第3章
神经网络

上一章我们学习了感知机。关于感知机，既有好消息，也有坏消息。好消息是，即便对于复杂的函数，感知机也隐含着能够表示它的可能性。上一章已经介绍过，即便是计算机进行的复杂处理，感知机（理论上）也可以将其表示出来。坏消息是，设定权重的工作，即确定合适的、能符合预期的输入与输出的权重，现在还是由人工进行的。上一章中，我们结合与门、或门的真值表人工决定了合适的权重。

神经网络的出现就是为了解决刚才的坏消息。具体地讲，神经网络的一个重要性质是它可以自动地从数据中学习到合适的权重参数。本章中，我们会先介绍神经网络的概要，然后重点关注神经网络进行识别时的处理。在下一章中，我们将了解如何从数据中学习权重参数。

3.1 从感知机到神经网络

神经网络和上一章介绍的感知机有很多共同点。这里，我们主要以两者的差异为中心，来介绍神经网络的结构。

3.1.1 神经网络的例子

用图来表示神经网络的话，如图3-1所示。我们把最左边的一列称为**输入层**，最右边的一列称为**输出层**，中间的一列称为**中间层**。中间层有时

也称为**隐藏层**。“隐藏”一词的意思是，隐藏层的神经元（和输入层、输出层不同）肉眼看不见。另外，本书中把输入层到输出层依次称为第0层、第1层、第2层（层号之所以从0开始，是为了方便后面基于Python进行实现）。图3-1中，第0层对应输入层，第1层对应中间层，第2层对应输出层。

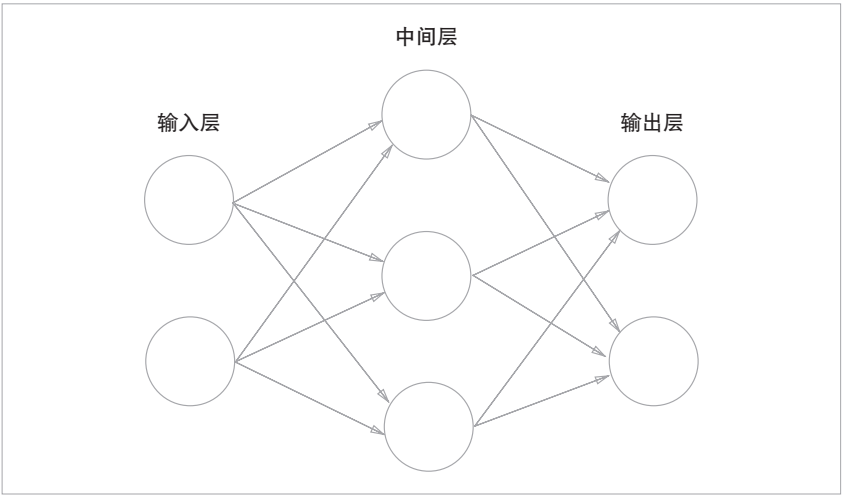

图3-1 神经网络的例子

图3–1中的网络一共由3层神经元构成，但实质上只有2层神经元有权重，因此将其称为“2层网络”。请注意，有的书也会根据构成网络的层数，把图3–1的网络称为“3层网络”。本书将根据实质上拥有权重的层数（输入层、隐藏层、输出层的总数减去1后的数量）来表示网络的名称。

只看图3-1的话，神经网络的形状类似上一章的感知机。实际上，就神经元的连接方式而言，与上一章的感知机并没有任何差异。那么，神经网络中信号是如何传递的呢？

3.1.2 复习感知机

在观察神经网络中信号的传递方法之前，我们先复习一下感知机。现在

来思考一下图3-2中的网络结构。

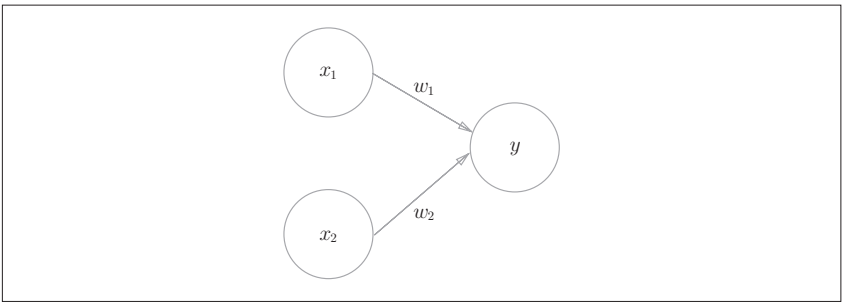

图3-2 复习感知机

图3-2中的感知机接收x_1和x_2两个输入信号，输出y。如果用数学式来表示图3-2中的感知机，则如式(3.1)所示。

$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leqslant 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases} \tag{3.1}$$

b是被称为偏置的参数，用于控制神经元被激活的容易程度；而w_1和w_2是表示各个信号的权重的参数，用于控制各个信号的重要性。

顺便提一下，在图3-2的网络中，偏置b并没有被画出来。如果要明确地表示出b，可以像图3-3那样做。图3-3中添加了权重为b的输入信号1。这个感知机将x_1、x_2、1三个信号作为神经元的输入，将其和各自的权重相乘后，传送至下一个神经元。在下一个神经元中，计算这些加权信号的总和。如果这个总和超过0，则输出1，否则输出0。另外，由于偏置的输入信号一直是1，所以为了区别于其他神经元，我们在图中把这个神经元整个涂成灰色。

现在将式(3.1)改写成更加简洁的形式。为了简化式(3.1)，我们用一个函数来表示这种分情况的动作(超过0则输出1，否则输出0)。引入新函数$h(x)$，将式(3.1)改写成下面的式(3.2)和式(3.3)。

$$y = h(b + w_1x_1 + w_2x_2) \tag{3.2}$$

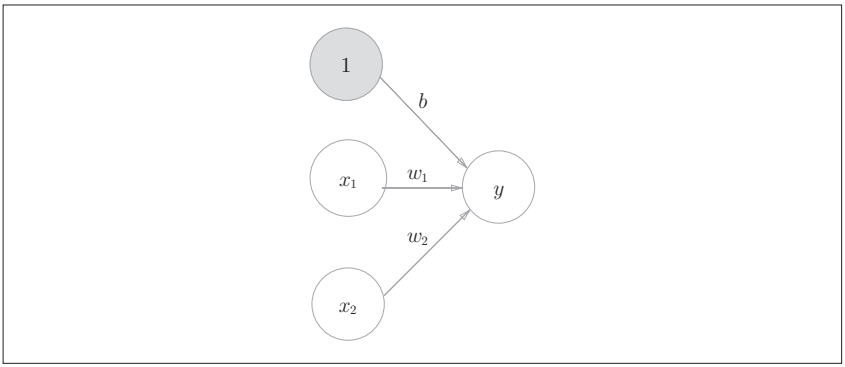

图3-3 明确表示出偏置

$$h(x) = \begin{cases} 0 & (x \leqslant 0) \\ 1 & (x > 0) \end{cases} \tag{3.3}$$

式(3.2)中，输入信号的总和会被函数$h(x)$转换，转换后的值就是输出y。然后，式(3.3)所表示的函数$h(x)$，在输入超过0时返回1，否则返回0。因此，式(3.1)和式(3.2)、式(3.3)做的是相同的事情。

3.1.3 激活函数登场

刚才登场的$h(x)$函数会将输入信号的总和转换为输出信号，这种函数一般称为**激活函数**(activation function)。如“激活”一词所示，激活函数的作用在于决定如何来激活输入信号的总和。

现在来进一步改写式(3.2)。式(3.2)分两个阶段进行处理，先计算输入信号的加权总和，然后用激活函数转换这一总和。因此，如果将式(3.2)写得详细一点，则可以分成下面两个式子。

$$a = b + w_1x_1 + w_2x_2 \tag{3.4}$$

$$y = h(a) \tag{3.5}$$

首先，式(3.4)计算加权输入信号和偏置的总和，记为a。然后，式(3.5)用$h()$函数将a转换为输出y。

之前的神经元都是用一个○表示的，如果要在图中明确表示出式(3.4)和式(3.5)，则可以像图3-4这样做。

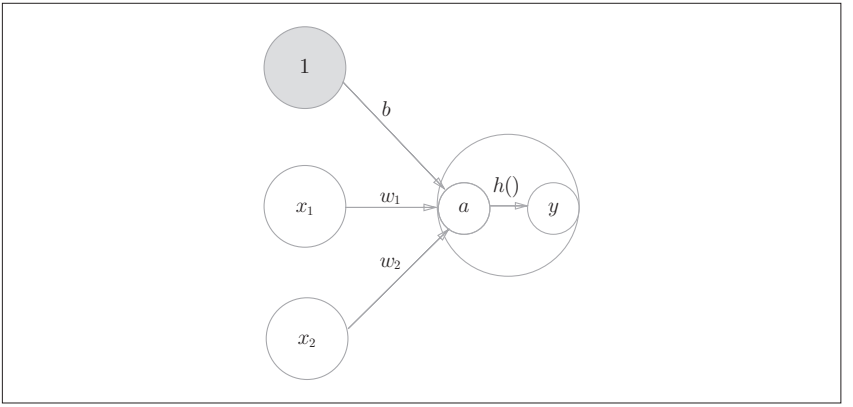

图3-4 明确显示激活函数的计算过程

如图3-4所示，表示神经元的○中明确显示了激活函数的计算过程，即信号的加权总和为节点a，然后节点a被激活函数$h()$转换成节点y。本书中，“神经元”和“节点”两个术语的含义相同。这里，我们称a和y为“节点”，其实它和之前所说的“神经元”含义相同。

通常如图3-5的左图所示，神经元用一个○表示。本书中，在可以明确神经网络的动作的情况下，将在图中明确显示激活函数的计算过程，如图3-5的右图所示。

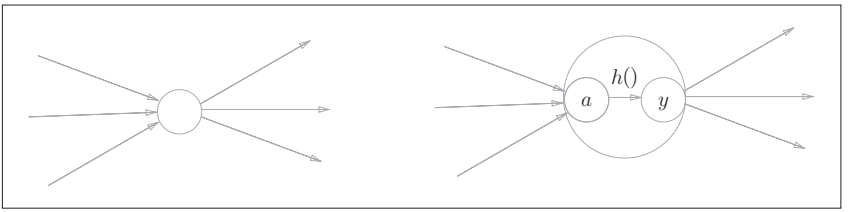

图3-5 左图是一般的神经元的图，右图是在神经元内部明确显示激活函数的计算过程的图（a表示输入信号的总和，$h()$表示激活函数，y表示输出）

下面，我们将仔细介绍激活函数。激活函数是连接感知机和神经网络的桥梁。

本书在使用“感知机”一词时，没有严格统一它所指的算法。一般而言，“朴素感知机”是指单层网络，指的是激活函数使用了阶跃函数[①]的模型。“多层感知机”是指神经网络，即使用sigmoid函数（后述）等平滑的激活函数的多层网络。

3.2 激活函数

式（3.3）表示的激活函数以阈值为界，一旦输入超过阈值，就切换输出。这样的函数称为“阶跃函数”。因此，可以说感知机中使用了阶跃函数作为激活函数。也就是说，在激活函数的众多候选函数中，感知机使用了阶跃函数。那么，如果感知机使用其他函数作为激活函数的话会怎么样呢？实际上，如果将激活函数从阶跃函数换成其他函数，就可以进入神经网络的世界了。下面我们就来介绍一下神经网络使用的激活函数。

3.2.1 sigmoid函数

神经网络中经常使用的一个激活函数就是式（3.6）表示的**sigmoid函数**（sigmoid function）。

$$h(x) = \frac{1}{1 + \exp(-x)} \tag{3.6}$$

式（3.6）中的$\exp(-x)$表示e^{-x}的意思。e是纳皮尔常数$2.7182\cdots$。式（3.6）表示的sigmoid函数看上去有些复杂，但它也仅仅是个函数而已。而函数就是给定某个输入后，会返回某个输出的转换器。比如，向sigmoid函数输入1.0或2.0后，就会有某个值被输出，类似$h(1.0) = 0.731\cdots$、$h(2.0) = 0.880\cdots$这样。

神经网络中用sigmoid函数作为激活函数，进行信号的转换，转换后的

① 阶跃函数是指一旦输入超过阈值，就切换输出的函数。

信号被传送给下一个神经元。实际上，上一章介绍的感知机和接下来要介绍的神经网络的主要区别就在于这个激活函数。其他方面，比如神经元的多层连接的构造、信号的传递方法等，基本上和感知机是一样的。下面，让我们通过和阶跃函数的比较来详细学习作为激活函数的sigmoid函数。

3.2.2 阶跃函数的实现

这里我们试着用Python画出阶跃函数的图（从视觉上确认函数的形状对理解函数而言很重要）。阶跃函数如式（3.3）所示，当输入超过0时，输出1，否则输出0。可以像下面这样简单地实现阶跃函数。

```
def step_function(x):
    if x > 0:
        return 1
    else:
        return 0
```

这个实现简单、易于理解，但是参数x只能接受实数（浮点数）。也就是说，允许形如step_function(3.0)的调用，但不允许参数取NumPy数组，例如step_function(np.array([1.0, 2.0]))。为了便于后面的操作，我们把它修改为支持NumPy数组的实现。为此，可以考虑下述实现。

```
def step_function(x):
    y = x > 0
    return y.astype(np.int)
```

上述函数的内容只有两行。由于使用了NumPy中的“技巧”，可能会有点难理解。下面我们通过Python解释器的例子来看一下这里用了什么技巧。下面这个例子中准备了NumPy数组x，并对这个NumPy数组进行了不等号运算。

```
>>> import numpy as np
>>> x = np.array([-1.0, 1.0, 2.0])
>>> x
array([-1.,  1.,  2.])
>>> y = x > 0
```

```
>>> y
array([False,  True,  True], dtype=bool)
```

对NumPy数组进行不等号运算后，数组的各个元素都会进行不等号运算，生成一个布尔型数组。这里，数组x中大于0的元素被转换为True，小于等于0的元素被转换为False，从而生成一个新的数组y。

数组y是一个布尔型数组，但是我们想要的阶跃函数是会输出int型的0或1的函数。因此，需要把数组y的元素类型从布尔型转换为int型。

```
>>> y = y.astype(np.int)
>>> y
array([0, 1, 1])
```

如上所示，可以用astype()方法转换NumPy数组的类型。astype()方法通过参数指定期望的类型，这个例子中是np.int型。Python中将布尔型转换为int型后，True会转换为1，False会转换为0。以上就是阶跃函数的实现中所用到的NumPy的“技巧”。

3.2.3 阶跃函数的图形

下面我们就用图来表示上面定义的阶跃函数，为此需要使用matplotlib库。

```
import numpy as np
import matplotlib.pylab as plt

def step_function(x):
    return np.array(x > 0, dtype=np.int)

x = np.arange(-5.0, 5.0, 0.1)
y = step_function(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # 指定y轴的范围
plt.show()
```

np.arange(-5.0, 5.0, 0.1)在−5.0到5.0的范围内，以0.1为单位，生成NumPy数组（[-5.0, -4.9, ···, 4.9]）。step_function()以该NumPy数组为参数，对数组的各个元素执行阶跃函数运算，并以数组形式返回运算结果。对数组x、y进行绘图，结果如图3-6所示。

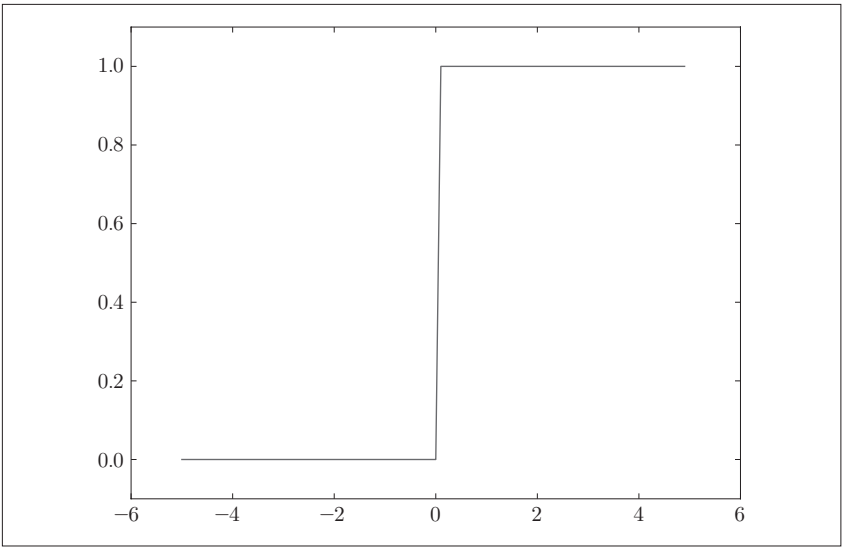

图3-6 阶跃函数的图形

如图3-6所示，阶跃函数以0为界，输出从0切换为1（或者从1切换为0）。它的值呈阶梯式变化，所以称为阶跃函数。

3.2.4 sigmoid函数的实现

下面，我们来实现sigmoid函数。用Python可以像下面这样写出式（3.6）表示的sigmoid函数。

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

这里，`np.exp(-x)`对应$\exp(-x)$。这个实现没有什么特别难的地方，但是要注意参数`x`为NumPy数组时，结果也能被正确计算。实际上，如果在这个sigmoid函数中输入一个NumPy数组，则结果如下所示。

```
>>> x = np.array([-1.0, 1.0, 2.0])
>>> sigmoid(x)
array([ 0.26894142,  0.73105858,  0.88079708])
```

之所以sigmoid函数的实现能支持NumPy数组，秘密就在于NumPy的广播功能（1.5.5节）。根据NumPy 的广播功能，如果在标量和NumPy数组之间进行运算，则标量会和NumPy数组的各个元素进行运算。这里来看一个具体的例子。

```
>>> t = np.array([1.0, 2.0, 3.0])
>>> 1.0 + t
array([ 2.,  3.,  4.])
>>> 1.0 / t
array([ 1.        ,  0.5       ,  0.33333333])
```

在这个例子中，标量（例子中是1.0）和NumPy数组之间进行了数值运算（+、/等）。结果，标量和NumPy数组的各个元素进行了运算，运算结果以NumPy数组的形式被输出。刚才的sigmoid函数的实现也是如此，因为np.exp(-x)会生成NumPy数组，所以1 / (1 + np.exp(-x))的运算将会在NumPy数组的各个元素间进行。

下面我们把sigmoid函数画在图上。画图的代码和刚才的阶跃函数的代码几乎是一样的，唯一不同的地方是把输出y的函数换成了sigmoid函数。

```
x = np.arange(-5.0, 5.0, 0.1)
y = sigmoid(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # 指定y轴的范围
plt.show()
```

运行上面的代码，可以得到图3-7。

3.2.5 sigmoid函数和阶跃函数的比较

现在我们来比较一下sigmoid 函数和阶跃函数，如图3-8所示。两者的不同点在哪里呢？又有哪些共同点呢？我们通过观察图3-8来思考一下。

观察图3-8，首先注意到的是“平滑性”的不同。sigmoid函数是一条平滑的曲线，输出随着输入发生连续性的变化。而阶跃函数以0为界，输出发生急剧性的变化。sigmoid函数的平滑性对神经网络的学习具有重要意义。

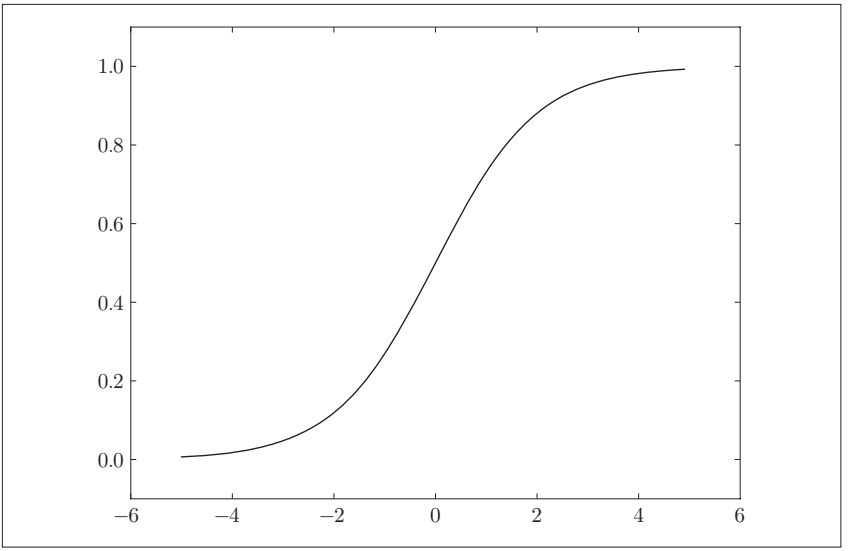

图3-7 sigmoid函数的图形

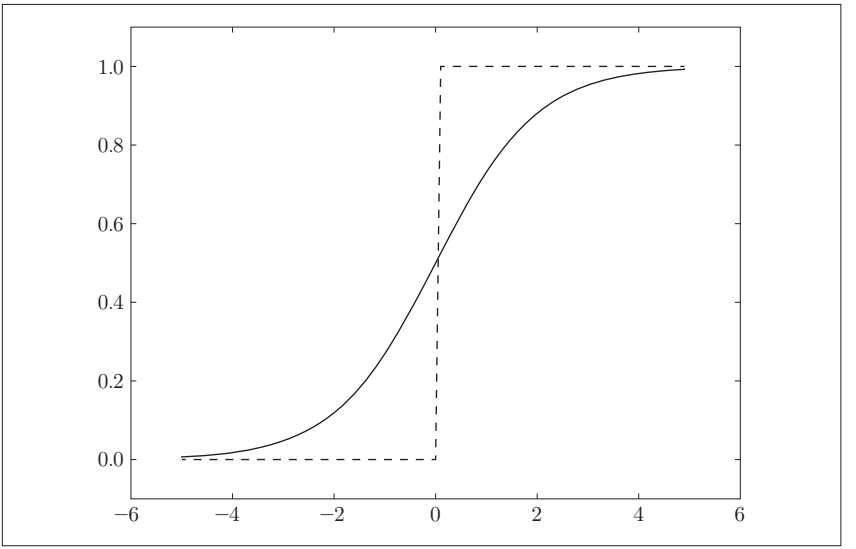

图3-8 阶跃函数与sigmoid函数（虚线是阶跃函数）

另一个不同点是，相对于阶跃函数只能返回0或1，sigmoid函数可以返回0.731…、0.880…等实数（这一点和刚才的平滑性有关）。也就是说，感知机中神经元之间流动的是0或1的二元信号，而神经网络中流动的是连续的实数值信号。

如果把这两个函数与水联系起来，则阶跃函数可以比作“竹筒敲石”[①]，sigmoid函数可以比作“水车”。阶跃函数就像竹筒敲石一样，只做是否传送水（0或1）两个动作，而sigmoid函数就像水车一样，根据流过来的水量相应地调整传送出去的水量。

接着说一下阶跃函数和sigmoid函数的共同性质。阶跃函数和sigmoid函数虽然在平滑性上有差异，但是如果从宏观视角看图3-8，可以发现它们具有相似的形状。实际上，两者的结构均是“输入小时，输出接近0（为0）；随着输入增大，输出向1靠近（变成1）”。也就是说，当输入信号为重要信息时，阶跃函数和sigmoid函数都会输出较大的值；当输入信号为不重要的信息时，两者都输出较小的值。还有一个共同点是，不管输入信号有多小，或者有多大，输出信号的值都在0到1之间。

3.2.6 非线性函数

阶跃函数和sigmoid函数还有其他共同点，就是两者均为**非线性函数**。sigmoid函数是一条曲线，阶跃函数是一条像阶梯一样的折线，两者都属于非线性的函数。

在介绍激活函数时，经常会看到“非线性函数”和“线性函数”等术语。函数本来是输入某个值后会返回一个值的转换器。向这个转换器输入某个值后，输出值是输入值的常数倍的函数称为线性函数（用数学式表示为$h(x) = cx$。c为常数）。因此，线性函数是一条笔直的直线。而非线性函数，顾名思义，指的是不像线性函数那样呈现出一条直线的函数。

① 竹筒敲石是日本的一种庭院设施。支点架起竹筒，一端下方置石，另一端切口上翘。在切口上滴水，水积多后该端下垂，水流出，另一端翘起，之后又因重力而落下，击石发出响声。——译者注

神经网络的激活函数必须使用非线性函数。换句话说，激活函数不能使用线性函数。为什么不能使用线性函数呢？因为使用线性函数的话，加深神经网络的层数就没有意义了。

线性函数的问题在于，不管如何加深层数，总是存在与之等效的“无隐藏层的神经网络”。为了具体地（稍微直观地）理解这一点，我们来思考下面这个简单的例子。这里我们考虑把线性函数$h(x) = cx$作为激活函数，把$y(x) = h(h(h(x)))$的运算对应3层神经网络[①]。这个运算会进行$y(x) = c \times c \times c \times x$的乘法运算，但是同样的处理可以由$y(x) = ax$（注意，$a = c^3$）这一次乘法运算（即没有隐藏层的神经网络）来表示。如本例所示，使用线性函数时，无法发挥多层网络带来的优势。因此，为了发挥叠加层所带来的优势，激活函数必须使用非线性函数。

3.2.7 ReLU函数

到目前为止，我们介绍了作为激活函数的阶跃函数和sigmoid函数。在神经网络发展的历史上，sigmoid函数很早就开始被使用了，而最近则主要使用**ReLU**（Rectified Linear Unit）函数。

ReLU函数在输入大于0时，直接输出该值；在输入小于等于0时，输出0（图3-9）。

ReLU函数可以表示为下面的式(3.7)。

$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leqslant 0) \end{cases} \tag{3.7}$$

如图3-9和式(3.7)所示，ReLU函数是一个非常简单的函数。因此，ReLU函数的实现也很简单，可以写成如下形式。

```
def relu(x):
    return np.maximum(0, x)
```

① 该对应只是一个近似，实际的神经网络运算比这个例子要复杂，但不影响后面的结论成立。
——译者注

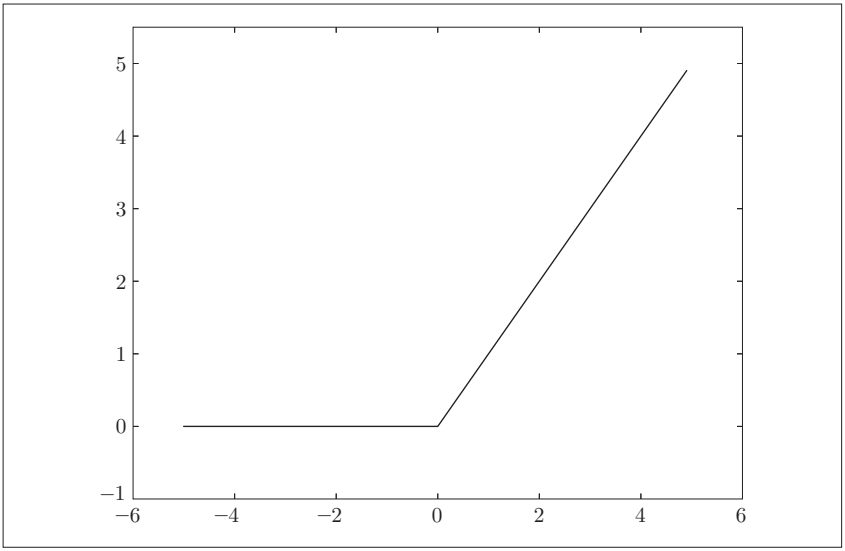

图3-9 ReLU函数

这里使用了NumPy的`maximum`函数。`maximum`函数会从输入的数值中选择较大的那个值进行输出。

本章剩余部分的内容仍将使用sigmoid函数作为激活函数，但在本书的后半部分，则将主要使用ReLU函数。

3.3 多维数组的运算

如果掌握了NumPy多维数组的运算，就可以高效地实现神经网络。因此，本节将介绍NumPy多维数组的运算，然后再进行神经网络的实现。

3.3.1 多维数组

简单地讲，多维数组就是“数字的集合”，数字排成一列的集合、排成长方形的集合、排成三维状或者（更加一般化的）N维状的集合都称为多维数组。下面我们就用NumPy来生成多维数组，先从前面介绍过的一维数组开始。

```
>>> import numpy as np
>>> A = np.array([1, 2, 3, 4])
>>> print(A)
[1 2 3 4]
>>> np.ndim(A)
1
>>> A.shape
(4,)
>>> A.shape[0]
4
```

如上所示，数组的维数可以通过np.dim()函数获得。此外，数组的形状可以通过实例变量shape获得。在上面的例子中，A是一维数组，由4个元素构成。注意，这里的A.shape的结果是个元组(tuple)。这是因为一维数组的情况下也要返回和多维数组的情况下一致的结果。例如，二维数组时返回的是元组(4,3)，三维数组时返回的是元组(4,3,2)，因此一维数组时也同样以元组的形式返回结果。下面我们来生成一个二维数组。

```
>>> B = np.array([[1,2], [3,4], [5,6]])
>>> print(B)
[[1 2]
 [3 4]
 [5 6]]
>>> np.ndim(B)
2
>>> B.shape
(3, 2)
```

这里生成了一个3 × 2的数组B。3 × 2的数组表示第一个维度有3个元素，第二个维度有2个元素。另外，第一个维度对应第0维，第二个维度对应第1维(Python的索引从0开始)。二维数组也称为**矩阵**(matrix)。如图3-10所示，数组的横向排列称为**行**(row)，纵向排列称为**列**(column)。

3.3.2 矩阵乘法

下面，我们来介绍矩阵(二维数组)的乘积。比如2 × 2的矩阵，其乘积可以像图3-11这样进行计算(按图中顺序进行计算是规定好了的)。

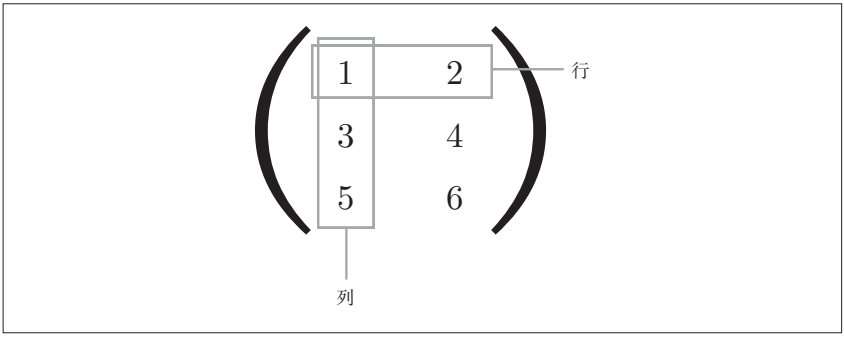

图3-10 横向排列称为行，纵向排列称为列

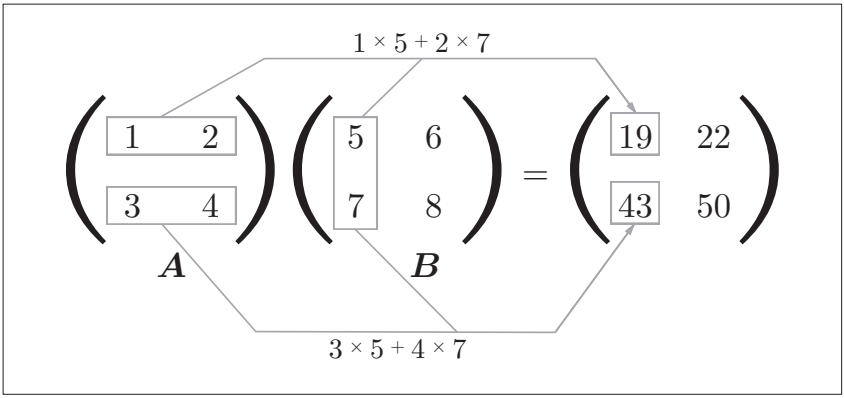

图3-11 矩阵的乘积的计算方法

如本例所示，矩阵的乘积是通过左边矩阵的行（横向）和右边矩阵的列（纵向）以对应元素的方式相乘后再求和而得到的。并且，运算的结果保存为新的多维数组的元素。比如，$\boldsymbol{A}$的第1行和$\boldsymbol{B}$的第1列的乘积结果是新数组的第1行第1列的元素，$\boldsymbol{A}$的第2行和$\boldsymbol{B}$的第1列的结果是新数组的第2行第1列的元素。另外，在本书的数学标记中，矩阵将用黑斜体表示（比如，矩阵$\boldsymbol{A}$），以区别于单个元素的标量（比如，a或b）。这个运算在Python中可以用如下代码实现。

```
>>> A = np.array([[1,2], [3,4]])
```

```
>>> A.shape
(2, 2)
>>> B = np.array([[5,6], [7,8]])
>>> B.shape
(2, 2)
>>> np.dot(A, B)
array([[19, 22],
       [43, 50]])
```

这里，$\boldsymbol{A}$和$\boldsymbol{B}$都是2×2的矩阵，它们的乘积可以通过NumPy的np.dot()函数计算（乘积也称为点积）。np.dot()接收两个NumPy数组作为参数，并返回数组的乘积。这里要注意的是，np.dot(A, B)和np.dot(B, A)的值可能不一样。和一般的运算（+或*等）不同，矩阵的乘积运算中，操作数（A、B）的顺序不同，结果也会不同。

这里介绍的是计算2×2形状的矩阵的乘积的例子，其他形状的矩阵的乘积也可以用相同的方法来计算。比如，2×3的矩阵和3×2 的矩阵的乘积可按如下形式用Python来实现。

```
>>> A = np.array([[1,2,3], [4,5,6]])
>>> A.shape
(2, 3)
>>> B = np.array([[1,2], [3,4], [5,6]])
>>> B.shape
(3, 2)
>>> np.dot(A, B)
array([[22, 28],
       [49, 64]])
```

2×3的矩阵$\boldsymbol{A}$和3×2的矩阵$\boldsymbol{B}$的乘积可按以上方式实现。这里需要注意的是矩阵的形状（shape）。具体地讲，矩阵$\boldsymbol{A}$的第1维的元素个数（列数）必须和矩阵$\boldsymbol{B}$的第0维的元素个数（行数）相等。在上面的例子中，矩阵$\boldsymbol{A}$的形状是2×3，矩阵$\boldsymbol{B}$的形状是3×2，矩阵$\boldsymbol{A}$的第1维的元素个数（3）和矩阵$\boldsymbol{B}$的第0维的元素个数（3）相等。如果这两个值不相等，则无法计算矩阵的乘积。比如，如果用Python计算2×3 的矩阵$\boldsymbol{A}$和2×2的矩阵$\boldsymbol{C}$的乘积，则会输出如下错误。

```
>>> C = np.array([[1,2], [3,4]])
>>> C.shape
```

```
(2, 2)
>>> A.shape
(2, 3)
>>> np.dot(A, C)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shapes (2,3) and (2,2) not aligned: 3 (dim 1) != 2 (dim 0)
```

这个错误的意思是，矩阵 $\boldsymbol{A}$ 的第1维和矩阵 $\boldsymbol{C}$ 的第0维的元素个数不一致（维度的索引从0开始）。也就是说，在多维数组的乘积运算中，必须使两个矩阵中的对应维度的元素个数一致，这一点很重要。我们通过图3-12再来确认一下。

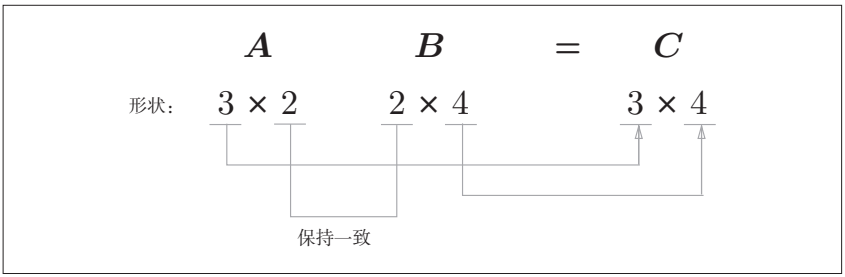

图3-12 在矩阵的乘积运算中，对应维度的元素个数要保持一致

图3-12中，3×2 的矩阵 $\boldsymbol{A}$ 和 2×4 的矩阵 $\boldsymbol{B}$ 的乘积运算生成了 3×4 的矩阵 $\boldsymbol{C}$。如图所示，矩阵 $\boldsymbol{A}$ 和矩阵 $\boldsymbol{B}$ 的对应维度的元素个数必须保持一致。此外，还有一点很重要，就是运算结果的矩阵 $\boldsymbol{C}$ 的形状是由矩阵 $\boldsymbol{A}$ 的行数和矩阵 $\boldsymbol{B}$ 的列数构成的。

另外，当 $\boldsymbol{A}$ 是二维矩阵、$\boldsymbol{B}$ 是一维数组时，如图3-13所示，对应维度的元素个数要保持一致的原则依然成立。

可按如下方式用Python实现图3-13的例子。

```
>>> A = np.array([[1,2], [3, 4], [5,6]])
>>> A.shape
(3, 2)
>>> B = np.array([7,8])
>>> B.shape
(2,)
>>> np.dot(A, B)
array([23, 53, 83])
```

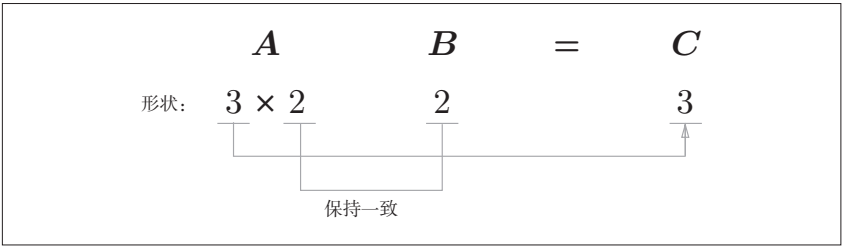

图3-13 A是二维矩阵、B是一维数组时，也要保持对应维度的元素个数一致

3.3.3 神经网络的内积

下面我们使用NumPy矩阵来实现神经网络。这里我们以图3-14中的简单神经网络为对象。这个神经网络省略了偏置和激活函数，只有权重。

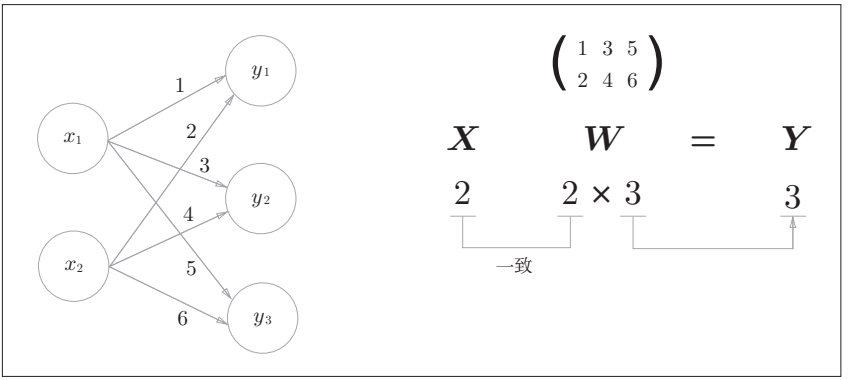

图3-14 通过矩阵的乘积进行神经网络的运算

实现该神经网络时，要注意$\boldsymbol{X}$、$\boldsymbol{W}$、$\boldsymbol{Y}$的形状，特别是$\boldsymbol{X}$和$\boldsymbol{W}$的对应维度的元素个数是否一致，这一点很重要。

```
>>> X = np.array([1, 2])
>>> X.shape
(2,)
>>> W = np.array([[1, 3, 5], [2, 4, 6]])
>>> print(W)
[[1 3 5]
 [2 4 6]]
>>> W.shape
```

```
(2, 3)
>>> Y = np.dot(X, W)
>>> print(Y)
[ 5  11  17]
```

如上所示，使用np.dot（多维数组的点积），可以一次性计算出$\boldsymbol{Y}$的结果。这意味着，即便$\boldsymbol{Y}$的元素个数为100或1000，也可以通过一次运算就计算出结果！如果不使用np.dot，就必须单独计算$\boldsymbol{Y}$的每一个元素（或者说必须使用for语句），非常麻烦。因此，通过矩阵的乘积一次性完成计算的技巧，在实现的层面上可以说是非常重要的。

3.4 3层神经网络的实现

现在我们来进行神经网络的实现。这里我们以图3-15的3层神经网络为对象，实现从输入到输出的（前向）处理。在代码实现方面，使用上一节介绍的NumPy多维数组。巧妙地使用NumPy数组，可以用很少的代码完成神经网络的前向处理。

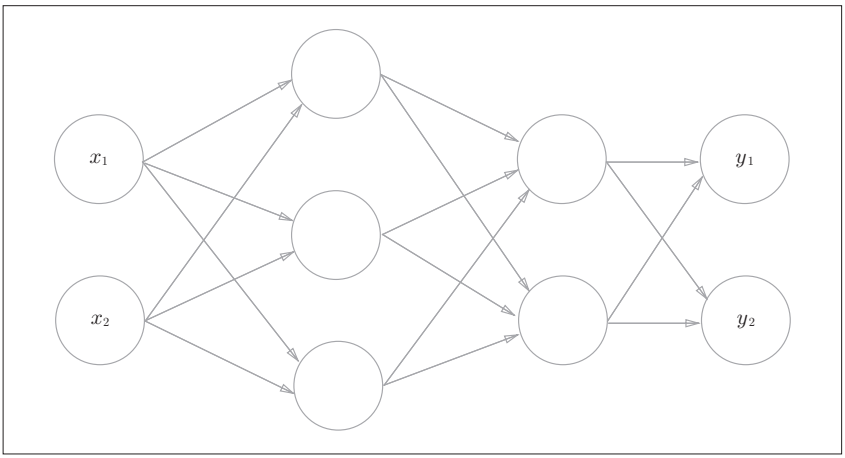

图3-15 3层神经网络：输入层（第0层）有2个神经元，第1个隐藏层（第1层）有3个神经元，第2个隐藏层（第2层）有2个神经元，输出层（第3层）有2个神经元

3.4.1 符号确认

在介绍神经网络中的处理之前，我们先导入 $w_{12}^{(1)}$、$a_1^{(1)}$ 等符号。这些符号可能看上去有些复杂，不过因为只在本节使用，稍微读一下就跳过去也问题不大。

本节的重点是神经网络的运算可以作为矩阵运算打包进行。因为神经网络各层的运算是通过矩阵的乘法运算打包进行的（从宏观视角来考虑），所以即便忘了（未记忆）具体的符号规则，也不影响理解后面的内容。

我们先从定义符号开始。请看图3-16。图3-16中只突出显示了从输入层神经元 x_2 到后一层的神经元 $a_1^{(1)}$ 的权重。

如图3-16所示，权重和隐藏层的神经元的右上角有一个“(1)”，它表示权重和神经元的层号（即第1层的权重、第1层的神经元）。此外，权重的右下角有两个数字，它们是后一层的神经元和前一层的神经元的索引号。比如，$w_{12}^{(1)}$ 表示前一层的第2个神经元 x_2 到后一层的第1个神经元 $a_1^{(1)}$ 的权重。权重右下角按照“后一层的索引号、前一层的索引号”的顺序排列。

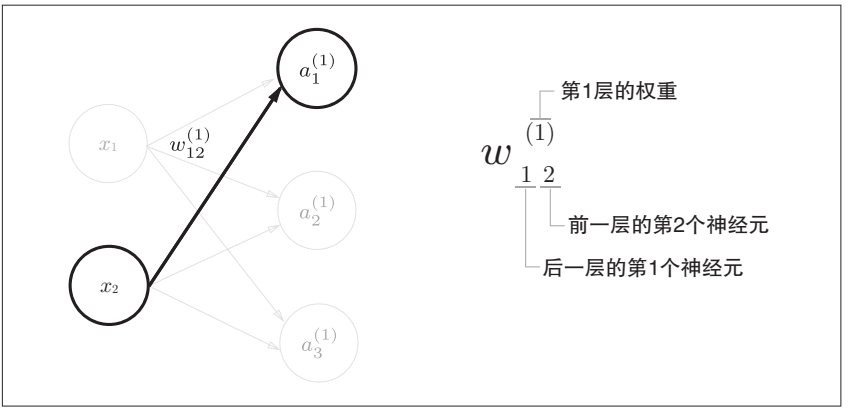

图3-16 权重的符号

3.4.2 各层间信号传递的实现

现在看一下从输入层到第1层的第1个神经元的信号传递过程，如图3-17所示。

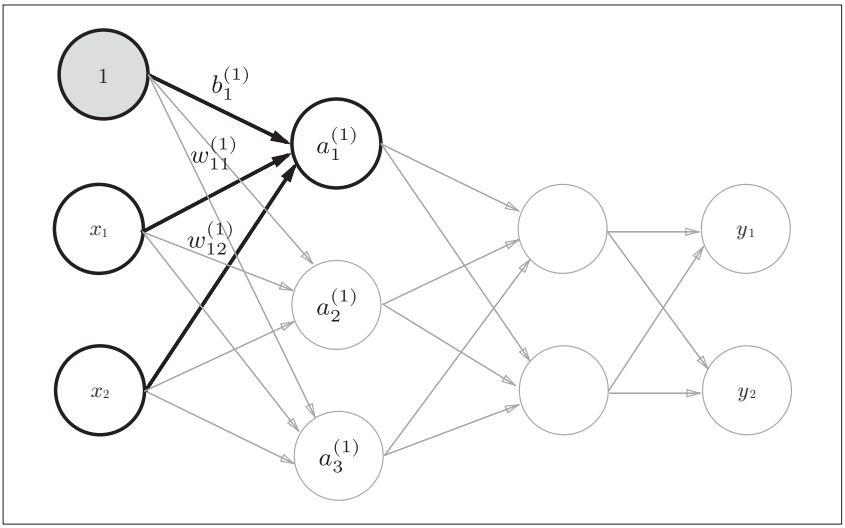

图3-17 从输入层到第1层的信号传递

图3-17中增加了表示偏置的神经元“1”。请注意，偏置的右下角的索引号只有一个。这是因为前一层的偏置神经元（神经元“1”）只有一个[①]。

为了确认前面的内容，现在用数学式表示 $a_1^{(1)}$。$a_1^{(1)}$ 通过加权信号和偏置的和按如下方式进行计算。

$$a_1^{(1)} = w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + b_1^{(1)} \tag{3.8}$$

① 任何前一层的偏置神经元“1”都只有一个。偏置权重的数量取决于后一层的神经元的数量（不包括后一层的偏置神经元“1”）。——译者注

此外，如果使用矩阵的乘法运算，则可以将第1层的加权和表示成下面的式(3.9)。

$$\boldsymbol{A}^{(1)} = \boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{B}^{(1)} \tag{3.9}$$

其中，$\boldsymbol{A}^{(1)}$、$\boldsymbol{X}$、$\boldsymbol{B}^{(1)}$、$\boldsymbol{W}^{(1)}$如下所示。

$$\boldsymbol{A}^{(1)} = \begin{pmatrix} a_1^{(1)} & a_2^{(1)} & a_3^{(1)} \end{pmatrix},\ \boldsymbol{X} = \begin{pmatrix} x_1 & x_2 \end{pmatrix},\ \boldsymbol{B}^{(1)} = \begin{pmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \end{pmatrix}$$

$$\boldsymbol{W}^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

下面我们用NumPy多维数组来实现式(3.9)，这里将输入信号、权重、偏置设置成任意值。

```
X = np.array([1.0, 0.5])
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])

print(W1.shape) # (2, 3)
print(X.shape)  # (2,)
print(B1.shape) # (3,)

A1 = np.dot(X, W1) + B1
```

这个运算和上一节进行的运算是一样的。`W1`是2×3的数组，`X`是元素个数为2的一维数组。这里，`W1`和`X`的对应维度的元素个数也保持了一致。

接下来，我们观察第1层中激活函数的计算过程。如果把这个计算过程用图来表示的话，则如图3-18所示。

如图3-18所示，隐藏层的加权和(加权信号和偏置的总和)用a表示，被激活函数转换后的信号用z表示。此外，图中$h()$表示激活函数，这里我们使用的是sigmoid函数。用Python来实现，代码如下所示。

```
Z1 = sigmoid(A1)

print(A1) # [0.3, 0.7, 1.1]
print(Z1) # [0.57444252, 0.66818777, 0.75026011]
```

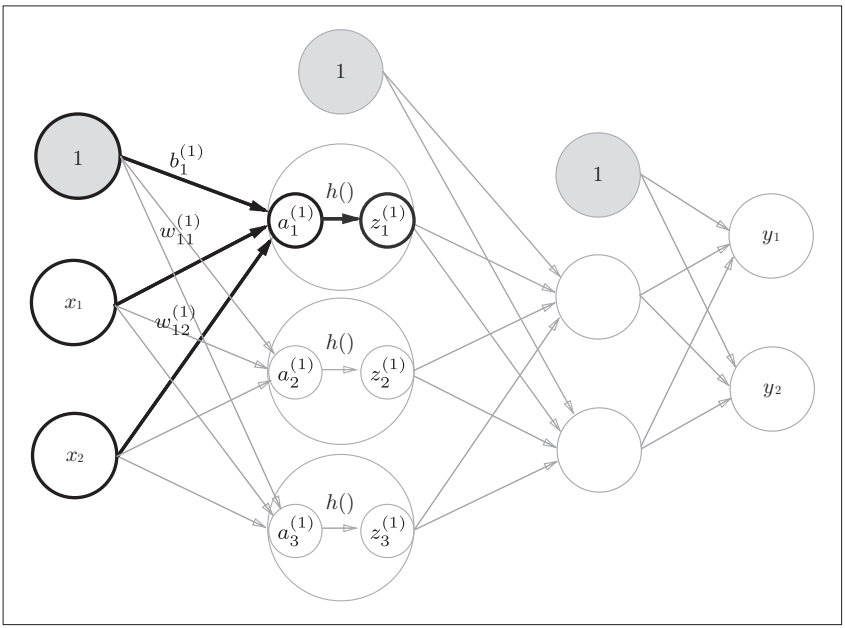

图3-18 从输入层到第1层的信号传递

这个sigmoid()函数就是之前定义的那个函数。它会接收NumPy数组，并返回元素个数相同的NumPy数组。

下面，我们来实现第1层到第2层的信号传递（图3-19）。

```
W2 = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
B2 = np.array([0.1, 0.2])

print(Z1.shape) # (3,)
print(W2.shape) # (3, 2)
print(B2.shape) # (2,)

A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)
```

除了第1层的输出（Z1）变成了第2层的输入这一点以外，这个实现和刚才的代码完全相同。由此可知，通过使用NumPy数组，可以将层到层的信号传递过程简单地写出来。

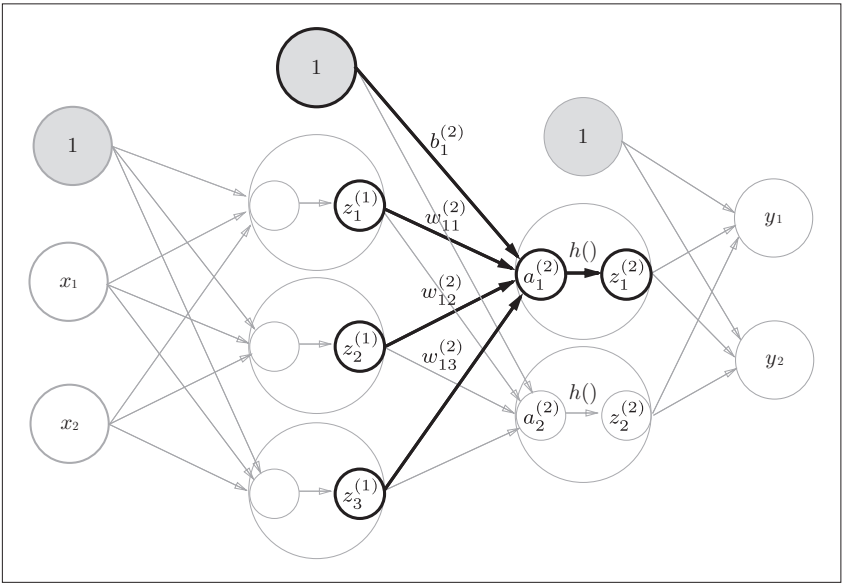

图3-19 第1层到第2层的信号传递

最后是第2层到输出层的信号传递(图3-20)。输出层的实现也和之前的实现基本相同。不过，最后的激活函数和之前的隐藏层有所不同。

```
def identity_function(x):
    return x

W3 = np.array([[0.1, 0.3], [0.2, 0.4]])
B3 = np.array([0.1, 0.2])

A3 = np.dot(Z2, W3) + B3
Y = identity_function(A3) # 或者Y = A3
```

这里我们定义了identity_function()函数(也称为“恒等函数”)，并将其作为输出层的激活函数。恒等函数会将输入按原样输出，因此，这个例子中没有必要特意定义identity_function()。这里这样实现只是为了和之前的流程保持统一。另外，图3-20中，输出层的激活函数用$\sigma()$表示，不同于隐藏层的激活函数$h()$(σ读作sigma)。

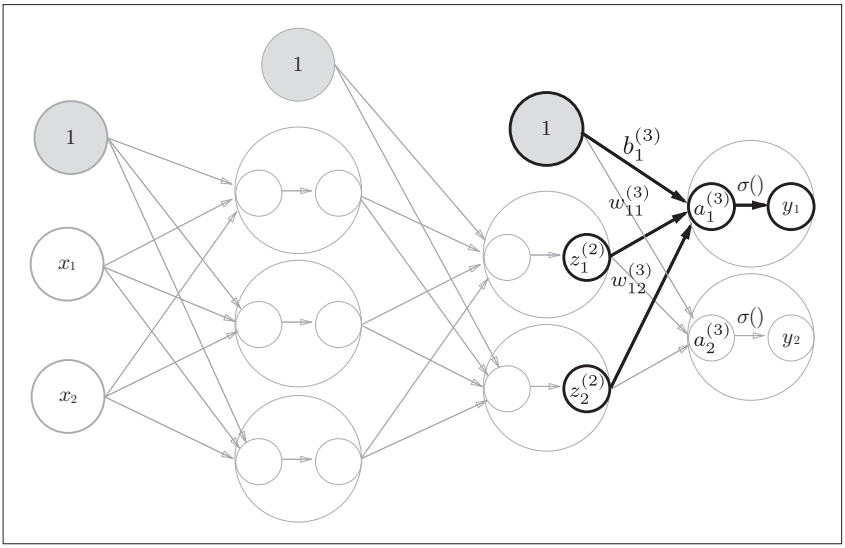

图3-20 从第2层到输出层的信号传递

输出层所用的激活函数，要根据求解问题的性质决定。一般地，回归问题可以使用恒等函数，二元分类问题可以使用sigmoid函数，多元分类问题可以使用softmax函数。关于输出层的激活函数，我们将在下一节详细介绍。

3.4.3 代码实现小结

至此，我们已经介绍完了3层神经网络的实现。现在我们把之前的代码实现全部整理一下。这里，我们按照神经网络的实现惯例，只把权重记为大写字母`W1`，其他的（偏置或中间结果等）都用小写字母表示。

```
def init_network():
    network = {}
    network['W1'] = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['W2'] = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
    network['b2'] = np.array([0.1, 0.2])
    network['W3'] = np.array([[0.1, 0.3], [0.2, 0.4]])
```

```
    network['b3'] = np.array([0.1, 0.2])

    return network

def forward(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = identity_function(a3)

    return y

network = init_network()
x = np.array([1.0, 0.5])
y = forward(network, x)
print(y) # [ 0.31682708  0.69627909]
```

这里定义了 `init_network()` 和 `forward()` 函数。`init_network()` 函数会进行权重和偏置的初始化，并将它们保存在字典变量 `network` 中。这个字典变量 `network` 中保存了每一层所需的参数（权重和偏置）。`forward()` 函数中则封装了将输入信号转换为输出信号的处理过程。

另外，这里出现了 forward（前向）一词，它表示的是从输入到输出方向的传递处理。后面在进行神经网络的训练时，我们将介绍后向（backward，从输出到输入方向）的处理。

至此，神经网络的前向处理的实现就完成了。通过巧妙地使用 NumPy 多维数组，我们高效地实现了神经网络。

3.5 输出层的设计

神经网络可以用在分类问题和回归问题上，不过需要根据情况改变输出层的激活函数。一般而言，回归问题用恒等函数，分类问题用 softmax 函数。

机器学习的问题大致可以分为分类问题和回归问题。分类问题是数据属于哪一个类别的问题。比如，区分图像中的人是男性还是女性的问题就是分类问题。而回归问题是根据某个输入预测一个（连续的）数值的问题。比如，根据一个人的图像预测这个人的体重的问题就是回归问题（类似“57.4kg”这样的预测）。

3.5.1 恒等函数和softmax函数

恒等函数会将输入按原样输出，对于输入的信息，不加以任何改动地直接输出。因此，在输出层使用恒等函数时，输入信号会原封不动地被输出。另外，将恒等函数的处理过程用之前的神经网络图来表示的话，则如图3-21所示。和前面介绍的隐藏层的激活函数一样，恒等函数进行的转换处理可以用一根箭头来表示。

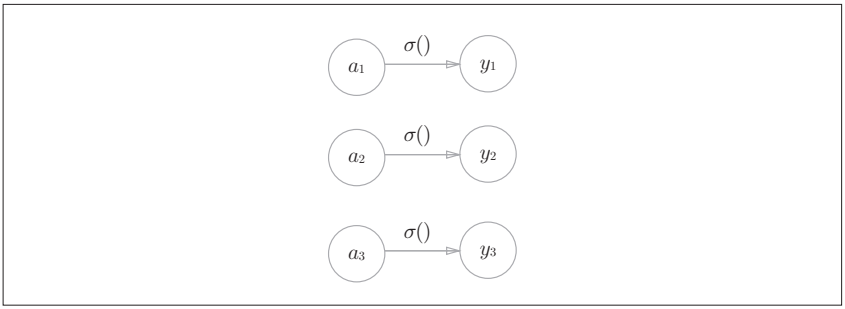

图3-21 恒等函数

分类问题中使用的softmax函数可以用下面的式（3.10）表示。

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)} \tag{3.10}$$

$\exp(x)$是表示e^x的指数函数（e是纳皮尔常数2.7182$\cdots$）。式（3.10）表示假设输出层共有n个神经元，计算第k个神经元的输出y_k。如式（3.10）所示，softmax函数的分子是输入信号a_k的指数函数，分母是所有输入信号的指数函数的和。

用图表示softmax函数的话，如图3-22所示。图3-22中，softmax函数的输出通过箭头与所有的输入信号相连。这是因为，从式(3.10)可以看出，输出层的各个神经元都受到所有输入信号的影响。

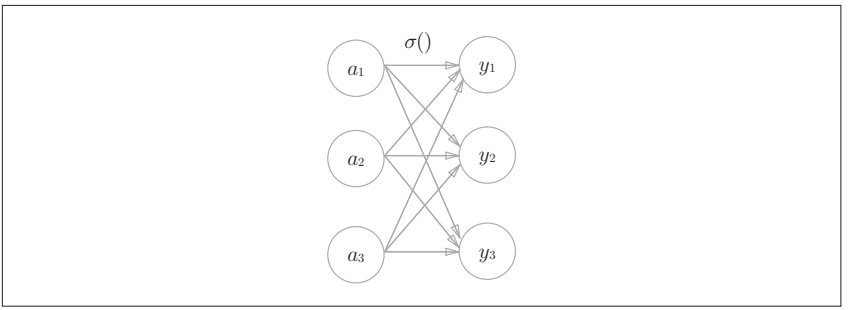

图3-22 softmax函数

现在我们来实现softmax函数。在这个过程中，我们将使用Python解释器逐一确认结果。

```
>>> a = np.array([0.3, 2.9, 4.0])
>>>
>>> exp_a = np.exp(a) # 指数函数
>>> print(exp_a)
[  1.34985881  18.17414537  54.59815003]
>>>
>>> sum_exp_a = np.sum(exp_a) # 指数函数的和
>>> print(sum_exp_a)
74.1221542102
>>>
>>> y = exp_a / sum_exp_a
>>> print(y)
[ 0.01821127  0.24519181  0.73659691]
```

这个Python实现是完全依照式(3.10)进行的，所以不需要特别的解释。考虑到后面还要使用softmax函数，这里我们把它定义成如下的Python函数。

```
def softmax(a):
    exp_a = np.exp(a)
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a

    return y
```

3.5.2 实现softmax函数时的注意事项

上面的softmax函数的实现虽然正确描述了式(3.10)，但在计算机的运算上有一定的缺陷。这个缺陷就是溢出问题。softmax函数的实现中要进行指数函数的运算，但是此时指数函数的值很容易变得非常大。比如，e^{10}的值会超过20000，e^{100}会变成一个后面有40多个0的超大值，e^{1000}的结果会返回一个表示无穷大的inf。如果在这些超大值之间进行除法运算，结果会出现“不确定”的情况。

计算机处理“数”时，数值必须在4字节或8字节的有限数据宽度内。这意味着数存在有效位数，也就是说，可以表示的数值范围是有限的。因此，会出现超大值无法表示的问题。这个问题称为溢出，在进行计算机的运算时必须(常常)注意。

softmax函数的实现可以像式(3.11)这样进行改进。

$$
\begin{aligned}
y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n}\exp(a_i)} &= \frac{\mathrm{C}\exp(a_k)}{\mathrm{C}\sum_{i=1}^{n}\exp(a_i)} \\
&= \frac{\exp(a_k + \log \mathrm{C})}{\sum_{i=1}^{n}\exp(a_i + \log \mathrm{C})} \\
&= \frac{\exp(a_k + \mathrm{C}')}{\sum_{i=1}^{n}\exp(a_i + \mathrm{C}')}
\end{aligned}
\tag{3.11}
$$

首先，式(3.11)在分子和分母上都乘上C这个任意的常数(因为同时对分母和分子乘以相同的常数，所以计算结果不变)。然后，把这个C移动到指数函数(exp)中，记为$\log \mathrm{C}$。最后，把$\log \mathrm{C}$替换为另一个符号C'。

式(3.11)说明，在进行softmax的指数函数的运算时，加上(或者减去)某个常数并不会改变运算的结果。这里的C'可以使用任何值，但是为了防止溢出，一般会使用输入信号中的最大值。我们来看一个具体的例子。

```
>>> a = np.array([1010, 1000, 990])
>>> np.exp(a) / np.sum(np.exp(a)) # softmax函数的运算
array([ nan,  nan,  nan])         # 没有被正确计算
>>>
>>> c = np.max(a) # 1010
>>> a - c
array([  0, -10, -20])
>>>
>>> np.exp(a - c) / np.sum(np.exp(a - c))
array([  9.99954600e-01,   4.53978686e-05,   2.06106005e-09])
```

如该例所示，通过减去输入信号中的最大值（上例中的c），我们发现原本为nan（not a number，不确定）的地方，现在被正确计算了。综上，我们可以像下面这样实现softmax函数。

```
def softmax(a):
    c = np.max(a)
    exp_a = np.exp(a - c) # 溢出对策
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a

    return y
```

3.5.3 softmax函数的特征

使用softmax()函数，可以按如下方式计算神经网络的输出。

```
>>> a = np.array([0.3, 2.9, 4.0])
>>> y = softmax(a)
>>> print(y)
[ 0.01821127  0.24519181  0.73659691]
>>> np.sum(y)
1.0
```

如上所示，softmax函数的输出是0.0到1.0之间的实数。并且，softmax函数的输出值的总和是1。输出总和为1是softmax函数的一个重要性质。正因为有了这个性质，我们才可以把softmax函数的输出解释为“概率”。

比如，上面的例子可以解释成y[0]的概率是0.018（1.8％），y[1]的概率是0.245（24.5％），y[2]的概率是0.737（73.7％）。从概率的结果来看，可以说“因为第2个元素的概率最高，所以答案是第2个类别”。而且，还可以回

答“有74％的概率是第2个类别，有25％的概率是第1个类别，有1％的概率是第0个类别”。也就是说，通过使用softmax函数，我们可以用概率的（统计的）方法处理问题。

这里需要注意的是，即便使用了softmax函数，各个元素之间的大小关系也不会改变。这是因为指数函数（$y = \exp(x)$）是单调递增函数。实际上，上例中a的各元素的大小关系和y的各元素的大小关系并没有改变。比如，a的最大值是第2个元素，y的最大值也仍是第2个元素。

一般而言，神经网络只把输出值最大的神经元所对应的类别作为识别结果。并且，即便使用softmax函数，输出值最大的神经元的位置也不会变。因此，神经网络在进行分类时，输出层的softmax函数可以省略。在实际的问题中，由于指数函数的运算需要一定的计算机运算量，因此输出层的softmax函数一般会被省略。

求解机器学习问题的步骤可以分为“学习”[①]和“推理”两个阶段。首先，在学习阶段进行模型的学习[②]，然后，在推理阶段，用学到的模型对未知的数据进行推理（分类）。如前所述，推理阶段一般会省略输出层的softmax函数。在输出层使用softmax函数是因为它和神经网络的学习有关系（详细内容请参考下一章）。

3.5.4 输出层的神经元数量

输出层的神经元数量需要根据待解决的问题来决定。对于分类问题，输出层的神经元数量一般设定为类别的数量。比如，对于某个输入图像，预测是图中的数字0到9中的哪一个的问题（10类别分类问题），可以像图3-23这样，将输出层的神经元设定为10个。

如图3-23所示，在这个例子中，输出层的神经元从上往下依次对应数字$0, 1, \cdots, 9$。此外，图中输出层的神经元的值用不同的灰度表示。这个例子

① “学习”也称为“训练”，为了强调算法从数据中学习模型，本书使用“学习”一词。——译者注

② 这里的“学习”是指使用训练数据、自动调整参数的过程，具体请参考第4章。——译者注

中神经元y_2颜色最深，输出的值最大。这表明这个神经网络预测的是y_2对应的类别，也就是“2”。

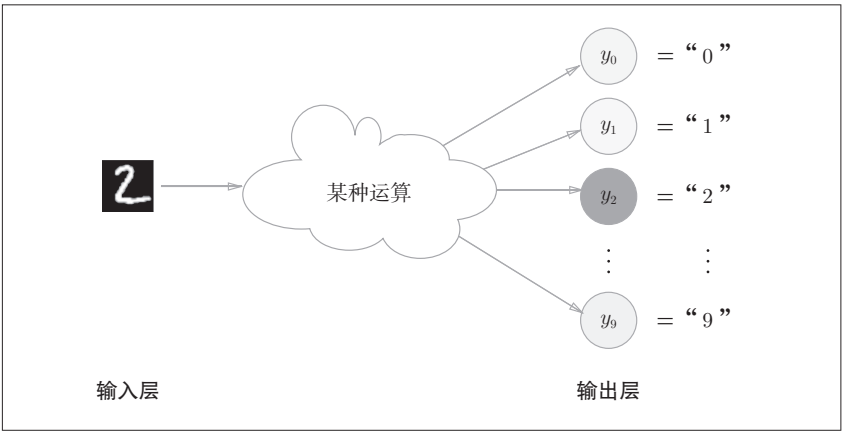

图3-23 输出层的神经元对应各个数字

3.6 手写数字识别

介绍完神经网络的结构之后，现在我们来试着解决实际问题。这里我们来进行手写数字图像的分类。假设学习已经全部结束，我们使用学习到的参数，先实现神经网络的“推理处理”。这个推理处理也称为神经网络的**前向传播**(forward propagation)。

和求解机器学习问题的步骤(分成学习和推理两个阶段进行)一样，使用神经网络解决问题时，也需要首先使用训练数据(学习数据)进行权重参数的学习；进行推理时，使用刚才学习到的参数，对输入数据进行分类。

3.6.1 MNIST数据集

这里使用的数据集是MNIST手写数字图像集。MNIST是机器学习领域最有名的数据集之一，被应用于从简单的实验到发表的论文研究等各种场合。实际上，在阅读图像识别或机器学习的论文时，MNIST数据集经常作为实验用的数据出现。

MNIST数据集是由0到9的数字图像构成的（图3-24）。训练图像有6万张，测试图像有1万张，这些图像可以用于学习和推理。MNIST数据集的一般使用方法是，先用训练图像进行学习，再用学习到的模型度量能在多大程度上对测试图像进行正确的分类。

图3-24 MNIST图像数据集的例子

MNIST的图像数据是28像素 × 28像素的灰度图像（1通道），各个像素的取值在0到255之间。每个图像数据都相应地标有“7”“2”“1”等标签。

本书提供了便利的Python脚本`mnist.py`，该脚本支持从下载MNIST数据集到将这些数据转换成NumPy数组等处理（`mnist.py`在`dataset`目录下）。使用`mnist.py`时，当前目录必须是`ch01`、`ch02`、`ch03`、…、`ch08`目录中的一个。使用`mnist.py`中的`load_mnist()`函数，就可以按下述方式轻松读入MNIST数据。

```
import sys, os
sys.path.append(os.pardir) # 为了导入父目录中的文件而进行的设定
from dataset.mnist import load_mnist

# 第一次调用会花费几分钟……
(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True,
normalize=False)

# 输出各个数据的形状
print(x_train.shape) # (60000, 784)
```

```
print(t_train.shape) # (60000,)
print(x_test.shape)  # (10000, 784)
print(t_test.shape)  # (10000,)
```

首先，为了导入父目录中的文件，进行相应的设定[①]。然后，导入dataset/mnist.py中的load_mnist函数。最后，使用load_mnist函数，读入MNIST数据集。第一次调用load_mnist函数时，因为要下载MNIST数据集，所以需要接入网络。第2次及以后的调用只需读入保存在本地的文件（pickle文件）即可，因此处理所需的时间非常短。

用来读入MNIST图像的文件在本书提供的源代码的dataset目录下。并且，我们假定了这个MNIST数据集只能从ch01、ch02、ch03、…、ch08目录中使用，因此，使用时需要从父目录（dataset目录）中导入文件，为此需要添加sys.path.append(os.pardir)语句。

load_mnist函数以“**(训练图像,训练标签)，(测试图像，测试标签)**”的形式返回读入的MNIST数据。此外，还可以像load_mnist(normalize=True, flatten=True, one_hot_label=False)这样，设置3个参数。第1个参数normalize设置是否将输入图像正规化为0.0～1.0的值。如果将该参数设置为False，则输入图像的像素会保持原来的0～255。第2个参数flatten设置是否展开输入图像（变成一维数组）。如果将该参数设置为False，则输入图像为1 × 28 × 28的三维数组；若设置为True，则输入图像会保存为由784个元素构成的一维数组。第3个参数one_hot_label设置是否将标签保存为one-hot表示（one-hot representation）。one-hot表示是仅正确解标签为1，其余皆为0的数组，就像[0,0,1,0,0,0,0,0,0,0]这样。当one_hot_label为False时，只是像7、2这样简单保存正确解标签；当one_hot_label为True时，标签则保存为one-hot表示。

---

① 观察本书源代码可知，上述代码在mnist_show.py文件中。mnist_show.py文件的当前目录是ch03，但包含load_mnist()函数的mnist.py文件在dataset目录下。因此，mnist_show.py文件不能跨目录直接导入mnist.py文件。sys.path.append(os.pardir)语句实际上是把父目录deep-learning-from-scratch加入到sys.path（Python的搜索模块的路径集）中，从而可以导入deep-learning-from-scratch下的任何目录（包括dataset目录）中的任何文件。——译者注

Python有pickle这个便利的功能。这个功能可以将程序运行中的对象保存为文件。如果加载保存过的pickle文件，可以立刻复原之前程序运行中的对象。用于读入MNIST数据集的`load_mnist()`函数内部也使用了pickle功能（在第2次及以后读入时）。利用pickle功能，可以高效地完成MNIST数据的准备工作。

现在，我们试着显示MNIST图像，同时也确认一下数据。图像的显示使用PIL（Python Image Library）模块。执行下述代码后，训练图像的第一张就会显示出来，如图3-25所示（源代码在`ch03/mnist_show.py`中）。

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist
from PIL import Image

def img_show(img):
    pil_img = Image.fromarray(np.uint8(img))
    pil_img.show()

(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True,
normalize=False)
img = x_train[0]
label = t_train[0]
print(label) # 5

print(img.shape)          # (784,)
img = img.reshape(28, 28) # 把图像的形状变成原来的尺寸
print(img.shape)          # (28, 28)

img_show(img)
```

这里需要注意的是，`flatten=True`时读入的图像是以一列（一维）NumPy数组的形式保存的。因此，显示图像时，需要把它变为原来的28像素 × 28像素的形状。可以通过`reshape()`方法的参数指定期望的形状，更改NumPy数组的形状。此外，还需要把保存为NumPy数组的图像数据转换为PIL用的数据对象，这个转换处理由`Image.fromarray()`来完成。

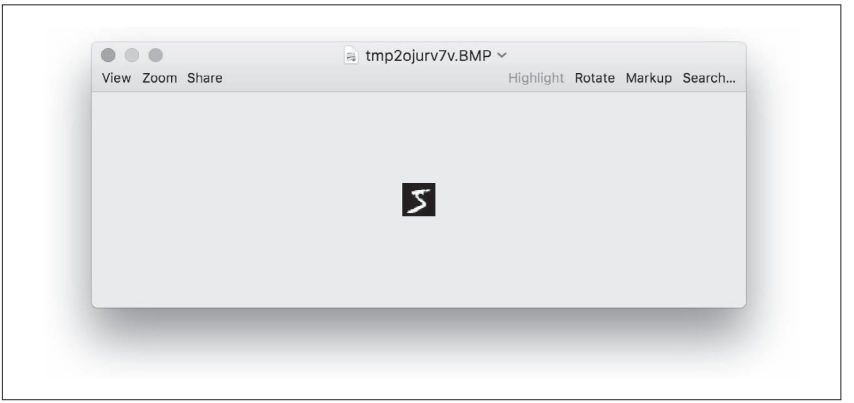

图3-25 显示MNIST图像

3.6.2 神经网络的推理处理

下面，我们对这个MNIST数据集实现神经网络的推理处理。神经网络的输入层有784个神经元，输出层有10个神经元。输入层的784这个数字来源于图像大小的$28 \times 28 = 784$，输出层的10这个数字来源于10类别分类（数字0到9，共10类别）。此外，这个神经网络有2个隐藏层，第1个隐藏层有50个神经元，第2个隐藏层有100个神经元。这个50和100可以设置为任何值。下面我们先定义`get_data()`、`init_network()`、`predict()`这3个函数（代码在`ch03/neuralnet_mnist.py`中）。

```
def get_data():
    (x_train, t_train), (x_test, t_test) = \
        load_mnist(normalize=True, flatten=True, one_hot_label=False)
    return x_test, t_test

def init_network():
    with open("sample_weight.pkl", 'rb') as f:
        network = pickle.load(f)

    return network

def predict(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']
```

```
    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = softmax(a3)

    return y
```

init_network()会读入保存在pickle文件sample_weight.pkl中的学习到的权重参数[①]。这个文件中以字典变量的形式保存了权重和偏置参数。剩余的2个函数，和前面介绍的代码实现基本相同，无需再解释。现在，我们用这3个函数来实现神经网络的推理处理。然后，评价它的**识别精度**(accuracy)，即能在多大程度上正确分类。

```
x, t = get_data()
network = init_network()

accuracy_cnt = 0
for i in range(len(x)):
    y = predict(network, x[i])
    p = np.argmax(y) # 获取概率最高的元素的索引
    if p == t[i]:
        accuracy_cnt += 1

print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

首先获得MNIST数据集，生成网络。接着，用for语句逐一取出保存在x中的图像数据，用predict()函数进行分类。predict()函数以NumPy数组的形式输出各个标签对应的概率。比如输出[0.1, 0.3, 0.2, ···, 0.04]的数组，该数组表示“0”的概率为0.1，“1”的概率为0.3，等等。然后，我们取出这个概率列表中的最大值的索引(第几个元素的概率最高)，作为预测结果。可以用np.argmax(x)函数取出数组中的最大值的索引，np.argmax(x)将获取被赋给参数x的数组中的最大值元素的索引。最后，比较神经网络所预测的答案和正确解标签，将回答正确的概率作为识别精度。

① 因为之前我们假设学习已经完成，所以学习到的参数被保存下来。假设保存在sample_weight.pkl文件中，在推理阶段，我们直接加载这些已经学习到的参数。——译者注

执行上面的代码后，会显示“Accuracy:0.9352”。这表示有93.52％的数据被正确分类了。目前我们的目标是运行学习到的神经网络，所以不讨论识别精度本身，不过以后我们会花精力在神经网络的结构和学习方法上，思考如何进一步提高这个精度。实际上，我们打算把精度提高到99％以上。

另外，在这个例子中，我们把load_mnist函数的参数normalize设置成了True。将normalize设置成True后，函数内部会进行转换，将图像的各个像素值除以255，使得数据的值在0.0～1.0的范围内。像这样把数据限定到某个范围内的处理称为**正规化**（normalization）。此外，对神经网络的输入数据进行某种既定的转换称为**预处理**（pre-processing）。这里，作为对输入图像的一种预处理，我们进行了正规化。

预处理在神经网络（深度学习）中非常实用，其有效性已在提高识别性能和学习的效率等众多实验中得到证明。在刚才的例子中，作为一种预处理，我们将各个像素值除以255，进行了简单的正规化。实际上，很多预处理都会考虑到数据的整体分布。比如，利用数据整体的均值或标准差，移动数据，使数据整体以0为中心分布，或者进行正规化，把数据的延展控制在一定范围内。除此之外，还有将数据整体的分布形状均匀化的方法，即数据**白化**（whitening）等。

3.6.3 批处理

以上就是处理MNIST数据集的神经网络的实现，现在我们来关注输入数据和权重参数的“形状”。再看一下刚才的代码实现。

下面我们使用Python解释器，输出刚才的神经网络的各层的权重的形状。

```
>>> x, _ = get_data()
>>> network = init_network()
>>> W1, W2, W3 = network['W1'], network['W2'], network['W3']
>>>
>>> x.shape
(10000, 784)
>>> x[0].shape
(784,)
>>> W1.shape
```

```
(784, 50)
>>> W2.shape
(50, 100)
>>> W3.shape
(100, 10)
```

我们通过上述结果来确认一下多维数组的对应维度的元素个数是否一致（省略了偏置）。用图表示的话，如图3-26所示。可以发现，多维数组的对应维度的元素个数确实是一致的。此外，我们还可以确认最终的结果是输出了元素个数为10 的一维数组。

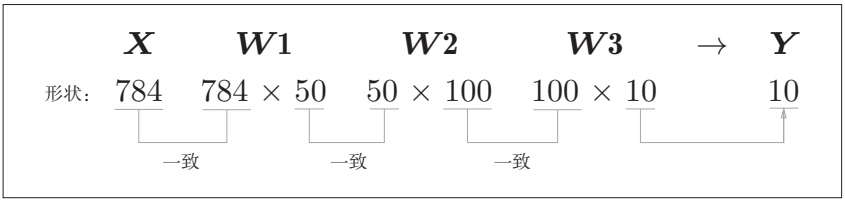

图3-26 数组形状的变化

从整体的处理流程来看，图3-26中，输入一个由784个元素（原本是一个28 × 28的二维数组）构成的一维数组后，输出一个有10个元素的一维数组。这是只输入一张图像数据时的处理流程。

现在我们来考虑打包输入多张图像的情形。比如，我们想用predict()函数一次性打包处理100张图像。为此，可以把$\boldsymbol{x}$的形状改为100 × 784，将100张图像打包作为输入数据。用图表示的话，如图3-27所示。

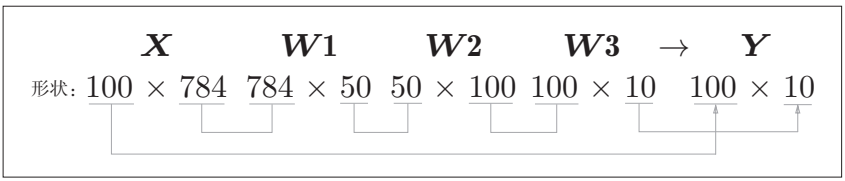

图3-27 批处理中数组形状的变化

如图3-27所示，输入数据的形状为100 × 784，输出数据的形状为100 × 10。这表示输入的100张图像的结果被一次性输出了。比如，x[0]和

y[0]中保存了第0张图像及其推理结果，x[1]和y[1]中保存了第1张图像及其推理结果，等等。

这种打包式的输入数据称为**批**（batch）。批有“捆”的意思，图像就如同纸币一样扎成一捆。

批处理对计算机的运算大有利处，可以大幅缩短每张图像的处理时间。那么为什么批处理可以缩短处理时间呢？这是因为大多数处理数值计算的库都进行了能够高效处理大型数组运算的最优化。并且，在神经网络的运算中，当数据传送成为瓶颈时，批处理可以减轻数据总线的负荷（严格地讲，相对于数据读入，可以将更多的时间用在计算上）。也就是说，批处理一次性计算大型数组要比分开逐步计算各个小型数组速度更快。

下面我们进行基于批处理的代码实现。这里用粗体显示与之前的实现的不同之处。

```
x, t = get_data()
network = init_network()

batch_size = 100 # 批数量
accuracy_cnt = 0

for i in range(0, len(x), batch_size):
    x_batch = x[i:i+batch_size]
    y_batch = predict(network, x_batch)
    p = np.argmax(y_batch, axis=1)
    accuracy_cnt += np.sum(p == t[i:i+batch_size])

print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

我们来逐个解释粗体的代码部分。首先是range()函数。range()函数若指定为range(start, end)，则会生成一个由start到end-1之间的整数构成的列表。若像range(start, end, step)这样指定3个整数，则生成的列表中的下一个元素会增加step指定的值。我们来看一个例子。

```
>>> list( range(0, 10) )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> list( range(0, 10, 3) )
[0, 3, 6, 9]
```

在range()函数生成的列表的基础上，通过x[i:i+batch_size]从输入数据中抽出批数据。x[i:i+batch_n]会取出从第i个到第i+batch_n个之间的数据。本例中是像x[0:100]、x[100:200]……这样，从头开始以100为单位将数据提取为批数据。

然后，通过argmax()获取值最大的元素的索引。不过这里需要注意的是，我们给定了参数axis=1。这指定了在100 × 10的数组中，沿着第1维方向（以第1维为轴）找到值最大的元素的索引（第0维对应第1个维度）[①]。这里也来看一个例子。

```
>>> x = np.array([[0.1, 0.8, 0.1], [0.3, 0.1, 0.6],
...     [0.2, 0.5, 0.3], [0.8, 0.1, 0.1]])
>>> y = np.argmax(x, axis=1)
>>> print(y)
[1 2 1 0]
```

最后，我们比较一下以批为单位进行分类的结果和实际的答案。为此，需要在NumPy数组之间使用比较运算符（==）生成由True/False构成的布尔型数组，并计算True的个数。我们通过下面的例子进行确认。

```
>>> y = np.array([1, 2, 1, 0])
>>> t = np.array([1, 2, 0, 0])
>>> print(y==t)
[True True False True]
>>> np.sum(y==t)
3
```

至此，基于批处理的代码实现就介绍完了。使用批处理，可以实现高速且高效的运算。下一章介绍神经网络的学习时，我们将把图像数据作为打包的批数据进行学习，届时也将进行和这里的批处理一样的代码实现。

① 矩阵的第0维是列方向，第1维是行方向。——译者注

3.7 小结

本章介绍了神经网络的前向传播。本章介绍的神经网络和上一章的感知机在信号的按层传递这一点上是相同的，但是，向下一个神经元发送信号时，改变信号的激活函数有很大差异。神经网络中使用的是平滑变化的sigmoid函数，而感知机中使用的是信号急剧变化的阶跃函数。这个差异对于神经网络的学习非常重要，我们将在下一章介绍。

本章所学的内容

- 神经网络中的激活函数使用平滑变化的sigmoid函数或ReLU函数。
- 通过巧妙地使用NumPy多维数组，可以高效地实现神经网络。
- 机器学习的问题大体上可以分为回归问题和分类问题。
- 关于输出层的激活函数，回归问题中一般用恒等函数，分类问题中一般用softmax函数。
- 分类问题中，输出层的神经元的数量设置为要分类的类别数。
- 输入数据的集合称为批。通过以批为单位进行推理处理，能够实现高速的运算。