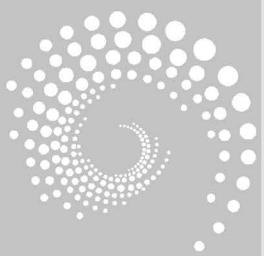
第3章

深度学习

CHAPTER 3



本章学习目标

- 了解神经网络的基本结构和发展历史
- 掌握梯度下降算法的原理及应用
- 利用 PyTorch 框架构建神经网络
- 通过两个项目案例，熟练掌握神经网络在回归与分类任务中的应用

本章将从神经网络的基础知识入手，逐步引导读者使用 PyTorch 构建和优化模型，并通过实际案例巩固所学内容。

3.1　神经网络

神经网络(Neural Network，NN)又称人工神经网络(Artificial Neural Network，ANN)，是一种通过模拟人脑神经元的输入、处理和输出过程，实现对信息的并行处理和自主学习的计算模型。自 20 世纪 80 年代以来，神经网络作为人工智能的重要分支，已经成为科学研究和工程应用中的关键技术之一。

3.1.1　神经网络简介

神经网络由多个相互连接的节点(或称神经元)组成，执行简单的计算并传递结果给网络中的下一级节点。这种结构使得神经网络具有强大的表征学习能力，可自动提取数据中的特征，并通过调整它们之间的连接权重来学习从输入到输出的映射，以实现对输入数据的分类、预测或其他任务。神经网络的结构如图 3.1 所示。

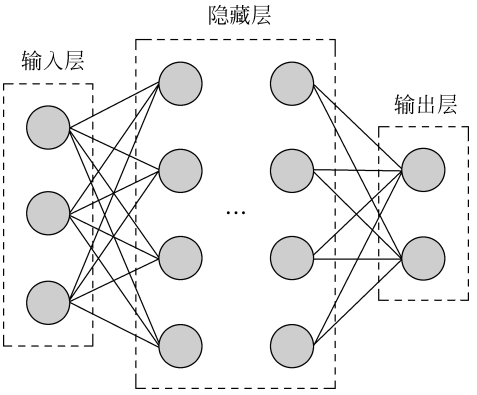


图 3.1　神经网络结构示意图

神经网络在构建之初，神经元之间的连线(权重)均为随机值，如同新生的婴儿，大脑一片空白，此时是不具备任何预测能力的。类比人类的幼年时期需要持续教育，智力水平才能获得发展和提高，这时需要将大量数据(题目)和标记(答案)作为训练内容，不断地“喂给”神经网络，而神经网络经过反复训练，学习到数据和标记之间的映射关系，这个过程称为模型训练，使用的输入数据和标记答案的集合称为“训练集”。通常来说，为检测模型的真实预测能力，需通过另外一组包含输入数据和标记答案的“测试集”来验证模型。模型预测值与标记(答案)之间的误差越小，说明模型越理想。

神经网络模型在训练期间，根据预测结果和标记之间的误差，通过不断调整神经元间的连接权重，尽可能减小二者之间的差距(损失值)。经过多轮训练，直到损失值不再变化或已经趋于 0，此时神经网络完成训练，预测能力获得提高，并最终可准确预测未知问题。

3.1.2　神经网络发展史

自 20 世纪 40 年代神经网络诞生以来，已经历了萌芽时期、创立时期、低谷时期和复兴时期共 4 个阶段，并在模式识别、计算机视觉和自然语言处理等领域得到了广泛应用。

1. 萌芽时期

神经网络的概念萌芽于 20 世纪初，当时心理学家和生物学家开始探索人类大脑的工作机制。德国心理学家威廉·冯特(Wilhelm Wundt)和美国神经科学家怀尔德·彭菲尔德(Wilder Graves Penfield)率先提出了神经元和神经网络的基本概念，并建立了早期的神经元模型，为神经网络的形成奠定了基础。随后，英国数学家阿兰·图灵(Alan Mathison

Turing)提出了图灵机模型,这一理论对后来的人工神经网络模型的发展产生了深远的影响。

2. 创立时期

1943 年,美国心理学家沃伦·麦卡洛克(Warren McCulloch)和数学家沃尔特·皮茨(Walter Pitts)提出了 MP 模型,这是第一个能够完整模拟神经元工作的数学模型,标志着神经网络的正式诞生。之后在 1957 年,弗兰克·罗森布拉特(Frank Rosenblatt)提出了感知机(Perceptron)算法,这是一种二层神经网络模型,可用于解决分类问题。与此同时,误差函数和梯度下降方法也被引入神经网络中,使得神经网络可以通过学习过程进行自我优化。

3. 低谷时期

受限于当时的计算能力和理论发展,1969 年,Marvin Minsky 和 Seymour Papert 在他们的著作《感知机》中指出感知机的局限性,即无法处理非线性问题,该书对人工智能的发展产生了重要影响,有关神经网络的研究第一次陷入长达十几年的低谷期。

进入 20 世纪 80 年代,人工智能领域的研究重点转向了专家系统。这些基于规则的系统在特定领域取得了一些成功,吸引了大量投资和研究资源。而神经网络在此期间未能取得预期的成果。尽管多层感知机和后向传播算法在 20 世纪 80 年代中期被重新发现并推广,但神经网络仍然面临诸多问题,如计算资源的限制和理论上的困境(梯度消失问题)。与专家系统相比,神经网络在实际应用中的表现相对较弱,导致学术界和工业界对神经网络的兴趣减弱,资金支持减少,许多研究人员转向其他领域。

在 20 世纪 90 年代中期,随着专家系统的局限性逐渐显现,人工智能领域再次面临困境。此时,神经网络的研究虽未完全停滞,但进展缓慢。虽然有支持向量机(SVM)等新算法的提出,但深度神经网络的训练仍然面临巨大挑战。训练深层网络所需的计算资源仍然不足,且在模型表现上未能显著超越传统方法。这一时期被认为是深度学习研究的"停滞(低谷)期"。

4. 复兴时期

2006 年,Geoffrey Hinton 团队利用受限玻尔兹曼机(RBM)和深度信念网络(DBN)进行逐层无监督预训练,再通过有监督的微调,解决了深层网络的梯度消失和梯度爆炸等难以训练的难题,为深度学习的崛起奠定了基础。

2012 年,Geoffrey Hinton 团队在 ImageNet 大规模视觉识别挑战赛(ImageNet Large Scale Visual Recognition Challenge,ILSVRC)中使用卷积神经网络(Convolutional Neural Network,CNN)取得了巨大成功。AlexNet 模型以极大的优势超越了传统的计算机视觉方法,错误率降低了近 10 个百分点。这一突破引发了全球范围内对深度学习的热潮,学术界和工业界开始广泛应用神经网络技术。

2012 年以后,深度学习迅速扩展到多个领域,如自然语言处理、语音识别、图像处理、自动驾驶等。科技公司如 Google、Meta 和 Microsoft 等纷纷投入大量资源进行深度学习研究和应用开发。

随着模型架构的不断创新(如 Transformer、GANs 等)和硬件技术的进步(如 GPU 和 TPU 的广泛应用),神经网络在人工智能领域的核心地位得到了进一步巩固,训练和部署大规模神经网络模型成为现实。例如,基于 Transformer 架构的大语言模型,如 GPT-3、GPT-4

等模型具有生成高质量自然语言文本的能力，广泛应用于教育、内容创作、编程辅助等领域。大语言模型的成功，标志着神经网络不仅在技术层面取得了突破，更在实际应用中获得了广泛认可。

3.1.3　神经网络模型

1. 单个神经元

神经网络模型(Neural Network Models，NNM)是模仿人脑神经元连接和工作原理的计算模型。它们由多个互联的节点(神经元)组成，单个神经元的结构如图 3.2 所示。

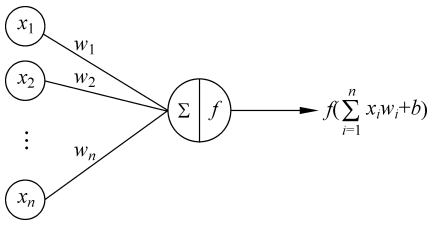


图 3.2　单个神经元结构示意图

在图 3.2 中，单个神经元是神经网络中最基本的运算单元。$X=(x_1,x_2,\cdots,x_n)$为输入数据，$W=(w_1,w_2,\cdots,w_n)$为当前神经元的连接权重，b 为偏置项，f 为激活函数。其中，W 和 b 称为该神经元的模型参数(简称参数)，神经网络的学习过程，就是寻求最佳模型参数 W 和 b 的过程。

激活函数 f 通常使用 Logistic 函数，常见形式包括 S 型函数(sigmoid)或双曲正切函数(tanh)。对于神经元为何需要引入激活函数，将在 3.2.2 节和 4.2.3 节中，结合具体案例详细阐释。

2. 神经网络的结构

神经网络是由多个神经元按照一定的规则(通常是“分层”)连接在一起而成的。不同层中神经元的数量通常根据实际需求设定。一般来说，一层神经元的数量表示神经网络的宽度，也是输入数据特征的维度，神经网络的层数表示神经网络的深度。模型的宽度越大，层次越深，含有的神经元个数就越多，网络表达能力越强，可以拟合的函数就越复杂。但是，如果神经网络中的神经元(参数)数量过多，模型会变得非常庞大，将显著提升计算时间和资源消耗。因为复杂的神经网络需要更多的训练时间和计算资源，一旦资源受限则将无法完成训练任务。此外，模型过于庞大时需要更多的训练数据，如果训练数据量不足以支撑对大量的模型参数进行优化，模型可能无法学到有效的特征，反而会受到数据稀疏性带来的负面影响。

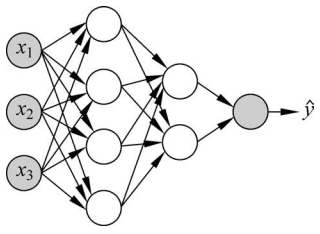


图 3.3　4 层神经网络结构示意图

因此，增加神经元并不是提高模型性能的最佳方式，选择合适的神经网络结构，才能更有效地处理特定类型的数据和顺利地完成训练任务。下面给出一个简单的神经网络的结构，如图 3.3 所示。

在图 3.3 中，给出了一个 4 层的神经网络。其中，最左层为输入层，最右层为输出层，中间的两层通常称为“隐藏层”，或简称为“隐层”。输入层中的圆形为输入数据，隐藏层和输出层中的圆形表示神经元，所有指向某个神经元的有向连接线为该模型的权重(模型参数)。除了输入层，任意层都可以将前一层的输出结果视为本层的数据输入。因此，在神经网络中数据计算结果能够逐层向前传递，并最终由输出层给出预测结果 $\hat{y}$，这个过程通常称为“前向传播”。

神经网络模型在训练过程中,得到的预测结果 $\hat{y}$ 与实际值 y 之间总是存在某种差距,称为损失(loss)。如果采用某种方法从 loss 开始,从后一层的结果向前一层进行反推,并逐层修改每个神经元对应的参数值,以使得 loss 值不断变小,这个过程称为“后向传播”。当 loss 值不再变化,或已收敛到一个较小值时,训练结束。通常来说,loss 值越小,预测的结果和真实值越接近,模型越优秀。

下面将通过一个现实生活中的“房价预测”问题,了解神经网络的工作过程,并动手实现一个房价预测模型,进行模型的训练、预测和评估。

3.2　学习案例 1:房价预测

通过房屋面积预测房价。本节重点学习内容如下。

- 通过线性回归模型预测房价。
- 如何获取模型的最佳参数。
- 如何评估模型的性能。

视频讲解

3.2.1　线性回归模型

线性回归(Linear Regression)是一种基于统计学和机器学习的方法,用于建模两个或多个变量之间的关系。其目的是通过拟合一个线性模型来预测一个因变量(目标变量)与一个或多个自变量(特征变量)之间的关系。

在现实生活中,房价受到多种因素的影响,如房屋面积、楼层以及地段等。为了简化问题,本书将仅选取房屋面积作为影响房价的自变量进行分析。将问题聚焦于单一因素,从而更容易进行建模与理解。房屋面积作为影响房价的自变量 x,房价为因变量 y,建立预测模型如式(3-1)所示。

$$\hat{y} = wx + b \tag{3-1}$$

式中:w——模型参数;

b——模型参数(偏置项)。

式(3-1)中,模型预测结果是 $\hat{y}$ 而非 y,这是因为 x 和 y 是观测结果,是客观事实存在,而 $\hat{y}$ 是模型计算得到的预测值,如图 3.4 所示。

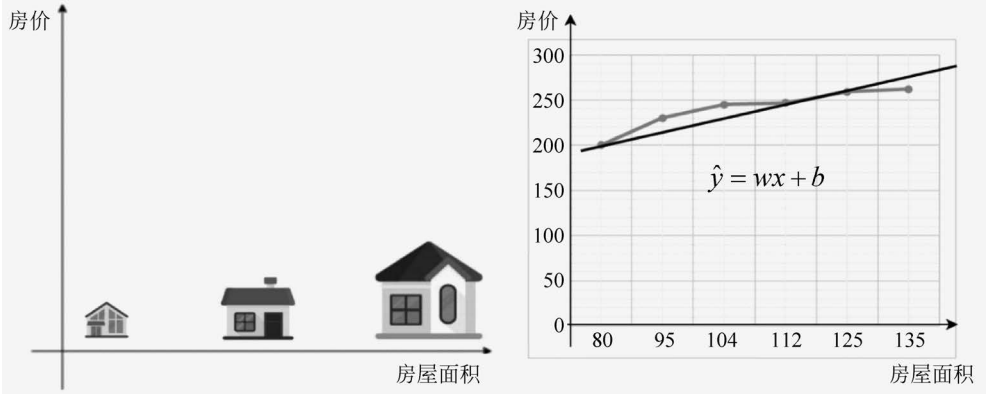


图 3.4　房价预测

通常来说，观测结果 x 和 y 称为“数据(特征)”和“标记”，(x,y)称为样本，多个样本的集合可用于模型的训练和测试，分别称为“训练集”和“测试集”。

根据市场调查，针对某地 2018 年的新房面积和售价情况，随机选取了其中 6 个样本，如表 3.1 所示。

表 3.1　某地 2018 年房屋价格情况

面积/m^2	售价/万元	面积/m^2	售价/万元
80	200	112	247
95	230	125	259
104	245	135	262

一般情况下，根据已观测数据，欲得到 $140m^2$ 的房屋价格，应该如何做呢？答案显而易见，通常的做法是：

- 首先，根据现有 6 个样本数据，建立房价预测模型 $\hat{y}=wx+b$。
- 然后，输入房屋面积数据 x，得到房屋预测价格 $\hat{y}$。


视频讲解

3.2.2　模型参数确定

对于线性模型 $\hat{y}=wx+b$，要根据 x 得到预测值 $\hat{y}$，需要确定参数 w 和 b 的值。根据现有数学知识，可以通过样本数据(x,y)来确定 w 和 b。下面通过 Python 代码来实现并求解。

首先，将样本集合定义为一个 6 行 2 列的数组对象，6 行对应 6 个样本，每行的 2 列分别对应“数据”和“标记”，然后将这些样本点在二维空间中描绘出来，观察这些点能否落在一条直线上，代码如下。

```
#代码示例 3-1
import numpy as np
data = np.array([[80, 200],
                 [95, 230],
                 [104, 245],
                 [112, 247],
                 [125, 259],
                 [135, 262]]
)
#绘制图形
import matplotlib.pyplot as plt
#提取数据特征到 X
X = data[:, 0]
#提取标记到 Y
Y = data[:, 1]
#在二维空间中打印坐标点(红色)
plt.scatter(X, Y, c="red")
plt.show()
```

程序运行结果如图 3.5 所示。

观察图 3.5，会发现，很难让所有的点都落在同一直线上。这是因为现实中的房价由多种因素决定，单纯只用面积来通过线性回归方程来预测房价，误差会很大。当然，如果能得到如图 3.6 所示的理想预测曲线，模型的预测准确性肯定是最佳的。

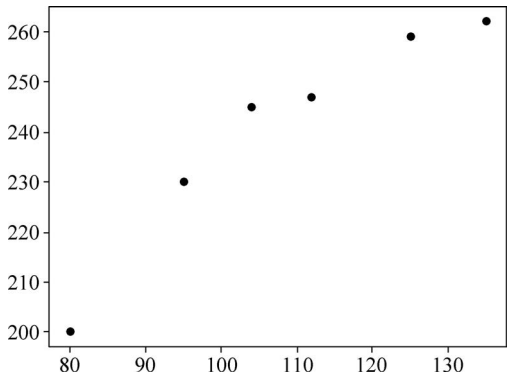


图 3.5　房屋面积与价格

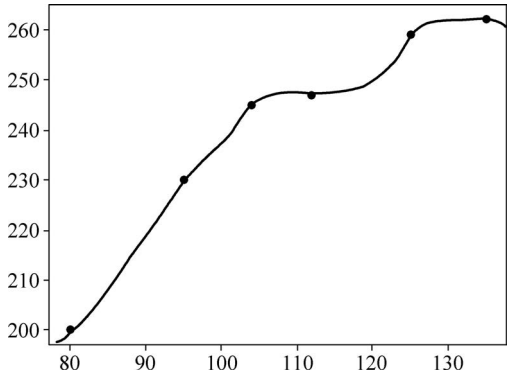


图 3.6　房屋面积与价格的理想预测曲线

如图 3.6 所示的曲线方程非常复杂,很难通过数学公式描述,求解会非常困难。为简化问题,通常采用线性回归方法获得预测方程(模型)的近似解。

一般地,采用二元一次方程组来求解线性回归模型 $y=wx+b$ 中的参数 w 和 b,最为简单和直接。从样本集合中,任取两个样本$[(x_1,y_1),(x_2,y_2)]$,如[[80,200],[95,230]],代入线性回归模型 $y=wx+b$ 中,则有

$$x_1=80,\quad y_1=200 \quad\rightarrow\quad 200=80\times w+b$$

$$x_2=95,\quad y_2=230 \quad\rightarrow\quad 230=95\times w+b$$

所以有

$$w=(y_2-y_1)/(x_2-x_1)$$

$$b=y_1-w\times x_1 \quad 或 \quad b=y_2-w\times x_2$$

采用方程组求解,使用任意两个样本就能确定 w 和 b 的值,现有 6 个样本数据:

[[80,200],
[95,230],
[104,245],
[112,247],
[125,259],
[135,262]]

任意两个样本$[[x_i,y_i],[x_j,y_j]]$的组合情况,将会有 $C_6^2=15$ 种可能。下面将利用代

码获得所有可能的组合情况。

```
#代码示例 3-2
import itertools
#首先,获得 15 种不同的样本组合数据,并将所有可能的组合保存在列表 com_lists 中
com_lists = list(itertools.combinations(data, 2))
#将 15 个不同的样本组合打印出来
for list in com_lists:
  print(list)
```

上述代码将打印出所有可能的样本数据组合(15 组),运行结果如下。

```
(array([ 80, 200]), array([ 95, 230]))
(array([ 80, 200]), array([104, 245]))
(array([ 80, 200]), array([112, 247]))
(array([ 80, 200]), array([125, 259]))
(array([ 80, 200]), array([135, 262]))
(array([ 95, 230]), array([104, 245]))
(array([ 95, 230]), array([112, 247]))
(array([ 95, 230]), array([125, 259]))
(array([ 95, 230]), array([135, 262]))
(array([104, 245]), array([112, 247]))
(array([104, 245]), array([125, 259]))
(array([104, 245]), array([135, 262]))
(array([112, 247]), array([125, 259]))
(array([112, 247]), array([135, 262]))
(array([125, 259]), array([135, 262]))
```

根据上述 15 组样本,分别计算可获得 15 组模型参数 w 和 b。下面将利用代码获得所有可能的模型参数:

```
#代码示例 3-3
import itertools
import numpy as np
#定义容器列表 ws 和 bs,分别存放不同组合获得的参数 w 和 b
ws = []
bs = []
#循环读取不同组合数据
for comlist in com_lists:
    x1, y1 = comlist[0]
    x2, y2 = comlist[1]
    w = (y2 - y1) / (x2 - x1)
    b = y1 - w * x1 #or b = y2 - w * x2
    ws.append(w)
    bs.append(b)
    print(f'({w:.2f}, {b:.2f})')
#计算所有 w 和 b 的平均值,并打印
mw, mb = np.mean(ws), np.mean(bs)
print(mw, mb)
```

程序运行结果,15 组模型参数 w 和 b,以及均值如下所示。

```
(2.00, 40.00)
(1.88, 50.00)
(1.47, 82.50)
(1.31, 95.11)
```

```
(1.13, 109.82)
(1.67, 71.67)
(1.00, 135.00)
(0.97, 138.17)
(0.80, 154.00)
(0.25, 219.00)
(0.67, 175.67)
(0.55, 187.97)
(0.92, 143.62)
(0.65, 173.96)
(0.30, 221.50)
1.0370514514185623    133.19792941461947
```

如果将预测模型的参数 w 和 b 分别设置为 15 组参数的平均值,则得到 $w=1.037$,$b=133.198$,那么预测模型为 $\hat{y}=1.037x+133.198$。

若要预测房屋面积为 $140m^2$ 时的房屋价格。令 $x=140$,将其代入公式 $\hat{y}=1.037x+133.198$ 中,可得 $\hat{y}=278.39$。此时,我们其实并不了解该模型预测结果的准确性。换言之,该值与实际值是否存在偏差,存在多大偏差,都是不知道的。

要了解预测模型的准确性,需根据测试样本进行评估。例如,逐一测试样本,将获得的预测值与标记值相减得到 loss 值,所有测试样本的平均 loss 值越小,说明预测模型的准确性越好。为避免求均值时遇到正负 loss 值相抵消的情况,通常采用均方误差来评估模型。

视频讲解

3.2.3 模型评估方法

均方误差(Mean Squared Error,MSE)是反映估计量与被估计量之间差异程度的一种度量。均方误差用于衡量数据偏离真实值的距离,是差值平方和的平均数。定义如式(3-2)所示。

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

式中:MSE——均方误差;

N——样本数量;

$\hat{y}$——预测值;

y——标记(观测值或理想值)。

评估预测模型前,通常将样本集合$[(x_1,y_1),(x_2,y_2),\cdots,(x_6,y_6)]$分为两部分:数据集合 $X=[x_1,x_2,\cdots,x_6]$和标记集合 $Y=[y_1,y_2,\cdots,y_6]$。将 X 输入预测模型,得到对应的预测值集合 $\hat{Y}=[\hat{y}_1,\hat{y}_2,\cdots,\hat{y}_6]$。这样就可以根据式(3-2)来计算 MSE,代码如下。

```
1   #代码示例 3-4  计算均方误差
2   import numpy as np
3   #losses 列表用于保存所有样本预测后得到的 loss 值
4   losses = []
5   i = 0
6   for x, y in data:
7       #进行预测
8       predict = mw * x + mb
```

```
        #计算损失
        loss = (y - predict) ** 2
        #打印损失
        i += 1
        print(f'第{i}个样本[{x},{y}]预测得到的 loss = {loss:.4f}')
        losses.append(loss)
    #打印平均损失
    print(f'6 个样本预测得到的 loss 均值为: {np.mean(losses):.4f}')print(np.mean(losses))
```

代码运算结果如下。

```
第 1 个样本[80,200]预测得到的 loss = 261.2117
第 2 个样本[95,230]预测得到的 loss = 2.9509
第 3 个样本[104,245]预测得到的 loss = 15.5924
第 4 个样本[112,247]预测得到的 loss = 5.5117
第 5 个样本[125,259]预测得到的 loss = 14.6640
第 6 个样本[135,262]预测得到的 loss = 125.4372
6 个样本预测得到的 loss 均值为: 70.8946
```

根据上述结果可知，6 个样本预测得到的均方误差(MSE)值为 70.89，误差仍然较大。通常来说希望预测模型的 MSE 越小越好，越小说明预测值越接近标记(理想)值，此时模型的预测准确率会更高。

那么，怎样才能找到最为理想的模型参数 w 和 b 的值，使得 MSE 获得最小值呢？通常会采用一种经典且有效的优化算法——梯度下降法，在训练过程中不断优化(更新)模型的参数 w 和 b，以实现 MSE 的最小化，进而得到最佳的模型拟合效果。


视频讲解

3.2.4　梯度下降算法

梯度下降算法是一种优化函数的迭代方法，通过沿着目标函数(如 MSE)的负梯度方向更新参数，从而逐步逼近目标函数的最小值。

算法的核心思想是：在每次迭代中，根据当前参数的位置计算目标函数的梯度，然后根据梯度的方向和步长调整参数值。通过不断迭代，目标函数的值将逐渐减小，最终达到局部最优解(最低点)，如图 3.7 所示。

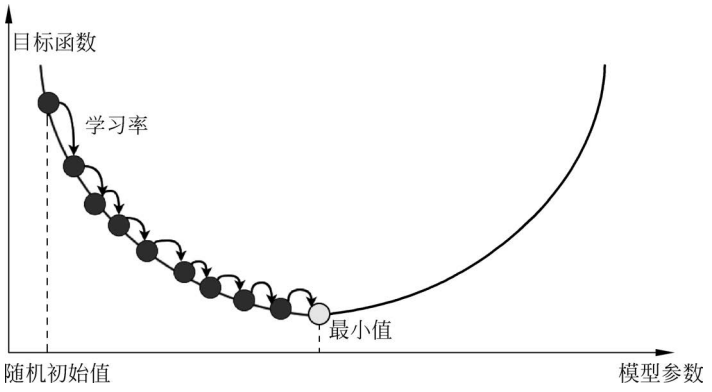


图 3.7　梯度下降算法

梯度下降法广泛应用于机器学习和深度学习中，用于优化模型的参数，从而最小化目标函数。通过不断迭代更新参数，梯度下降法帮助模型逐步接近最优状态(获得最优化参数)，

从而提升模型的预测准确率。

1. 相关概念

1) 目标函数

目标函数(Objective Function)是模型优化过程中需要最小化或最大化的函数。在机器学习中,通常是损失函数(Loss Function),如均方误差(MSE)或交叉熵损失(Cross-Entropy Loss)等。梯度下降法通过迭代优化使目标函数的值逐渐减小,从而提高模型的性能(预测准确率)。

2) 梯度

对于单变量目标函数 $f(x)$,梯度就是导数 $\mathrm{d}f(x)/\mathrm{d}x$,它表示目标函数 $f(x)$在 x 处的变化率。如果导数为正,表示函数在该处随着 x 的增大而增大;如果为负,表示函数随着 x 的增大而减小。

对于多变量目标函数 $f(x_1,x_2,\cdots,x_n)$,梯度是一个向量,由函数对每个变量的偏导数组成,定义如式(3-3)所示。

$$\nabla f(x_1,x_2,\cdots,x_n)=\left(\frac{\partial f}{\partial x_1},\frac{\partial f}{\partial x_2},\cdots,\frac{\partial f}{\partial x_n}\right) \tag{3-3}$$

梯度向量指示了目标函数 $f(x)$在当前位置上升最快的方向,同时其大小表示在该方向上的上升速率。因此,梯度下降法选择沿着梯度的反方向来优化参数,达到逐步减小目标函数值的目标。理想状态下,参数的优化过程持续到梯度接近零,此时目标函数达到最小值或接近最小值,算法收敛。模型参数的优化过程结束。

3) 学习率

学习率(Learning Rate)是控制每次梯度更新步幅大小的超参数。它决定了在每次迭代中,模型的参数应该沿着梯度方向移动的距离。

想象一下,"梯度下降过程"可以类比为在一条山路上寻找下山的最佳路径,学习率决定了下山(梯度下降)的速度,即每一步迈多大。

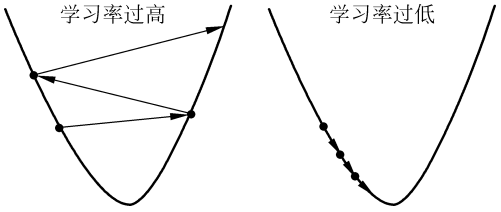


图 3.8 学习率过高和过低的情况

如果学习率设置太高,那么下山的步伐太大,可能会跨过山谷的最低点,一直在山坡上来回奔跑,永远抵达不了那个地势最低(梯度为零)的地方。如果学习率设置太小,就好比每次只迈出一小步,虽然不会错过山谷的最低点,但需要走很久才能到达目的地,如图 3.8 所示。

找到合适的学习率,就像找到合适的下山步伐,既不会跑过头,也不会走得太慢,这样才能又快又稳地到达山谷的最低点,同时也获得了模型的最佳参数。

4) 收敛性

收敛性是指梯度下降算法在多次迭代后,目标函数的值逐渐接近最小值并趋于稳定,在一个很小的范围内波动的过程。需要注意的是,算法的收敛性受学习率、目标函数的定义以及模型初始参数的选择等多种因素影响。

2. 公式推导

以均方误差(MSE)损失函数为例,利用梯度下降算法来迭代更新预测模型 $\hat{y}=wx+b$ 中的参数 w 和 b。

1) 损失函数定义

为统一表示不同类型的损失函数,通常用 J 来指代损失函数,无论是均方误差(MSE)、交叉熵损失(Cross-Entropy Loss)或是其他类型的损失函数。

一般地,预测模型可能像式(3-1)所示,用一个简单函数来表示,也可能是更为复杂的函数形式(模型中包含多个参数)。为统一描述这些模型中的参数,通常用向量$\boldsymbol{\theta}$ 来表示。那么对于 $\hat{y}=wx+b$,则有$\boldsymbol{\theta}=(\theta_1,\theta_2)$,其中,$\theta_1=w$,$\theta_2=b$。

因此,损失函数可以统一用 $J(\boldsymbol{\theta})$表示。如果 $\hat{y}=wx+b$ 表示为 $\hat{y}=f_{\boldsymbol{\theta}}(x)$,则 MSE 的定义修改为如式(3-4)所示。

$$J(\boldsymbol{\theta})=\frac{1}{N}\sum_{i=1}^{N}(f_{\boldsymbol{\theta}}(x_i)-y_i)^2 \tag{3-4}$$

式中:$J(\boldsymbol{\theta})$——均方误差损失函数;

N——样本数量;

$f_{\boldsymbol{\theta}}(x)$——预测模型,其中,$f_{\boldsymbol{\theta}}(x)=\theta_1 x+\theta_2$,并有 $\theta_1=w$,$\theta_2=b$;

x_i——第 i 个样本;

y_i——第 i 个标记。

2) 梯度下降算法

梯度下降算法的目标是确定最优的$\boldsymbol{\theta}$ 值,以得到最小的代价(损失)$J(\boldsymbol{\theta})$,记为 $J_{\min}(\boldsymbol{\theta})$。按此思路,通过循环迭代的方式,沿着梯度的反方向调整参数,逐步减小 $J(\boldsymbol{\theta})$,直到 $J(\boldsymbol{\theta})=J_{\min}(\boldsymbol{\theta})$。定义如式(3-5)所示。

$$\boldsymbol{\theta}:=\boldsymbol{\theta}-\alpha\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\theta}}J(\boldsymbol{\theta}) \tag{3-5}$$

式中:$\boldsymbol{\theta}$——预测模型参数向量;

α——学习率,模型参数更新的步长,其值通常较小。

3) 房价预测模型参数求解

首先,根据房价预测模型确定损失函数,房价预测模型为 $f_{\boldsymbol{\theta}}(x)=wx+b$,则损失函数定义如式(3-6)所示。

$$\mathrm{MSE}=\sum_{i=1}^{N}(wx_i+b-y_i)^2 \tag{3-6}$$

然后,根据损失函数 MSE,计算房价预测模型参数 w 和 b 的梯度$\frac{\nabla\mathrm{MSE}}{\nabla w}$和$\frac{\nabla\mathrm{MSE}}{\nabla b}$的定义如式(3-7)和式(3-8)所示。

$$\frac{\nabla\mathrm{MSE}}{\nabla w}=2\times\sum_{i=1}^{N}(wx_i+b-y_i)\cdot x_i \tag{3-7}$$

$$\frac{\nabla\mathrm{MSE}}{\nabla b}=2\times\sum_{i=1}^{N}(wx_i+b-y_i) \tag{3-8}$$

最后,根据房价预测模型参数的梯度值,结合学习率 α,分别更新 w 和 b 的值,如式(3-9)和式(3-10)所示。

$$w := w - \alpha \cdot \frac{\nabla \mathrm{MSE}}{\nabla w} \tag{3-9}$$

$$b := b - \alpha \cdot \frac{\nabla \mathrm{MSE}}{\nabla b} \tag{3-10}$$

3. 算法实现

利用 Python 实现梯度下降算法,代码如下。

```
#代码示例 3-5  实现梯度下降算法
#预测模型: f(x) = wx+b,用于预测房屋的真实价格
import numpy as np
data = np.array([ [80, 200],
                  [95, 230],
                  [104, 245],
                  [112, 247],
                  [125, 259],
                  [135, 262]]
     )
N = len(data)
#求解 f(x) = wx + b,其中,(x,y)来自 data,y 为标记数据
#目标: y 与 f(x)之间的差距尽量小
#初始化参数
w, b, lr = 1, 1, 0.00001
#梯度下降的函数
def gradientdecent(w, b, data, lr):
    #loss 为均方误差,wpd 为 w 的偏导数,bpd 为 b 的偏导数
    loss, wpd, bpd = 0, 0, 0
    for xi, yi in data:
        loss += (w * xi + b - yi) ** 2          #计算 MSE
        bpd += (w * xi + b - yi) * 2            #计算 loss/b 偏导数
        wpd += (w * xi + b - yi) * 2 * xi       #计算 loss/m 偏导数
    #更新 w 和 b
    loss = loss / N
    wpd = wpd / N
    bpd = bpd / N
    w = w - wpd * lr
    b = b - bpd * lr
    return loss, w, b
#训练过程
epochs = 5000000
for epoch in range(epochs):
    mse, w, b = gradientdecent(w, b, data, lr)
    if epoch % 2000000 == 0:
        print(f"loss = {mse:.4f},m = {w:.4f},b = {b:.4f}")
```

程序运行结果如下。

```
loss = 178.3555,w = 1.7144,b = 52.5560
loss = 87.8729,w = 1.4481,b = 82.2635
loss = 57.8181,w = 1.2947,b = 99.3849
```

```
loss = 47.8351, w = 1.2062, b = 109.2525
loss = 44.5191, w = 1.1553, b = 114.9396
```

根据上述结果可知，目前得到的最优参数为 $w=1.1553$，$b=114.9396$。此时，损失函数值最小值 loss=44.5191。

读者可以自行修改上述代码中的 w、b 和 lr 的初值，以及训练轮数 epoches 值，验证是否能使得 loss 值更小，以获得更优的模型参数值。


视频讲解

3.2.5　PyTorch 中的梯度下降

在实际应用中，神经网络模型通常由多层构成，每层包含多个神经元，因此模型的参数量往往非常庞大。在这种情况下，手动编写代码来计算梯度变得极其复杂且不可行。为了解决这一问题，PyTorch 框架提供了自动求导功能。通过这一功能，PyTorch 可以自动计算模型参数的梯度，并根据梯度更新参数，从而大大简化了训练过程。

1. 相关概念

为了让 PyTorch 框架自动计算梯度并更新模型的参数 θ，需要将 θ 定义为 PyTorch 的张量类型，并将其 requires_grad 属性设置为 True。这样，PyTorch 会自动为这些张量生成相应的梯度计算函数，并保存梯度计算的结果。在使用 PyTorch 框架进行模型参数更新之前，还需要进一步深入理解两个相关概念："前向传播"和"后向传播"。

1）前向传播

在实际应用中，仅包含一个神经元的单层神经网络通常无法有效解决现实问题。与之相对，多层神经网络能够通过多层计算进行信息处理，从而提高模型的表达能力。在进行预测时，输入数据经过神经网络的逐层传递与计算，最终生成预测结果。这个过程称为"前向传播"。

前向传播过程包括以下 4 部分。

- 输入层：将输入数据 x 传递给神经网络模型的第一个隐藏层。
- 隐藏层：每个隐藏层的神经元对输入进行加权求和，并将结果传递到下一层；那么当前隐藏层 l（不包括数据层）的输出如式(3-11)所示。

$$z^{(l)}=w^{(l)}a^{(l-1)}+b^{(l)} \tag{3-11}$$

式中：$z^{(l)}$——当前第 l 层（不包括数据层）的加权和；

$w^{(l)}$——第 l 层的参数值（权重）；

$a^{(l-1)}$——上一层的输出；

$b^{(l)}$——偏置项。

- 输出层：隐藏层的输出经过计算得到最终的预测结果 $\hat{y}$。
- 损失函数计算：将预测结果 $\hat{y}$ 与标记值 y 进行比较，计算损失 $J(\theta)$。

2）后向传播

"后向传播"与"前向传播"完全相反，后向传播主要用于优化模型的参数。在后向传播过程中，根据 $J(\theta)$，从网络的最后一层开始，首先计算该层每个参数的梯度，然后优化相应的模型参数；并随后通过链式法则逐层从后向前，计算前一层的梯度并优化该层模型参数，直到所有模型参数都得到更新。将整个过程分为三部分，详述如下。

- 计算模型参数的梯度：从输出层开始，计算损失函数 $J(\theta)$ 对各层参数的偏导数，即梯度。
- 后向传播：通过链式法则，逐层向前传播误差，并计算每层模型参数(以 w 为例)的梯度。例如，当前层 l 的梯度计算如式(3-12)所示。

$$\frac{\partial J}{\partial w^{(l)}}=\delta^{(l)}\cdot a^{(l-1)} \tag{3-12}$$

式中：$\delta^{(l)}$——当前第 l 层的误差项；

$a^{(l-1)}$——上一层的输出。

- 更新参数：使用计算出的梯度更新每层的参数(以 w 为例)，如式(3-13)所示。

$$w^{(l)}:=w^{(l)}-\alpha\frac{\partial J}{\partial w^{(l)}} \tag{3-13}$$

式中：$w^{(l)}$——第 l 层的参数(权重)；

α——学习率(控制更新步长)。

2. 自动计算梯度

为了确定哪些模型参数需要计算梯度并自动更新，PyTorch 框架将模型参数封装为张量形式，并提供了 requires_grad 属性。当将模型参数张量 θ 的 requires_grad 属性设置为 True 时，在前向传播过程中，PyTorch 会在该张量的整个前向传播过程中生成计算梯度的函数，并预留存储空间以保存梯度。

下面通过一个简单的例子，对 4 个模型参数张量 w_1、w_2、b_1 和 b_2，在定义张量对象时分别采用显式设置“requires_grad＝True”“requires_grad＝False”和采用默认设置，以观察 PyTorch 自动计算梯度的工作过程。代码示例如下。

```
#代码示例 3-6
import torch
#对 4 个张量自动赋值,并显式设置张量 w1 和 w2
w1 = torch.randn(1, requires_grad=True)          #设置为 True
w2 = torch.randn(1, requires_grad=False)         #设置为 False
#采用默认(隐式)设置张量 b1 和 b2
b1 = torch.randn(1)
b2 = torch.randn(1)
#打印 4 个张量
print(f'w1---->{w1}')
print(f'w2---->{w2}')
print(f'b1---->{b1}')
print(f'b2---->{b2}')
#打印 4 个张量的梯度计算函数
print(w1.grad_fn, w1.grad)
print(w2.grad_fn, w2.grad)
print(b1.grad_fn, b1.grad)
print(b2.grad_fn, b2.grad)
```

程序运行结果如图 3.9 所示。

根据图 3.9 中的代码输出结果可知，PyTorch 会将张量的 requires_grad 属性默认设置为 False，因此，要为张量生成梯度计算函数，必须显式将该张量的 requires_grad 属性赋值为 True。图 3.9 中的最后 4 行用于打印 w_1、w_2、b_1 和 b_2 的梯度函数和梯度值，结果全部为

None。这说明 4 个张量 w_1、w_2、b_1 和 b_2 定义结束后，PyTorch 未赋予其梯度计算函数和梯度值。

```
w1---->tensor([0.7859], requires_grad=True)
w2---->tensor([-1.6938])
b1---->tensor([-0.4415])
b2---->tensor([-2.9920])
None None
None None
None None
None None
```

图 3.9　输出结果

下面定义“前向传播”函数 forward，并调用该函数。然后观察这 4 个张量 w_1、w_2、b_1 和 b_2 是否会发生变化，即 PyTorch 是否会为其赋予相应的梯度计算函数和梯度值。代码如下。

```
#代码示例 3-7　针对参数 w1、b1、w2 和 b2 分别构造前向传播函数
def forward1(x):
    global w1, b1
    return w1 * x + b1
def forward2(x):
    global w2, b2
    return w2 * x + b2
#对于数据 x=2, 标记 y=5,预测模型 f(x) = wx + b,则有 w=2,b=1
x = 2
y = 5
#前向传播,构建了计算图
predict1 = forward1(x)
predict2 = forward2(x)
#打印参数 w1、b1、w2 和 b2 的梯度计算函数和梯度值
print(w1.grad_fn, w1.grad)
print(w2.grad_fn, w2.grad)
print(b1.grad_fn, b1.grad)
print(b2.grad_fn, b2.grad)
```

```
None None
None None
None None
None None
```

图 3.10　输出结果

程序运行结果如图 3.10 所示。

根据图 3.10 中的输出结果可知，前向传播过程发生后，PyTorch 并未为 4 个张量赋予梯度计算函数和梯度值。这是因为 PyTorch 采用的是动态计算图(Dynamic Computational Graph)技术，在前向传播过程中，PyTorch 只记录下各个操作的顺序和依赖关系，梯度计算函数只能在模型预测完成后，显式调用后向传播函数 backward()时才能建立。

当调用后向传播函数 backward()时，PyTorch 会基于前向传播时构建的计算图，开始反向追踪计算，自动计算每个张量的梯度。在这个阶段，PyTorch 会为每个 requires_grad=True 的模型参数建立对应的梯度计算函数，并计算这些梯度。验证代码如下。

```
#代码示例 3-8
#1.前向传播,构建了计算图
predict1 = forward1(x)
predict2 = forward2(x)
#2.构造损失(代价)函数
loss1 = (y - predict1) ** 2
loss2 = (y - predict2) ** 2
#3.查看梯度计算函数
print(f'loss1.grad_fn--->{loss1.grad_fn}')
print(f'loss2.grad_fn--->{loss2.grad_fn}')
print(f'w1.grad_fn--->{w1.grad_fn}')
```

```
print(f'w2.grad_fn--->{w2.grad_fn}')
print(f'b1.grad_fn--->{b1.grad_fn}')
print(f'b2.grad_fn--->{b2.grad_fn}')
#4.后向传播
loss1.backward()
#请思考,下面的代码在运行过程中,为何会出现异常?
loss2.backward()
#打印4个模型参数的梯度计算函数
print(w1.grad)
print(w2.grad)
print(b1.grad)
print(b2.grad)
```

程序运行结果如图 3.11 所示。

```
loss1.grad_fn--><PowBackward0 object at 0x00000265DB4DE340>
loss2.grad_fn--->None
w1.grad_fn--->None
w2.grad_fn--->None
b1.grad_fn--->None
b2.grad_fn--->None

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In [11], line 18
     16 loss1.backward()
     17 #请思考，下面的代码在运行过程中，为何会出现异常?
---> 18 loss2.backward()
     19 #打印4个模型参数的梯度计算函数
     20 print(w1.grad)
```

图 3.11 输出结果

对图 3.11 中的运行结果进行分析可知,代码示例 3-8 中的第 6、7 行执行了 loss1 和 loss2 的计算过程,此时前向传播过程结束。

- loss1 包含梯度计算函数,打印结果是:loss1.grad_fn--><PowBackward0 object at 0x0000020FBD5CB070>。
- 由于计算 loss2 的前向传播过程中不存在属性"requires_grad"为 True 的张量,因此 loss2 无须计算梯度,自然不会有梯度计算函数,所以 loss2 的梯度计算函数打印结果为 loss2.grad_fn--->None。

另外,模型的 4 个参数 w_1、w_2、b_1 和 b_2 仍然没有梯度值,打印结果为

```
w1.grad_fn--->None
w2.grad_fn--->None
b1.grad_fn--->None
b2.grad_fn--->None
```

根据上述实验结果可知,只有"后向传播"过程执行后,属性"requires_grad"为 True 的张量 w_1 才会得到梯度值,而 b_1、w_2 和 b_2 仍然不会得到梯度值。

在图 3.11 中,"RuntimeError"清楚地表明第 18 行代码发生运行时错误。这是由于 loss2 没有梯度计算函数,因此执行后向传播过程出现了错误。那么将第 18 行代码注释掉,然后再次运行,则程序运行结果如图 3.12 所示。

如图 3.12 所示,loss1 调用 backward()后,模型参数 w_1 的梯度值为 tensor([-28.6057]),由于其他三个模型参数 w_2、b_1 和 b_2 没有显式设置其"requires_grad"属性值为 True,因此这三个模型参数不具有梯度计算函数,也未获得梯度值。

```
loss1.grad_fn--><PowBackward0 object at 0x00000265DADC3A30>
loss2.grad_fn--->None
w1.grad_fn--->None
w2.grad_fn--->None
b1.grad_fn--->None
b2.grad_fn--->None
tensor([-28.6057])
None
None
None
```

图 3.12　输出结果

验证 PyTorch 自动计算得到的梯度值与手工计算得到梯度值是否相等，代码如下。

```
#代码示例 3-9
#初始化模型参数
w1 = torch.randn(1, requires_grad = True)
b1 = torch.randn(1,requires_grad = False)
def forward1(x):
    global w1, b1
    return w1 * x + b1
predict1 = forward1(x)
loss1 = (y - predict1) ** 2
loss1.backward()
#手工计算模型参数 w1 的偏导值 mpd1,并与 w1.grad 比较
wpd = (w1 * x + b1 - y) * 2 * x      #计算 loss/m 偏导数
print(wpd == w1.grad)
```

上述代码运行的输出结果为 True，表明 PyTorch 自动计算的梯度值与手动计算的梯度值完全一致。借助 PyTorch 框架对模型参数梯度的自动管理，不仅有效简化了手动计算梯度的复杂过程，还使得在大规模多层神经网络中实现梯度的自动计算成为可能。这种特性显著提升了开发效率，同时降低了出错概率，为深度学习模型的快速构建与优化提供了强有力支持。

3. 多层网络的梯度计算

为了简化问题描述，我们给出一个只包含两个神经元的多层网络。两个神经元对应的前向传播函数分别为 forward1 和 forward2，如图 3.13 所示。

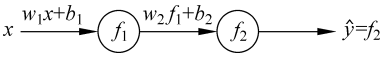


图 3.13　一个简单的多层神经网络

在图 3.13 中，前向传播过程(forward)分为两步，分别是

$$f_1 = \text{forward1}(x) = w_1 x + b_1$$

$$f_2 = \text{forward2}(f_1) = w_2 f_1 + b_2$$

在函数 forward2(z)中，其输入数据 z 为 forward1(x)的输出结果 f_1。多层神经网络的梯度计算代码如下。

```
#代码示例 3-10
#初始化模型参数
w1 = torch.randn(1, requires_grad = True)
b1 = torch.randn(1, requires_grad = True)
w2 = torch.randn(1, requires_grad = True)
b2 = torch.randn(1, requires_grad = True)
#定义前向传播函数
```

```
def forward1(x):
    global w1, b1
    return w1 * x + b1
def forward2(z):
    global w2, b2
    return w2 * z + b2
#1.前向传播,构建计算图
f1 = forward1(x)
f2 = forward2(f1)
#2.构造损失(代价)函数
loss = (y - f2) ** 2
#3.查看梯度计算函数
print(f'loss.grad_fn--->{loss.grad_fn}')
#4.后向传播
loss.backward()
print(w1.grad)
print(w2.grad)
print(b1.grad)
print(b2.grad)
```

程序运行结果如图 3.14 所示。

```
loss.grad_fn--><PowBackward0 object at 0x0000020FBD5A36D0>
tensor([-33.6368])
tensor([44.0505])
tensor([-16.8184])
tensor([-16.4741])
```

图 3.14 程序运行结果

在图 3.14 中,张量 loss 获得梯度计算函数,w_1、b_1、w_2 和 b_2 得到其梯度值,分别为 −33.6368,44.0505,−16.8184 和−16.4741。

4. 梯度清空

在 PyTorch 中,求解张量的梯度的方法是 torch.autograd.backward,若多次运行该函数,会将计算得到的梯度累加起来,代码如下。

```
#代码示例 3-11 梯度清空
#定义张量 w,其 requires_grad = True
w = torch.ones(4, requires_grad = True)
x = 2
#将 x 作为输入,两个前向传播过程 y 和 z
y = (w * x + 1).sum() #sum()的原因是: 只能对标量调用 backward()
z = (w * x).sum()
print(w)
print(y)
print(z)
```

程序运行结果如图 3.15 所示。

```
tensor([1., 1., 1., 1.], requires_grad=True)
tensor(12., grad_fn=<SumBackward0>)
tensor(8., grad_fn=<SumBackward0>)
```

图 3.15 程序运行结果

此时，尝试分别调用张量 y 和 z 的 backward()方法，由于 y 和 z 并非损失函数，所以此时的后向传播是没有任何意义的，我们只是观察由 y 得到的 w 的梯度值 w.grad，以及由 z 得到的 w 的梯度值 w.grad，代码如下。

```
#代码示例 3-12　调用 y 和 z 的后向传播(自动计算梯度)
y.backward()
print("dy/dw 导数:", w.grad)
z.backward()
print("dz/dw 导数:", w.grad)
```

程序运行结果如图 3.16 所示。

根据图 3.16 中的结果，可知如果对张量 y 和 z 分别求 w 的梯度，关于 w 的梯度计算值会累加到 w.grad 中。因此，计算张量 w 的梯度之后，需要进行梯度清空，避免下次计算梯度累加，代码如下。

```
#代码示例 3-13
#首先定义张量 x,其 requires_grad = True
w = torch.ones(4, requires_grad = True)
x = torch.tensor(2.0)
#将 x 作为输入,两个前向传播过程,结果赋值给 y 和 z
y = (w * x + 1).sum() #sum()的原因是: 只能对标量调用 backward()
z = (w * x).sum()
y.backward()
print("dy/dw 导数:", w.grad)
#梯度清空
w.grad.zero_()
z.backward()
print("dz/dw 导数:", w.grad)
```

程序运行结果如图 3.17 所示。

```
… dy/dw导数: tensor([2., 2., 2., 2.])
  dz/dw导数: tensor([4., 4., 4., 4.])
```

图 3.16　程序运行结果 1

```
… dy/dw导数: tensor([2., 2., 2., 2.])
  dz/dw导数: tensor([2., 2., 2., 2.])
```

图 3.17　程序运行结果 2

根据图 3.17 中的程序运行结果可知，执行梯度清空操作可以有效避免梯度的累积现象。这一点在预测模型的训练过程中尤为重要，由于训练需要通过大量样本进行循环迭代计算，如果未及时清空梯度，可能导致梯度累积影响模型的优化过程。因此，在每次迭代中，确保梯度被正确清空是深度学习训练中的关键步骤之一。

视频讲解

3.2.6　模型训练中的优化问题

根据前面章节学习的知识，从某种视角来看，神经网络可以视为一个非常复杂且有大量参数的复合函数(函数链条)。利用 PyTorch 框架，可以有效管理预测模型中的参数，在前向传播过程中，会将每个 requires_grad 属性为 True 的模型参数的操作顺序和依赖关系依次记录，并在后向传播过程中，逐一计算参数的梯度，并更新该参数。模型训练过程实际上就是不断迭代更新模型参数的过程。在后向传播阶段，梯度下降算法被用来计算每个模型参数的梯度方向，并沿着反方向逐步调整参数，目标是使损失函数的值达到最小，这意味着模型在训练过程中，会不断地朝着最优模型参数的方向前进。梯度下降过程如图 3.18 所示。

通常来说,可以将梯度下降算法类比为一个人下山的过程。假设一个人被困在山上,需要尽快下山到达谷底。然而,由于大雾弥漫,视野受到限制,无法看清楚下山的路径。只能依靠身边的地形来摸索前进的方向。由于下山的起点是随机的,加上只能沿着局部可见的坡度向下移动,因此很容易被困在一个山坳里,误以为自己已经到达了最低点,而实际上还没有走到真正的山谷底部。这种情况通常被称为"陷入局部最优",即虽然找到了一条看似不错的路径,但并未达到全局最优的目标,如图 3.19 所示。

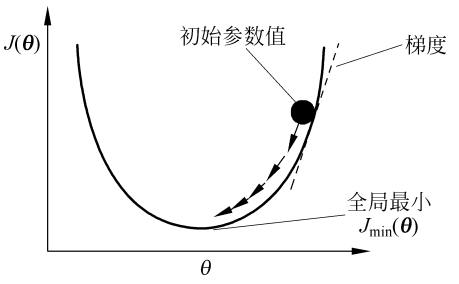


图 3.18　梯度下降过程

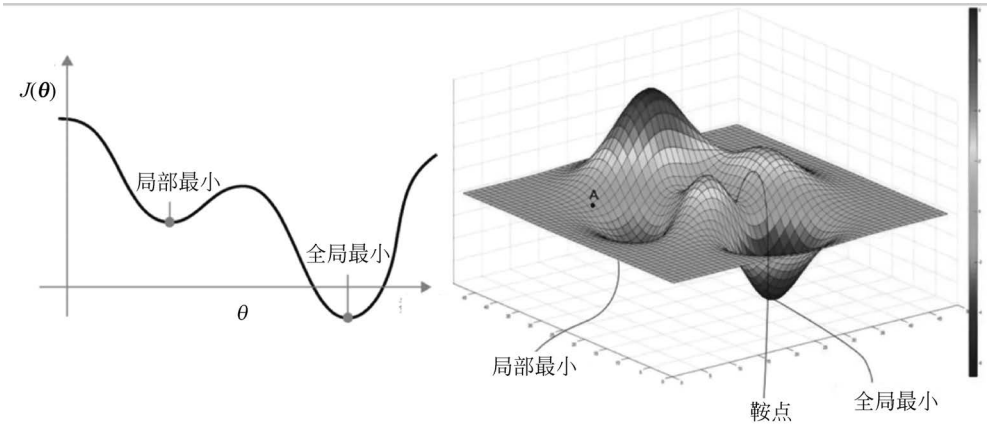


图 3.19　梯度下降过程中的局部最小和全局最小点

如图 3.19 所示,由于模型参数θ的初始值通常为随机数,所以如果起始点落入二维图中的"局部最小"区域,或三维图中的"鞍点"周边或"局部最小"区域,使用梯度下降算法,可能会始终无法摆脱困境,而陷入局部最优之中。

为了避免在训练过程中陷入局部最优的窘境,研究者们提出了一些有效的策略或方法来优化模型训练过程,统称为"优化方法"。

在 PyTorch 中,目前提供了多种优化器,它们能够帮助我们在模型训练中避免陷入局部最优,找到更优的参数,从而提升训练的效率,以及模型的性能。

1. 常见的 PyTorch 优化器

在 PyTorch 中,优化器(Optimizer)的主要作用是通过不断调整模型参数来最小化损失函数,从而提高模型的性能。优化器可以被看作一种管理工具,专门负责根据预定的优化策略更新模型参数权重,以实现自动和高效的模型训练过程。

PyTorch 提供了多种优化器,每种优化器都具有不同的参数更新策略,以适应不同类型的模型和任务需求。常见的优化器有以下几种。

1) 随机梯度下降

SDG(Stochastic Gradient Descent)是最简单且广泛使用的一种优化器。它在每次迭代中根据当前的梯度信息更新参数。定义如式(3-14)所示。

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta) \tag{3-14}$$

式中：θ——模型参数；

α——学习率；

$J(\theta)$——损失函数。

2）动量法

在 SGD 的基础上引入了“动量”，使参数更新时不仅考虑当前梯度，还考虑之前的梯度积累。定义如式(3-15)所示。

$$\theta = \theta - \alpha \cdot v_t, \quad v_t = \gamma \cdot v_{t-1} + \nabla_\theta J(\theta) \tag{3-15}$$

式中：v_t——当前的动量项；

γ——动量因子。

3）均方根传播

RMSProp(Root Mean Square Propagation)对每个参数使用不同的学习率，它根据过去的梯度信息动态调整学习率，适合处理非平稳的目标函数。定义如式(3-16)所示。

$$\theta = \theta - \alpha \cdot \frac{\nabla_\theta J(\theta)}{\sqrt{E[g^2]} + \dot{o}} \tag{3-16}$$

式中：$E[g^2]$——梯度平方的滑动平均值；

$\dot{o}$——防止分母为零的数值较小常数，通常取值为 10^{-8}。

4）自适应矩估计

Adam 是一种自适应学习率的优化算法，结合了动量法和 RMSProp 的优点，能够适应不同特征的学习速率。定义如式(3-17)所示。

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \dot{o}} \tag{3-17}$$

式中：$\hat{m}_t$——一阶矩的估计值；

$\hat{v}_t$——二阶矩的估计值。

2. 损失函数

在机器学习中，损失函数和优化器是模型训练过程中的两个关键组件，它们在模型训练过程中扮演着关键角色。损失函数用于衡量模型预测值与真实值之间的差异，是模型优化的目标；优化器则负责根据损失函数提供的梯度信息调整模型参数，从而逐步逼近最优解。

优化器的设计和选择往往受到损失函数的影响。不同的损失函数具有不同的梯度特性，如梯度大小、梯度分布的不均衡性等。这些特性直接影响优化器在更新参数时的效率和稳定性。因此，在实际应用中，为了获得理想的训练效果，需根据具体任务合理选择损失函数和优化器的组合，以充分发挥二者的协同作用。以下是一些常见的损失函数和优化器的组合建议。

1）均方误差

MSE 是一种常用的回归任务损失函数，特别适用于目标变量为连续值的情况。其核心思想是通过计算预测值与真实值之间的平方差来衡量模型的预测误差，并将所有误差取平均值。在 PyTorch 框架中，均方误差通过 torch.nn.MSELoss()实现。

常用的优化器包括：

- SGD 常用于简单的回归问题。
- Adam 能够自适应调整学习率，通常收敛会更快，常用于复杂的回归任务中。

2) 交叉熵损失

交叉熵损失适用于分类任务，包括多元交叉熵(Categorical Cross-Entropy，CE)损失和二元交叉熵(Binary Cross-Entropy，BCE)损失，分别对应多分类(激活函数 Softmax)和二分类(激活函数 Sigmoid)。

在 PyTorch 框架中，多分类损失函数定义为 torch. nn. CrossEntropyLoss()，二分类损失函数定义为 torch. nn. BCELoss()。

常用的优化器包括：

- Adam 优化器通常在多分类任务中表现稳定且收敛速度快。
- Momentum 帮助加速 SGD 的收敛，因此适用于一些大型数据集的情况。
- RMSprop 对具有不同梯度尺度的分类任务表现良好。

3) 绝对误差

绝对误差(MAE)适用于回归任务，尤其是当数据中有许多异常值的情况。这是因为 MSE 将误差平方处理，对较大的误差惩罚更重，因此异常值可能对模型训练产生较大影响，导致模型过度拟合这些异常值，降低了整体性能。而 MAE 只计算误差的绝对值，能够更好地抵御异常值的影响，使模型更关注数据的整体趋势，提升模型的鲁棒性。在 PyTorch 框架中的定义形式为 torch. nn. L1Loss()。

常用的优化器包括：

- 对于绝对误差损失函数，Adam 的自适应学习率调节可以带来较好的效果。
- SGD 在一定程度上也能提供很好的适应性，但可能需要更精细的学习率调整。

4) KL 散度

KLD(Kullback-Leibler Divergence)是一种常用的损失函数，通常用于生成模型中的分布匹配任务，例如，变分自编码器(Variational Autoencoder，VAE)中用来度量两种概率分布之间的差异。在 PyTorch 框架中的定义形式为 torch. nn. KLDivLoss()。

常用的优化器包括：

- Adam 能够稳定训练过程并加速收敛。
- RMSprop 通常在分布匹配任务中也表现良好。

5) 自定义损失函数

通常来说，标准损失函数(如均方误差、交叉熵损失等)能够满足大部分机器学习任务的需求。然而，对于某些特殊的任务或模型，标准损失函数可能无法满足其特定需求。在这种情况下，需要自定义损失函数。

常用的优化器包括：

- Adam 通常是一个通用且有效的选择，适用于多种自定义损失函数。
- Momentum 也常用于特定的自定义损失函数，但可能需要对 SGD 进行调优。

综合来说，Adam、SGD 和 RMSprop 是较常使用的优化器，其中，Adam 具有自适应学习率和动量特性，是一个通用且有效的优化器，适用于大多数损失函数和任务；SGD 特别适用于需要细粒度控制学习率的任务，并且在大规模数据集上表现稳定；RMSprop 对处理

不同梯度尺度的任务表现良好，适合于一些特定的损失函数。

选择合适的损失函数和优化器组合需要根据具体的任务和数据来确定。实验和调优是找到最佳组合的重要步骤。

3. 模型训练与优化

通过学习 PyTorch 框架中定义的损失函数和优化器，可以对 3.2.4 节中的房价预测算法进行重新设计与编码。具体来说，利用合适的损失函数与优化器，可以更高效地优化模型的训练过程，提高预测性能。以下是重新设计的代码示例。

```
#代码示例 3-14  利用 f(x) = wx + b 预测房屋的真实价格(一)
import numpy as np
import torch
import torch.nn as nn
#样本数据
data = np.array([
            [80, 200],
            [95, 230],
            [104, 245],
            [112, 247],
            [125, 259],
            [135, 262]]
)
#初始化参数
Xs = torch.tensor(data[:, 0], dtype = torch.float32)
Ys = torch.tensor(data[:, 1], dtype = torch.float32)
w = torch.randn(1, dtype = torch.float32, requires_grad = True)
b = torch.randn(1, dtype = torch.float32, requires_grad = True)
#定义前向传播函数
def forward(x):
    return w * x + b
#定义损失函数和优化器
lossFun = nn.MSELoss()
learning_rate = 0.00001
epoches = 5000000
optimizer = torch.optim.SGD([w, b], lr = learning_rate)
#开始模型训练
for epoch in range(epoches):
    #1.前向传播
    y_pre = forward(Xs)
    #2.计算损失
    loss = lossFun (Ys, y_pre)
    #3.后向传播(计算梯度)
    loss.backward()
    #4. 优化器进行更新权重
    optimizer.step()
    #5.优化器来清空梯度
    optimizer.zero_grad()
    if epoch % 1000000 == 0:
        print(f"epoch:{epoch}, w = {w.item():.8f},b = {b.item():.8f},loss = {l:.8f}")
```

最终运行结果，得到的最优参数为 $w=1.147\,982$，$b=115.753\,219\,6$，此时，损失函数值最小值 loss=44.190 418 24。

基于本节的学习内容，读者可以尝试在模型训练过程中应用不同的优化策略。例如，调

整模型参数的初始值、修改超参数(如学习率或批量大小),或更换优化器(如使用 Adam 或 RMSProp 等)。通过对代码的调整和实验,探索是否能够获得性能更优的预测模型,使损失值进一步降低,并加深对训练过程和优化方法的理解。

4. PyTorch 中的组件

PyTorch 提供了许多功能强大的组件,包括全连接层、卷积神经网络和循环神经网络等。这些组件具有高度灵活性,能够帮助研究人员和工程师轻松构建从简单到复杂的神经网络模型,从而满足多样化的任务需求。

得益于其灵活的设计和强大的功能,PyTorch 广泛应用于计算机视觉、自然语言处理等领域,成为深度学习研究与实际应用中的重要工具。无论是快速原型开发还是复杂模型的部署,PyTorch 都能够提供高效的支持。

本节将介绍 PyTorch 中最基础和重要的组件之一——全连接层(又称线性层)。在 PyTorch 中,全连接层由 torch.nn.Linear()定义,用于实现简单的线性变换,如式(3-18)所示。

$$f(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{W}^{\mathrm{T}} + \boldsymbol{b} \tag{3-18}$$

式中:$\boldsymbol{W}$——二维权重矩阵;

$\boldsymbol{b}$——偏置向量;

$\boldsymbol{x}$——输入数据。

注意,在式(3-18)中,$\boldsymbol{W}$ 是二维的权重矩阵,两个维度的长度分别对应输入数据 $\boldsymbol{x}$ 的维度 r 和输出结果 $f(\boldsymbol{x})$的维度 c,如图 3.20 所示。

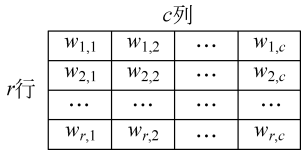


图 3.20 W 权重矩阵

使用 torch.nn.Linear(in_features, out_features)来创建线性层时,其构造函数包含以下两个关键参数。

- in_features:输入数据的维度,即输入张量包含数据(特征)的个数。
- out_features:输出特征的维度,即线性变换后输出张量包含数据(特征)的个数。

由此可知:

- 输入数据 $\boldsymbol{x}$ 的维度:(…,in_feature)。
- 参数矩阵 $\boldsymbol{W}$ 的维度:(in_feature,out_feature)。
- 输出结果 $f(\boldsymbol{x})$的维度:(…,out_feature)。

例如,torch.nn.Linear(10, 5)表示输入数据 $\boldsymbol{x}$ 包含 10 个特征,经过线性变换后输出结果 $f(\boldsymbol{x})$包含 5 个特征,且模型参数 $\boldsymbol{W}$ 的矩阵形状为(10,5)。

下面将使用 PyTorch 提供的全连接层组件 torch.nn.Linear()对房价预测的实现代码进行改写。通过 torch.nn.Linear(),可以直接实现输入与输出之间的线性映射,进一步简化代码结构并提高代码可读性,使得模型的定义更加清晰和模块化。代码如下。

```
#代码示例 3-15  利用 f(x) = wx+b,预测房屋的真实价格(二)
import numpy as np
import torch
import torch.nn as nn
#样本数据
data = np.array([[80, 200], [95, 230], [104, 245], [112, 247], [125, 259], [135, 262]])
#初始化参数
Xs = torch.tensor(data[:, 0], dtype=torch.float32).view(-1, 1)
```

```
9   Ys = torch.tensor(data[:, 1], dtype = torch.float32).view( - 1, 1)
10  #定义全连接层对象
11  linear = torch.nn.Linear(1, 1)          #输入维度是一维的面积,输出维度是一维的房价
12  #定义前向传播函数
13  def forward(x):
14      return linear(x)
15  #定义损失函数和优化器
16  lossFunc = nn.MSELoss()
17  learning_rate = 0.00001
18  epoches = 5000000
19  optimizer = torch.optim.SGD(linear.parameters(), lr = learning_rate)
20  #开始模型训练
21  for epoch in range(epoches):
22      #1.前向传播
23      y_pre = forward(Xs)
24      #2.计算损失
25      loss = lossFunc (Ys, y_pre)
26      #3.后向传播(计算梯度)
27      loss.backward()
28      #4.优化器进行更新权重
29      optimizer.step()
30      #5.优化器来清空梯度
31      optimizer.zero_grad()
32      if epoch % 20000 == 0:
33          #从 linear 对象中获取当前的权重和偏置
34          w, b = linear.weight.item(), linear.bias.item()
35          print(f"epoch:{epoch}, w = {w:.8f},b = {b:.8f},loss = {loss:.8f}")
```

上述代码中,第 8、9 行通过 view(－1, 1)操作进行维度对齐,将 Xs 和 Ys 转换为二维张量,形状为(n_samples, 1),确保输入和输出数据的维度符合 PyTorch 的要求。

对于.view(－1, 1)中的两个参数:

- －1 表示将自动推断张量在第一个维度上的大小,以匹配原始数据的总长度。
- 1 表示第二个维度固定为 1,即每个样本对应一个目标值。

代码第 19 行中的 linear.parameters()是一个可迭代对象,包含模型中所有可训练的参数。通过将模型 linear 的参数传递给优化器,优化器可以在训练过程中自动管理这些参数,以简化编程过程。

由于线性函数被 torch.nn.Linear()对象替代,为了方便打印模型权重 w 和偏置 b,在第 34 行代码中,通过 linear.weight 和 linear.bias 获取当前的权重 w 和偏置 b,并在第 35 行打印其内容。这样有助于对模型的参数变化进行追踪和理解,确保权重和偏置的正确性。

最后,本书将为读者介绍一个 PyTorch 中的重要组件“torch.nn.Module”,是 PyTorch 中所有神经网络模块的基类。无论是构建简单的线性层,还是设计复杂的卷积神经网络,都可以通过继承 torch.nn.Module 来实现。

通过继承 torch.nn.Module 类,用户可以轻松定义、组织和管理复杂的神经网络模型。其模块化的设计大大提高了代码的可读性和可维护性,同时简化了参数管理和前向传播的实现。无论是用于分类、回归,还是其他深度学习任务,torch.nn.Module 都提供了强有力的支持。

在后续章节中,将详细介绍核心组件 torch.nn.Module,并在之后的学习中使用它。


视频讲解

3.2.7 神经网络类

在使用 PyTorch 建立自定义神经网络类时,通常需要继承 nn.Module 类。在自定义类中,需要重写两个主要方法:构造函数(__init__)和前向传播函数(forward)。具体代码如下。

```
#代码示例 3-16  神经网络类的定义
class Model(nn.Module):
    #初始化方法用于模型参数传递和模型中的组件定义
    def __init__(self):
    #前向传播,用于构建动态计算图
    def forward(self, x):
```

在自定义神经网络类中,构造函数主要用于定义网络的各个层及其参数,为前向传播过程做准备;而前向传播函数则负责实现输入数据在网络中的传递和计算,完成模型的推理过程。

1. 定义神经网络类

为了更好地理解神经网络的构建过程,接下来以房价预测为例,定义一个简单的神经网络类实现。在这个例子中,将使用 PyTorch 来构建一个基础的神经网络模型,并通过继承 nn.Module 类来定义神经网络的结构。具体实现代码如下。

```
#代码示例 3-17  构建自定义神经网络类,并创建模型对象
import torch
#自定义神经网络类,继承 torch.nn.Module
class MyModel(torch.nn.Module):
    #构造函数,其中,in_features 和 out_features 分别指定输入数据和输出数据的维度
    def __init__(self, in_features, out_features):
        super(MyModel, self).__init__()
        #构建线性层
        self.linear = torch.nn.Linear( in_features, out_features)
    #向前传播,构建计算图
    def forward(self, x):
        out = self.linear(x)
        return out
#输入数据维度
input_len = 1
#输出数据维度
output_len = 1
#创建自定义神经网络类 MyModel 的对象 model
model = MyModel(input_len, output_len)
#打印该模型对象的主要结构
print(model)
#打印所有命名对象
for name, param in model.named_parameters():
    print(f"Name: {name}, Shape: {param.shape}, Values: {param.data}")
```

在上述代码中:

- 第 3～13 行自定义了神经网络类 MyModel,并在构造函数中定义了一个实例属性(线性层)self.linear,该线性(全连接)层通过 torch.nn.Linear(in_features, out_

features)创建。其中,in_features 和 out_features 分别指定输入数据和输出数据的维度。

- 建立模型对象 model 后,第 21 行用于打印该模型对象的主要结构,目前只包含一个 torch.nn.Linear 对象(包括该对象的输入、输出维度等信息)。
- 第 23 行中,模型 model 的 named_parameters()方法将返回所有命名对象,并依次取出每个命名对象的名称和对象属性。
- 第 24 行,将依次打印模型中参数的名称、形状(shape)和参数值,如图 3.21 所示。

```
MyModel(
  (linear): Linear(in_features=1, out_features=1, bias=True)
)
Name: linear.weight, Shape: torch.Size([1, 1]), Values: tensor([[-0.7367]])
Name: linear.bias, Shape: torch.Size([1]), Values: tensor([0.9223])
```

图 3.21　模型结构和命名参数对象列表

2. 生成模拟数据

在实际应用中,模型的训练通常需要大量的样本数据。为了便于实验并快速获取更多样本数据,本节将介绍如何使用机器学习库 Scikit-learn(简称 sklearn)中的工具生成线性回归模拟数据。

下面将利用 datasets.make_regression 方法生成一组包含 100 个样本的线性回归数据,并通过 Matplotlib 对生成的数据进行可视化。以下代码展示了具体实现过程。

```
1  #代码示例 3-18　利 sklearn 库生成模拟数据
2  import numpy as np
3  from sklearn import datasets
4  import matplotlib.pyplot as plt
5  #随机生成 100 个带噪声的线性回归样本点
6  X, Y = datasets.make_regression(n_samples=100, n_features=1, noise=20, random_state=12)
7  plt.plot(X, Y, "ro")      #红色圆点
8  plt.show()
```

上述代码的第 6 行,调用 sklearn 库中的 datasets 类的 make_regression 方法生成 100 个样本,其中参数:

- noise=20,表示在目标值(因变量)中加入正态分布噪声(大小为 20,取值范围为 0～100)。
- random_state=12,用于设置随机种子值为 12,从而确保生成的随机数据在每次运行时保持一致。这种设置使实验结果具有可重复性,即使随机性本质上是伪随机的。

由于生成样本时引入了噪声,因此生成的数据点将围绕回归线分布,而不是完美地落在回归线上,运行结果如图 3.22 所示。

通常来说,当 noise 的值较低时,利用 make_regression 方法生成样本数据将更接近线性分布,数据点会集中在回归线附近,此时线性回归模型更容易拟合;当 noise 的值较高时,生成的数据将包含更多的随机波动,数据点将分布得更为散乱,回归线的拟合难度增加,这种情况更能反映实际数据中的噪声特征。

下面尝试将 noise 值设置为 0,以验证这些点是否能够完美地落在回归线上,代码如下。

```
#代码示例 3-19　随机生成 100 个无噪声线性回归样本点
X0, Y0 = datasets.make_regression(n_samples = 100, n_features = 1, noise = 0, random_state = 12)
plt.plot(X0, Y0, "ro")　　#红色圆点
plt.show()
```

程序运行结果如图 3.23 所示。

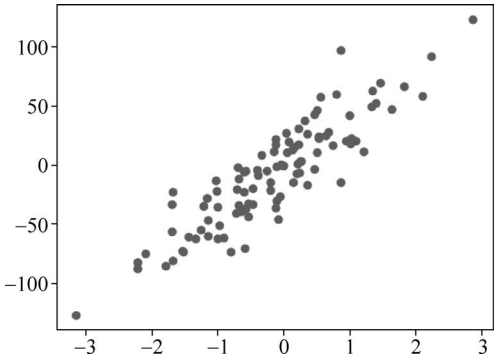


图 3.22　sklearn 生成 100 个带噪声的线性回归样本点

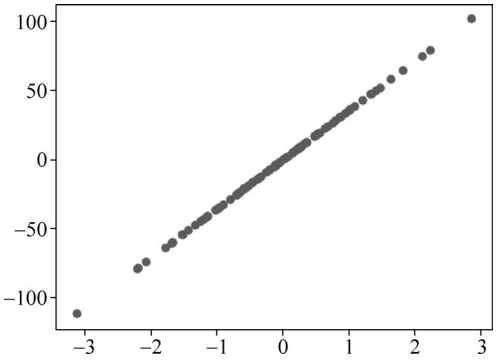


图 3.23　sklearn 生成 100 个无噪声的线性回归样本点

在使用 sklearn 库生成线性回归模拟数据时,需要注意生成的数据类型与 PyTorch 默认的数据类型存在差异。为了将这些数据应用于 PyTorch 的训练过程,必须进行相应的数据类型转换。以下是修改后的代码示例。

```
#代码示例 3-20
import torch
#将 NumPy 的数据类型转成 tensor
Xs = torch.from_numpy(X.astype(np.float32))
Ys = torch.from_numpy(Y.astype(np.float32)).view(-1, 1)
#查看 Xs 和 Ys 将 NumPy 的数据类型转成 tensor
print(f'X.shape:{X.shape}')
print(f'Y.shape:{Y.shape}')
print(f'Xs.shape:{Xs.shape}')
print(f'Ys.shape:{Ys.shape}')
```

在上述代码中:

- 第 4、5 行代码将 NumPy 格式的数据转换为 PyTorch 的张量类型(torch.Tensor)。
- 第 5 行代码在将 NumPy 格式的数据转换为 PyTorch 的张量类型后,进行了维度对齐操作,Ys 获得与 Xs 完全相同的二维张量形状,以确保输入和输出数据的维度符合要求。

最终运行结果如图 3.24 所示。

```
X.shape:(100, 1)
Y.shape:(100,)
Xs.shape:torch.Size([100, 1])
Ys.shape:torch.Size([100, 1])
```

图 3.24　线性回归模拟数据的维度

对于房价预测模型,房屋特征数据通常会包含多个维度,例如,“面积”“地段”“楼层”等。但是,为了简化问题和便于理解,当前模型的输入数据仅包含一个数据特征——“面积”。通过这种简化处理,可以专注于单一特征“面积”对目标值“房价”的影响,使学习者更直观地理解模型的基本原理和运行机制。因此,100 个面积数据构成输入数据集 X,其 shape 为(100,1),而输出的房价预测值默认只有一个维度“价

格”，因而这里对应的 100 个价格数据构成标记数据集 Y，其 shape 为(100,)。

由于 X 和 Y 的 shape 不一致，需要进行维度对齐。因此，在使用 PyTorch 封装 X 和 Y 为张量后，代码第 5 行利用.view(－1，1)将 Y 的维度从一维提升到二维(100,1)，否则在计算损失时，会因为维度问题而提示错误。

3. 模型训练与预测

在使用自定义的神经网络类 MyModel 创建预测模型 model 后，需继续创建损失函数对象 lossFun 和优化器对象 optimizer。接着，利用生成的 100 个模拟线性回归数据样本(Xs,Ys)对模型进行训练。以下是完整的代码实现。

```
#代码示例 3-21
import torch
#1.搭建自己的神经网络,并创建模型对象
class MyModel(torch.nn.Module):
    #初始化函数,根据参数,定义神经网络的组件
    def __init__(self, in_features, out_features):
        super(MyModel, self).__init__()
        self.linear = torch.nn.Linear(in_features, out_features)
    #前向传播,构建计算图
    def forward(self, x):
        out = self.linear(x)
        return out
input_len = 1
output_len = 1
model = MyModel(input_len, output_len)
#2.定义损失(代价)函数 loss
lossFun = torch.nn.MSELoss()
#3.定义优化器 optimizer,利用其管理模型参数
optimizer = torch.optim.SGD(model.parameters(),lr=0.1)
#4.模型训练
epoches = 100
for epoch in range(epoches):
    #4.1 通过 model(Xs) 调用 forward,前向传播,构建计算图
    pred = model(Xs)
    #4.2 利用 loss(Ys, pred)计算模型的损失
    loss = lossFun(Ys,pred)
    #4.3 利用 loss.backward() 进行后向传播,计算模型的梯度
    loss.backward()
    #4.4 利用 optimizer.step() 更新权重
    optimizer.step()
    #4.5 利用 optimizer.zero_grad() 清空梯度
    optimizer.zero_grad()
    #打印结果
    if epoch%10 == 0:
        w,b = model.parameters()
        print(f"loss={loss.item():.8f},w={w.item():.4f},b = {b.item()}")
```

程序运行结果如图 3.25 所示。

在代码示例 3-21 中，所设置的超参数 epoches＝100，lr＝0.1，显然不够合理。此外，由于样本数据中存在 20%的噪声，这也导致了最终的损失值(loss)较高。读者可以根据所学

```
…  loss=2098.71704102,w=8.3272,b = -1.5836896896362305
   loss=411.81124878,w=35.8880,b = -2.5938704013824463
   loss=399.67202759,w=38.2142,b = -1.7533917427062988
   loss=399.52005005,w=38.4426,b = -1.5776232481002808
   loss=399.51748657,w=38.4682,b = -1.5498062372207642
   loss=399.51745605,w=38.4714,b = -1.545767068862915
   loss=399.51742554,w=38.4718,b = -1.5452003479003906
   loss=399.51742554,w=38.4718,b = -1.5451213121414185
   loss=399.51742554,w=38.4718,b = -1.5451111793518066
   loss=399.51742554,w=38.4718,b = -1.5451104640960693
```

图 3.25　程序运行结果 1

知识,调整超参数设置,以优化训练过程并降低损失值。

当训练完成后,可以利用模型的预测结果绘制回归直线,以直观展示线性回归模型的拟合效果,代码如下。

```
1   #代码示例 3-22  使用模型进行预测
2   with torch.no_grad():
3       predicted = model(Xs).numpy()
4   #打印样本点和预测曲线,进行比较
5   plt.plot(X, Y, "ro")              #红色样本点
6   plt.plot(X, predicted,"b")        #蓝色样本点连成一条直线
7   plt.show()
```

上述代码中的第 2 行 with torch.no_grad(),使用上下文管理器 with 禁用梯度计算。该功能在模型预测阶段非常实用且重要,这是因为在模型预测过程中无须进行“后向传播”,自然就不需要计算梯度。禁用梯度计算可以有效提高计算效率和减少内存消耗。运行结果如图 3.26 所示。

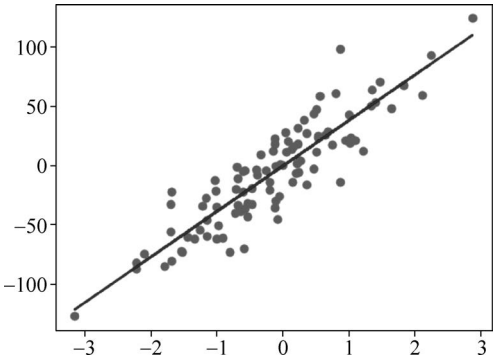


图 3.26　程序运行结果 2

3.3　学习案例 2:乳腺癌预测

在机器学习中,回归和分类是两种常见的任务类型。通常情况下,回归的目标是通过输入数据预测一个连续值,如房价、温度或股票价格等。而分类的目标则是将输入数据分配到预定义的类别中,例如,判断邮件是否为垃圾邮件,或识别图片中的物体类别。本节将通过一个基于多特征数据的二分类案例,详细讲解二分类模型的实现过程。本节重点学习的内容包括:

- 如何处理外部数据文件。
- 数据集划分与标准化。
- 构建基于多特征的二分类神经网络。


视频讲解

3.3.1　数据处理

1. 读取数据文件

在机器学习和数据分析领域，公共数据集或专业领域的数据文件通常以 .csv 文件格式进行存储(逗号分隔值文件)。对于.csv 文件，通常使用 pandas 库来进行读取和处理。下面给出一个读取 .csv 文件的代码示例。

```
1   #代码示例 3-23
2   import pandas as pd
3   #预测乳腺癌样本数据文件为 CSV 格式,使用 pandas 对数据集合进行加载
4   df = pd.read_csv("./data/breast_cancer.csv")
5   #打印读取的数据对象 df
6   print(df)
```

上述代码运行后，pandas 对象 df 中将包含该.csv 文件的全部内容，部分打印结果如图 3.27 和图 3.28 所示。

```
…       mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0             17.99         10.38          122.80     1001.0          0.11840
1             20.57         17.77          132.90     1326.0          0.08474
2             19.69         21.25          130.00     1203.0          0.10960
3             11.42         20.38           77.58      386.1          0.14250
4             20.29         14.34          135.10     1297.0          0.10030
..              ...           ...             ...        ...              ...
564           21.56         22.39          142.00     1479.0          0.11100
565           20.13         28.25          131.20     1261.0          0.09780
566           16.60         28.08          108.30      858.1          0.08455
567           20.60         29.33          140.10     1265.0          0.11780
568            7.76         24.54           47.92      181.0          0.05263
```

图 3.27　乳腺癌部分特征数据

```
    worst concave points  worst symmetry  worst fractal dimension  target
0                 0.2654          0.4601                  0.11890       0
1                 0.1860          0.2750                  0.08902       0
2                 0.2430          0.3613                  0.08758       0
3                 0.2575          0.6638                  0.17300       0
4                 0.1625          0.2364                  0.07678       0
..                   ...             ...                      ...     ...
564               0.2216          0.2060                  0.07115       0
565               0.1628          0.2572                  0.06637       0
566               0.1418          0.2218                  0.07820       0
567               0.2650          0.4087                  0.12400       0
568               0.0000          0.2871                  0.07039       1

[569 rows x 31 columns]
```

图 3.28　乳腺癌部分特征数据和标记数据

从图 3.27 和图 3.28 可知，该数据集合中共有 569 条数据，每条数据有 30 个和乳腺癌相关的病变特征，最后一列 target 是这 30 个病变特征的患者是否患有乳腺癌的诊断结果，target 列值为 0 表示健康，值为 1 表示患乳腺癌。

此外,还可以通过打印该 df 对象的 shape,以及 df 所包含的列对象 columns 和行对象 index,观察其结果,以便后续进行数据处理,如图 3.29 所示。

```
print(df.shape)
print(df.columns)
print(df.index)
```

✓ 0.0s

```
(569, 31)
Index(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
       'mean smoothness', 'mean compactness', 'mean concavity',
       'mean concave points', 'mean symmetry', 'mean fractal dimension',
       'radius error', 'texture error', 'perimeter error', 'area error',
       'smoothness error', 'compactness error', 'concavity error',
       'concave points error', 'symmetry error', 'fractal dimension error',
       'worst radius', 'worst texture', 'worst perimeter', 'worst area',
       'worst smoothness', 'worst compactness', 'worst concavity',
       'worst concave points', 'worst symmetry', 'worst fractal dimension',
       'target'],
      dtype='object')
RangeIndex(start=0, stop=569, step=1)
```

图 3.29 数据集对象 df 的属性

如图 3.29 所示,观察 df 对象的属性:

- 其形状 df.shape 为 569 行,31 列。
- 列对象 df.columns 包含 31 列,其中最后一列为 target。
- 行索引 df.index 的范围为 0～569(不包括 569),步长为 1。

通过了解其结构,可利用 pandas 的切片功能,将表中的特征数据和标记分开,分别获得数据特征集合 X 和标记集合 Y,代码如下。

```
#代码示例 3-24
#df 每行数据有 30 个乳腺病理特征, 最后一列表示是否患有乳腺癌
#X 赋值特征数据
X = df[df.columns[0:-1]].values
#Y 赋值标记数据
Y = df[df.columns[-1]].values
print(X.shape, Y.shape)
```

上述代码的运行结果为(569,30)(569,)。说明:X 为二维 NumPy 数组,Y 为一维 NumPy 数组。在后续处理过程中,需要注意二者数据维度的不同。

2. 构建数据集

在前述章节中,在评估房价预测模型时,使用了全部的 8 个样本数据,因此训练数据与测试数据完全相同,这种做法其实存在很严重的问题。设想一下,如果考试题目与平时的学习内容完全一致,那么很难准确评估学生的真实学习效果。为了有效检验学生的真实水平,教师会设计与平时练习完全不同的考试内容。只要确保这些测试题不会在平时的练习过程中出现,就能保证测试题能够准确反映学生的实际学习水平。

这一原则在预测模型的评估中同样适用。为确保预测模型评估的有效性,在机器学习任务中,通常会将数据集划分为“训练集”和“测试集”两部分,这样可以使测试数据与训练数据相互独立,从而更真实地反映模型的泛化能力。

通常,可利用 sklearn 库中的 train_test_split 方法,将原数据按比例随机分为训练集和

测试集。代码如下。

```
#代码示例 3-25　将数据集按比例随机分为训练集和测试集
from sklearn.model_selection import train_test_split
#按照 0.8 和 0.2 的比例随机划分数据集合
X_train, X_test, Y_train, Y_test = train_test_split(X,Y,test_size=0.2,random_state=1234)
print(f'X_train.shape={X_train.shape}')
print(f'Y_train.shape={Y_train.shape}')
print(f'X_test.shape={X_test.shape}')
print(f'Y_test.shape={Y_test.shape}')
```

在机器学习中,数据集的划分比例取决于具体任务和数据量。“训练集”和“测试集”的划分比例通常采用 70%∶30% 或 80%∶20%,前者适用于数据量较大的情况,后者适用于对模型泛化能力要求较高的情况。

在代码示例 3-25 中的第 4 行,train_test_split 方法的 test_size 参数值设置为 0.2,将会选取 20% 的数据作为测试集。最终结果分别打印出训练集和测试集的特征数据 X 和标记 Y 的 shape,如图 3.30 所示。

```
X_train.shape=(455, 30)
Y_train.shape=(455,)
X_test.shape=(114, 30)
Y_test.shape=(114,)
```

图 3.30　训练集和测试集的 shape 值

根据图 3.30 可知,共有 569 个数据样本,其中,训练集数据 455 条、测试集数据 114 条。划分比例采用 80%∶20%。

3. 数据预处理

在机器学习中,为加速模型的训练过程,通常需要对原始数据进行“标准化”处理。标准化的目的是将数据转换为均值为 0、标准差为 1 的标准正态分布。原始数据标准化处理后,数据中的各个特征值会被缩放到一个统一的尺度,确保不同特征对模型训练的影响程度一致。这在使用像梯度下降这样的优化算法时尤为重要,因为标准化后的数据能够使得梯度计算更加平稳,从而加快模型的收敛速度。

接下来,将通过一个例子来演示如何使用 sklearn.preprocessing 中的 StandardScaler 类对数据进行标准化处理。具体代码如下。

```
#代码示例 3-26　对数据集进行标准化
from sklearn.preprocessing import StandardScaler
#创建标准化对象
sc = StandardScaler()
#对特征进行标准化
X_train_sta = sc.fit_transform(X_train)
X_test_sta = sc.fit_transform(X_test)
print(X_train[0, :5])
print(X_train_sta[0, :5])
```

在上述代码中的第 6、7 行,使用了 sc.fit_transform()方法对输入数据进行标准化处理。该方法由标准化对象 sc(StandardScaler 的实例对象)完成,主要包含以下两个步骤。

- 计算数据的统计量:用 fit()方法计算输入数据的统计量,包括均值和标准差。这些统计量将用于后续的标准化过程。
- 数据转换:由 transform()方法使用之前计算得到的均值和标准差,按式(3-19)对数

据进行标准化处理。

$$z=\frac{X-\mu}{\sigma} \tag{3-19}$$

式中：X——原始数据；

μ——数据特征的均值；

σ——数据特征的标准差。

为观察标准化后的效果，第 8、9 行分别打印了训练集在标准化前(X_train)和标准化后(X_train_sta)的第 0 行的前 5 个值，结果如表 3.2 所示。

表 3.2　数据样本中第 0 行前 5 个值在标准化前和标准化后的比较

训练集	第 0 列	第 1 列	第 2 列	第 3 列	第 4 列	…
X_train	1.288e+01	1.822e+01	8.445e+01	4.931e+02	1.218e−01	…
X_train_sta	−0.361 808 27	−0.265 210 11	−0.317 157 02	−0.467 138 41	1.803 826 09	…

在模型训练前，需要将输入数据 X 和标记 Y 封装到张量对象 Xs 和 Ys 中。此外，由于 Xs 和 Ys 的维度不同，为避免后续处理出现问题，还须对训练集中的标记 Ys_train 和测试集中的标记 Ys_test 进行维度对齐处理。代码如下。

```
#代码示例 3-27　将数据封装到 PyTorch 张量中
import torch
import numpy as np
#将 NumPy 数据封装到张量对象中
Xs_train = torch.from_numpy(X_train_sta.astype(np.float32))
Ys_train = torch.from_numpy(Y_train.astype(np.float32))
Xs_test = torch.from_numpy(X_test_sta.astype(np.float32))
Ys_test = torch.from_numpy(Y_test.astype(np.float32))
#将标记集合 Ys_train 和 Ys_test 转成二维
Ys_train = Ys_train.view(-1, 1)
Ys_test = Ys_test.view(-1, 1)
print(f'Xs_train.shape={Xs_train.shape}')
print(f'Ys_train.shape={Ys_train.shape}')
print(f'Xs_test.shape={Xs_test.shape}')
print(f'Ys_test.shape={Ys_test.shape}')
```

上述代码中，第 10、11 行中的.view(−1, 1)执行了维度变换，将 Ys 的维度对齐到 Xs，将一维的 Ys_train 和 Ys_test 转成二维的张量类型。最后，分别打印 Xs 和 Ys 的训练集和测试集，结果如图 3.31 所示。

```
…
Xs_train.shape=torch.Size([455, 30])
Ys_train.shape=torch.Size([455, 1])
Xs_test.shape=torch.Size([114, 30])
Ys_test.shape=torch.Size([114, 1])
```

图 3.31　预处理后的训练集和测试集的 shape 值

根据图 3.31 中 Ys_train 和 Ys_test 的 Shape 值，对比图 3.30，可知其已经从一维张量转换为二维张量。

视频讲解

3.3.2　神经网络类定义

采用线性模型 $f(x)=wx+b$ 来预测乳腺癌，通常会有两种预测结果：患病或健康(用 1 表示患病，0 表示健康)，则该预测模型属于二分类模型。

1. 二分类模型

二分类模型是一种用于解决二元分类问题的机器学习模型，其目标是将输入样本划分为两个互斥的类别，如“正例”和“反例”或“是”和“否”，该模型广泛应用于垃圾邮件检测、疾病诊断、情感分析和信用风险评估等任务。

每个样本 $\boldsymbol{x}$（特征向量）具有 30 个特征维，$f(\boldsymbol{x})=1$ 为“患病”，$f(\boldsymbol{x})=0$ 为“健康”，那么应该有 $f(\boldsymbol{x}) \in [0,1]$。但实际上，线性模型 $f(\boldsymbol{x})$ 的输出值范围通常为 $(-\infty, +\infty)$，那么怎样才能使 $f(\boldsymbol{x})$ 输出值落在 $[0,1]$ 区间呢？

2. 激活函数

Sigmoid 函数是一种常用的激活函数，特别适用于二分类任务。其作用是将输入的任意实数值映射到 $[0, 1]$ 区间，从而将模型输出转换为概率形式，便于解释为属于某一类别的可能性。具体而言，Sigmoid 函数的数学定义如式(3-20)所示。

$$\sigma(z)=\frac{1}{1+\mathrm{e}^{-z}} \tag{3-20}$$

式中：z——输入值；

e——自然常数。

Sigmoid 函数的输出图像如图 3.32 所示。

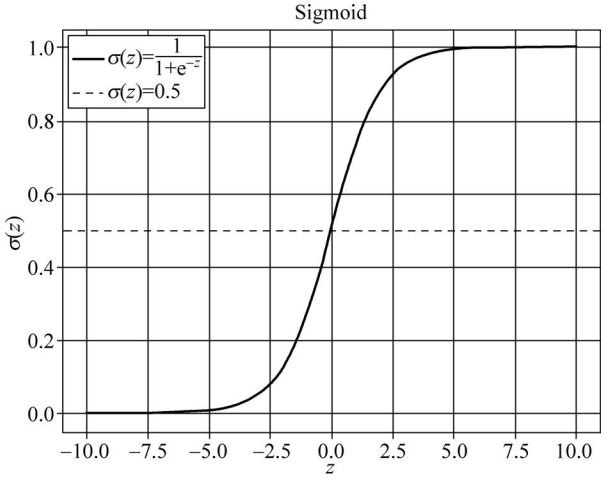


图 3.32　Sigmoid 函数的输出图像

注意，式(3-20)的输出总是一个 0～1 的实数。因此，通过 Sigmoid 函数输出的值可以理解为“属于某个类别的概率”。

通常情况下，Sigmoid 函数输出结果会与预设的阈值（如 0.5）进行比较，以完成分类决策。当输出值大于阈值时，模型预测该样本属于类别 1；反之，预测该样本属于类别 0。该过程可以使用式(3-21)表示如下。

$$\hat{y}=\begin{cases}1, & \sigma(z)>0.5\\ 0, & \sigma(z)\leqslant 0.5\end{cases} \tag{3-21}$$

3. 损失函数

在二分类预测模型中,常用的损失函数是二元交叉熵损失(Binary Cross-Entropy Loss,BCE),该函数用于度量模型预测的概率分布与真实标记之间的差异,其具体形式如下。

$$\text{Loss}(y,\hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i\log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)] \tag{3-22}$$

式中:y_i——第 i 个标记值(真实的观测输出值);

$\hat{y}_i$——第 i 个模型的预测值。

BCE 在分类问题中能够处理连续的概率输出,从而避免极端的梯度消失问题。此外,由于使用了对数函数,损失函数能对那些极度偏离真实值的预测施加更大的惩罚,从而引导模型快速学习。

在 PyTorch 中,BCE 通常可通过 torch.nn.BCELoss 或 torch.nn.BCEWithLogitsLoss 来实现。其中,BCEWithLogitsLoss 将 Sigmoid 激活函数与二元交叉熵损失结合在一起,为模型训练提供了一种更加高效的方式。

4. 定义神经网络类

乳腺癌预测神经网络类定义如下。

```
#代码示例 3-28  自定义乳腺癌预测模型 Breast Cancer Prediction Model,简称为 BCPM
class BCPM(torch.nn.Module):
    #定义构造(初始化)函数
    def __init__(self, in_features):
        super(BCPM, self).__init__()       #调用父类的构造函数
        #1.构建线性层
        self.linear = torch.nn.Linear(in_features, 1)
        #2.构建激活函数层
        self.sigmoid = torch.nn.Sigmoid()
    #重写了父类的 forward 函数,前向传播
    def forward(self, x):
        pred = self.linear(x)
        out = self.sigmoid(pred)
        return out
```

神经网络类 BCPM 定义完成后,还须进行:

- 确定损失函数和学习率。
- 创建神经网络类的模型对象。
- 构建优化器对象,并为优化器指定模型参数和学习率。

代码如下。

```
#代码示例 3-29
#损失函数公式定义
lossFun = torch.nn.BCELoss()
#学习率,迭代次数
learning_rate = 0.1
num_epochs = 100
#获取样本量和特征数,创建模型
```

```
8   n_samples, n_features = X.shape
9   model = BCPM(n_features)
10  #创建优化器
11  optimizer = torch.optim.SGD(model.parameters(), lr = learning_rate)
12  #打印模型、打印模型参数
13  print(model)
14  i = 0
15  for name, param in model.named_parameters():
16      i += 1
17      print(f"第{i}个参数: ")
18      print(f"Name: {name}, \nShape: {param.shape}, \nValues: {param.data}")
```

上述代码中，创建乳腺癌预测模型 model 后，第 13 行打印 model 的模型结构，第 17、18 行打印 model 的参数情况，结果如图 3.33 所示。

```
BCPM(
  (linear): Linear(in_features=30, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
第1个参数:
Name: linear.weight,
Shape: torch.Size([1, 30]),
Values: tensor([[ 0.1074,  0.0854,  0.0839,  0.0936,  0.1224,  0.1664,  0.1825,  0.0812,
          0.0931,  0.1522, -0.0081, -0.1243,  0.1811, -0.0664,  0.1301, -0.0117,
          0.0620,  0.0914,  0.0651, -0.1162, -0.1087,  0.1810,  0.1582,  0.0808,
          0.0810, -0.1596,  0.0174,  0.0404, -0.0037, -0.0838]])
第2个参数:
Name: linear.bias,
Shape: torch.Size([1]),
Values: tensor([-0.0277])
```

图 3.33　BCPM 类对象的结构和模型参数情况


视频讲解

3.3.3　模型训练与预测

在定义损失函数和优化器后，预测模型的训练过程通常包括以下步骤。

- 前向传播：将输入数据传入模型，通过前向传播计算得到预测结果。
- 损失计算：利用预测结果和真实值(标记)计算损失。
- 反向传播：通过反向传播算法，计算模型参数的梯度。
- 参数更新：根据梯度值，由优化器对模型参数进行优化。
- 梯度清零：清除优化器中的梯度信息，以避免累积影响后续计算。
- 循环迭代：重复上述操作，直到达到设定的训练次数，或者当损失值低于预设阈值时终止训练。

模型训练过程如下。

```
#代码示例 3-30　模型训练
#当 loss 小于该阈值 threshold_value 时，停止训练
threshold_value = 0.00001
#迭代训练
for epoch in range(num_epochs):
    #1.通过模型的前向传播，调用 forward()方法，得到预测结果
    ys_pred = model(Xs_train)
    #2.根据预测结果和真实标记值计算损失(标量值)
    ls = lossFun(ys_pred, Ys_train)
    #3.通过后向传播，获取模型参数的梯度
```

```
    ls.backward()
    #4.根据计算获得的梯度,更新模型参数的权重
    optimizer.step()
    #5.进行梯度的清空
    optimizer.zero_grad()
    #打印训练中的 loss 变化情况
    if epoch % 5 == 0:
        print(f"epoch:{epoch},loss={ls.item():.4f}")
    #6.循环上面的操作,直到预定训练次数,或者损失小于阈值 threshold_value 为止
    if ls.item() <= threshold_value:
        break
print("模型训练完成! loss={0}".format(ls))
```

上述代码执行结果如图 3.34 所示。

```
…  epoch:0,loss=1.0203
   epoch:5,loss=0.3734
   epoch:10,loss=0.2646
   epoch:15,loss=0.2153
   epoch:20,loss=0.1859
   epoch:25,loss=0.1660
   epoch:30,loss=0.1516
   epoch:35,loss=0.1407
   epoch:40,loss=0.1320
   epoch:45,loss=0.1250
   epoch:50,loss=0.1192
   epoch:55,loss=0.1142
   epoch:60,loss=0.1100
   epoch:65,loss=0.1062
   epoch:70,loss=0.1030
   epoch:75,loss=0.1000
   epoch:80,loss=0.0974
   epoch:85,loss=0.0950
   epoch:90,loss=0.0929
   epoch:95,loss=0.0909
   模型训练完成! loss=0.08940503746271133
```

图 3.34 BCPM 预测模型的训练过程

如图 3.34 所示,训练开始后,损失函数值(loss)呈现持续下降的趋势,直至迭代结束,最终 loss=0.0894。如果希望获得的 loss 足够小,一定要低于设定阈值(threshold_value),那么训练过程可能需要消耗更多的时间和计算资源,这可能是得不偿失的。通常来说,在满足模型预测性能要求(正确率超过预期)的前提下,并非必须追求最低的损失值。在实际应用中,应综合考虑训练中要消耗的计算资源以及需求满足程度,以平衡模型性能与训练效率。

下面使用测试集(完全区别于训练数据的另外的数据集)来预测诊断结果,并计算模型预测的准确率。代码如下。

```
#代码示例 3-31  预测过程中无须后向传播,因此无须计算梯度
with torch.no_grad():
    ys_pred = model(Xs_test)
    #计算出来的结果是 0~1 的数,将数据进行四舍五入,得到 0 或 1
    ys_pred_cls = ys_pred.round()
    #统计结果
    acc = ys_pred_cls.eq(Ys_test).sum().numpy() / float(Ys_test.shape[0])
    print(f"准确率:{acc.item():.4f}")
```

上述代码中的第 2 行,with torch.no_grad()是 PyTorch 中常用的一段上下文管理器代

码，它的作用是临时关闭自动求导机制。这意味着，在 with torch. no_grad()语句块中的所有计算，都不会构建计算图，自然也不会计算和存储梯度。这不仅可以节省内存和加快计算速度，还能避免在训练过程中不小心进行了不必要的梯度计算，导致错误的参数更新。

最终，程序输出的正确率为 0.9479，虽然正确率超过 90%，说明当前模型可以很好地进行乳腺癌的诊断。但模型的正确率仍然有提高空间，读者可以根据已经学习的知识，对上述代码进行优化，以获得更高的模型正确率。

习题

一、选择题

1. 正确定义神经网络模型参数 p 的选项为(　　)。
   A. p = torch. randn(3, 5, requoires_grad=True)
   B. p = torch. randn(3, 5, requires_grad=False)
   C. p = torch. randn(3, 5)
2. 神经网络是由(　　)个神经元按照一定的规则连接在一起形成的。
   A. 1　　B. 2　　C. 多
3. 二分类问题对应的交叉熵函数，在 PyTorch 中的实现为(　　)。
   A. torch. nn. CrossEntropyLoss ()　　B. torch. nn. BCELoss()
   C. torch. nn. MSELoss()　　D. torch. nn. KLDivLoss ()

二、简答题

1. 简述 PyTorch 中有哪些常用的优化器。
2. 在定义损失函数和优化器后，对预测模型的训练包括哪些步骤？

三、编程练习题

1. 针对代码示例 3-30，修改预测模型训练过程中的超参数(调整训练次数、学习率、阈值 threshold_value 等)，重新训练模型并测试，以得到更高的模型正确率。

2. 修改代码示例 3-24 的代码，随机选取 30 个特征中的 20 个特征作为训练数据，分析数据特征数减少的情况下，模型预测的准确性是降低还是升高？请根据实验结果进行分析，并讨论训练数据特征数量对模型训练和预测的影响。