a3, w3) + b3
    a4 = softmax(z4)
    return a4, a3, a2
```

在上述代码中，第 1 行中各个变量的信息在 3.7.1 节内容中已经介绍过，这里就不再赘述了。第 2～3 行用于对第 1 个全连接层进行计算，对应式(3-54)中的计算过程。第 4～5 行用于对第 2 个全连接层进行计算，对应式(3-55)中的计算过程。第 6～7 行则用于对输出层进行计算，对应式(3-56)中的计算过程。第 8 行用于返回最后的预测结果，但由于 a3 和 a2 在反向传播的计算过程中需要用到，所以也进行了返回。

5. 反向传播实现

接着实现反向传播，用于计算参数梯度，实现代码如下：

```
1  def backward(a4, a3, a2, a1, w3, w2, y):
2      m = a4.shape[0]
3      delta4 = a4 - y
4      grad_w3 = 1 / m * np.matmul(a3.T, delta4)
5      grad_b3 = 1 / m * np.sum(delta4, axis = 0)
6      delta3 = np.matmul(delta4, w3.T) * (a3 * (1 - a3))
7      grad_w2 = 1 / m * np.matmul(a2.T, delta3)
8      grad_b2 = 1 / m * (np.sum(delta3, axis = 0))
9      delta2 = np.matmul(delta3, w2.T) * (a2 * (1 - a2))
10     grad_w1 = 1 / m * np.matmul(a1.T, delta2)
11     grad_b1 = 1 / m * (np.sum(delta2, axis = 0))
12     return [grad_w1, grad_b1, grad_w2, grad_b2, grad_w3, grad_b3]
```

在上述代码中，第 2 行表示获取样本个数。第 3～5 行则根据式(3-65)来计算 delta4、grad_w3 和 grad_b3，其形状分别为[m，10]、[1024，10]和[10]。第 6～8 行则根据式(3-66)来计算 delta3、grad_w2 和 grad_b2，其形状分别为[m，1024]、[1024，1024]和[1024]。第 9～11 行则根据式(3-67)来计算 delta2、grad_w1 和 grad_b1，其形状分别为[m，1024]、[784，1024]和[1024]。第 12 行用于返回最后所有权重参数对应的梯度。

6. 模型训练实现

在实现完上述所有过程后便可以实现整个模型的训练过程，实现代码如下：

```
1  def train(x_data, y_data):
2      input_nodes, hidden_nodes = 28 * 28, 1024
3      output_nodes,epochs = 10, 2
4      lr,batch_size,losses = 0.03,64,[]
5      w1 = np.random.uniform( - 0.3, 0.3, [input_nodes, hidden_nodes])
6      b1 = np.zeros(hidden_nodes)
7      w2 = np.random.uniform( - 0.3, 0.3, [hidden_nodes, hidden_nodes])
8      b2 = np.zeros(hidden_nodes)
9      w3 = np.random.uniform( - 0.3, 0.3, [hidden_nodes, output_nodes])
10     b3 = np.zeros(output_nodes)
11     for epoch in range(epochs):
12         for i, (x, y) in enumerate(gen_batch(x_data,y_data,batch_size)):
13             logits, a3, a2 = forward(x, w1, b1, w2, b2, w3, b3)
14             y_one_hot = np.eye(output_nodes)[y]
15             loss = crossEntropy(y_one_hot, logits)
16             grads = backward(logits, a3, a2, x, w3, w2, y_one_hot)
17             w1, b1, w2, b2, w3, b3 = gradient_descent(grads,
18                                 [w1, b1, w2, b2, w3, b3], lr)
19             losses.append(loss)
```

```
20              if i % 5 == 0:
21                  acc = accuracy(y, logits)
22                  print(f"Epos[{epoch+1}/{epochs}]
                                batch[{i}/{len(x_data) //batch_size}]"
23                      f" -- Acc: {round(acc, 4)} -- loss: {round(loss, 4)}")
24      acc = evaluate(x_data, y_data, forward, w1, b1, w2, b2, w3, b3)
25      print(f"Acc: {acc}")
```

在上述代码中，第 2～4 行表示定义相关的变量参数，包括输入特征数、隐藏层节点数、分类数和学习率等。第 5～10 行表示定义不同层的参数值，并进行相应初始化。第 11～23 行开始迭代训练整个模型，其中第 12 行用于遍历数据集中每个小批量的样本，第 13 行用于进行前向传播计算，第 14 行用于将原始真实标签转换为 One-Hot 编码，第 15 行用于计算损失值，第 16 行用于反向传播计算所有参数的梯度值，第 17～18 行通过梯度下降算法来更新参数，第 20～23 行用于每隔 5 个小批量计算一次准确率。第 24 行用于计算模型在整个数据集上的准确率。关于准确率将在 3.9 节中进行介绍。

上述代码运行结束后便可得到类似如下的输出结果：

```
Epochs[1/2] -- batch[0/937] -- Acc: 0.0625 -- loss: 7.1358
Epochs[1/2] -- batch[5/937] -- Acc: 0.1406 -- loss: 2.3524
Epochs[1/2] -- batch[10/937] -- Acc: 0.2188 -- loss: 2.2945
......
Epochs[2/2] -- batch[925/937] -- Acc: 0.9844 -- loss: 0.1114
Epochs[2/2] -- batch[930/937] -- Acc: 0.9844 -- loss: 0.0674
Epochs[2/2] -- batch[935/937] -- Acc: 1.0 -- loss: 0.0276
Acc: 0.9115333333333333
```

同时，还可以对网络模型在训练过程中保存的损失值进行可视化，如图 3-32 所示。

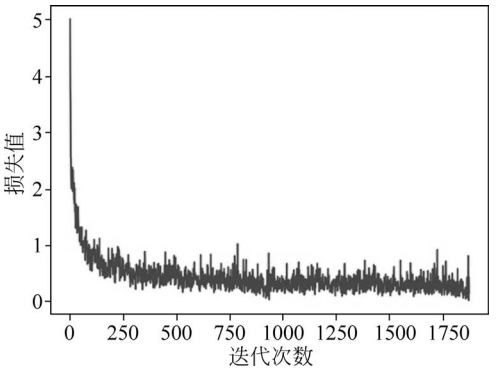


图 3-32　梯形面积预测损失图

7. 模型预测实现

在完成模型训练之后，便可将其运用在新样本上，以此来预测其对应的结果，实现代码如下：

```
def prediction(x, w1, b1, w2, b2, w3, b3):
    x = x.reshape(-1, 784)
    logits, _, _ = forward(x, w1, b1, w2, b2, w3, b3)
    return np.argmax(logits, axis=1)
```

在上述代码中，第 1 行用于传入带预测的样本点及网络模型前向传播时所依赖的 6 个权重参数。第 2 行用于确保带预测样本为 m 行 784 列的形式。第 3～4 行则使模型进行前向传播并返回预测的结果。

最后，可以通过如下代码来完成模型的训练与预测过程：

```
1  if __name__ == '__main__':
2      x, y = load_dataset()
3      losses, w1, b1, w2, b2, w3, b3 = train(x, y)
4      visualization_loss(losses)
5      y_pred = prediction(x[0], w1, b1, w2, b2, w3, b3)
6      print(f"预测标签为{y_pred}, 真实标签为{y[0]}")
#预测标签为[5], 真实标签为 5
```

到此，对于如何从零实现多层神经网络分类模型就介绍完了。

3.7.3　小结

本节首先通过一个 3 层的神经网络来回顾和梳理了分类模型前向传播的详细计算过程，然后根据 3.3 节中介绍的内容导出了模型在反向传播过程中权重参数的梯度计算公式；最后，一步一步详细地介绍了如何从零开始实现这个 3 层网络的分类模型，包括分类数据集的构建、损失函数的计算、模型的正向传播和反向传播过程，以及如何对新样本进行预测等。

3.8　回归模型评估指标

3.1 节～3.4 节介绍了如何建模线性回归（包括多变量与多项式回归）及如何通过 PyTorch 来快速搭建模型并求解，但是对于一个创建出来的模型应该怎样来对其进行评估呢？换句话说，这个模型到底怎样呢？

以最开始的房价预测为例，现在假设求解得到了如图 3-33 所示的两个模型 $h_1(x)$ 与 $h_2(x)$，那么应该选哪一个呢？抑或在不能可视化的情况下，应该如何评估模型的好与坏呢？

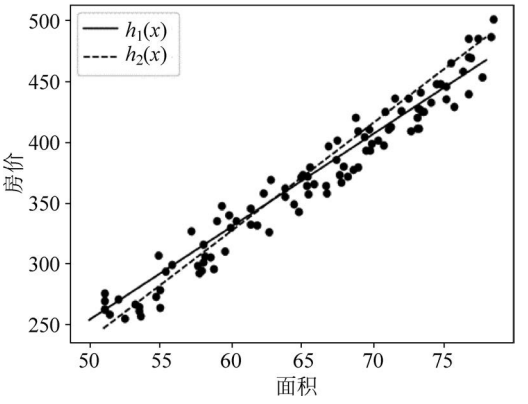


图 3-33　不同模型对房价的预测结果

在回归任务中，常见的评估指标(Metric)有平均绝对误差(Mean Absolute Error，MAE)、均方误差(Mean Square Error，MSE)、均方根误差(Root Mean Square Error，RMSE)、平均绝对百分比误差(Mean Absolute Percentage Error，MAPE)和决定系数(Coefficient of Determination)等，其中用得最为广泛的是MAE和MSE。下面依次来对这些指标进行介绍，同时在所有的计算公式中，m 均表示样本数量、$y^{(i)}$ 均表示第 i 个样本的真实值、$\hat{y}^{(i)}$ 均表示第 i 个样本的预测值。

3.8.1 常见回归评估指标

1. 平均绝对误差(MAE)

MAE用来衡量预测值与真实值之间的平均绝对误差，定义如下：

$$\mathrm{MAE}=\frac{1}{m}\sum_{i=1}^{m}|y^{(i)}-\hat{y}^{(i)}| \tag{3-69}$$

其中，MAE$\in[0,+\infty)$，其值越小表示模型越好，实现代码如下：

```
def MAE(y, y_pre):
    return np.mean(np.abs(y - y_pre))
```

2. 均方误差(MSE)

MSE用来衡量预测值与真实值之间的误差平方，定义如下：

$$\mathrm{MSE}=\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}-\hat{y}^{(i)})^2 \tag{3-70}$$

其中，MSE$\in[0,+\infty)$，其值越小表示模型越好，实现代码如下：

```
def MSE(y, y_pre):
    return np.mean((y - y_pre) ** 2)
```

3. 均方根误差(RMSE)

RMSE是在MSE的基础上取算术平方根而来，其定义如下：

$$\mathrm{RMSE}=\sqrt{\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}-\hat{y}^{(i)})^2} \tag{3-71}$$

其中，RMSE$\in[0,+\infty)$，其值越小表示模型越好，实现代码如下：

```
def RMSE(y, y_pre):
    return np.sqrt(MSE(y, y_pre))
```

4. 平均绝对百分比误差(MAPE)

MAPE和MAE类似，只是在MAE的基础上做了标准化处理，其定义如下：

$$\mathrm{MAPE}=\frac{100\%}{m}\sum_{i=1}^{m}\left|\frac{y^{(i)}-\hat{y}^{(i)}}{y^{(i)}}\right| \tag{3-72}$$

其中，MAPE$\in[0,+\infty)$，其值越小表示模型越好，实现代码如下：

```
def MAPE(y, y_pre):
    return np.mean(np.abs((y - y_pre) / y))
```

5. R^2 评价指标

决定系数 R^2 是线性回归模型中 sklearn 默认采用的评价指标，其定义如下：

$$R^2 = 1 - \frac{\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^{m}(y^{(i)} - \bar{y})^2} \tag{3-73}$$

其中，$R^2 \in (-\infty, 1]$，其值越大表示模型越好，$\bar{y}$ 表示真实值的平均值，实现代码如下：

```
def R2(y, y_pre):
    u = np.sum((y - y_pre) ** 2)
    v = np.sum((y - np.mean(y)) ** 2)
    return 1 - (u / v)
```

3.8.2　回归指标示例代码

有了这些评估指标后，在训练模型时就可以选择其中的一些指标对模型的精度进行评估了，示例代码如下：

```
if __name__ == '__main__':
    y_true = 2 * np.random.randn(200) + 1
    y_pred = np.random.randn(200) + y_true
    print(f"MAE: {MAE(y_true, y_pred)}\n"
          f"MSE: {MSE(y_true, y_pred)}\n"
          f"RMSE: {RMSE(y_true, y_pred)}\n"
          f"MAPE: {MAPE(y_true, y_pred)}\n"
          f"R2: {R2(y_true, y_pred)}\n")
```

在上述代码中，第 2～3 行用来生成模拟的真实标签与预测值。第 4～8 行则表示不同指标下的评价结果。最后，上述代码运行结束后输出的结果如下：

```
MAE: 0.7395229164418393
MSE: 0.8560928033277224
RMSE: 0.9252528321100792
MAPE: 2.2088106952308864
R2: -0.2245663206367467
```

3.8.3　小结

本节首先通过一个示例介绍了为什么需要引入评估指标，即如何评价一个回归模型的优与劣，然后逐一介绍了 5 种常用的评估指标和实现方法；最后，还逐一展示了评价指标的示例用法。

3.9　分类模型评估指标

如同回归模型一样，分类模型在训练结束后同样需要一种测度来对模型的结果进行评判，以便于进行下一步流程。相较于回归模型的评估指标，分类模型的评估指标则相对更多且考虑的情况也更为繁杂。在接下来的这节内容中，将从零开始一步一步地详细介绍分类

任务中的几种常见的评估指标及其实现方法。

3.9.1 准确率

首先介绍分类任务中最常用的且最简单的评估指标准确率(Accuracy)。假定现在有一个猫狗识别程序,并且假定狗为正类别(Positives),猫为负类别(Negatives)。程序在对 12 张狗图片和 10 张猫图片进行识别后,判定其中 8 张图片为狗,14 张图片为猫。待程序识别完毕后,经人工核对在这 8 张程序判定为狗的图片中仅有 5 张图片的确为狗,14 张被判定为猫的图片中仅有 7 张的确为猫。

因此,准确率的定义为预测正确的样本数在总样本数中的占比,即在上述例子中程序的准确率为

$$\text{Accuracy}=\frac{\text{预测正确的样本数}}{\text{总样本数}}=\frac{5+7}{12+10}\approx 0.545 \tag{3-74}$$

以上就是准确率的定义及计算过程。

虽然准确率的计算过程简单,并且十分容易理解,但是准确率却存在着一个不容忽视的弊端。例如,现在需要训练一个癌细胞诊断模型来识别癌细胞,并且在训练数据中其中负样本(非癌细胞)有 10 万个,而正样本(癌细胞)只有 200 个。假如某个模型将其中的 105 个预测为正样本,将 100 095 个预测为负样本。最终经过核对后发现,正样本中有 5 个预测正确,负样本中有 99 900 个样本预测正确。那么此时该模型在训练集上的准确率为

$$\text{Accuracy}=\frac{99\,900+5}{100\,000+200}\approx 0.997 \tag{3-75}$$

但显然,这样的一个模型对于辅助医生决策来讲并没有任何作用。如果模型极端一点将所有的样本都预测为负样本,则模型的准确率高达 0.998,因此,在面对类似这种样本不均衡的任务中,并不能将准确率作为评估模型的唯一指标。此时就需要引入精确率和召回率来作为新的评价指标。

3.9.2 精确率与召回率计算

我们仍旧以上面的猫狗识别任务为例。在这 8 张被程序判定为狗的图片中仅有 5 张图片的确为狗,因此这 5 张图片就被称为预测正确的正样本(True Positives,TP),而余下的 3 张被称为预测错误的正样本(False Positives,FP)。同时,在这 14 张被程序判定为猫的图片中,仅有 7 张的确为猫,即预测正确的负样本(True Negatives,TN),而余下的 7 张被称为预测错误的负样本(False Negatives,FN)。

真实	预测	
	P	N
P	TP	FN
N	FP	TN

真实	预测	
	P	N
P	5	7
N	3	7

图 3-34 混淆矩阵图

此时,根据这一识别结果,便可以得到如图 3-34 所示的混淆矩阵(Confuse Matrix)。

如何来读这个混淆矩阵呢?读的时候首先横向看,然后纵向看。例如读 TP 的时候,首

先横向表示真实的正样本，其次纵向表示预测的正样本，因此 TP 表示的就是将正样本预测为正样本的个数，即预测正确，因此，同理共有以下 4 种情况。

(1) TP：表示将正样本预测为正样本，即预测正确。

(2) FN：表示将正样本预测为负样本，即预测错误。

(3) FP：表示将负样本预测为正样本，即预测错误。

(4) TN：表示将负样本预测为负样本，即预测正确。

如果此时突然问 FP 表示什么含义，则该怎样迅速地回答出来呢？我们知道 FP 从字面意思来看表示的是错误的正类，也就是说实际上它并不是正类，而是错误的正类，即实际上为负类，因此，FP 表示的就是将负样本预测为正样本的含义。再看一个 FN，其字面意思为错误的负类，也就是说实际上它表示的是正类，因此 FN 的含义就是将正样本预测为负样本。

在定义完上述 4 种分类情况后就能得出各种场景下的计算指标公式，如式(3-76)～式(3-78)所示。

$$\text{Accuracy}=\frac{\text{TP}+\text{TN}}{\text{TP}+\text{FP}+\text{FN}+\text{TN}} \tag{3-76}$$

$$\text{Precision}=\frac{\text{TP}}{\text{TP}+\text{FP}} \tag{3-77}$$

$$\text{Recall}=\frac{\text{TP}}{\text{TP}+\text{FN}} \tag{3-78}$$

$$F_{\text{score}}=(1+\beta^2)\frac{\text{Precision}\cdot\text{Recall}}{\beta^2\cdot\text{Precision}+\text{Recall}} \tag{3-79}$$

注意：当 F_{score} 中 $\beta=1$ 时称为 F_1 值，同时 F_1 也是用得最多的 F_{score} 评价指标。

在这里，我们又一次根据不同的定义形式得到了准确率的计算方式，但其本质依旧等同于式(3-74)。同时还可以看到，精确率计算的是预测对的正样本在整个预测为正样本中的比重，而召回率计算的是预测对的正样本在整个真实正样本中的比重，因此一般来讲，召回率越高也就意味着这个模型寻找正样本的能力越强(例如在判断是否为癌细胞时，寻找正样本癌细胞的能力就十分重要)，而 F_{score} 则是精确率与召回率的调和平均。

因此，根据精确率和召回率的定义，还可以通过更直观的图示来进行说明，如图 3-35 所示。

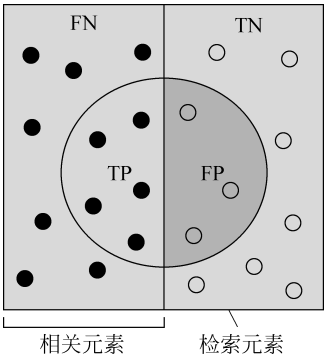


图 3-35　分类情况分布图

左侧的所有实心样本点为正样本(相关元素)，右侧的所有空心点为负样本，中间的圆形区域为模型预测的正样本(检索元素)，即圆形左侧为模型将正样本预测为正样本的情况，右侧为模型将负样本预测为正样本的情况。例如现在可以想象这么一个场景，某一次在

使用搜索引擎搜索相关内容(正样本)时,搜索引擎一共检索并返回了 30 个搜索页面(搜索引擎认为的正样本),而搜索引擎返回的结果就相当于图 3-35 中对应的圆形区域,所以精确率和召回率还可以通过图 3-36 来形象地进行表示。

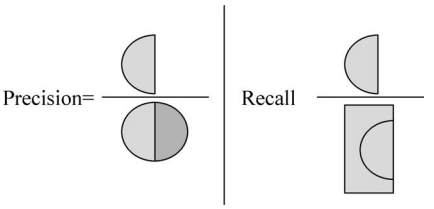


图 3-36 精确率召回率图示

从图 3-36 中更能直观地看出,精确率计算的是预测正确的正样本在整个被预测为正样本中的占比,而召回率计算的是预测正确的正样本在所有真实正样本中的占比。

在有了上述各项指标的定义之后,下面就来计算示例中各项指标的实际值。

1. 准确率

根据式(3-76)可得,上述示例中模型的准确率为

$$\text{Accuracy}=\frac{5+7}{5+3+7+7}\approx 0.545 \tag{3-80}$$

2. 精确率与召回率

$$\text{Precision}=\frac{5}{5+3}=0.625 \tag{3-81}$$

$$\text{Recall}=\frac{5}{5+7}\approx 0.417 \tag{3-82}$$

$$F_1=(1+1^2)\times\frac{5/8\times 5/12}{1^2\times 5/8+5/12}=0.5 \tag{3-83}$$

因此,对于 3.9.1 节中癌细胞识别模型中的结果来讲,其精确率和召回率分别为

$$\text{Precision}=\frac{5}{105}\approx 0.048 \tag{3-84}$$

$$\text{Recall}=\frac{5}{200}=0.025 \tag{3-85}$$

根据式(3-84)和式(3-85)中的结果可以看出,尽管该模型的准确率达到了 0.997,但是从精确率和召回率来看,这个模型显然是非常糟糕的。

3.9.3 准确率与召回率的区别

介绍到这里可能有的读者会问,在上述问题中既然精确率和召回率都能够解决准确率所带来的弊端问题,那么可不可以只用其中一个呢?答案是不可以。下面再次以上面的癌细胞判别模型为例,并以 3 种情况来进行说明。

情况一:将所有样本均预测为正样本,此时有 TP=200,FP=100 000,TN=0,FN=0,则

$$\text{Accuracy}=\frac{200}{100\,200}\approx 0.002$$

$$\text{Precision}=\frac{200}{100\,000+200}\approx 0.002$$

$$\text{Recall}=\frac{200}{200+0}=1.0 \tag{3-86}$$

情况二：将其中50个样本预测为正样本，将100 150个样本预测为负样本。最终经过核对后发现，正样本中有50个预测正确，负样本中有100 000个样本预测正确。此时有TP=50，FP=0，TN=100 000，FN=150，则

$$\begin{aligned}\text{Accuracy}&=\frac{50+100\,000}{100\,200}\approx 0.999\\ \text{Precision}&=\frac{50}{50+0}=1.0\\ \text{Recall}&=\frac{50}{50+150}=0.25\end{aligned} \tag{3-87}$$

情况三：将其中的210个样本预测为正样本，将99 990个样本预测为负样本。最终经过核对后发现，正样本中有190个预测正确，负样本中有99 980个样本预测正确。此时有TP=190，FP=20，TN=99 980，FN=10，则

$$\begin{aligned}\text{Accuracy}&=\frac{190+99\,980}{100\,200}\approx 0.999\\ \text{Precision}&=\frac{190}{190+20}\approx 0.905\\ \text{Recall}&=\frac{190}{190+10}=0.95\end{aligned} \tag{3-88}$$

根据这3种情况下模型的表现结果可以知道，如果仅从单一指标来看，则无论是准确率、精确率还是召回率都不能全面地评估一个模型，并且至少应该选择精确率和召回率同时作为评价指标。同时可以发现，精确率和召回率之间在一定程度上存在着某种相互制约的关系，即如果一味地只追求提高精确率，则召回率可能会很低，反之亦然。

所以，在实际情况中会根据需要来选择不同的侧重点，当然最理想的情况就是在取得高召回率的同时还能保持较高的精确率。最后，也可以直接计算 F_{score} 来进行综合评估，例如上述3种情况对应的 F_1 值分别为0.039、0.4和0.927，因此，对于一个分类模型来讲，如果想要在精确率和召回率之间取得一个较好的平衡，则最大化 F_1 值是一个有效的方法。

3.9.4　多分类下的指标计算

经过以上内容的介绍，对于分类任务下的准确率、精确率、召回率和 F 值已经有了一定的了解，但这里需要注意的一个问题是，通常在绝大多数任务中并不会明确哪一类别是正样本，哪一类别是负样本，例如之前介绍的手写体识别任务，对于每个类别来讲都可以计算其各项指标，但是准确率依旧只有一个。

假设有以下三分类任务的预测值与真实值，结果如下：

```
1  y_true = [1, 1, 1, 0, 0, 0, 2, 2, 2, 2]
2  y_pred = [1, 0, 0, 0, 2, 1, 0, 0, 2, 2]
```

真实	预测		
	0	1	2
0	1	1	1
1	2	1	0
2	2	0	2

图 3-37　多分类混淆矩阵

根据这一结果，便可以得到一个混淆矩阵，如图 3-37 所示。

由于是多分类，所以也就不止正样本和负样本两个类别，此时这张图该怎么读呢？方法还是同图 3-34 中的一样，先横向看，再纵向看。例如第 2 行灰色单元格中的 1 表示的就是将真实值 0 预测为 0 的样本个数（预测正确），接着右边的 1 表示的就是将真实值 0 预测为 1 的个数，第 3 行灰色单元格中的 1 表示的就是将真实值 1 预测为 1 的个数，第 4 行灰色单元格中的 2 表示的就是将真实值 2 预测为 2 的个数，也就是说只有对角线上的值才表示模型预测正确的样本数量。接下来开始对每个类别的各项指标进行计算。

1. 对于类别 0

在上面我们介绍过，精确率计算的是预测对的正样本在整个预测为正样本中的比重。根据图 3-37 可知，对于类别 0 来讲，预测对的正样本（类别 0）的数量为 1，而整个预测为正样本的数量为 5，因此，类别 0 对应的精确率为

$$\text{Precision}=\frac{1}{1+2+2}=0.2 \tag{3-89}$$

同时，召回率计算的是预测对的正样本在整个真实正样本中的比重。根据图 3-37 可知，对于类别 0 来讲，预测对的正样本（类别 0）的数量为 1，而整个真实正样本 0 的个数为 3（图 3-37 中第 2 行的 3 个 1），因此，对于类别 0 来讲其召回率为

$$\text{Recall}=\frac{1}{1+1+1}=0.33 \tag{3-90}$$

因此，其 F_1 值为

$$F_1=\frac{2\times 0.2\times 0.33}{0.2+0.33}=0.25 \tag{3-91}$$

2. 对于类别 1

对于类别 1 来讲，预测对的正样本（类别 1）的数量为 1，而整个预测为类别 1 的样本数量为 2，因此，其精确率为

$$\text{Precision}=\frac{1}{1+1+0}=0.5 \tag{3-92}$$

同理，其召回率和 F_1 值分别为

$$\text{Recall}=\frac{1}{1+2}=0.33$$

$$F_1=\frac{2\times 0.5\times 0.33}{0.5+0.33}=0.40 \tag{3-93}$$

3. 对于类别 2

$$\text{Precision}=\frac{2}{1+0+2}=0.67$$

$$\text{Recall}=\frac{2}{2+0+2}=0.50$$

$$F_1 = \frac{2 \times 0.67 \times 0.50}{0.67 + 0.50} = 0.57 \tag{3-94}$$

4. 整体准确率

$$\text{Accuracy} = \frac{1+1+2}{1+1+1+2+1+0+2+0+2} = 0.4 \tag{3-95}$$

此时，对于多分类场景下各个类别评价指标的计算就介绍完了，不过有读者可能会发现这里每个类别下都有 3 个评估值，如果有 5 个或 10 个类别，则观察起来简直难以想象，但如果想要衡量模型整体的精确率、召回率或者 F 值，则该怎么处理呢？对于分类结果整体的评估结果常见的做法有两种：第 1 种是取算术平均，第 2 种是加权平均[3]。

(1) 算术平均：所谓算术平均也叫作宏平均(Macro Average)，也就是等权重地对各类别的评估值进行累加求和。例如对于上述三分类任务来讲，其精确率、召回率和 F_1 值分别为

$$\begin{aligned}
\text{Precision} &= \frac{1}{3} \times (0.20 + 0.50 + 0.67) = 0.46 \\
\text{Recall} &= \frac{1}{3} \times (0.33 + 0.33 + 0.50) = 0.39 \\
F_1 &= \frac{1}{3} \times (0.25 + 0.40 + 0.57) = 0.41
\end{aligned} \tag{3-96}$$

(2) 加权平均：所谓加权平均就是以不同的加权方式来对各类别的评估值进行累加求和。这里只介绍一种用得最多的加权方式，即按照各类别的样本数在总样本中的占比进行加权。对于图 3-37 中的分类结果来讲，加权后的精确率、召回率和 F_1 值分别为

$$\begin{aligned}
\text{Precision} &= \frac{3}{10} \times 0.2 + \frac{3}{10} \times 0.50 + \frac{4}{10} \times 0.67 = 0.48 \\
\text{Recall} &= \frac{3}{10} \times 0.33 + \frac{3}{10} \times 0.33 + \frac{4}{10} \times 0.50 = 0.40 \\
F_1 &= \frac{3}{10} \times 0.25 + \frac{3}{10} \times 0.40 + \frac{4}{10} \times 0.57 = 0.42
\end{aligned} \tag{3-97}$$

最后，再来介绍如何编码实现上述各项指标的计算。根据图 3-34 和图 3-37 可知，计算各项评估值的关键便是如何计算得到这个混淆矩阵。在这里，可以借助 sklearn 框架中的 sklearn.metrics.confusion_matrix 来完成混淆矩阵的计算。不过，更方便地，还可以直接使用 sklearn.metrics 中的 classification_report 模块来完成所有指标的计算过程。

各项指标的计算过程的示例代码如下：

```
from sklearn.metrics import classification_report
if __name__ == '__main__':
    y_true = [1, 1, 1, 0, 0, 0, 2, 2, 2, 2]
    y_pred = [1, 0, 0, 0, 2, 1, 0, 0, 2, 2]
    result = classification_report(y_true, y_pred,
            target_names=['class 0', 'class 1', 'class 2'])
    print(result)
```

上述代码运行结束后便可以得到如下所示的结果：

```
1                precision recall f1-score support
2       class 0      0.20     0.33     0.25      3
3       class 1      0.50     0.33     0.40      3
4       class 2      0.67     0.50     0.57      4
5       accuracy                       0.40      10
6       macro avg    0.46     0.39     0.41      10
7   weighted avg     0.48     0.40     0.42      10
```

在上述结果中,第 2～3 行表示各个类别下的不同评估结果。第 5～7 行分别是准确率、宏平均和加权平均,可以同式(3-96)和式(3-97)进行对比。

3.9.5 Top-*K* 准确率

在上面的内容中,我们详细地介绍了分类任务中常见的 4 种评估指标,看上去似乎已经够用了,但事实上在某些场景下这类指标还是过于严格,尤其是在图片分类任务中。例如有一个五分类模型,某个样本的真实标签为第 0 个类别,而模型预测结果的概率分布为[0.32,0.1,0.2,0.05,0.33]。如果是取概率值最大的索引作为类别,则该样本的预测结果将为第 4 个类别,但是我们就一定能说模型表现的结果很差吗?

因此,一种可行的做法就是采用 Top-*K* 准确率来进行评估。不过什么是 Top-K 准确率呢? 用一句话概括,Top-*K* 准确率就是用来计算预测结果中概率值最大的前 *K* 个结果包含正确标签的占比。换句话说,平常所讲的准确率其实就是 Top-1 准确率。例如对于上面的例子来讲,如果采用 Top-2 的计算方式,则该预测结果就算是正确的,因为概率值排序第 2 位的 0.32 对应的类别 0 就是样本真实的结果。

因此可以看出,Top-*K* 准确率考虑的是预测结果中最有可能的 *K* 个结果是否包含真实标签,如果包含,则算预测正确,如果不包含,则算预测错误,所以这里也能得出一个结论,*K* 值越大计算得到的 Top-*K* 准确率就会越高,极端情况下如果取 *K* 值为分类数,则得到的准确率就一定是 1,但通常情况下只会看模型的 Top-1、Top-3 或 Top-5 准确率。

介绍完了什么是 Top-*K* 准确率,下面就来看如何通过代码实现。从上面的介绍可以知道,想要计算 Top-*K* 准确率,首先需要得到的就是预测概率中最大的前 *K* 个值所对应的预测标签。在 PyTorch 中可以通过 torch.topk()函数来返回前 *K* 个值及其对应的索引,实现代码如下:

```
1  def calculate_top_k_accuracy(logits, targets, k = 2):
2      values, indices = torch.topk(logits, k = k, sorted = True)
3      y = torch.reshape(targets, [-1, 1])
4      correct = (y == indices) * 1.
5      top_k_accuracy = torch.mean(correct) * k
6      return top_k_accuracy
```

在上述代码中,第 1 行 logits 表示每个样本预测的概率分布形状为[m,n],targets 表示每个样本的真实标签形状为[m,]。第 2 行用于返回降序后前 *K* 个值及其对应的索引(类标)。第 4 行用于对比预测结果的 *K* 个值中是否包含正确标签中的结果。第 5 行用于计算最后的准确率。

最后,可以通过以下方式来使用上述方法:

```
if __name__ == '__main__':
    logits = torch.tensor([[0.1, 0.3, 0.2, 0.4],
                           [0.5, 0.01, 0.9, 0.4]])
    y = torch.tensor([3, 0])
    print(calculate_top_k_accuracy(logits, y, k=1).item()) #0.5
    print(calculate_top_k_accuracy(logits, y, k=2).item()) #1.0
```

在上述代码中，从第 2～3 行的预测结果和第 4 行的真实标签可以看出，如果 $k=1$，则只有第 1 个样本预测正确，此时的准确率为 0.5，如果 $k=2$，则两个样本都预测正确，此时准确率为 1。

3.9.6　小结

本节首先介绍了分类任务中最常见的且最容易理解的评估指标准确率，然后由准确率的弊端引出了为什么需要召回率和精确率，并介绍了两者的调和形式 F 值；最后详细介绍了多分类场景下各项指标的计算方式及其编码实现，同时也介绍了某些特定场景下模型 Top-K 准确率的计算原理和实现方式。

3.10　过拟合与正则化

经过前面几节内容的介绍，对于深度学习的理念及最基本的回归和分类模型已经有了清晰的认识。在接下来的内容中，将逐步介绍深度学习中关于模型优化的一些基本内容，包括模型的过拟合、正则化和丢弃法等。

3.10.1　模型拟合

3.3 节首次引入了梯度下降这一优化算法，以此来最小化线性回归中的目标函数，并且在经过多次迭代后便可以得到模型中对应的参数。此时可以发现，模型的参数是一步一步地根据梯度下降算法更新而来的，直至目标函数收敛，也就是说这是一个循序渐进的过程，因此，这一过程也被称作拟合(Fitting)模型参数的过程，当这个过程执行结束后就会产生多种拟合后的状态，例如过拟合(Overfitting)和欠拟合(Underfitting)等。

3.8 节介绍了几种评估回归模型常用的指标，但现在有一个问题：当 MAE 或者 RMSE 越小时就代表模型越好吗？还是说在某种条件下其越小就越好呢？细心的读者可能一眼便明了，肯定是在某种条件下其越小所对应的模型才越好。那么这其中到底是怎么回事呢？

假设现在有一批样本点，它本是由函数 $\sin(x)$ 生成(现实中并不知道)的，但由于其他因素的缘故，使我们得到的样本点并没有准确地落在曲线 $\sin(x)$ 上，而是分布在其附近，如图 3-38 所示。

黑色圆点为训练集，黑色曲线为样本真实的分布曲线。现在需要根据训练集来建立并训练模型，然后得到相应的预测函数。现在分别用 3 个不同的模型 A、B 和 C(复杂度依次增加，例如更多的网络层数和神经元个数等)来分别根据这 12 个样本点进行建模，最终便可

以得到如图 3-39 所示的结果。

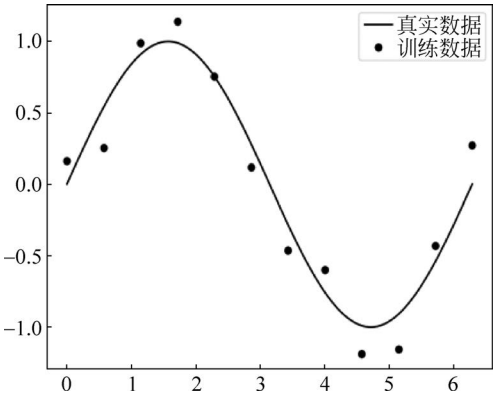


图 3-38　正弦样本点图形

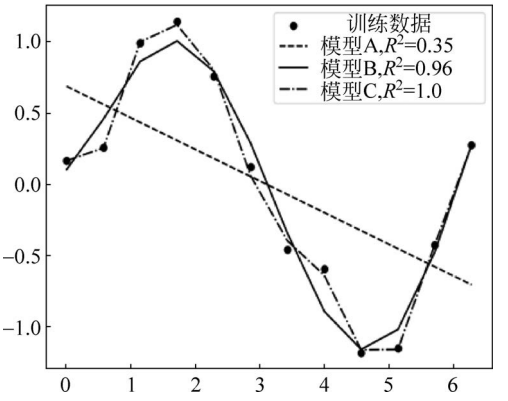


图 3-39　正弦样本点拟合图形

从图 3-39 中可以看出，随着模型复杂度的增加，R^2 指标的值也越来越大（$R^2 \in (-\infty, 1]$），并且在模型 C 中 R^2 还达到了 1.0，但是最后就应该选择模型 C 吗？

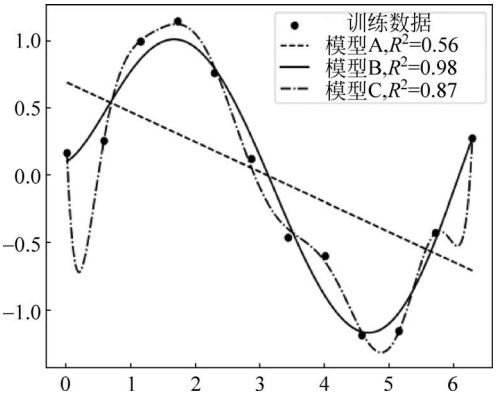


图 3-40　正弦样本点过拟合图形

不知道又过了多久，突然一名客户要买你的这个模型进行商业使用，同时客户为了评估这个模型的效果自己又带来了一批新的含有标签的数据（虽然模型 C 已经用 R^2 测试过，但客户并不完全相信，万一你对这个模型作弊呢）。于是你拿着客户的新数据（也是由 sin（x）所生成的），然后分别用上面的 3 个模型进行预测，并得到了如图 3-40 所示的可视化结果。

各个曲线表示根据新样本预测值绘制得到的结果。此时令你感到奇怪的是，为什么模型 B 的结果会好于模型 C 的结果？问题出在哪里？其原因在于，当第 1 次通过这 12 个样本点进行建模时，为了尽可能地使“模型好（表现形式为 R^2 尽可能大）”而使用了非常复杂的模型，尽管最后每个训练样本点都“准确无误”地落在了预测曲线上，但是这却导致最后模型在新数据上的预测结果严重地偏离了其真实值。

3.10.2　过拟合与欠拟合概念

在机器学习领域中，通常将建模时所使用的数据叫作训练集（Training Dataset），例如图 3-38 中的 12 个样本点。将测试时所使用的数据集叫作测试集（Testing Dataset）。同时把模型在训练集上产生的误差叫作训练误差（Training Error），把模型在测试集上产生的误差叫作泛化误差（Generalization Error），最后也将整个拟合模型的过程称作训练（Training）[4]。

进一步地讲，将 3.10.1 节中模型 C 所产生的现象叫作过拟合(Overfitting)，即模型在训练集上的误差很小，但在测试集上的误差却很大，也就是泛化能力弱；相反，将其对立面模型 A 所产生的现象叫作欠拟合(Underfitting)，即模型训练集和测试集上的误差都很大；同时，将模型 B 对应的现象叫作恰拟合(Goodfitting)，即模型在训练集和测试集上都有着不错的效果。

同时，需要说明的是，3.10.1 节仅以回归任务为例来向读者直观地介绍了什么是过拟合与欠拟合，但并不代表这种现象只出现在回归模型中，事实上所有的深度学习模型都会存在着这样的问题，因此一般来讲，所谓过拟合现象指的是模型在训练集上表现很好，但在测试集上表现糟糕；欠拟合现象是指模型在两者上的表现都十分糟糕，而恰拟合现象是指模型在训练集上表现良好(尽管可能不如过拟合时好)，但同时在测试集上也有着不错的表现。

3.10.3　解决欠拟合与过拟合问题

1. 如何解决欠拟合问题

经过上面的描述已经对欠拟合有了一个直观的认识，所谓欠拟合就是训练出来的模型根本不能较好地拟合现有的训练数据。在深度学习中，要解决欠拟合问题相对来讲较为简单，主要分为以下两种方法。

(1) 重新设计更为复杂的模型，例如增加网络的深度、增加神经元的个数或者采用更为复杂的网络架构(如 Transformer)。

(2) 减小正则化系数，当模型出现欠拟合现象时，可以通过减小正则化中的惩罚系数来减缓欠拟合现象，这一点将在 3.10.4 节中进行介绍。

2. 如何解决过拟合问题

对于如何有效地缓解模型的过拟合现象，常见的做法主要分为以下 4 种方法。

(1) 收集更多数据，这是一个最为有效但实际操作起来又是最为困难的一种方法。训练数据越多，在训练过程中也就越能纠正噪声数据对模型所造成的影响，使模型不易过拟合，但是对于新数据的收集往往有较大的困难。

(2) 降低模型复杂度，当训练数据过少时，使用较为复杂的模型极易产生过拟合现象，例如 3.10.1 节中的示例，因此可以通过适当减少模型的复杂度来达到缓解模型过拟合的现象。

(3) 正则化方法，在出现过拟合现象的模型中加入正则化约束项，以此来降低模型过拟合的程度，这部分内容将在 3.10.4 节中进行介绍。

(4) 集成方法，将多个模型集成在一起，以此来达到缓解模型过拟合的目的。

3. 如何避免过拟合

为了避免训练出来的模型产生过拟合现象，在模型训练之前一般会将获得的数据集划分成两部分，即训练集与测试集，并且两者的比例一般为 7∶3，其中训练集用来训练模型(降低模型在训练集上的误差)，然后用测试集来测试模型在未知数据上的泛化误差，观察是否产生了过拟合现象[1]。

但是由于一个完整的模型训练过程通常会先用训练集训练模型，再用测试集测试模型，而在绝大多数情况下不可能第1次就选择了合适的模型，所以又会重新设计模型(如调整网络层数、调整正则化系数等)进行训练，然后用测试集进行测试，因此在不知不觉中，测试集也被当成了训练集在使用，所以这里还有另外一种数据的划分方式，即训练集、验证集(Validation Data)和测试集，并且这三者的比例一般为7∶2∶1，此时的测试集一般通过训练集和验证集选定模型后为最后的测试所用。

在实际训练中应该选择哪种划分方式呢？这一般取决于训练者对模型的要求程度。如果要求严苛就划分为3份，如果不那么严格，则可以划分为两份，也就是说并没有硬性的标准。

3.10.4 泛化误差的来源

根据3.10.3节内容可以知道，模型产生过拟合的现象表现为在训练集上误差较小，而在测试集上误差较大，并且还讲道，之所以会产生过拟合现象是由于训练数据中可能存在一定的噪声，而在训练模型时为了尽可能地做到拟合每个样本点(包括噪声)，往往就会使用复杂的模型。最终使训练出来的模型在很大程度上受到了噪声数据的影响，例如真实的样本数据可能更符合一条直线，但是由于个别噪声的影响使训练出来的是一条曲线，从而使模型在测试集上表现糟糕，因此，可以将这一过程看作由糟糕的训练集导致了糟糕的泛化误差，但是，如果仅仅从过拟合的表现形式来看，糟糕的测试集(噪声多)则可能导致糟糕的泛化误差。

在接下来的内容中，将分别从这两个角度来介绍正则化(Regularization)方法中最常用的 ℓ_2 正则化是如何来解决这一问题的。

这里以线性回归为例，首先来看一下在线性回归的目标函数后面再加上一个 ℓ_2 正则化项的形式。

$$J=\frac{1}{2m}\sum_{i=1}^{m}\left[y^{(i)}-\left(\sum_{j=1}^{n}w_j x_j^{(i)}+b\right)\right]^2+\frac{\lambda}{2n}\sum_{j=1}^{n}(w_j)^2,\quad \lambda>0 \tag{3-98}$$

在式(3-98)中的第2项便是新加入的 ℓ_2 正则化项(Regularization Term)，那它有什么作用呢？根据3.1.3节中的内容可知，当真实值与预测值之间的误差越小(表现为损失值趋于0)时，也就代表着模型的预测效果越好，并且可以通过最小化目标函数来达到这一目的。由式(3-98)可知，为了最小化目标函数 J，第2项的结果也必将逐渐地趋于0。这使最终优化求解得到的 w_j 均会趋于0，进而得到一个平滑的预测模型。这样做的好处是什么呢？

3.10.5 测试集导致的泛化误差

所谓测试集导致糟糕的泛化误差是指训练集本身没有多少噪声，但由于测试集含有大量噪声，使训练出来的模型在测试集上没有足够的泛化能力，从而产生了较大的误差。这种情况可以看作模型过于准确而出现了过拟合现象。正则化方法是怎样解决这个问题的呢？

$$y=\sum_{j=1}^{n}x_j w_j+b \tag{3-99}$$

假如式(3-99)所代表的模型就是根据式(3-98)中的目标函数训练而来的，此时当某个新输入样本(含噪声)的某个特征维度由训练时的 x_j 变成了现在的$(x_j+\Delta x_j)$，那么其预测输出就由训练时的 $\hat{y}$ 变成了现在的 $\hat{y}+\Delta x_j w_j$，即产生了 $\Delta x_j w_j$ 的误差，但是，由于 w_j 接近于 0，所以这使模型最终只会产生很小的误差。同时，如果 w_j 越接近于 0，则产生的误差就会越小，这意味着模型越能抵抗噪声的干扰，在一定程度上越能提升模型的泛化能力[4]。

由此便可以知道，在原始目标函数中加入正则化项，便能够使训练得到的参数趋于平滑，进而能够使模型对噪声数据不再那么敏感，缓解了模型的过拟合现象。

3.10.6　训练集导致的泛化误差

所谓训练集导致糟糕的泛化误差是指，由于训练集中包含了部分噪声，所以导致在训练模型的过程中为了能够尽可能地最小化目标函数而使用了较为复杂的模型，使最终得到的模型并不能在测试集上有较好的泛化能力(如 3.10.1 节中的示例)，但这种情况完全是因为模型不合适而出现了过拟合的现象，而这也是最常见的过拟合的原因。ℓ_2 正则化方法又是怎样解决在训练过程中就能够降低对噪声数据的敏感度的呢？为了便于后面的理解，先从图像上来直观地理解正则化到底对目标函数做了什么。

左右两边黑色实线为原始目标函数，黑色虚线为加了 ℓ_2 正则化后的目标函数，如图 3-41 所示。可以看出黑色实线的极值点均发生了明显改变，并且不约而同地都更靠近原点。

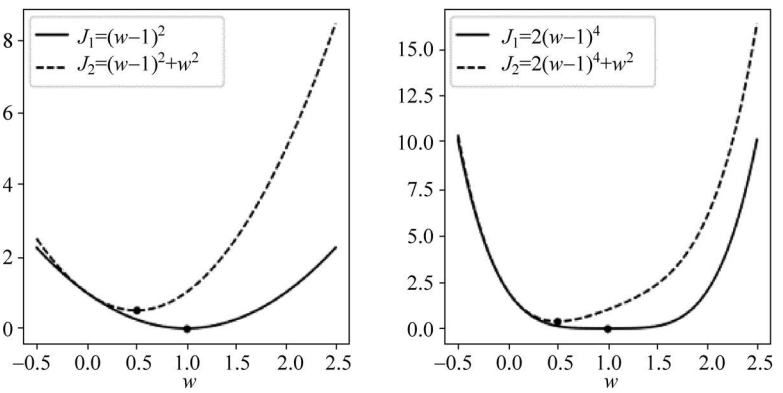


图 3-41　ℓ_2 正则化图形

再来看一张包含两个参数的目标函数在加入 ℓ_2 正则化后的结果，如图 3-42 所示。

图中黑色虚线为原始目标函数的等高线，黑色实线为施加正则化后目标函数的等高线。可以看出，目标函数的极值点同样也发生了变化，从原始的(0.5，0.5)变成了(0.0625，0.25)，而且也更靠近原点(w_1 和 w_2 变得更小了)。到此似乎可以发现，正则化能够使原始目标函数极值点发生改变，并且同时还有使参数趋于 0 的作用。事实上也正是因为这个原因才使 ℓ_2 正则化具有缓解过拟合的作用，但原因是什么呢？

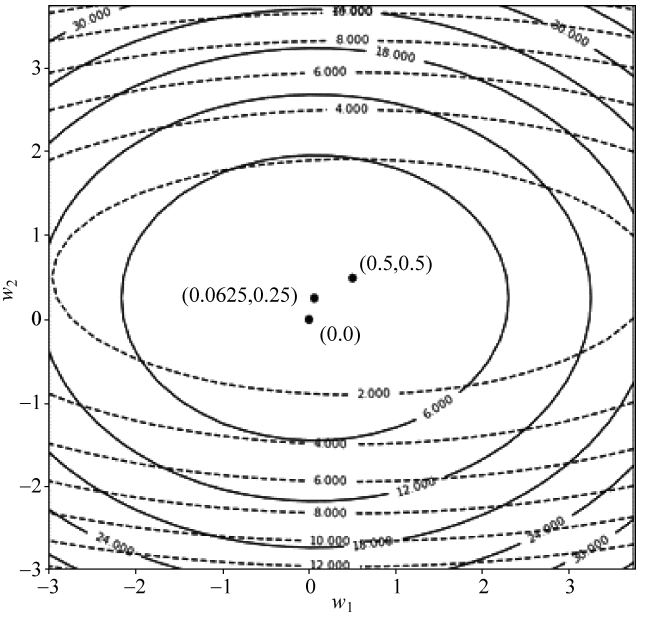


图 3-42 ℓ_2 正则化投影图形

3.10.7 ℓ_2 正则化原理

以目标函数 $J_1=1/6(w_1-0.5)^2+(w_2-0.5)^2$ 为例，其取得极值的极值点为(0.5,0.5)，并且 J_1 在极值点处的梯度为(0,0)。当对其施加正则化 $R=(w_1^2+w_2^2)$ 后，由于 R 的梯度方向是远离原点的(因为 R 为一个二次曲面)，所以给目标函数加入正则化，实际上等价于给目标函数施加了一个远离原点的梯度。通俗点讲，正则化给原始目标函数的极值点施加了一个远离原点的梯度(甚至可以想象成施加了一个力的作用)，因此，这也就意味着对于施加正则化后的目标函数 $J_2=J_1+R$ 来讲，J_2 的极值点(0.0625,0.25)相较于 J_1 的极值点(0.5,0.5)更加靠近于原点，而这也就是 ℓ_2 正则化的本质。

注意：在通过梯度下降算法最小化目标函数的过程中，需要得到的是负梯度方向，因此上述极值点会向着原点的方向移动。

假如有一个模型 A，它在含有噪声的训练集上表现异常出色，使目标函数 $J_1(\hat{w})$ 的损失值等于 0(也就是拟合到了每个样本点)，即在 $w=\hat{w}$ 处取得了极值。现在，在 J_1 的基础上加入 ℓ_2 正则化项构成新的目标函数 J_2，然后来分析一下通过最小化 J_2 求得的模型 B 到底发生了什么样的变化。

$$J_1=\frac{1}{2m}\sum_{i=1}^{m}\left[y^{(i)}-\left(\sum_{j=1}^{n}x_j^{(i)}w_j+b\right)\right]^2$$

$$J_2=J_1+\frac{\lambda}{2n}\sum_{j=1}^{n}(w_j)^2,\quad \lambda>0 \tag{3-100}$$

从式(3-100)可知，由于 J_2 是由 J_1 加入正则化项构成的，同时根据先前的铺垫可知，J_2 将在离原点更近的极值点 $w=\tilde{w}$ 处取得 J_2 的极值，即通过最小化含正则化项的目标函数 J_2，将得到 $w=\tilde{w}$ 这个最优解，但是需要注意，此时的 $w=\tilde{w}$ 将不再是 J_1 的最优解，即 $J_1(\tilde{w})\neq 0$，因此通过最小化 J_2 求得的最优解 $w=\tilde{w}$ 将使 $J_1(\tilde{w})>J_1(\hat{w})$，而这就意味着模型 B 比模型 A 更简单了，也就代表从一定程度上缓解了 A 的过拟合现象。

同时，由式(3-98)可知，通过增大参数 λ 的取值可以对应增大正则化项所对应的梯度，而这将使最后得到更加简单的模型(参数值更加趋于 0)。也就是 λ 越大，在一定程度上越能缓解模型的过拟合现象，因此，参数 λ 又叫作惩罚项(Penalty Term)或者惩罚系数。

最后，从上面的分析可知，在第 1 种情况中 ℓ_2 正则化可以看作使训练好的模型不再对噪声数据那么敏感，而对于第 2 种情况来讲，ℓ_2 正则化则可以看作使模型不再那么复杂，但其实两者的原理归结起来都是一回事，那就是通过较小的参数取值，使模型变得更加简单。

3.10.8　ℓ_2 正则化中的参数更新

在给目标函数施加正则化后也就意味着其关于参数的梯度发生了变化。不过幸运的是正则化被加在原有的目标函数中，因此其关于参数 w 的梯度也只需加上惩罚项中对应参数的梯度，同时关于偏置 b 的梯度并没有改变。

以线性回归为例，根据式(3-98)可知，目标函数关于 w_j 的梯度为

$$\frac{\partial J}{\partial w_j}=-\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}-\left(\sum_{j=1}^{n}x_j^{(i)}w_j+b\right)\right]x_j^{(i)}+\frac{\lambda}{n}w_j \tag{3-101}$$

因此，对于任意目标函数 J 来讲，其在施加 ℓ_2 正则化后的梯度下降迭代公式为

$$w=w-\alpha\left(\frac{\partial J}{\partial w}+\frac{\lambda}{n}w\right)=\left(1-\alpha\frac{\lambda}{n}\right)w-\alpha\frac{\partial J}{\partial w} \tag{3-102}$$

从式(3-102)可以看出，相较于之前的梯度下降更新公式，ℓ_2 正则化会令权重 w 先用自身乘以小于 1 的系数，再减去不含惩罚项的梯度，这也将使模型参数在迭代训练的过程中以更快的速度趋近于 0，因此 ℓ_2 正则化又叫作权重衰减(Weight Decay)法[1]。

3.10.9　ℓ_2 正则化示例代码

在介绍完 ℓ_2 正则化的原理后，下面以加入正则化的线性回归模型为例进行演示。完整代码见 Code/Chapter03/C16_L2Regularization/main.py 文件。

1. 制作数据集

由于这里要模拟模型的过拟合现象，所以需要先制作一个容易导致过拟合的数据集，例如特征数量远大于训练样本数量，具体的代码如下：

```
def make_data():
    np.random.seed(1)
    n_train, n_test, n_features = 80, 110, 150
    w, b = np.random.randn(n_features, 1) * 0.01, 0.01
    x = np.random.normal(size=(n_train + n_test, n_features))
```

```
    y = np.matmul(x, w) + b
    y += np.random.normal(scale = 0.3, size = y.shape)
    x = torch.tensor(x, dtype = torch.float32)
    y = torch.tensor(y, dtype = torch.float32)
    x_train, x_test = x[:n_train, :], x[n_train:, :]
    y_train, y_test = y[:n_train, :], y[n_train:, :]
    return x_train, x_test, y_train, y_test
```

在上述代码中，第 1 行用于设定一个随机种子，保证每次生成的数据一样，使结果可复现。第 3 行用来指定训练样本、测试样本和特征的数量。第 4～7 行用于生成原始样本并在真实值中加入相应的噪声。第 8～9 行用于将 NumPy 中的向量转换为 PyTorch 中的张量。第 10～12 行用于划分数据集并返回。

2. 定义 l_2 惩罚项

由于整个线性回归的模型定义和训练部分的代码在 3.2.2 节房价预测实现中已经介绍过，因此这里就不再赘述了，只是介绍如何在原始目标函数中加入 l_2 惩罚项，示例代码如下：

```
def train(x_train, x_test, y_train, y_test, lambda_term = 0.):
    ......
    loss = nn.MSELoss() #定义损失函数
    optimizer = torch.optim.SGD(net.parameters(), lr = lr,
                                weight_decay = lambda_term)
    loss_train,loss_test = [], []
    for epoch in range(epochs):
        logits = net(x_train)
        l = loss(logits, y_train)
        loss_train.append(l.item())
        logits = net(x_test)
        ll = loss(logits, y_test)
        loss_test.append(ll.item())
        optimizer.zero_grad()
        l.backward()
        optimizer.step() #执行梯度下降
    return loss_train, loss_test
```

在上述代码中，第 3 行用于定义损失函数。第 4 行用于指定优化器，其中 weight_decay 参数便是 ℓ_2 正则化中的惩罚项系数，在默认情况下为 0，即不使用正则化。第 7～9 行用于在训练集上进行正向传播并保存对应的损失值。第 10～12 行则用于在测试集上进行正向传播并保存损失值。第 13～15 行用于在训练集上进行反向传播并更新模型参数。第 16 行用于分别返回模型在训练集和测试集上的损失值。

在定义完上述各个函数后，便可以用来分别训练带正则化项和不带正则化项（将 lambda_term 参数设为 0）的线性回归模型，最终得到的损失变化如图 3-43 所示。

在图 3-43 中，左边为未添加正则化项时训练误差和测试误差的走势。可以明显地看出模型在测试集上的误差远大于在训练集上的误差，这就是典型的过拟合现象。右图为使用正则化后模型的训练误差和测试误差，可以看出虽然训练误差有些许增加，但是测试误差在很大程度上得到了降低[1]。这就说明正则化能够很好地缓解模型的过拟合现象。

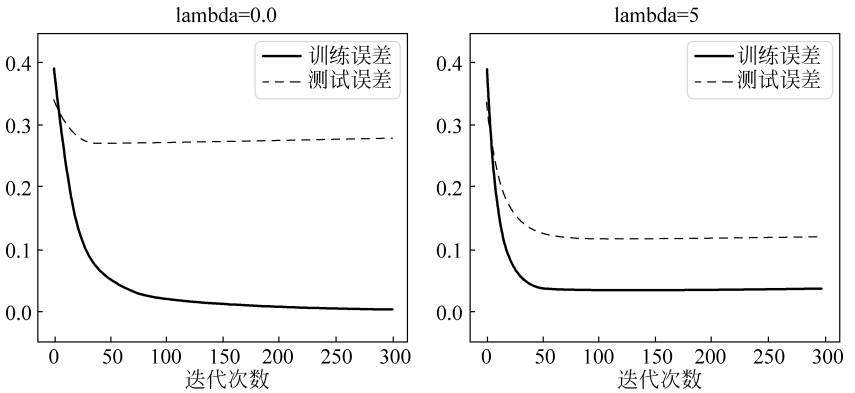


图 3-43　ℓ_2 正则化损失图

3.10.10　ℓ_1 正则化原理

在介绍完 ℓ_2 正则化后再来简单地看一下 ℓ_1 正则化背后的思想原理。如式(3-103)所示，这便是加入 ℓ_1 正则化后的线性回归目标函数。

$$J=\frac{1}{2m}\sum_{i=1}^{m}\left[y^{(i)}-\left(\sum_{j=1}^{n}w_j x_j^{(i)}+b\right)\right]^2+\frac{\lambda}{2n}\sum_{j=1}^{n}|w_j|,\quad \lambda>0 \tag{3-103}$$

在式(3-103)中第 2 项便是新加入的 ℓ_1 正则化项，可以看出它与 ℓ_2 正则化的差别在于前者是各个参数的绝对值之和，而后者是各个参数的平方之和。那么 ℓ_1 正则化又是如何解决模型过拟合现象的呢？

以单变量线性回归为例，并且假设此时只有一个样本，在对其施加 ℓ_1 正则化之前和之后的目标函数如式(3-104)所示。

$$\begin{aligned}J_1&=\frac{1}{2}(wx+b-y)^2\\ J_2&=J_1+\lambda|w|\end{aligned} \tag{3-104}$$

由式(3-104)可知，目标函数 J_1 和 J_2 关于权重 w 的梯度分别为

$$\begin{aligned}\frac{\partial J_1}{\partial w}&=x(wx+b-y)\\ \frac{\partial J_2}{\partial w}&=\begin{cases}x(wx+b-y)+\lambda, & w>0\\ x(wx+b-y)-\lambda, & w<0\end{cases}\end{aligned} \tag{3-105}$$

进一步由梯度下降算法可得两者的参数更新公式为

$$\begin{aligned}w&=w-\alpha\frac{\partial J_1}{\partial w}=w-\alpha\cdot x(wx+b-y)\\ w&=w-\alpha\frac{\partial J_2}{\partial w}=\begin{cases}w-\alpha\cdot[x(wx+b-y)+\lambda], & w>0\\ w-\alpha\cdot[x(wx+b-y)-\lambda], & w<0\end{cases}\end{aligned} \tag{3-106}$$

为了更好地观察式(3-106)中两者的差异，令 $\phi=x(wx+b-y)$，$\alpha=1$，此时有

$$w=w-\alpha\frac{\partial J_1}{\partial w}=w-\phi$$

$$w=w-\alpha\frac{\partial J_2}{\partial w}=\begin{cases}(w-\lambda)-\phi, & w>0\\(w+\lambda)-\phi, & w<0\end{cases} \tag{3-107}$$

此时根据式(3-107)中两者对比可知,对于施加 ℓ_1 正则化后的目标函数来讲,当 $w>0$ 时,w 会先减去 λ;当 $w<0$ 时,w 会先加上 λ,所以在这两种情况下更新后的 w 都会更加趋于 0,而这也就是 ℓ_1 正则化同样能缓解模型过拟合的原因。

3.10.11 ℓ_1 与 ℓ_2 正则化差异

根据前面几节内容的介绍可知,ℓ_1 和 ℓ_2 正则化均能够使得到的参数趋于 0(接近于 0),但是对于 ℓ_1 正则化来讲它却能够使模型参数更加稀疏,即直接使模型对应的参数变为 0(不仅是接近)。那么 ℓ_1 正则化是如何产生这一结果的呢?

以单变量线性回归为例,对其分别施加 ℓ_1 和 ℓ_2 正则化后,目标函数关于参数 w 的梯度分别为

$$\ell_1:\begin{cases}x(wx+b-y)+\lambda, & w>0\\x(wx+b-y)-\lambda, & w<0\end{cases}$$

$$\ell_2:x(wx+b-y)+\lambda w \tag{3-108}$$

根据式(3-108)可知,对于 ℓ_1 正则化来讲,只要满足条件 $|x(wx+b-y)|<\lambda$,那么带有 ℓ_1 正则化的目标函数总能保持,当 $w<0$ 时单调递减,当 $w>0$ 时单调递增,即此时一定能在 $w=0$ 产生最小值。对于 ℓ_2 正则化来讲,当 $w=0$ 时,只要 $x(wx+b-y)\neq 0$,那么带有 ℓ_2 正则化的目标函数便不可能在 $w=0$ 处产生最小值。也就是说,对于 ℓ_1 正则化来讲只需满足条件 $-\lambda<x(wx+b-y)<\lambda$,便可以在 $w=0$ 处取得最小值,而对于 ℓ_2 正则化来讲,只有满足条件 $x(wx+b-y)=0$ 时,才可能在 $w=0$ 处取得最小值,因此,相较于 ℓ_2 正则化,ℓ_1 正则化更能够使模型产生稀疏解。

当然,还可以从另外一个比较直观的角度来解释为什么 ℓ_1 正则化更容易产生稀疏解。从本质上看,带正则化的目标函数实际上等价于带约束条件的原始目标函数,即为了缓解模型的过拟合现象可以对原始目标函数的解空间施加一个约束条件,而这个约束条件便可以是 ℓ_1 或者 ℓ_2 正则化。例如对于带 ℓ_1 约束条件的目标函数有

$$\min\frac{1}{2m}\sum_{i=1}^{m}\left[y^{(i)}-\left(\sum_{j=1}^{n}w_jx_j^{(i)}+b\right)\right]^2$$

$$\text{s.t.}\sum_{j=1}^{n}|w_j|\leqslant t,\quad t>0 \tag{3-109}$$

从式(3-109)可以看出,这是一个典型的带有不等式约束条件的极值求解问题,并且可以得到对应的拉格朗日函数

$$\mathcal{L}(w,b)=\frac{1}{2m}\sum_{i=1}^{m}\left[y^{(i)}-\left(\sum_{j=1}^{n}w_j x_j^{(i)}+b\right)\right]^2+\mu\left(\sum_{j=1}^{n}\mid w_j\mid -t\right),\quad \mu\geqslant 0 \tag{3-110}$$

其中，μ 被称为拉格朗日乘子，可以看出本质上它等同于式(3-103)中的惩罚系数 λ，同时由于 t 为常数，所以式(3-110)与目标函数(3-103)等价。

根据式(3-110)可以画出在二维特征条件下原始目标函数与 ℓ_1 约束条件下解的分布情况(同理还可以画出 ℓ_2 约束条件下的解空间)，如图 3-44 所示。

在图 3-44 中，椭圆曲线表示目标函数对应的等高线，菱形和圆形分别表示 ℓ_1 和 ℓ_2 约束条件下对应的解空间。从图 3-44 中可以看出，目标函数在 ℓ_1 约束条件下于 p 处取得最小值，在 ℓ_2 约束条件下于 q 处取得最小值。此时可以发现，由于 ℓ_1 约束条件下的解空间为菱形，因此相较于 ℓ_2 约束，ℓ_1 更容易在顶点处产生极值，这就导致 ℓ_1 约束更能使模型产生稀疏解。同时，还可以通过一张更明显的图示来进行说明，如图 3-45 所示。

从图 3-45 可以看出，对于不同的约束条件来讲，相较于 ℓ_2 约束条件目标函数更容易在 ℓ_1 约束条件下产生稀疏解。

最后，在实际运用过程中，还可以将两种正则化方式结合到一起，即弹性网络(Elastic-Net)惩罚[5]，如式(3-111)所示。

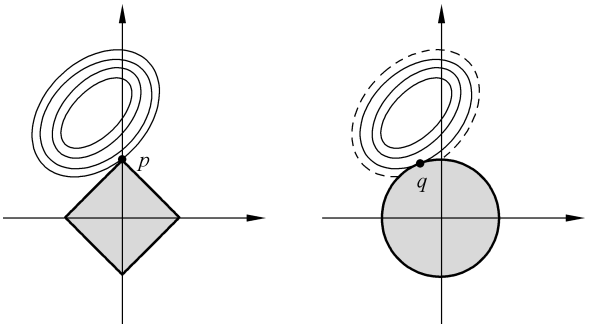


图 3-44　两种约束条件下目标函数对应的最优解

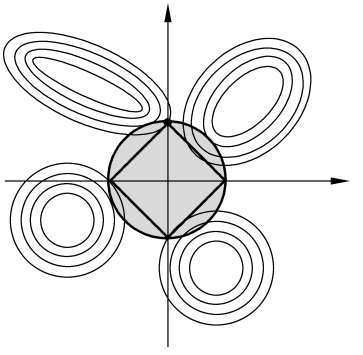


图 3-45　不同目标函数在两种约束条件下的最优解

$$\lambda\sum_{i=1}^{n}(\beta\mid w_i\mid+(1-\beta)w_i^2),0\leqslant\beta\leqslant 1 \tag{3-111}$$

其中，β 用来控制 ℓ_1 和 ℓ_2 惩罚项各自所占的比重，可以看出当 $\beta=0$ 时式(3-111)便等价于 ℓ_2 正则化，当 $\beta=1$ 时则等价于 ℓ_1 正则化。

最后，由于 PyTorch 中没有直接提供调用 ℓ_1 正则化的方法，因此需要通过以下方式来使用，示例代码如下：

```
def train(x_train, x_test, y_train, y_test, lambda_term = 0.):
    ...
    optimizer = torch.optim.SGD(net.parameters(), lr = lr) #定义优化器
    loss_train = []
    for epoch in range(epochs):
        logits = net(x_train)
```

```
7           l = loss(logits, y_train)
8           for p_name in net.state_dict():
9               if 'bias' not in p_name:
10                  p_value = net.state_dict()[p_name]
11                  l += lambda_term/len(x_train) * torch.norm(p_value, 1)
12      ...
```

在上述代码中，第 3 行用于指定优化器，并且不需要指定 weight_decay 参数。第 8～9 行用于遍历模型中的所有参数，并将偏置过滤掉。第 10～11 行用于分别对模型中的参数进行 ℓ_1 正则化处理。

3.10.12 丢弃法

在深度学习中，除了通过正则化方法来缓解模型的过拟合现象外，还有一种常用的处理方式，即丢弃法(DropOut)[6]。丢弃法的思想是在模型的训练过程中，根据某一概率分布随机将其中一部分神经元忽略(乘以一个只含 0 和 1 的掩码矩阵)的做法，并且对于每次前向传播来讲忽略部分神经元的位置都是不尽相同的，因此从另一个角度来看每次执行梯度下降时优化的都是不同模型对应的参数。

如图 3-46 所示，这便是在原始网络结构的基础上对输出层进行 DropOut 后可能的两种结果，其中虚线表示被丢弃的神经元，其作为输入在下一层线性组合时对应位置的值便为 0。可以看出，丢弃法这一思想相当于引入了类似自举聚合(Bootstrap Aggregation，Bagging)的集成学习思想，可以被认为集成了大量深层神经网络的 Bagging 方法[7]。

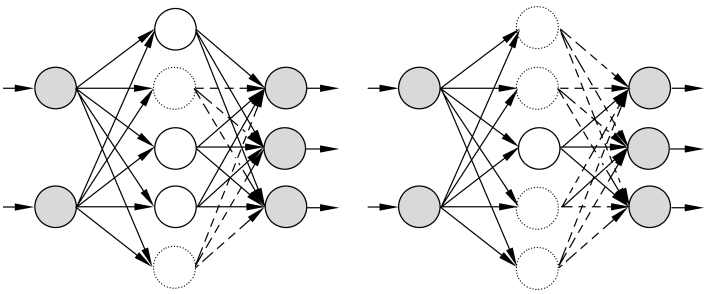


图 3-46 DropOut 示例图

同时，需要注意的是模型在测试或者称为推理(Inference)过程中，并不需要进行随机丢弃操作，一方面是为了保证模型每次输出结果相同；另一方面是因为如果在测试过程进行了随机丢弃，则此时相当于仅使用了整个集成模型中的一个模型。但此时又引入了一个新的问题，就是训练阶段和测试阶段模型输出的不一致性，而这种行为不一致性会导致测试阶段网络的输出尺度与训练阶段不同，从而影响网络的预测性能。

例如有一个网络层，丢弃率设置为 $p=0.2$。如果在训练阶段神经元的原始输出为 $x=2$，而该神经元被保留的概率为 $1-p=0.8$，则在训练阶段的输出应该为 $1-p=2.5$ 或 0。也就是说，在训练阶段每个神经元的输出会被放大 2.5 倍，这样做是为了补偿因丢弃神经元造成的输出期望的降低。在测试阶段所有神经元都被保留，因此不需要放大，所以输出值就是 $x=2$ 不变。

具体地，设某一层中神经元 o_i 被丢弃的概率为 p，即随机变量 η_i 为 0 和 1 的概率分别为 p 和 $1-p$，则有

$$o_i'=\frac{\eta_i}{1-p}o_i \tag{3-112}$$

其中，o_i'为使用丢弃法后的结果。

在式(3-112)中之所以还要除以$(1-p)$，是为了使施加丢弃法后的结果的期望等于作用丢弃法之前的结果，即保持训练阶段和测试阶段输出结果期望的一致性，所以有

$$E(o_i')=E(o_i')\frac{1}{1-p}o_i=\frac{1-p}{1-p}o_i=o_i \tag{3-113}$$

在介绍完丢弃法的基本原理后，下面开始介绍其具体实现过程。首先需要实现函数来完成整个 DropOut 操作，实现代码如下：

```
1  def DropOut(a, drop_pro = 0.5, training = True):
2      if not training:
3          return a
4      assert 0 <= drop_pro <= 1
5      if drop_pro == 1:
6          return refs.zeros_like(a)
7      if drop_pro == 0:
8          return a
9      keep_pro = 1 - drop_pro
10     scale = 1 / keep_pro
11     mask = refs.uniform(a.shape, low = 0.0, high = 1.0,
12                         dtype = torch.float32, device = a.device) < keep_pro
13     return refs.mul(refs.mul(a, mask), scale)
```

在上述代码中，第 1 行中 a 表示输入的网络层，drop_pro 表示神经元被丢弃的概率，training 表示当前是否处于训练状态。第 2～3 行表示如果是推理阶段，则直接返回原始值。第 4 行用于判断丢弃比例的合法取值。第 5～8 行表示返回特殊情况下对应的结果。第 9～13 行则是式(3-112)计算过程的体现，其中第 11～12 行会根据均匀分布返回一个只包含 0 和 1 的掩码矩阵，第 13 行表示原始输入先乘以掩码矩阵再进行缩放。

到此对于 DropOut 的计算过程就实现完了，可以直接把它当作一个函数进行调用。不过为了能将其作为 PyTorch 中的网络层添加到 nn.Sequential()中进行使用，还需要将其封装成一个 nn.Module 类对象，实现代码如下：

```
1  class MyDropOut(nn.Module):
2      def __init__(self, p = 0.5):
3          super(MyDropOut, self).__init__()
4          self.p = p
5
6      def forward(self, x):
7          return DropOut(x, drop_pro = self.p, training = self.training)
```

在上述代码中，第 1 行表示继承 PyTorch 中的 nn.Module 类，所有想要作为一个网络层来使用的类都需要继承该类，在后续内容中也会对此进行介绍。第 2～4 行用于初始化相应的参数，其中 p 表示丢弃率。第 6～7 行用于定义前向传播过程，其中 self.training 用于

获取模型当前的状态(训练或推理)。

最后,可以通过以下方式来使用 MyDropOut(),代码如下:

```
if __name__ == '__main__':
    a = torch.randn([2, 10])
    op_DropOut = MyDropOut(p = 0.2)
    print(op_DropOut(a))
```

输入结果如下:

```
tensor([[ - 1.4843, - 1.0103, - 0.0000, 0.7997, 0.9650,
          0.4117, - 0.6568, - 0.4334, 1.5951, - 0.9222],
          [ - 1.2151, 2.4469, - 1.9339, - 0.6010, - 0.0000,
          0.0342, - 1.1552, 0.0000,0.1395, 1.7941]])
```

在上述结果中,取值为 0 的位置便是被丢弃的位置。

上述完整代码见 Code/Chapter03/Code/Chapter03/C17_DropOut/main.py 文件。

3.10.13 小结

本节首先通过示例详细介绍了如何通过 ℓ_2 正则化方法来缓解模型的过拟合现象,以及介绍了为什么 ℓ_2 正则化能够使模型变得更简单,其次介绍了加入正则化后原有梯度更新公式的变化之处,其仅仅加上了正则化项对应的梯度,然后通过一个示例来展示了 ℓ_2 正则化的效果,与此同时还介绍了另外一种常见的 ℓ_1 正则化方法并详细对比了 ℓ_1 正则化和 ℓ_2 正则化的差异之处;最后介绍了深度学习中另外一种常见的缓解模型过拟合的丢弃法及其实现方式。

3.11 超参数与交叉验证

在深度学习中,除了通过训练集根据梯度下降算法训练得到的权重参数之外,还有另外一类,即通过手动设置的超参数(Hyper Parameter),而超参数的选择对于模型最终的表现也至关重要。在接下来的内容中,将介绍到目前为止已经接触过的几个超参数及其选择方式。

3.11.1 超参数介绍

在之前的介绍中,我们知道了模型中的权重参数可以通过训练集利用梯度下降算法得到,但超参数又是什么呢? 所谓超参数是指那些不能通过数据集训练得到的参数,但它的取值同样会影响最终模型的效果,因此同样重要。到目前为止,一共接触过 4 个超参数,只是第 1 次出现时并没有提起其名字,在这里再做一个细致的总结。这 4 个超参数分别是:学习率 α、惩罚系数 λ、网络层数、丢弃率。

1. 学习率 α

在 3.3.3 节中介绍梯度下降算法原理时,首次介绍了梯度下降算法的迭代更新公式,见式(3-12),并且讲过 α 用来控制每次向前跳跃的距离,较大的 α 可以更快地跳到谷底并找到最优解,但是过大的 α 同样能使目标函数在峡谷的两边来回振荡,以至于需要多次迭代才可以得到最优解,甚至可能因为出现梯度爆炸现象而使目标函数发散。

相同模型采用不同的学习率后，经梯度下降算法在同一初始位置优化后的结果如图 3-47 所示，其中黑色五角星表示全局最优解(Global Optimum)，ite 表示迭代次数。

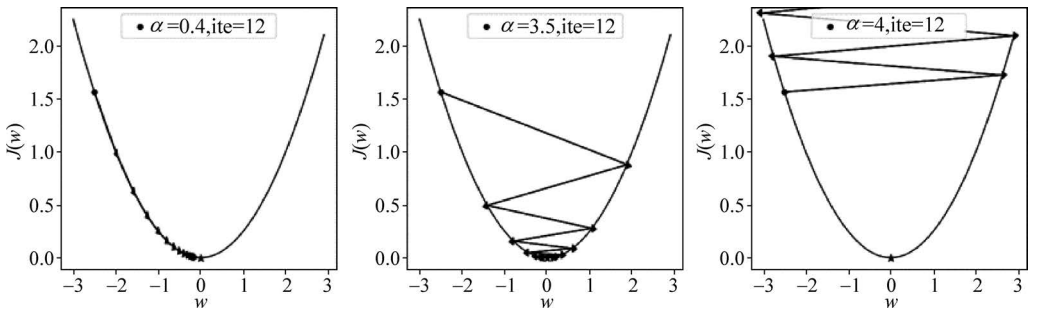


图 3-47　凸函数优化过程

当学习率为 0.4 时，模型大概在迭代 12 次后就基本达到了全局最优解。当学习率为 3.5 时，模型在大约迭代 12 次后同样能够收敛于全局最优解附近，但是，当学习率为 4.1 时，此时的模型已经处于发散状态。可以发现，由于模型的目标函数为凸形函数(例如线性回归)，所以尽管使用了较大的学习率 3.5，目标函数依旧能够收敛，但在后面的学习过程中，遇到更多的情况便是非凸型的目标函数，此时的模型对于学习率的大小将会更加敏感。

一个非凸形的目标函数如图 3-48 所示，三者均从同一初始点开始进行迭代优化，只是各自采用了不同的学习率，其中黑色五角星表示全局最优解，ite 表示迭代次数。

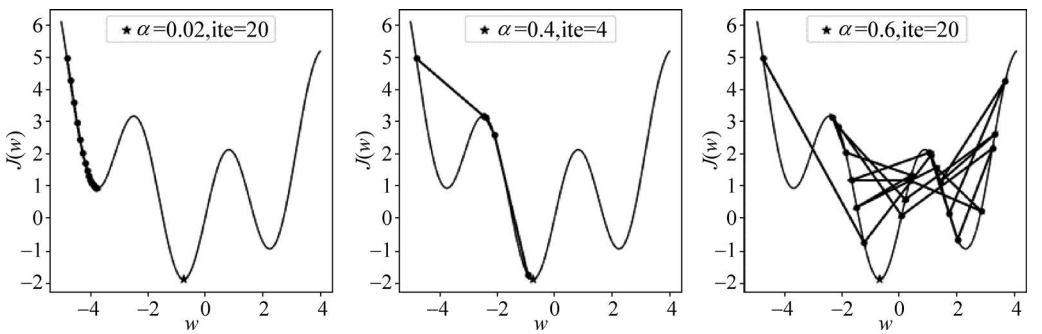


图 3-48　非凸形函数优化过程

从图 3-48 可以看出，当采用较小的学习率 0.02 时，模型在迭代 20 次后陷入了局部最优解(Local Optimum)，并且可以知道此时无论再继续迭代多少次，其依旧会收敛于此处，因为它的梯度已经开始接近于 0，而使参数无法得到更新。当采用较大一点的学习率 0.4 时，模型在迭代 4 次后便能收敛于全局最优解附近。当采用的学习率为 0.6 时，模型在这 20 次的迭代过程中总是来回振荡的，并且没有一次接近于全局最优解。

从上面两个示例的分析可以得出，学习率的大小对于模型的收敛性及收敛速度有着严重的影响，并且非凸函数在优化过程中对于学习率的敏感性更大。同时值得注意的是，所谓学习率过大或者过小，在不同模型间没有可比性。例如在上面凸函数的图示中当学习率为 0.4 时可能还算小，但是在非凸函数的这个例子中 0.4 已经算是相对较大的了。图示代码参见 Code/Chapter03/C18_HyperParams/visual.py 文件。

2. 惩罚系数 λ

从 3.10.9 节内容中正则化的实验结果可知，超参数 λ 表示对模型的惩罚力度。λ 越大也就意味着对模型的惩罚力度越大，最终训练得到的模型也就相对越简单，在一定程度上可以看作环境模型的过拟合现象，但是这并不代表 λ 越大越好，过大的 λ 将会降低模型的拟合能力，使最终得到的结果呈现出欠拟合的状态，因此，在模型的训练过程中，也需要选择一个合适的 λ 来使模型的泛化能力尽可能更好。

3. 网络层数

在 3.5.8 节内容中，我们介绍过可以通过增加网络模型的深度来提高模型的特征表达能力，以此来提高后续任务的精度，但是具体的层数需要人为地进行设定，因此网络层数也是深度学习中的一个重要超参数。

4. 丢弃率

在 3.10.12 节内容中，我们介绍过可以通过在训练网络时随机丢弃一部分神经元来近似达到集成模型的效果，以此来缓解模型的过拟合现象，但是从丢弃法的原理可知，对于参数神经元的丢弃率来讲它同样是一个需要手动设定的超参数，不同的取值对模型有不同的影响，因此需要根据经验或者交叉验证进行选择，不过通常情况下会将 0.5 作为默认值。

经过上面的介绍，我们明白了超参数对于模型最终的性能有着重要的影响。那么到底应该如何选择这些超参数呢？对于超参数的选择，首先可以列出各个参数的备选取值，例如 $\alpha=[0.001,0.03,0.1,0.3,1]$，$\lambda=[0.1,0.3,1,3,10]$（通常可以以 3 的倍数进行扩大），然后根据不同的超参数进行组合训练，从而得到不同的模型（例如这里就有 25 个备选模型），然后通过 3.11.2 节所要介绍的交叉验证来确立模型。

不过随着介绍的模型越来越复杂，就会出现更多的超参数组合，训练一个模型会花费一定的时间，因此，对于模型调参的一个基本要求就是要理解各个参数的含义，这样才可能更快地排除不可能的参数取值组合，以便于更快地训练出可用的模型。

3.11.2 模型选择

当在对模型进行改善时，自然而然地就会出现很多备选模型，而目的便是尽可能地选择一个较好的模型，但如何选择一个好的模型呢？通常来讲有两种方式：第 1 种便是 3.10.3 节中介绍过的将整个数据集划分成 3 部分的方式；第 2 种则是使用 K 折交叉验证（K Fold Cross Validation）[4] 的方式。对于第 1 种方法，其步骤为先在训练集上训练不同的模型，然后在验证集上选择其中表现最好的模型，最后在测试集上测试模型的泛化能力，但是这种做法的缺点在于，对于数据集的划分可能恰好某一次划分出来的测试集含有比较怪异的数据，导致模型表现出来的泛化误差也很糟糕，此时就可以通过 K 折交叉验证来解决此类问题。

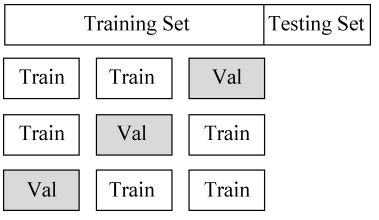


图 3-49 交叉验证划分图

以 3 折交叉验证为例，首先需要将整个完整的数据集分为训练集与测试集两部分，并且同时再将训练集划分成 3 份，每次选择其中的两份作为训练数据，另外一份作为验证数据对模型进行训练与验证，最后选择平均误差最小的模型，如图 3-49 所示。

假设现在有 4 个不同的备选模型，其各自在不同验证集上的误差如表 3-1 所示。根据得到的结果，可以选择平均误差最小的模型 2 作为最终选择的模型，然后将其用于整个大的训练集训练一次，最后用测试集测试其泛化误差。当然，还有一种简单的交叉验证方式，即一开始并不划分出测试集，而是直接将整个数据划分成为 K 份进行交叉验证，然后选择平均误差最小的模型即可。

表 3-1　3 折交叉验证划分结果

划分方式			模型 1	模型 2	模型 3	模型 4
Train	Train	Val	0.4	0.3	0.55	0.5
Train	Val	Train	0.3	0.45	0.35	0.35
Val	Train	Train	0.5	0.35	0.3	0.3
平均误差			0.4	0.37	0.4	0.38

3.11.3　基于交叉验证的手写体分类

在详细介绍完模型超参数及交叉验证的相关原理后，下面将通过一个实际的示例来介绍如何运用交叉验证去选择一个合适的深度学习模型。这里依旧以 MNIST 数据集为例进行介绍，完整示例代码可以参见 Code/Chapter03/C18_HyperParams/main.py 文件。

1. 载入数据

首先，需要载入原始数据，实现代码如下：

```
1  def load_dataset():
2      data_train = MNIST(root = '~/Datasets/MNIST', train = True, download = True,
3                         transform = transforms.ToTensor())
4      data_test = MNIST(root = '~/Datasets/MNIST', train = False, download = True,
5                         transform = transforms.ToTensor())
6      return data_train, data_test
```

在上述代码中，第 2～5 行用于分别载入训练数据和测试数据，同时这里需要注意的是由于需要通过交叉验证来选择模型，因此这里暂时没有直接返回训练集和测试集对应的 DataLoader 迭代器。

2. 定义网络结果

接下来，需要定义网络模型的整体框架，实现代码如下：

```
1  def get_model(input_node = 28 * 28,
2                hidden_nodes = 1024,
3                hidden_layers = 0,
4                output_node = 10,
5                p = 0.5):
6      net = nn.Sequential(nn.Flatten())
7      for i in range(hidden_layers):
8          net.append(nn.Linear(input_node, hidden_nodes))
9          net.append(nn.DropOut(p = p))
10         input_node = hidden_nodes
11     net.append(nn.Linear(input_node, output_node))
12     return net
```

在上述代码中，第 1～5 行是相关超参数的默认值。第 6 行只定义了一个 Flatten()层，

因为不知道需要定义多少隐藏层。第 7～10 行根据输入的超参数来确定隐藏层的个数。第 11～12 行则是加入最后的输出层，并返回最后的模型。

3．模型训练

进一步地，需要定义一个函数来完成单个模型的训练过程，实现代码如下：

```
1  def train(train_iter, val_iter, net, lr = 0.03, weight_decay = 0., epochs = 1):
2      loss = nn.CrossEntropyLoss() #定义损失函数
3      optimizer = torch.optim.SGD(net.parameters(), lr = lr,
                          weight_decay = weight_decay) #定义优化器
4      for epoch in range(epochs):
5          for i, (x, y) in enumerate(train_iter):
6              logits = net(x)
7              l = loss(logits, y)
8              optimizer.zero_grad()
9              l.backward()
10             optimizer.step() #执行梯度下降
11     return evaluate(train_iter, net), evaluate(val_iter, net)
```

在上述代码中，第 1 行 train_iter 表示训练集样本迭代器，val_iter 表示验证集样本迭代器，net 表示网络模型，lr 表示学习率，weight_decay 表示 l_2 惩罚项系数，epochs 表示迭代轮数。第 2～11 行为标准的模型训练过程，不再赘述。第 12 行用于分别返回模型在训练集和验证集上的准确率，其中 evaluate()函数的实现如下：

```
1  def evaluate(data_iter, net):
2      net.eval()
3      with torch.no_grad():
4          acc_sum, n = 0.0, 0
5          for x, y in data_iter:
6              logits = net(x)
7              acc_sum += (logits.argmax(1) == y).float().sum().item()
8              n += len(y)
9          net.train()
10         return round(acc_sum / n, 4)
```

在上述代码中，第 2 行用于将模型的状态设定为推理状态，因为在推理阶段不需要进行 DropOut 操作，同时这一步 PyTorch 内部已经帮我们实现，所以不需要自己写逻辑判断是否要进行 DropOut 操作。第 3 行表示模型在推理阶段时不计算梯度信息，有利于减少内存消耗和提高计算速度。第 4～8 行用于对迭代器中的所有样本进行预测，然后记录预测正确的数量和总数量。第 9 行表示推理阶段完毕后需要将模型的当前状态设定为训练状态。第 10 行用于返回最后的准确率。

4．交叉验证

在实现完单个模型的训练过程后，下面需要定义一个函数来实现整个交叉验证的执行逻辑，实现代码如下：

```
1  def cross_validation(data_train,k = 2,
2                       batch_size = 128,input_node = 28 * 28,
3                       hidden_nodes = 1024,hidden_layers = 0,
4                       output_node = 10,p = 0.5,
5                       weight_decay = 0.,lr = 0.03):
6      model = get_model(input_node,hidden_nodes,hidden_layers,output_node,p)
```

```
    kf = KFold(n_splits=k)
    val_acc_his = []
    for i, (train_idx, val_idx) in
                    enumerate(kf.split(np.arange(len(data_train)))):
        train_sampler = SubsetRandomSampler(train_idx)
        val_sampler = SubsetRandomSampler(val_idx)
        train_iter = DataLoader(data_train, batch_size=batch_size,
                                sampler=train_sampler)
        val_iter = DataLoader(data_train, batch_size=batch_size,
                              sampler=val_sampler)
        train_acc, val_acc = train(train_iter, val_iter, model, lr,
                                   weight_decay)
        val_acc_his.append(val_acc)
        print(f"#Fold {i} train acc:{train_acc},val acc:{val_acc} OK.")
    return np.mean(val_acc_his), model
```

在上述代码中，第 2～5 行用于传入模型对应的参数或者超参数。第 6 行用于根据当前这一组参数得到对应的网络模型。第 7 行用于实例化一个用于产生 K 折交叉验证样本索引的类。第 9 行用于生成每一折验证时所取训练样本和验证样本的索引。第 10～13 行用于根据索引在原始样本中取到对应训练或验证部分的样本，并构造相应的迭代器。第 14 行则通过当前这一组参数和样本得到模型在训练集和验证集上的准确率。第 17 行用于返回模型在交叉验证上的平均准确率和对应的模型。

5. 模型选择

在完成上述所有辅助函数的实现之后，接下来便可以列出所有的备选超参数组合，然后通过交叉验证逐一训练和验证每个模型，并输出最优的模型，实现代码如下：

```
if __name__ == '__main__':
    data_train, data_test = load_dataset()
    k, batch_size = 3, 128
    input_node, hidden_nodes = 28 * 28, 1024
    output_node = 10
    hyp_hidden_layers, hyp_p = [0, 2], [0.5, 0.7]
    hyp_weight_decay, hyp_lr = [0., 0.01], [0.01, 0.03]
    best_val_acc,no_model = 0, 1
    best_model, best_params = None, None
    total_models = len(hyp_hidden_layers) * len(hyp_p) *
                   len(hyp_weight_decay) * len(hyp_lr)
    print(f"#Total model{total_models},fitting times{k*total_models}")
    for hidden_layer in hyp_hidden_layers:
        for p in hyp_p:
            for weight_decay in hyp_weight_decay:
                for lr in hyp_lr:
                    print(f"#Fitting model [{no_model}/{total_models}]")
                    no_model += 1
                    mean_val_acc, model =
                    cross_validation(data_train=data_train,
                        k=k, batch_size=batch_size,input_node=input_node,
                        hidden_nodes=hidden_nodes,
                        hidden_layers=hidden_layer,
                        output_node=output_node,
                        p=p,weight_decay=weight_decay,lr=lr)
```

```
22                  params = {"hidden_layer": hidden_layer, "p": p,
23                            "weight_decay": weight_decay, "lr": lr,
24                            "mean_val_acc": mean_val_acc}
25                  if mean_val_acc > best_val_acc:
26                      best_val_acc = mean_val_acc
27                      best_model = model
28                      best_params = params
29                  print(f"{params}\n")
30      test_iter = DataLoader(data_test, batch_size = 128)
31      print(f"The best model params: {best_params},"
32            f"acc on test: {evaluate(test_iter, best_model)}")
```

在上述代码中，第 2 行用来载入原始数据样本。第 3～7 行分别用来定义模型的参数及备选的超参数。第 10～11 行用来计算模型的个数及需要拟合的次数并打印输出。第 12～15 行用于构造各个超参数组合。第 18～21 行用于根据当前的超参数组合进行模型的交叉验证。第 25～28 行用来保存在交叉验证中表现最好的模型。第 30 行则通过测试集来测试交叉验证中最优模型的泛化能力。

最后，在上述代码运行结束后，便可以得到类似如下所示的输出结果：

```
1   #Total model 16, fitting times 48
2   #Fitting model [1/16].....
3   #Fold 0 train acc: 0.824, val acc: 0.8194 finished.
4   #Fold 1 train acc: 0.8464, val acc: 0.8415 finished.
5   #Fold 2 train acc: 0.8565, val acc: 0.8618 finished.
6   {'hidden_layer':0,'p':0.5,'weight_decay':0,'lr':0.01,'mean_val_acc':0.84}
7       ...
8   #Fitting model [2/16]...
9   {'hidden_layer':0,'p':0.5,'weight_decay':0,'lr':0.03,'mean_val_acc':0.87}
10    #Fitting model [3/16]...
11      ...
12    The best model params: {'hidden_layer': 2, 'p': 0.5, 'weight_decay': 0.0, 'lr': 0.03, 'mean_
      val_acc': 0.8874}, acc on test: 0.9072
```

根据上述输出结果可知，在第 12 行中列出了最优模型的超参数组合，并且其在最终测试集上的准确率为 0.9072。

3.11.4 小结

本节首先介绍了什么是超参数，以及几个常见超参数能够给模型带来什么样的影响，然后详细介绍了什么是交叉验证及如何通过交叉验证来选择模型；最后，一步一步地从零介绍了基于手写体分类任务的模型筛选过程。

3.12 激活函数

在 3.1.6 节内容中介绍过，神经网络中多次线性组合后的输出结果仍旧只是原始特征的线性组合，它并没有增加模型的复杂度。为了增加模型的表达能力需要对每层的输出结果通过一个激活函数（Activation Function）来进行一次非线性变换，然后将其作为下一层网络的输入。在深度学习中，常见的激活函数包括 Sigmoid、Tanh、ReLU 和 LeakyReLU

等，下面对这些激活函数分别进行介绍。以下所有图示及实现代码可以参见 Code/Chapter03/C19_Activation/main.py 文件。

3.12.1　Sigmoid 激活函数

1. 原理

Sigmoid 激活函数的作用是将神经元的输出值映射到区域(0,1)中，同时它是最常用的具有指数形式的激活函数，其计算过程如下所示

$$g(x)=\frac{1}{1+\mathrm{e}^{-x}} \tag{3-114}$$

其导数为

$$g'(x)=\frac{\mathrm{e}^{-x}}{(1+\mathrm{e}^{-x})^2}=g(x)(1-g(x)) \tag{3-115}$$

根据式(3-114)和式(3-115)可以分别画出两者的函数图像，如图 3-50 所示。

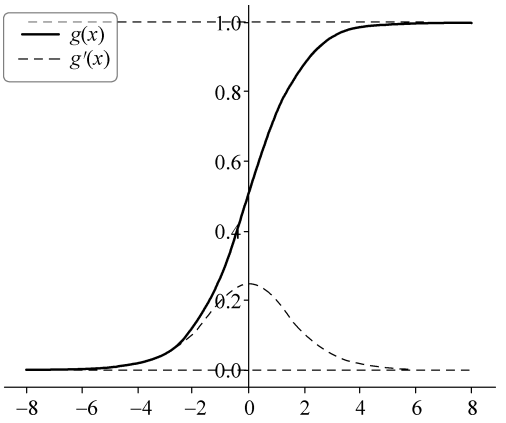


图 3-50　Sigmoid 激活函数图像

在图 3-50 中，由于经过 Sigmoid 函数非线性变换后，其值域将被限定在 0 到 1 的开区间内，所以可以发现当输入值在 0 附近时，Sigmoid 激活函数就类似于一个线性函数；当输入值在函数两端时，将对输入形成抑制作用，即梯度趋于 0 附近。根据图 3-50 中 Sigmoid 激活函数的导数 $g'(x)$的图像可知，$g'(x)\in\left(0,\frac{1}{4}\right)$，即当使用 Sigmoid 作为激活函数时会减缓参数的更新速度，因此，Sigmoid 激活函数的最大缺点在于当神经元的输入值过大或者过小时都容易引起神经元的饱和现象，即梯度消失问题，使网络模型的训练过程变慢，甚至停止。通常，此时会对输入进行标准化操作，例如对每个特征维度进行归一化操作。

2. 实现

在理解了 Sigmoid 激活函数的基本原理之后，再来介绍如何实现与使用。根据式(3-114)可知，Sigmoid 的实现代码如下：

```
def sigmoid(x):
    return 1 / (1 + torch.exp(-x))
```

需要将 Sigmoid 操作当作一个网络层来使用，因此需要构造一个继承自 nn.Module 的类，实现代码如下：

```
class MySigmoid(nn.Module):
    def forward(self, x):
        return sigmoid(x)
```

最后，可以通过以下方式来使用：

```
def test_Sigmoid():
    x = torch.randn([2, 5], dtype=torch.float32)
    net = nn.Sequential(MySigmoid())
    y = net(x)
    print(f"Sigmoid 前：{x}")
    print(f"Sigmoid 后：{y}")
```

上述代码运行结束后，便可以得到类似如下的结果：

```
Sigmoid 前:tensor([[-9.01e-02, 1.11e+00, -6.33e-01, 3.26e-01, -7.47e-01],
                   [-8.38e-01, 1.89e-01, -4.15e-01, -5.49e-01, 1.11e-04]])
Sigmoid 后:tensor([[0.4775, 0.7525, 0.3467, 0.5810, 0.3214],
                   [0.3034, 0.5473, 0.3977, 0.3661, 0.5000]])
```

当然，在 PyTorch 中还可以直接通过 nn.Sigmoid()来使用 Sigmoid 激活函数。

3.12.2 Tanh 激活函数

1. 原理

Tanh 激活函数也叫作双曲正切激活函数，其作用效果与 Sigmoid 激活函数类似且本质上仍旧属于类 Sigmoid 激活函数，它们都使用指数来进行非线性变换。Tanh 激活函数会将神经元的输入值压缩到 (−1,1)中，与 Sigmoid 相比 Tanh 的激活值具有更大的激活范围。其数学定义如下

$$g(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{3-116}$$

其导数为

$$g'(x)=1-(g(x))^{2} \tag{3-117}$$

根据式(3-116)和式(3-117)可以分别画出两者的函数图像，如图 3-51 所示。

Tanh 激活函数可以看作放大并且平移后的 Sigmoid 函数。与 Sigmoid 函数类似，当输入值在 0 附近时，Tanh 类似于一个线性函数；当输入值在两端时，将对输入形成抑制作用，因此，当神经元的输入值过大或者过小时其依旧存在梯度消失的问题，同时其指数计算也会加大网络的计算开销。从 Tanh 的导数图像来看，其具有更大的梯度范围(0,1]，能够在网络训练时加快训练速度。

2. 实现

在有了 Sigmoid 激活函数的实现示例后，Tanh 就相对容易了。根据式(3-116)可知，

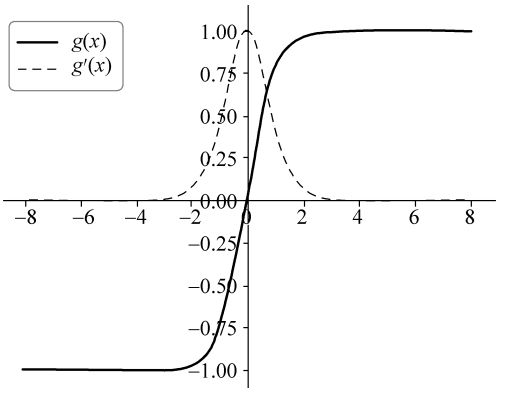


图 3-51　Tanh 函数图像

Tanh 的实现代码如下：

```
1  def tanh(x):
2      p = torch.exp(x) - torch.exp(-x)
3      q = torch.exp(x) + torch.exp(-x)
4      return p / q
5
6  class MyTanh(nn.Module):
7      def forward(self, x):
8          return tanh(x)
9
10 def test_Tanh():
11     x = torch.randn([2, 5], dtype=torch.float32)
12     net = nn.Sequential(MyTanh())
13     y = net(x)
14     print(f"Tanh 前: {x}")
15     print(f"Tanh 后: {y}")
```

在上述代码中，第 1～4 行用于实现基础的 Tanh 计算过程。第 6～8 行用于将其定义为一个网络层。第 10～15 行为使用示例，最后将会输入类似如下的结果：

```
1  Tanh 前: tensor([[-0.2107, 0.3643, 0.3670, 0.3385, 0.7338],
2                   [ 1.0832, -0.3375, 2.1993, -1.1353, 0.9691]])
3  Tanh 后: tensor([[-0.2076, 0.3490, 0.3514, 0.3261, 0.6254],
4                   [ 0.7944, -0.3253, 0.9757, -0.8128, 0.7483]])
```

在 PyTorch 中可以直接通过 nn.Tanh()来使用 Tanh 激活函数。

3.12.3　ReLU 激活函数

1. 原理

ReLU 激活函数的全称为线性修正单元(Rectified Linear Unit，ReLU)，是目前深度学习中使用最为广泛的非线性激活函数，它能够将神经元的输入值映射到[0，+∞)范围，其数学定义如下

$$g(x)=\begin{cases}x, & x\geqslant 0\\ 0, & x<0\end{cases}=\max(0,x) \tag{3-118}$$

其导数为

$$g'(x)=\begin{cases}1, & x>0\\ 0, & x\leqslant 0\end{cases} \tag{3-119}$$

值得注意的是，尽管 $g(x)$ 在 $x=0$ 处不可导，但是在实际处理时可以取其导数为 0。

进一步，根据式(3-118)和式(3-119)可以分别画出两者的函数图像，如图 3-52 所示。

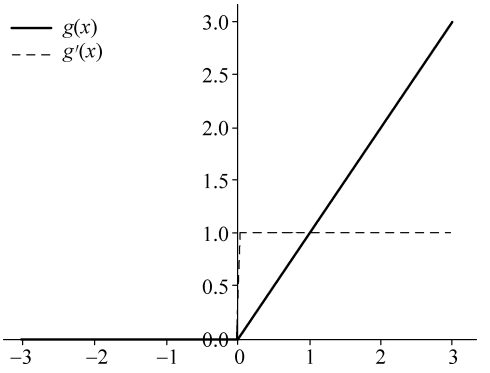


图 3-52 ReLU 函数图像

虽然 ReLU 激活函数整体上是一个非线性函数，但是其在原点两边均为线性函数，因此，采用 ReLU 激活函数的神经元只需进行加、乘和比较操作，使网络在训练过程中能够很大程度上降低运算复杂度，从而提高计算效率。同时从优化的角度来看，相比于 Sigmoid 和 Tanh 激活函数的两端饱和性，ReLU 激活函数为左端饱和函数，因此当 $x>0$ 时其梯度始终为 1，这在很大程度上缓解了网络梯度消失的问题，加速了网络的收敛速度，但同时，由于 ReLU 激活函数在 $x<0$ 时，其激活值始终保持为 0，因此在网络的训练过程中容易造成神经元“死亡”的现象。

2. 实现

根据式(3-118)可知，ReLU 激活函数的实现代码如下：

```
1  def relu(x):
2      mask = x >= 0.
3      return x * mask
4
5  class MyReLU(nn.Module):
6      def forward(self, x):
7          return relu(x)
8
9  def test_ReLU():
10     x = torch.randn([2, 5], dtype = torch.float32)
11     net = nn.Sequential(MyReLU())
12     y = net(x)
13     print(f"ReLU 前: {x}")
14     print(f"ReLU 后: {y}")
```

在上述代码中，第 2 行用于判断哪些位置上的元素大于 0，将会返回一个只含 True 和

False 的向量。第 3 行则用于计算最后的输出值，True 和 False 将分别被视为 1 和 0 参与计算。

最终将会输出类似如下的结果：

```
ReLU 前: tensor([[ 0.4586, -2.1994, 0.6357, -1.7937, 0.1907],
                 [1.1383, 0.9027, 1.8619, -0.9388, -0.1586]])
ReLU 后: tensor([[0.4586, -0.0000, 0.6357, -0.0000, 0.1907],
                 [1.1383, 0.9027, 1.8619, -0.0000, -0.0000]])
```

在 PyTorch 中也可以直接通过 nn.ReLU()来使用 ReLU 激活函数。

3.12.4　LeakyReLU 激活函数

1. 原理

LeakyReLU 激活函数即带泄露的修正线性单元，其总体上与 ReLU 激活函数一样，只是在 $x<0$ 的部分保持了一个很小的梯度。这样使神经元在非激活状态时也能有一个非零的梯度，以此来更新参数，避免了 ReLU 激活函数中神经元永远不能被激活的问题，其数学定义如下：

$$g(x)=\begin{cases}x, & x\geqslant 0\\ \gamma x, & x<0\end{cases}=\max(0,x)+\gamma\min(0,x) \tag{3-120}$$

其导数为

$$g'(x)=\begin{cases}1, & x>0\\ -\gamma, & x\leqslant 0\end{cases} \tag{3-121}$$

其中，$\gamma\geqslant 0$，并且尽管 $g(x)$在 $x=0$ 处不可导，但是在实际处理时可以取其导数为 $-\gamma$。

根据式(3-120)和式(3-121)可以分别画出两者的函数图像，如图 3-53 所示。

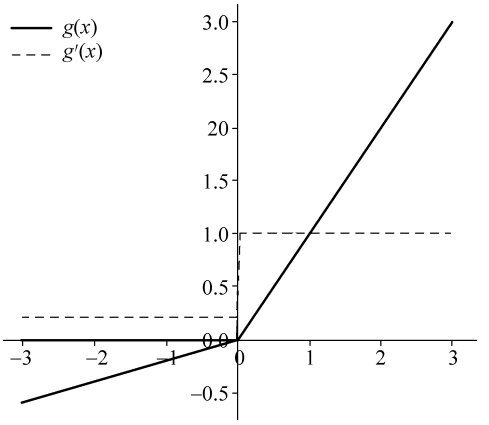


图 3-53　LeakeyReLU 激活函数图像

从图 3-53 可知，与 ReLU 激活函数的主要区别在于当 $x\leqslant 0$ 时，LeakyReLU 仍旧存在一个较小激活值 $-\gamma$，从而不会造成神经元的“死亡”现象。

2. 实现

根据式(3-120)可知，LeakyReLU 激活函数的实现代码如下：

```
1  def leakyrelu(x, gamma = 0.2):
2      y = (x >= 0) * x + gamma * (x < 0) * x
3      return y
4
5  class MyLeakyReLU(nn.Module):
6      def __init__(self, gamma = 0.2):
7          super(MyLeakyReLU, self).__init__()
8          self.gamma = gamma
9      def forward(self, x):
10         return leakyrelu(x, self.gamma)
11
12 def test_LeakyReLU():
13     x = torch.randn([2, 5], dtype = torch.float32)
14     net = nn.Sequential(MyLeakyReLU(0.2))
15     y = net(x)
16     print(f"LeakyReLU 前: {x}")
17     print(f"LeakyReLU 后: {y}")
```

在上述代码中,第 2 行便是式(3-120)的实现过程。第 5～10 行则用于将其封装为一个网络层。

最终将会输出类似如下的结果:

```
1  LeakyReLU 前: tensor([[ - 0.0888, - 0.5845, - 0.8447, - 0.9255, 1.1864],
2                        [ 0.7030, - 0.2215, - 0.7323, 1.4960, 0.7068]])
3  LeakyReLU 后: tensor([[ - 0.0178, - 0.1169, - 0.1689, - 0.1851, 1.1864],
4                        [ 0.7030, - 0.0443, - 0.1465, 1.4960, 0.7068]])
```

在 PyTorch 中也可以直接通过 nn.LeakyReLU()来使用 LeakyReLU 激活函数。

3.12.5 小结

本节首先回顾了在深度学习中为什么需要进行非线性变换,然后分别介绍了 4 种常见激活函数 Sigmoid、Tanh、ReLU 和 LeakyReLU 的原理和计算过程。最后详细介绍了各个激活函数的实现过程和使用示例。

3.13 多标签分类

3.5.5 节介绍了在单标签分类问题中模型损失的度量方法,即交叉熵损失函数,但是在实际应用中还会遇到多标签分类(Multi-Label Class)的情况,即对于每个样本来讲都可能存在不止一个正确标签的情况。例如在文本分类这一场景中,同一条文本可能涉及“体育”“娱乐”等多个类别标签。在接下来的内容中,将会详细介绍在多标签分类任务中两种常见的损失评估方法,以及在多标签分类场景中的模型评价指标。

3.13.1 Sigmoid 损失

在多标签分类场景中,第 1 种损失衡量方式就是将原始输出层的 Softmax 操作替换为 Sigmoid 操作,然后通过计算输出层与标签之间的 Sigmoid 交叉熵来作为误差的衡量标准,

具体计算公式为

$$\text{loss}(\boldsymbol{y},\hat{\boldsymbol{y}})=-\frac{1}{C}\sum_{i=1}^{m}\left[\boldsymbol{y}^{(i)}\cdot\log\left(\frac{1}{1+\exp(-\hat{\boldsymbol{y}}^{(i)})}\right)+(1-\boldsymbol{y}^{(i)})\cdot\log\left(\frac{\exp(-\hat{\boldsymbol{y}}^{(i)})}{1+\exp(-\hat{\boldsymbol{y}}^{(i)})}\right)\right] \tag{3-122}$$

其中，C 表示类别数量，$\boldsymbol{y}^{(i)}$ 和 $\hat{\boldsymbol{y}}^{(i)}$ 均为一个向量，分别用来表示真实标签和未经任何激活函数处理的网络输出值。

从式(3-122)可以发现，这种误差损失衡量方式其实就是在逻辑回归中用来衡量预测概率与真实标签之间误差的方法。

在 PyTorch 中，可以通过 torch.nn 模块中的 MultiLabelSoftMarginLoss 类来完成损失的计算，示例代码如下：

```
1  def Sigmoid_loss(y_true, y_pred):
2      loss = nn.MultiLabelSoftMarginLoss(reduction = 'mean')
3      print(loss(y_pred, y_true)) #0.5927
4
5  if __name__ == '__main__':
6      y_true = torch.tensor([[1, 1, 0, 0], [0, 1, 0, 1]], dtype = torch.int16)
7      y_pred = torch.tensor([[0.2, 0.5, 0, 0], [0.1, 0.5, 0, 0.8]],
                             dtype = torch.float32)
8      Sigmoid_loss(y_true, y_pred)
```

在上述代码中，第 6～7 行构造了两个样本的预测结果和真实标签，并且每个样本均有两个类别。同时，需要注意的是 MultiLabelSoftMarginLoss 默认返回的是所有样本损失的均值，可以通过将参数 reduction 指定为 mean 或 sum 来指定返回的类型。

在完成模型的训练过程后，可以通过以下方式来得到模型的预测结果：

```
1  def prediction(logits, K):
2      y_pred = np.argsort(-logits, axis=-1)[:, :K]
3      print("预测标签:", y_pred)
4      p = np.vstack([logits[r, c] for r, c in enumerate(y_pred)])
5      print("预测概率:", p)
6  prediction(y_pred, 2)
```

在上述代码中，第 1 行中 K 表示多标签的数量。运行结束以后，便可以得到如下结果：

```
1  预测标签: tensor([[1, 0], [3, 1]])
2  预测概率: [[0.5 0.2] [0.8 0.5]]
```

在上述输出结果中，第 1～2 行便是每个样本对应每个类别的标签，并且是以概率值递减进行排序的。

3.13.2　交叉熵损失

在衡量多标签分类损失的方法中，除了 Sigmoid 损失以外还有一种常用的损失函数。这种损失函数本质上是在单标签分类中用到的交叉熵损失函数的扩展版，单标签可以看作其中的一种特例情况，其具体计算公式为

$$\text{loss}(\boldsymbol{y},\hat{\boldsymbol{y}})=-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{q}\boldsymbol{y}_j^{(i)}\log\hat{\boldsymbol{y}}_j^{(i)} \tag{3-123}$$

其中，$\boldsymbol{y}_j^{(i)}$ 表示第 i 个样本第 j 个类别的真实值；$\hat{\boldsymbol{y}}_j^{(i)}$ 表示第 i 个样本第 j 个类别的输出经过 Softmax 处理后的结果。

例如对于如下样本来讲：

```
y_true = np.array([[1, 1, 0, 0], [0, 1, 0, 1.]])
y_pred = np.array([[0.2, 0.5, 0.1, 0], [0.1, 0.5, 0, 0.8]])
```

经过 Softmax 处理后的结果如下：

```
[[0.24549354 0.33138161 0.22213174 0.20099311]
[0.18482871 0.27573204 0.16723993 0.37219932]]
```

此时，根据式(3-123)可知，对于上述两个样本来讲其损失值为

$$\text{loss}=-\frac{1}{2}(1\cdot\log(0.245)+1\cdot\log(0.331)+1\cdot\log(0.275)+1\cdot\log(0.372))\approx 2.392 \tag{3-124}$$

由于 PyTorch 中并没有直接提供对应的实现，所以需要自己动手实现，示例代码如下：

```
def cross_entropy(logits, y):
    s = torch.exp(logits)
    logits = s / torch.sum(s, dim = 1, keepdim = True)
    c = -(y * torch.log(logits)).sum(dim = -1)
    return torch.mean(c)

if __name__ == '__main__':
    loss = cross_entropy(y_pred,y_true)
    print(loss)#2.392
```

在介绍完两种不同的损失度量方法后，再来看如何对多标签分类任务中模型的预测结果进行评估。根据多标签分类任务的性质，评估指标整体上可以分为两类：不考虑部分正确的评估指标和考虑部分正确的评估指标。下面开始分别进行介绍。

3.13.3 不考虑部分正确的评估指标

1. 绝对匹配率

所谓绝对匹配率(Exact Match Ratio)是指，对于每个样本来讲除非每个标签的预测结果均正确，否则认为该样本的预测结果为错误。也就是说只有预测值与真实值完全相同的情况下才算预测正确，因此其计算公式为

$$\text{MR}=\frac{1}{m}\sum_{i=1}^{m}I(y^{(i)}==\hat{y}^{(i)}) \tag{3-125}$$

其中，n 表示样本总数；$I(\cdot)$为指示函数(Indicator Function)，当 $y^{(i)}$ 完全等同于 $\hat{y}^{(i)}$ 时取 1，否则为 0。

从式(3-125)可以看出，MR 值越大，表示分类的准确率越高。

例如现有以下真实值和预测值：

```
y_true = np.array([[0, 1, 0, 1], [0, 1, 1, 0], [0, 0, 1, 1]])
y_pred = np.array([[0, 1, 1, 0], [0, 1, 1, 0], [1, 1, 0, 0]])
```

那么其对应的 MR 就应该是 0.333，因为只有第 2 个样本才算预测正确。此时，可以直接通过 sklearn.metrics 模块中的 accuracy_score 方法来完成计算[8]，示例代码如下：

```
from sklearn.metrics import accuracy_score
print(accuracy_score(y_true,y_pred)) #0.33333333
```

2. 0-1 损失

除了绝对匹配率之外，还有另外一种与之计算过程恰好相反的评估指标，即 0-1 损失(Zero-One Loss)。绝对准确率计算的是完全预测正确的样本占总样本数的比例，而 0-1 损失计算的则是预测错误的样本占总样本的比例，因此对于上面的预测值和真实值来讲，其 0-1 损失就应该为 0.667。对应的计算公式为

$$L_{0\text{-}1}=\frac{1}{m}\sum_{i=1}^{m}I(y^{(i)}\neq\hat{y}^{(i)}) \tag{3-126}$$

此时，可以通过 sklearn.metrics 模块中的 zero_one_loss 方法来完成计算[8]，示例代码如下：

```
from sklearn.metrics import zero_one_loss
print(zero_one_loss(y_true,y_pred))#0.66666
```

3.13.4　考虑部分正确的评估指标

从上面的两种评估指标可以看出，不管是绝对匹配率还是 0-1 损失，两者在计算结果时都没有考虑部分正确的情况，而这对于模型的评估来讲显然是不够准确的。例如，假设某个样本的正确标签为[1,0,0,1]，模型的预测标签为[1,0,1,0]。可以看到，尽管模型没有把该样本的所有标签都预测正确，但是同样也预测正确了一部分，因此，一种可取的做法就是将部分预测正确的结果也考虑进去[9]。

为了实现这一想法，文献[10]中提出了在多标签分类场景下的准确率(Accuracy)、精确率(Precision)、召回率(Recall)、F_1 值(F_1-Measure)和汉明损失(Hamming Loss)计算方法，整体思想类似于 3.9 节中的内容，下面逐一进行介绍。

1. 准确率

对于准确率来讲，其计算公式为

$$\text{Accuracy}=\frac{1}{m}\sum_{i=1}^{m}\frac{|y^{(i)}\cap\hat{y}^{(i)}|}{|y^{(i)}\cup\hat{y}^{(i)}|} \tag{3-127}$$

从式(3-127)可以看出，准确率计算的其实是所有样本的平均准确率，而对于每个样本来讲，准确率就是预测正确的标签数在整个预测为正确或真实为正确标签数中的占比。例如对于某个样本来讲，其真实标签为[0,1,0,1]，预测标签为[0,1,1,0]。那么该样本对应的准确率为

$$\text{Acc}=\frac{1}{1+1+1}=\frac{1}{3} \tag{3-128}$$

因此，对于如下真实结果和预测结果来讲：

```
y_true = np.array([[0, 1, 0, 1], [0, 1, 1, 0], [0, 0, 1, 1]])
y_pred = np.array([[0, 1, 1, 0], [0, 1, 1, 0], [1, 1, 0, 0]])
```

其准确率为

$$\text{Accuracy} = \frac{1}{3} \times \left(\frac{1}{3} + \frac{2}{2} + \frac{0}{4} \right) \approx 0.4444 \tag{3-129}$$

对于式(3-127)所示的计算过程来讲，其对应的实现代码如下[11]：

```
def Accuracy(y_true, y_pred):
    count = 0
    for i in range(y_true.shape[0]):
        p = sum(np.logical_and(y_true[i], y_pred[i]))
        q = sum(np.logical_or(y_true[i], y_pred[i]))
        count += p / q
    return count / y_true.shape[0]
print(Accuracy(y_true, y_pred)) #0.4444
```

2. 精确率

对于精确率来讲，其计算公式为

$$\text{Precision} = \frac{1}{m} \sum_{i=1}^{m} \frac{|y^{(i)} \bigcap \hat{y}^{(i)}|}{|\hat{y}^{(i)}|} \tag{3-130}$$

从式(3-130)可以看出，精确率其实计算的是所有样本的平均精确率，而对于每个样本来讲，精确率就是预测正确的标签数在整个预测为正确的标签数中的占比。例如对于某个样本来讲，其真实标签为[0,1,0,1]，预测标签为[0,1,1,0]。那么该样本对应的精确率为

$$\text{Pre} = \frac{1}{1+1} = \frac{1}{2} \tag{3-131}$$

因此，对于上面的真实值和预测值来讲，其精确率为

$$\text{Precision} = \frac{1}{3} \times \left(\frac{1}{2} + \frac{2}{2} + \frac{0}{2} \right) \approx 0.5 \tag{3-132}$$

对于式(3-130)所示的计算过程来讲，其对应的实现代码如下：

```
def Precision(y_true, y_pred):
    count = 0
    for i in range(y_true.shape[0]):
        if sum(y_pred[i]) == 0:
            continue
        count += sum(np.logical_and(y_true[i], y_pred[i]))/sum(y_pred[i])
    return count / y_true.shape[0]
print(Precision(y_true, y_pred))#0.5
```

3. 召回率

对于召回率来讲，其计算公式为

$$\text{Recall} = \frac{1}{m} \sum_{i=1}^{m} \frac{|y^{(i)} \bigcap \hat{y}^{(i)}|}{|y^{(i)}|} \tag{3-133}$$

从式(3-133)可以看出，召回率其实计算的是所有样本的平均召回率，而对于每个样本

来讲，召回率就是预测正确的标签数在整个正确的标签数中的占比。

因此，对于上面的真实值和预测值来讲，其召回率为

$$\text{Recall}=\frac{1}{3}\times\left(\frac{1}{2}+\frac{2}{2}+\frac{0}{2}\right)\approx 0.5 \tag{3-134}$$

对于式(3-134)所示的计算过程来讲，其对应的实现代码如下：

```
def Recall(y_true, y_pred):
    count = 0
    for i in range(y_true.shape[0]):
        if sum(y_true[i]) == 0:
            continue
        count += sum(np.logical_and(y_true[i], y_pred[i]))/sum(y_true[i])
    return count / y_true.shape[0]
print(Recall(y_true, y_pred))#0.5
```

4. F_1 值

对于 F_1 值来讲，其计算公式为

$$F_1=\frac{1}{m}\sum_{i=1}^{m}\frac{2|y^{(i)}\cap\hat{y}^{(i)}|}{|y^{(i)}|+|\hat{y}^{(i)}|} \tag{3-135}$$

从式(3-135)可以看出，F_1 计算的也是所有样本的平均 F_1 值，因此，对于上面的真实值和预测值来讲，其 F_1 值为

$$F_1=\frac{2}{3}\times\left(\frac{1}{4}+\frac{2}{4}+\frac{0}{4}\right)\approx 0.5 \tag{3-136}$$

对于式(3-135)所示的计算过程来讲，其对应的实现代码如下：

```
def F1Measure(y_true, y_pred):
    count = 0
    for i in range(y_true.shape[0]):
        if (sum(y_true[i]) == 0) and (sum(y_pred[i]) == 0):
            continue
        p = sum(np.logical_and(y_true[i], y_pred[i]))
        q = sum(y_true[i]) + sum(y_pred[i])
        count += (2 * p) / q
    return count / y_true.shape[0]
print(F1Measure(y_true, y_pred))#0.5
```

在上述 4 项指标中都是值越大对应模型的分类效果越好。同时，从式(3-127)、式(3-130)、式(3-133)和式(3-135)可以看出，在多标签场景下各项指标尽管在计算步骤上与单标签场景有所区别，但是两者在计算各个指标时所秉承的思想却是类似的。

当然，对于后面 3 个指标的计算，还可以直接通过 sklearn 库中的对应方法来完成，示例代码如下：

```
from sklearn.metrics import precision_score, recall_score, f1_score
print(precision_score(y_true = y_true,y_pred = y_pred,average = 'samples'))#0.5
print(recall_score(y_true = y_true, y_pred = y_pred, average = 'samples'))#0.5
print(f1_score(y_true,y_pred,average = 'samples'))#0.5
```

除了前面已经介绍的 6 种评估指标外，下面再介绍最后一种更加直观的衡量方法——

汉明损失(Hamming Loss)[8]。

5. 汉明损失

对于汉明损失来讲,它的计算公式为

$$\text{Hamming Loss}=\frac{1}{mq}\sum_{i=1}^{m}\sum_{j=1}^{q}I(y_j^{(i)}\neq\hat{y}_j^{(i)}) \tag{3-137}$$

其中,$y_j^{(i)}$ 表示第 i 个样本的第 j 个标签;q 表示一种有多少个类别。

从式(3-137)可以看出,汉明损失衡量的是所有样本中,预测错的标签数在整个标签数中的占比,所以对于汉明损失来讲,其值越小表示模型的表现结果越好,现有如下真实结果和预测结果:

```
y_true = np.array([[0, 1, 0, 1], [0, 1, 1, 0], [0, 0, 1, 1]])
y_pred = np.array([[0, 1, 1, 0], [0, 1, 1, 0], [1, 1, 0, 0]])
```

其汉明损失为

$$\text{Hamming Loss}=\frac{1}{3\times4}\times(2+0+4)\approx0.5 \tag{3-138}$$

对于式(3-138)所示的计算过程来讲,其对应的实现代码如下:

```
def Hamming_Loss(y_true, y_pred):
    count = 0
    for i in range(y_true.shape[0]):
        p = np.size(y_true[i] == y_pred[i])
        q = np.count_nonzero(y_true[i] == y_pred[i])
        count += p - q
    return count / (y_true.shape[0] * y_true.shape[1])
```

同时也可以通过 sklearn.metrics 中的 hamming_loss 方法来进行计算,示例代码如下:

```
from sklearn.metrics import hamming_loss
print(hamming_loss(y_true, y_pred)) #0.5
```

尽管在这里介绍了 7 种不同的评估指标,但是在多标签分类中仍然还有其他不同的评估方法,具体可以参见文献[9]。例如还可以通过 sklearn.metric 模块中的 multilabel_confusion_matrix 方法来分别计算多标签中每个类别的准确率、召回率等;最后来计算每个类别各项指标的平均值。有兴趣的读者可以自行去探索。

3.13.5 小结

本节首先介绍了两种在多标签分类场景中常用的模型损失函数,即 Sigmoid 损失和扩展交叉熵损失;接着分别介绍了不考虑部分正确和考虑部分正确的评估指标,包括绝对匹配率、0-1 损失、准确率、召回率等的原理和实现方法。