第 3 章　单层感知器与线性神经网络

本章将要学习的主要内容是神经网络算法的基础——单层感知器和人工神经网络（ANN）。实际上，单层感知器、人工神经网络的设计是从生物体的神经网络结构获得灵感的。模仿生物神经网络，我们构造出了单层感知器，在单层感知器的基础上经过不断的优化才得到了后来的神经网络算法。

3.1　生物神经网络

生物神经网络一般是指由生物的大脑神经元和细胞等组成的网络，用于产生生物的意识，帮助生物进行思考和行动。

神经细胞是构成神经系统的基本单元，简称为神经元。如图 3.1 所示，神经元主要由 3 部分构成：细胞体；轴突；树突。

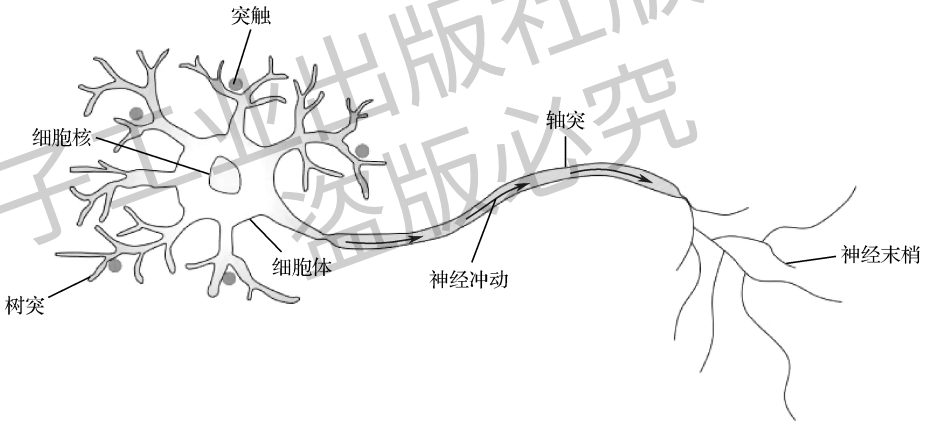


图 3.1　神经元的结构

每个神经元伸出的突起分两种，即树突和轴突。树突分支比较多，每个分支还可以再分支，长度一般比较短，作用是接收信号；轴突只有一个，从细胞体的一个凸出部分伸出，长度一般比较长，作用是把从树突和细胞表面传入细胞体的神经信号传出到其他神经元。轴突的末端分为许多小支，连接到其他神经元的树突上。

大脑可视作由 1000 多亿神经元组成的神经网络。神经元的信息传递和处理是一种电化学活动。树突由于电化学作用接受外界的刺激，通过胞体内的活动体现为轴突电位。当轴突电位达到一定的值时，则形成神经脉冲或动作电位，然后通过轴突末梢传递给其他的神经元。从控制论的观点来看，这一过程可以看作一个多输入单输出非线性系统的动态过程。

3.2　单层感知器

3.2.1　单层感知器介绍

受到生物神经网络的启发，计算机学家弗兰克 • 罗森布拉特（Frank Rosenblatt）在 20 世纪 60 年代提出了一种模拟生物神经网络的的人工神经网络结构，称为感知器（Perceptron）。图 3.2 为单层感知器的结构。

图 3.2 中，x_1、x_2 和 x_3 为输入信号，类似于生物神经网络中的树突；w_1、w_2 和 w_3 分别为 x_1、x_2 和 x_3 的权值，它可以调节输入信号的大小，让输入信号变大（w>0）、不变（w=0）或者减小（w<0）。可以理解为生物神经网络中的信号作用，信号经过树突传递到细胞核的过程中信号会发生变化。

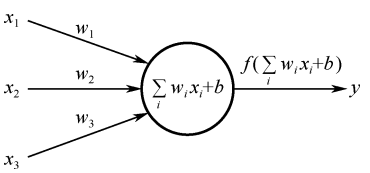


图 3.2　单层感知器的结构

公式 $\sum_i (w_i x_i)+b$ 表示细胞的输入信号在细胞核的位置进行汇总 $\sum_i w_i x_i$，然后再加上该细胞本身自带的信号 b。b 一般称为**偏置值（Bias）**，相当于是神经元内部自带的信号。

$f(x)$称为激活函数，可以理解为信号在轴突上进行的线性或非线性变化。在单层感知器中最开始使用的激活函数是 sign(x)激活函数。该函数的特点是当 x>0 时，输出值为 1；当 x=0 时，输出值为 0；当 x<0 时，输出值为−1。sign(x)函数如图 3.3 所示。

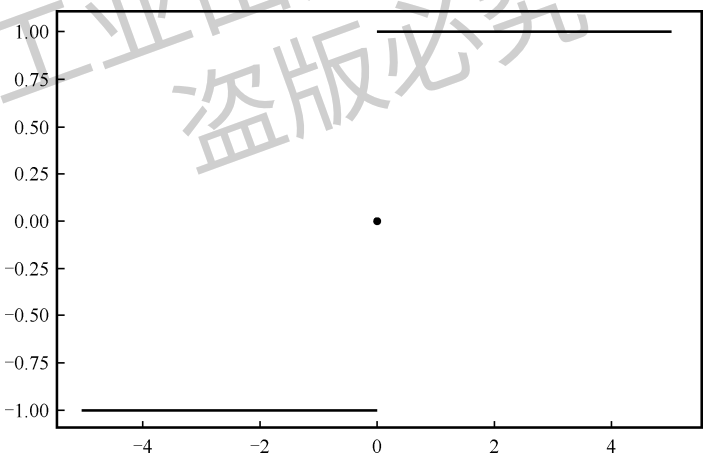


图 3.3　sign(x)函数

y 就是 $f\left(\sum_i (w_i x_i)+b\right)$，为单层感知器的输出结果。

3.2.2　单层感知器计算举例

假如一个单层感知器有 3 个输入：x_1、x_2 和 x_3，同时已知 b=−0.6，$w_1=w_2=w_3$=0.5，那么根据单层感知器的计算公式 $f\left(\sum_i (w_i x_i)+b\right)$ 我们就可以得到如图 3.4 所示的计算结果。

x_1	x_2	x_3	y
0	0	0	−1
0	0	1	−1
0	1	0	−1
0	1	1	1
1	0	0	−1
1	0	1	1
1	1	0	1
1	1	1	1

图 3.4　单层感知器的计算

$x_1=0, x_2=0, x_3=0$：　$\text{sign}(0.5\times0+0.5\times0+0.5\times0-0.6)=-1$
$x_1=0, x_2=0, x_3=1$：　$\text{sign}(0.5\times0+0.5\times0+0.5\times1-0.6)=-1$
$x_1=0, x_2=1, x_3=0$：　$\text{sign}(0.5\times0+0.5\times1+0.5\times0-0.6)=-1$
$x_1=0, x_2=1, x_3=1$：　$\text{sign}(0.5\times0+0.5\times1+0.5\times1-0.6)=1$
$x_1=1, x_2=0, x_3=0$：　$\text{sign}(0.5\times1+0.5\times0+0.5\times0-0.6)=-1$
$x_1=1, x_2=0, x_3=1$：　$\text{sign}(0.5\times1+0.5\times0+0.5\times1-0.6)=1$
$x_1=1, x_2=1, x_3=0$：　$\text{sign}(0.5\times1+0.5\times1+0.5\times0-0.6)=1$
$x_1=1, x_2=1, x_3=1$：　$\text{sign}(0.5\times1+0.5\times1+0.5\times1-0.6)=1$

3.2.3　单层感知器的另一种表达形式

单层感知器的另一种表达形式如图 3.5 所示。

其实这种表达形式跟 3.2.1 小节中的单层感知器是一样的，只不过是把偏置值 b 变成了输入 $w_0\times x_0$，其中 $x_0=1$。所以 $w_0\times x_0$ 实际上就是 w_0，把 $\sum_i(w_ix_i)$ 公式展开得到 $w_1\times x_1+w_2\times x_2+w_3\times x_3+w_0$。所以这两个单层感知器的表达不一样，但是计算结果是一样的。图 3.5 所示的表达形式更加简洁，更适合使用矩阵来进行运算。

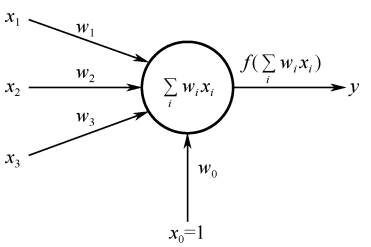


图 3.5　单层感知器的另一种表达形式

3.3　单层感知器的学习规则

3.3.1　单层感知器的学习规则介绍

感知器的学习规则就是指感知器中的权值参数训练的方法，本小节我们暂时先不解释这个学习规则是怎么推导出来的，等第 4 章我们讲到 Delta 学习规则的时候再来解释感知器的学习规则是如何推导的。这里我们可以先接受下面的公式即可。

在 3.2.3 小节中我们已知单层感知器的表达式可以写成

$$y = f\left(\sum_{i=0}^{n}(w_i x_i)\right) \tag{3.1}$$

式（3.1）中：y 表示感知器的输出；f 是 sign 激活函数；n 是输入信号的个数。

$$\Delta w_i = \eta(t-y)x_i \tag{3.2}$$

式（3.2）中，Δw_i 表示第 i 个权值的变化；η 表示**学习率（Learning Rate）**，用来调节权值变化的大小；t 是正确的标签（Target）。

因为单层感知器的激活函数为 sign 函数，所以 t 和 y 的取值都为±1。

$t=y$ 时，Δw_i 为 0；$t=1$，$y=-1$ 时，Δw_i 为 $2x_i\eta$；$t=-1$，$y=1$ 时，Δw_i 为 $-2x_i\eta$。由式（3.2）可以推出：

$$\Delta w_i = \pm 2\eta x_i \tag{3.3}$$

权值的调整公式为

$$w_i := w_i + \Delta w_i \tag{3.4}$$

3.3.2　单层感知器的学习规则计算举例

假设有一个单层感知器如图 3.2 所示，输入 $x_0=1$、$x_1=0$ 和 $x_2=-1$，权值 $w_0=-5$、$w_1=0$ 和 $w_2=0$，学习率 $\eta=1$，正确的标签 $t=1$（注意，在这个例子中，偏置值 b 用 $w_0 \times x_0$ 来表示，x_0 的值固定为 1）。

Step1：计算感知器的输出。

$$\begin{aligned} y &= f\left(\sum_{i=0}^{n}(w_i x_i)\right) \\ &= \text{sign}(-5\times 1+0\times 0+0\times(-1)+0) \\ &= \text{sign}(-5) \\ &= -1 \end{aligned}$$

由于 $y=-1$ 与正确的标签 $t=1$ 不相同，所以需要对感知器中的权值进行调节。

$$\begin{aligned} \Delta w_0 &= \eta(t-y)x_0 = 1\times(1+1)\times 1 = 2 \\ \Delta w_1 &= \eta(t-y)x_1 = 1\times(1+1)\times 0 = 0 \\ \Delta w_2 &= \eta(t-y)x_2 = 1\times(1+1)\times(-1) = -2 \\ w_0 &:= w_0 + \Delta w_0 = -5+2 = -3 \\ w_1 &:= w_1 + \Delta w_1 = 0+0 = 0 \\ w_2 &:= w_2 + \Delta w_2 = 0-2 = -2 \end{aligned}$$

Step2：重新计算感知器的输出。

$$\begin{aligned} y &= f\left(\sum_{i=0}^{n}(w_i x_i)\right) \\ &= \text{sign}(-3\times 1+0\times 0+(-2)\times(-1)+0) \\ &= \text{sign}(-1) \\ &= -1 \end{aligned}$$

由于 $y=-1$ 与正确的标签 $t=1$ 不相同，所以需要对感知器中的权值进行调节。

$$\begin{aligned} \Delta w_0 &= \eta(t-y)x_0 = 1\times(1+1)\times 1 = 2 \\ \Delta w_1 &= \eta(t-y)x_1 = 1\times(1+1)\times 0 = 0 \end{aligned}$$

$$\Delta w_2 = \eta(t-y)x_2 = 1\times(1+1)\times(-1) = -2$$
$$w_0 := w_0 + \Delta w_0 = -3+2 = -1$$
$$w_1 := w_1 + \Delta w_1 = 0+0 = 0$$
$$w_2 := w_2 + \Delta w_2 = -2-2 = -4$$

Step3：重新计算感知器的输出。

$$\begin{aligned} y &= f\left(\sum_{i=0}^{n}(w_i x_i)\right) \\ &= \text{sign}(-1\times1+0\times0+(-4)\times(-1)+0) \\ &= \text{sign}(3) \\ &= 1 \end{aligned}$$

由于 y=1 与正确的标签 t=1 相同，说明感知器经过训练后得到了我们想要的结果，这样我们就可以结束训练了。

如果将上面的例子写成 Python 程序，则可以得到代码 3-1。

代码 3-1：单层感知器学习规则计算举例

```
# 导入 numpy 科学计算包
import numpy as np
# 定义输入
x0 = 1
x1 = 0
x2 = -1
# 定义权值
w0 = -5
w1 = 0
w2 = 0
# 定义正确的标签
t = 1
# 定义学习率 lr(learning rate)
lr = 1
# 定义偏置值
b = 0
# 循环一个比较大的次数，如 100
for i in range(100):
    # 打印权值
    print(w0,w1,w2)
    # 计算感知器的输出
    y = np.sign(w0 * x0 + w1 * x1 + w2*x2)
    # 如果感知器的输出不等于正确的标签
    if(y != t):
        # 更新权值
        w0 = w0 + lr * (t-y) * x0
        w1 = w1 + lr * (t-y) * x1
        w2 = w2 + lr * (t-y) * x2
    # 如果感知器的输出等于正确的标签
    else:
        # 训练结束
        print('done')
        # 退出循环
            break
```

运行结果如下：

```
-5 0 0
-3 0 -2
-1 0 -4
done
```

下面我们还可以用矩阵运算的方式来完成同样的计算，代码 3-2 为以矩阵运算的方式来进行单层感知器学习规则的计算。

代码 3-2：单层感知器学习规则计算举例（矩阵计算）

```
# 导入 numpy 科学计算包
import numpy as np
# 定义输入，用大写字母表示矩阵
# 一般我们习惯用一行表示一个数据，如果存在多个数据，则用多行来表示
X = np.array([[1,0,-1]])
# 定义权值，用大写字母表示矩阵
# 神经网络中权值的定义可以参考神经网络输入和输出神经元的个数
# 本例子中，输入神经元的个数为 3 个，输出神经元的个数为 1 个，所以可以定义 3 行 1 列的矩阵
W = np.array([[-5],
              [0],
              [0]])
# 定义正确的标签
t = 1
# 定义学习率 lr(learning rate)
lr = 1
# 定义偏置值
b = 0
# 循环一个比较大的次数，如 100
for i in range(100):
    # 打印权值
    print(W)
    # 计算感知器的输出，np.dot 可以看作矩阵乘法
    y = np.sign(np.dot(X,W))
    # 如果感知器的输出不等于正确的标签
    if(y != t):
        # 更新权值
        # X.T 表示 X 矩阵的转置
        # 这里一个步骤可以完成代码 3-1 中下面 3 行代码完成的事情：
        # w0 = w0 + lr * (t-y) * x0
        # w1 = w1 + lr * (t-y) * x1
        # w2 = w2 + lr * (t-y) * x2
        W = W + lr * (t - y) * X.T
    # 如果感知器的输出等于正确的标签
    else:
        # 训练结束
        print('done')
        # 退出循环
            break
```

运行结果如下：

```
  [[-5]
   [ 0]
   [ 0]]
 [[-3]
   [ 0]
   [-2]]
 [[-1]
   [ 0]
   [-4]]
done
```

3.4　学习率

学习率是人为设定的一个超参数，主要是在训练阶段用来控制模型参数调整的快慢。关于学习率，主要有 3 个要点需要注意：

（1）学习率 η 的取值范围一般为 0～1；

（2）学习率太大，容易造成权值调整不稳定；

（3）学习率太小，模型参数调整太慢，迭代次数太多。

你可以想象一下在洗热水澡的时候：如果每次调节的幅度很大，那水温不是太热，就是太冷，很难得到一个合适的水温；如果一开始的时候水很冷，每次调节的幅度都非常小，那么需要调节很多次，花很长时间才能得到一个合适的水温。学习率的调整也是这样一个道理。图 3.6 表示不同大小的学习率对模型训练的影响。

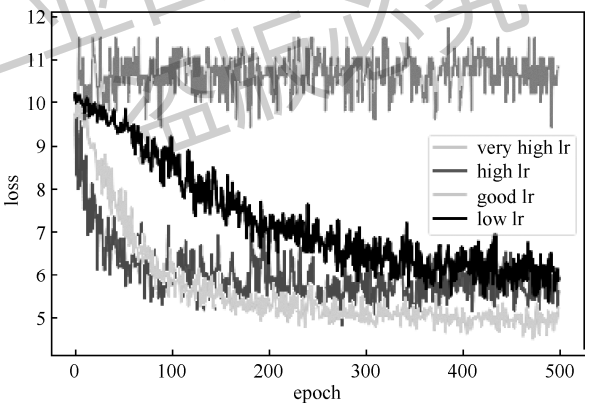


图 3.6　不同大小的学习率对模型训练的影响

图 3.6 中的纵坐标 **loss** 代表**代价函数（Loss Function）**，在后面的章节中有更详细的介绍，这里我们可以把它近似理解为模型的预测值与真实值之间的误差。我们训练模型的主要目的就是为了降低 loss 值，减少模型的预测值与真实值之间的误差。横坐标 **epoch** 代表模型的**迭代周期**，把所有的训练数据都训练一遍可以称为迭代了一个周期。

从图 3.6 中我们可以看到，如果使用非常大的学习率（very high lr）来训练模型，loss 会一直处于一个比较大的位置，模型不能收敛，这肯定不是我们想要的结果。如果使用比较大的学习率（high lr）来训练模型，loss 会下降很快，但是最终不能得到比较小的 loss，所以结果也不理想。如果使用比较小的学习率（low lr）来训练模型，模型收敛的速度会很

慢，模型需要等待很长时间才能收敛。最理想的结果是使用合适的学习率（good lr）来训练模型，使用合适的学习率，模型的 loss 会下降得比较快，并且最后的 loss 也能够下降到一个比较小的位置，结果最理想。

看到这里大家可能会有一个疑问，学习率的值到底取多少比较合适？这个问题其实是没有明确答案的，需要根据建模的经验和测试才能找到合适的学习率。但学习率的选择也有一些小的技巧（Trick）可以使用，比如说最开始我们设置一个学习率为 0.01，经过测试，我们发现学习率太小了，需要调大一点，那么我们可以改成 0.03。如果 0.03 还需要调大，我们可以调到 0.1。同理，如果 0.01 太大了，需要调小，那么我们可以调到 0.003。如果 0.003 还需要调小，我们可以调到 0.001。所以常用的学习率可以选择：

1，0.3，0.1，0.03，0.01，0.003，0.001，0.0003，0.0001 ...

当然，这也不是绝对的，其他的学习率的取值你也可以去尝试。

3.5　模型的收敛条件

通常模型的收敛条件可以有以下 3 个：

（1）loss 小于某个预先设定的较小的值；

（2）两次迭代之间权值的变化已经很小了；

（3）设定最大迭代次数，当迭代次数超过最大迭代次数时停止。

第 1 个条件很容易理解，模型的训练目的就是为了减小 loss，那么我们可以设定一个比较小的数值，每一次训练的时候，我们都同时计算一下 loss 的大小，当 loss 小于某个预先设定的阈值时，就可以认为模型收敛了，那么就可以结束训练。

第 2 个条件的意思是，每一次训练我们可以记录模型权值的变化，如果我们发现两次迭代之间模型的权值变化已经很小了，则说明模型已经几乎不需要做权值的调整了，那么就可以认为模型收敛，可以结束训练。

第 3 个条件是用得最多的方式。我们可以预先设定一个比较大的模型迭代周期，如迭代 100 次，或者 10000 次，或者 1000000 次等（需要根据实际情况来选择）。模型完成规定次数的训练之后，我们就可以认为模型训练完毕。如果达到我们设置的训练次数以后我们发现模型还没有训练好，那我们可以继续增加训练次数，让模型继续训练就可以了。

3.6　模型的超参数和参数的区别

模型的**超参数（Hyperparameters）**是机器学习或者深度学习中经常用到的一个概念，我们可以认为其是根据经验来人为设置的一些与模型相关的参数。比如说前面提到的学习率，学习率需要根据经验来人为设置。比如模型的迭代次数，也是需要在模型训练之前预先进行人为设置。

而前面提到的权值和偏置值则是**参数（Parameters）**，一般指的是模型中需要训练的变量。我们会给权值和偏置值进行初始化，随机赋值。模型在训练的过程中会不断调节这些参数，进行模型优化。

3.7　单层感知器分类案例

题目：假设我们有 4 个 2 维的数据，数据的特征分别是(3,3),(4,3),(1,1),(2,1)。(3,3)和(4,3)这两个数据的标签为 1，(1,1)和(2,1)这两个数据的标签为−1。构建神经网络来进行分类。

思路：我们要分类的数据是二维数据，所以只需要 2 个输入节点（一般输入数据有几个特征，我们就设置几个输入神经元），我们可以把神经元的偏置值也设置成一个输入节点，使用 3.2.3 小节中的方式，这样我们需要 3 个输入节点。

输入数据有 4 个：(1,3,3),(1,4,3),(1,1,1),(1,2,1)。

数据对应的标签：(1,1,−1,−1)。

初始化权值 w_1,w_2,w_3：取 0～1 的随机数。

学习率 lr（learning rate）：设置为 0.1。

激活函数：sign 函数。

我们可以构建一个如图 3.7 所示的单层感知器。

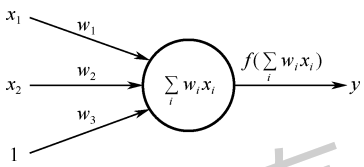


图 3.7　单层感知器

如代码 3-3 所示为单层感知器应用案例。

代码 3-3：单层感知器应用案例

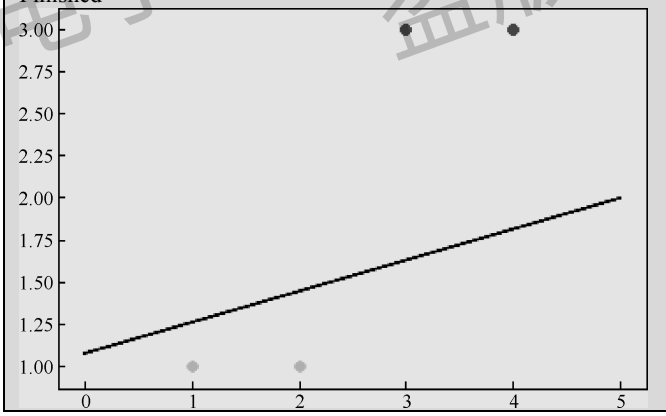
```
import numpy as np
import matplotlib.pyplot as plt
# 定义输入，我们习惯上用一行代表一个数据
X = np.array([[1,3,3],
              [1,4,3],
              [1,1,1],
              [1,2,1]])
# 定义标签，我们习惯上用一行表示一个数据的标签
T = np.array([[1],
              [1],
              [-1],
              [-1]])

# 权值初始化，3 行 1 列
# np.random.random 可以生成 0～1 的随机数
W = np.random.random([3,1])
# 学习率设置
lr = 0.1
# 神经网络的输出
Y = 0

# 更新一次权值
def train():
    # 使用全局变量 W
    global W
    # 同时计算 4 个数据的预测值
    # Y 的形状为(4,1)-4 行 1 列
    Y = np.sign(np.dot(X,W))
    # T - Y 得到标签值与预测值的误差 E，其形状为(4,1)
    E = T - Y
```

```
    # X.T 表示 X 的转置矩阵，形状为(3,4)
    # 我们一共有 4 个数据，每个数据 3 个值。定义第 i 个数据的第 j 个特征值为 xij
    # 如第 1 个数据的第 2 个特征值为 x12
    # X.T.dot(T - Y)为一个 3 行 1 列的数据
    # 第 1 行等于：x00×e0+x10×e1+x20×e2+x30×e3，它会调整第 1 个神经元对应的权值
    # 第 2 行等于：x01×e0+x11×e1+x21×e2+x31×e3，它会调整第 2 个神经元对应的权值
    # 第 3 行等于：x02×e0+x12×e1+x22×e2+x32×e3，它会调整第 3 个神经元对应的权值
    # X.shape 表示 X 的形状，X.shape[0]得到 X 的行数，表示有多少个数据
    # X.shape[1]得到的列数，表示每个数据有多少个特征值
    # 这里的公式跟式（3.2）看起来有些不同，原因是这里的计算是矩阵运算，式（3.2）是单个元素的
      计算。如果在草稿上仔细推算，则你会发现它们的本质是一样的
    delta_W = lr * (X.T.dot(E)) / X.shape[0]
        W = W + delta_W
# 训练 100 次
for i in range(100):
    #更新一次权值
    train()
    # 打印当前训练次数
    print('epoch:',i + 1)
    # 打印当前权值
    print('weights:',W)
    # 计算当前输出
    Y = np.sign(np.dot(X,W))
    # .all()表示 Y 中的所有值跟 T 中的所有值都对应相等时结果才为真
    if(Y == T).all():
        print('Finished')
        # 跳出循环
        break

#————————————以下为画图部分————————————#
# 正样本的 x、y 坐标
x1 = [3,4]
y1 = [3,3]
# 负样本的 x、y 坐标
x2 = [1,2]
y2 = [1,1]

# 计算分类边界线的斜率和截距
# 神经网络的信号总和为 w0×x0+w1×x1+w2×x2
# 当信号总和大于 0 时经过激活函数，模型的预测值会得到 1
# 当信号总和小于 0 时经过激活函数，模型的预测值会得到-1
# 所以，当信号总和 w0×x0+w1×x1+w2×x2=0 时，其为分类边界线表达式
# 我们在画图的时候，把 x1，x2 分别看作平面坐标系中的 x 和 y
# 可以得到：w0 + w1×x + w2 × y = 0
# 经过通分：y = -w0/w2 - w1×x/w2，因此可以得到：
k = - W[1] / W[2]
d =  -W[0] / W[2]
# 设定两个点
xdata = (0,5)
# 通过两个点来确定一条直线，用红色的线画出分界线
plt.plot(xdata,xdata * k + d,'r')
# 用蓝色的点画出正样本
```

```
plt.scatter(x1,y1,c='b')
# 用黄色的点画出负样本
plt.scatter(x2,y2,c='y')
# 显示图案
plt.show()
```

运行结果如下：

```
epoch: 1
weights: [[0.83669451]
 [0.58052698]
 [0.25564497]]
epoch: 2
weights: [[0.73669451]
 [0.43052698]
 [0.15564497]]
epoch: 3
weights: [[0.63669451]
 [0.28052698]
 [0.05564497]]
……
epoch: 16
weights: [[-0.01330549]
 [0.13052698]
 [0.20564497]]
epoch: 17
weights: [[-0.11330549]
 [-0.01947302]
 [ 0.10564497]]
Finished
```



因为权值的初始化使用的是随机的初始化方式，所以每一次训练的周期和画出来的图可能都是不一样的。这里我们可以看到单层感知器的一个问题，虽然单层感知器可以顺利地完成分类任务，但是使用单层感知器来做分类的时候，最后得到的分类边界距离某一个类别比较近，而距离另一个类别比较远，并不是一个特别理想的分类效果。图 3.8 中的分类效果应该才是比较理想的分类效果，分界线在两个类别比较中间的位置。

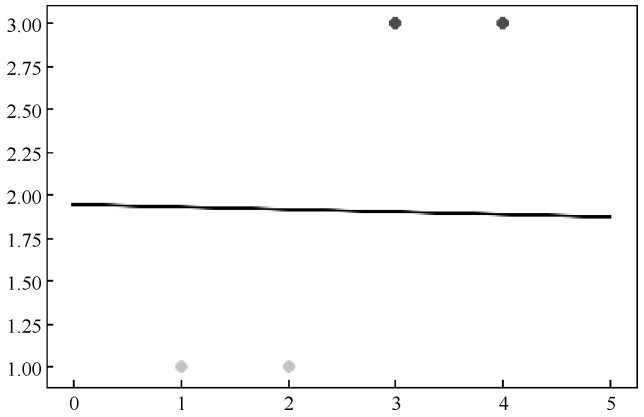


图 3.8　单层感知器比较理想的分类边界

3.8　线性神经网络

3.8.1　线性神经网络介绍

线性神经网络跟单层感知器非常类似，只是把单层感知器的 sign 激活函数改成了 purelin 函数：

$$y = x \tag{3.5}$$

purelin 函数也称为线性函数，如图 3.9 所示。

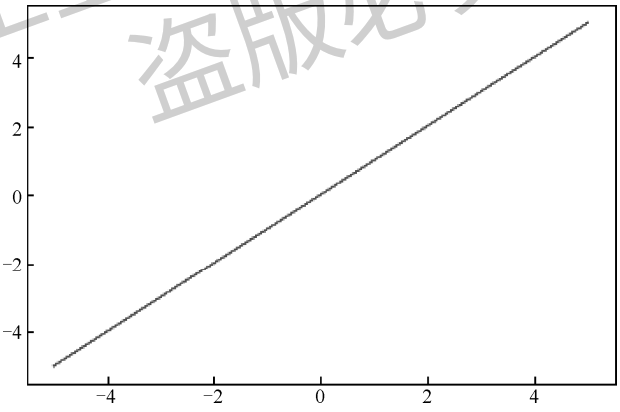


图 3.9　purelin 函数

3.8.2　线性神经网络分类案例

参考“单层感知器案例”，我们这次使用线性神经网络来完成相同的任务。线性神经网络的程序跟单层感知器的程序非常相似，大家可以思考一下需要修改哪些地方。

大家可以仔细阅读代码 3-4，找到修改了的部分。

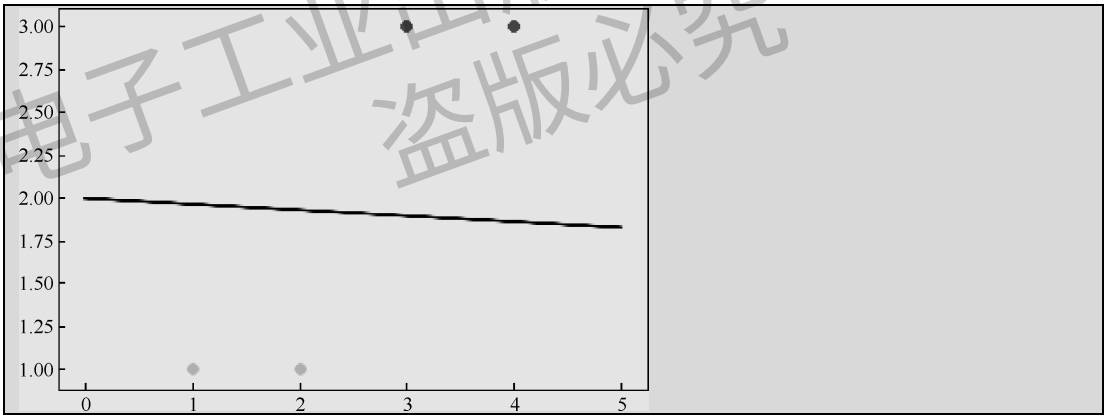
代码 3-4：线性神经网络案例

```
import numpy as np
```

```
import matplotlib.pyplot as plt
# 定义输入，我们习惯上用一行代表一个数据
X = np.array([[1,3,3],
              [1,4,3],
              [1,1,1],
              [1,2,1]])
# 定义标签，我们习惯上用一行表示一个数据的标签
T = np.array([[1],
              [1],
              [-1],
              [-1]])

# 权值初始化，3 行 1 列
# np.random.random 可以生成 0～1 的随机数
W = np.random.random([3,1])
# 学习率设置
lr = 0.1
# 神经网络的输出
Y = 0

# 更新一次权值
def train():
    # 使用全局变量 W
    global W
    # 同时计算 4 个数据的预测值
    # Y 的形状为(4,1)-4 行 1 列
    Y = np.dot(X,W)
    # T - Y 得到标签值与预测值的误差 E，其形状为(4,1)
    E = T - Y
    # X.T 表示 X 的转置矩阵，形状为(3,4)
    # 我们一共有 4 个数据，每个数据 3 个值。定义第 i 个数据的第 j 个特征值为 xij
    # 如第 1 个数据的第 2 个特征值为 x12
    # X.T.dot(T - Y)为一个 3 行 1 列的数据
    # 第 1 行等于：x00×e0+x10×e1+x20×e2+x30×e3，它会调整第 1 个神经元对应的权值
    # 第 2 行等于：x01×e0+x11×e1+x21×e2+x31×e3，它会调整第 2 个神经元对应的权值
    # 第 3 行等于：x02×e0+x12×e1+x22×e2+x32×e3，它会调整第 3 个神经元对应的权值
    # X.shape 表示 X 的形状，X.shape[0]得到 X 的行数，表示有多少个数据
    # X.shape[1]得到的列数，表示每个数据有多少个特征值
    # 这里的公式跟式（3.2）看起来有些不同，原因是这里的计算是矩阵运算，式（3.2）是单个元素的
      计算。如果在草稿上仔细推算，你会发现它们的本质是一样的
    delta_W = lr * (X.T.dot(E)) / X.shape[0]
      W = W + delta_W
# 训练 100 次
for i in range(100):
    #更新一次权值
    train()

#————————————以下为画图部分————————————#
# 正样本的 x、y 坐标
x1 = [3,4]
y1 = [3,3]
# 负样本的 x、y 坐标
```

```
x2 = [1,2]
y2 = [1,1]

# 计算分类边界线的斜率和截距
# 神经网络的信号总和为 w0×x0+w1×x1+w2×x2
# 当信号总和大于 0 时，经过激活函数，模型的预测值会得到 1
# 当信号总和小于 0 时，经过激活函数，模型的预测值会得到-1
# 所以当信号总和 w0×x0+w1×x1+w2×x2=0 时，其为分类边界线表达式
# 我们在画图的时候把 x1，x2 分别看作平面坐标系中的 x 和 y
# 可以得到：w0 + w1×x + w2 × y = 0
# 经过通分：y = -w0/w2 - w1×x/w2，因此可以得到：
k = - W[1] / W[2]
d =  -W[0] / W[2]
# 设定两个点
xdata = (0,5)
# 通过两个点来确定一条直线，用红色的线画出分界线
plt.plot(xdata,xdata * k + d,'r')
# 用蓝色的点画出正样本
plt.scatter(x1,y1,c='b')
# 用黄色的点画出负样本
plt.scatter(x2,y2,c='y')
# 显示图案
plt.show()
```

运行结果如下：



线性神经网络的程序有两处对单层感知器程序进行了修改。

第一处是在 train()函数中，将“Y = np.sign(np.dot(X,W))”改成了“Y = np.dot(X,W)”。因为线性神经网络的激活函数是 $y=x$，所以这里就不需要“np.sign()”了。

第二处是在 for i in range(100)中，把原来的：

```
# 训练 100 次
for i in range(100):
    #更新一次权值
    train()
    # 打印当前训练次数
    print('epoch:',i + 1)
    # 打印当前权值
```

```
    print('weights:',W)
    # 计算当前输出
    Y = np.sign(np.dot(X,W))
    # .all()表示 Y 中的所有值跟 T 中的所有值都对应相等时结果才为真
    if(Y == T).all():
        print('Finished')
        # 跳出循环
        break
```

改成了：

```
# 训练 100 次
for i in range(100):
    #更新一次权值
    train()
```

在单层感知器中，当 $y=t$ 时，Δw 就会为 0，模型训练就结束了，所以可以提前跳出循环。单层感知器使用的模型收敛条件是两次迭代模型的权值已经不再发生变化，此时则可以认为模型收敛。

而在线性神经网络中，y 会一直逼近 t 的值，但一般不会得到等于 t 的值，所以可以对模型不断进行优化。线性神经网络使用的模型收敛条件是设置一个最大迭代次数，当训练了一定次数后就可以认为模型收敛了。

对比单层感知器和线性神经网络所得到的结果，我们可以看得出线性神经网络所得到的结果会比单层感知器得到的结果更理想。但是线性神经网络也还不够优秀，当使用它处理非线性问题的时候，它就不能很好地完成工作了。

3.9　线性神经网络处理异或问题

首先我们先来回顾一下异或运算：

（1）0 与 0 异或等于 0；

（2）0 与 1 异或等于 1；

（3）1 与 0 异或等于 1；

（4）1 与 1 异或等于 0。

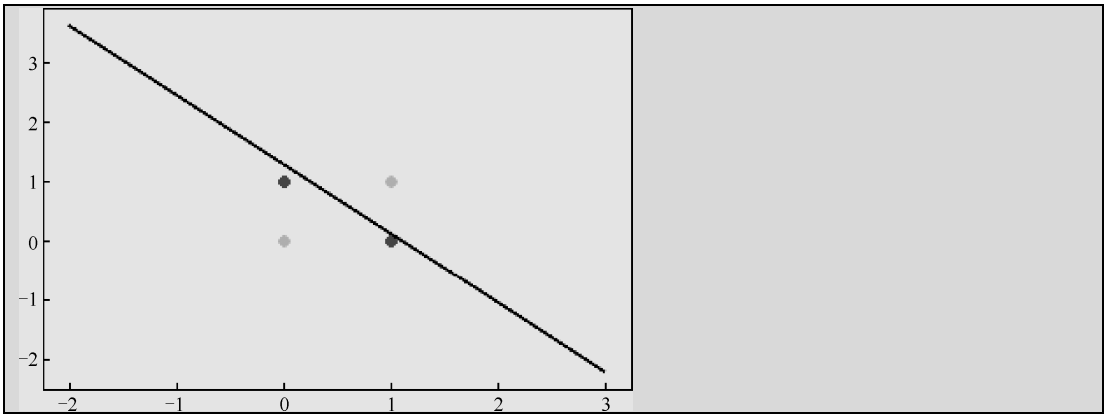
线性神经网络处理异或问题的代码如代码 3-5 所示。

代码 3-5：线性神经网络处理异或问题

```
import numpy as np
import matplotlib.pyplot as plt
# 输入数据
# 4 个数据分别对应 0 与 0 异或、0 与 1 异或、1 与 0 异或、1 与 1 异或
X = np.array([[1,0,0],
              [1,0,1],
              [1,1,0],
              [1,1,1]])
# 标签，分别对应 4 种异或情况的结果
# 注意，这里我们使用-1 作为负标签
T = np.array([[-1],
```

```
          [1],
          [1],
          [-1]])

# 权值初始化，3 行 1 列
# np.random.random 可以生成 0～1 的随机数
W = np.random.random([3,1])

# 学习率设置
lr = 0.1
# 神经网络的输出
Y = 0

# 更新一次权值
def train():
    # 使用全局变量 W
    global W
    # 计算网络预测值
    Y = np.dot(X,W)
    # 计算权值的改变
    delta_W = lr * (X.T.dot(T - Y)) / X.shape[0]
        # 更新权值
        W = W + delta_W
# 训练 100 次
for i in range(100):
    #更新一次权值
    train()

#————————以下为画图部分————————#
# 正样本
x1 = [0,1]
y1 = [1,0]
# 负样本
x2 = [0,1]
y2 = [0,1]

#计算分界线的斜率和截距
k = - W[1] / W[2]
d = - W[0] / W[2]

# 设定两个点
xdata = (-2,3)
# 通过两个点来确定一条直线，用红色的线画出分界线
plt.plot(xdata,xdata * k + d,'r')
# 用蓝色的点画出正样本
plt.scatter(x1,y1,c='b')
# 用黄色的点画出负样本
plt.scatter(x2,y2,c='y')
# 显示图案
plt.show()
```

运行结果如下：



从运行结果我们能够看出用一条直线并不能把异或问题中的两个类别给划分开来，因为这是一个非线性的问题，所以可以使用非线性的方式来进行求解。其中一种方式就是我们可以给神经网络加入非线性的输入。代码 3-5 中的输入信号只有 3 个信号，即 x_0,x_1,x_2，我们可以利用这 3 个信号得到带有非线性特征的输入，即 $x_0,x_1,x_2,x_1\times x_1,x_1\times x_2,x_2\times x_2$，其中 $x_1\times x_1,x_1\times x_2,x_2\times x_2$ 为非线性特征。引入非线性输入的线性神经网络如图 3.10 所示。

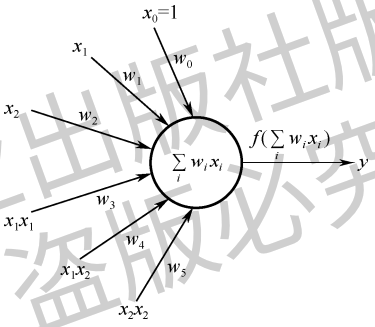


图 3.10　引入非线性输入的线性神经网络

线性神经网络引入非线性特征解决异或问题的代码如代码 3-6 所示。

代码 3-6：线性神经网络引入非线性特征解决异或问题

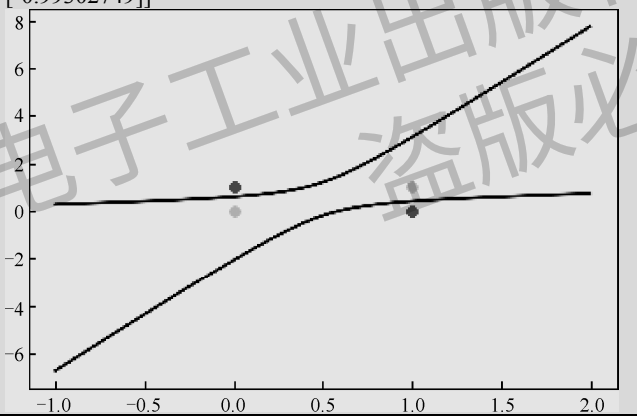
```
import numpy as np
import matplotlib.pyplot as plt
# 输入数据
# 原来 X 的 3 个特征分别为 x0、x1、x2
# X = np.array([[1,0,0],
#               [1,0,1],
#               [1,1,0],
#               [1,1,1]])
# 给网络输入非线性特征
# 现在 X 的 6 个特征分别为 x0、x1、x2、x1×x1、x1×x2、x2×x2
X = np.array([[1,0,0,0,0,0],
              [1,0,1,0,0,1],
              [1,1,0,1,0,0],
              [1,1,1,1,1,1]])
# 标签，分别对应 4 种异或情况的结果
```

```
T = np.array([[-1],
              [1],
              [1],
              [-1]])
# 权值初始化，6 行 1 列
# np.random.random 可以生成 0～1 的随机数
W = np.random.random([6,1])
# 学习率设置
lr = 0.1
# 神经网络的输出
Y = 0

# 更新一次权值
def train():
    # 使用全局变量 W
    global W
    # 计算网络预测值
    Y = np.dot(X,W)
    # 计算权值的改变
    delta_W = lr * (X.T.dot(T - Y)) / X.shape[0]
        # 更新权值
        W = W + delta_W
# 训练 1000 次
for i in range(1000):
    #更新一次权值
    train()

# 计算模型预测结果并打印
Y = np.dot(X,W)
print(Y)

#————————————以下为画图部分————————————#
# 正样本
x1 = [0,1]
y1 = [1,0]
# 负样本
x2 = [0,1]
y2 = [0,1]

# 神经网络信号的总和为：w0x0+w1x1+w2x2+w3x1x1+w4x1x2+w5x2x2
# 当 w0x0+w1x1+w2x2+w3x1x1+w4x1x2+w5x2x2=0 时，其为分类边界线
# 其中 x0 为 1，我们可以把 x1，x2 分别看作平面坐标系中的 x 和 y
# 可以得到：w0 + w1x + w2y + w3xx + w4xy + w5yy = 0
# 通分可得：w5y^2 + (w2+w4x)y + w0 + w1x + w3x^2 = 0
# 其中 a = w5, b = w2+w4x, c = w0 + w1x + w3x^2
# 根据一元二次方程的求根公式：ay^2+by+c=0，y=[-b±(b^2-4ac)^1/2 ]/2a

def calculate(x,root):
    # 定义参数
    a = W[5]
    b = W[2] + x * W[4]
    c = W[0] + x * W[1] + x * x * W[3]
```

```
    # 有两个根
    if root == 1:
        return (- b + np.sqrt(b * b - 4 * a * c)) / (2 * a)
    if root == 2:
        return (- b - np.sqrt(b * b - 4 * a * c)) / (2 * a)

# 从-1～2 之间均匀生成 100 个点
xdata = np.linspace(-1,2,100)
# 使用第一个求根公式计算出来的结果画出第一条红线
plt.plot(xdata,calculate(xdata,1),'r')
# 使用第二个求根公式计算出来的结果画出第二条红线
plt.plot(xdata,calculate(xdata,2),'r')
# 蓝色点表示正样本
plt.plot(x1,y1,'bo')
# 黄色点表示负样本
plt.plot(x2,y2,'yo')
# 绘图
plt.show()
```

运行结果如下：

```
[[-0.98650596]
 [ 0.990989  ]
 [ 0.990989  ]
[-0.99302749]]
```



从输出的预测值中我们可以看出，预测值与真实标签的数值是非常接近的，几乎相等，说明预测值很符合我们想要的结果。而从输出图片中也能观察到两条曲线的内部是负样本所属的类别，两条曲线的外部是正样本所属的类别。这两条曲线很好地把两个类别区分开了。

线性神经网络可以通过引入非线性的输入特征来解决非线性问题，但这并不是一种非常好的解决方案。

下一章我们将介绍一种新的神经网络，即 BP 神经网络。通过学习 BP 神经网络，我们可以获得更好的解决问题的思路。