

# [鱼书] 第3章 神经网络

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。

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。

## 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。

### 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**。

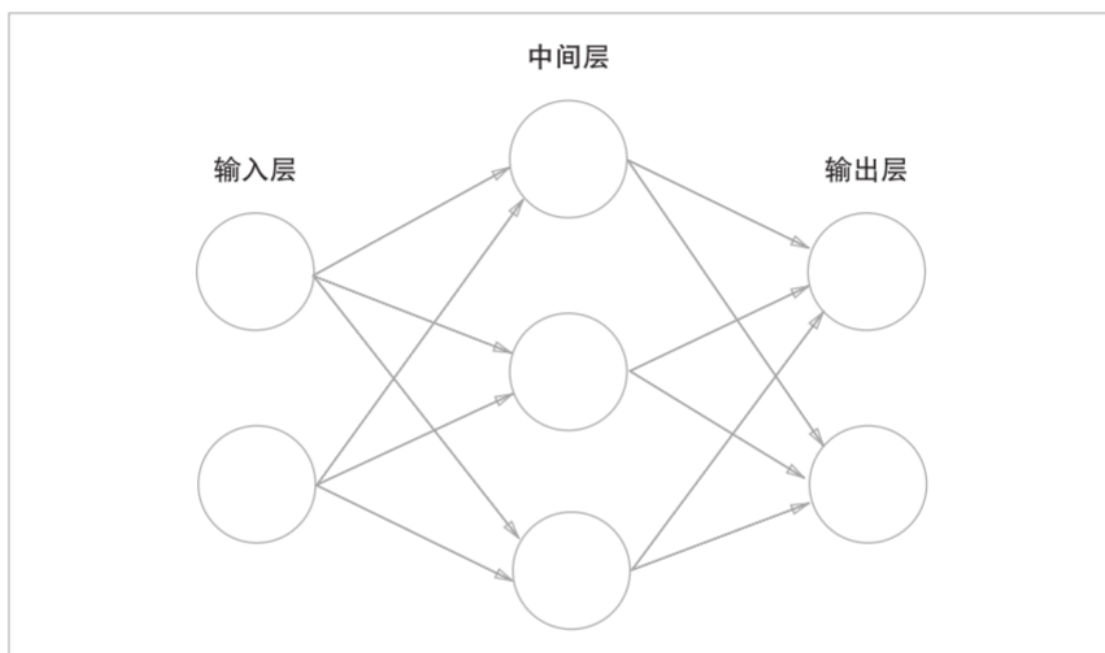


图 3-1 神经网络的例子

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？

## 复习感知机

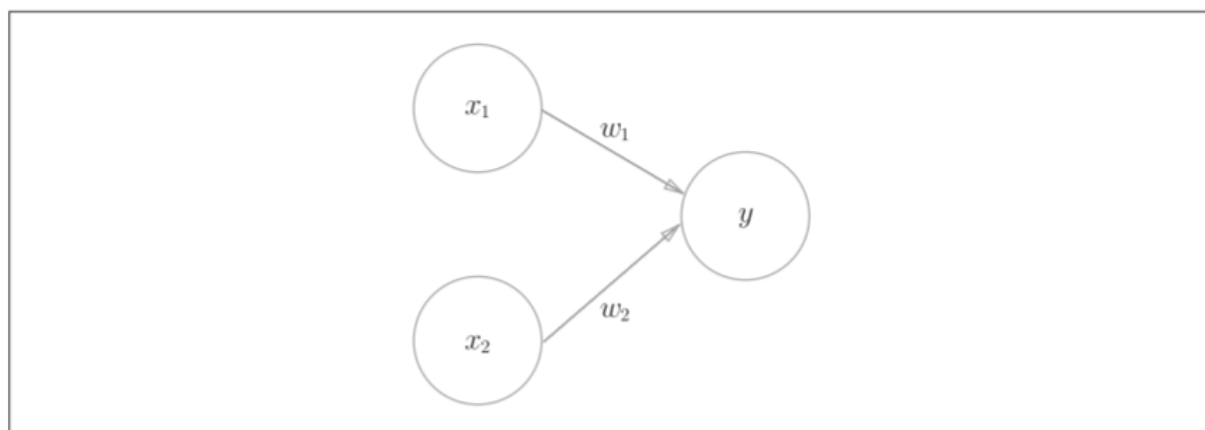


图 3-2 复习感知机

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。

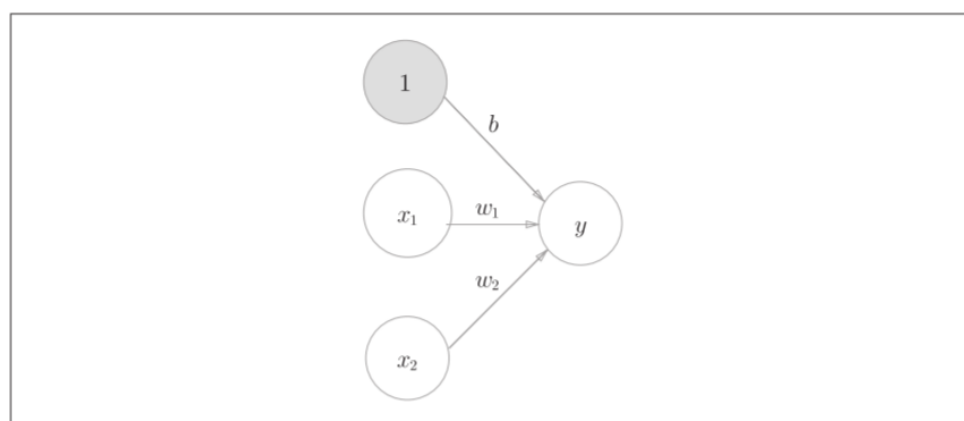


图 3-3 明确表示出偏置

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

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）。

$$y = h(b + w_1x_1 + w_2x_2) \quad (3.2)$$

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

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。

## 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。

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。

$$a = b + w_1x_1 + w_2x_2 \quad (3.4)$$

$$y = h(a) \quad (3.5)$$

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  $y$ 。

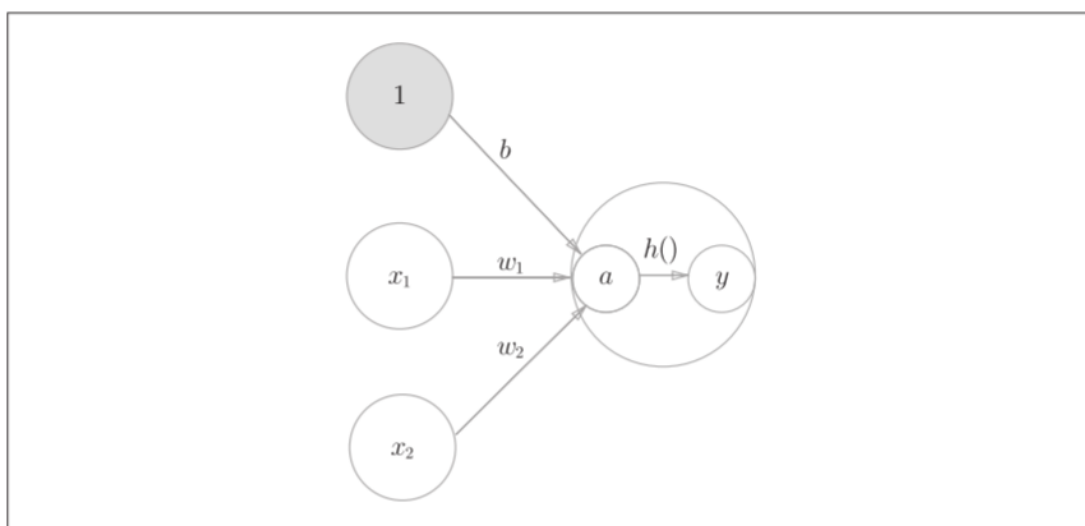


图3-4 明确显示激活函数的计算过程

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的模型。“多层感知机”是指神经网络，即使使用 sigmoid 函数（后述）等平滑的激活函数的多层网络。

## 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**”。可以说感知机中使用了阶跃函数作为激活函数。

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。

## sigmoid函数

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

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。

## 阶跃函数的实现

```
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。

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

最后，return中的布尔型数组，Python中将布尔型转换为int型后，True会转换为1，False会转换为0。

## 阶跃函数的图形

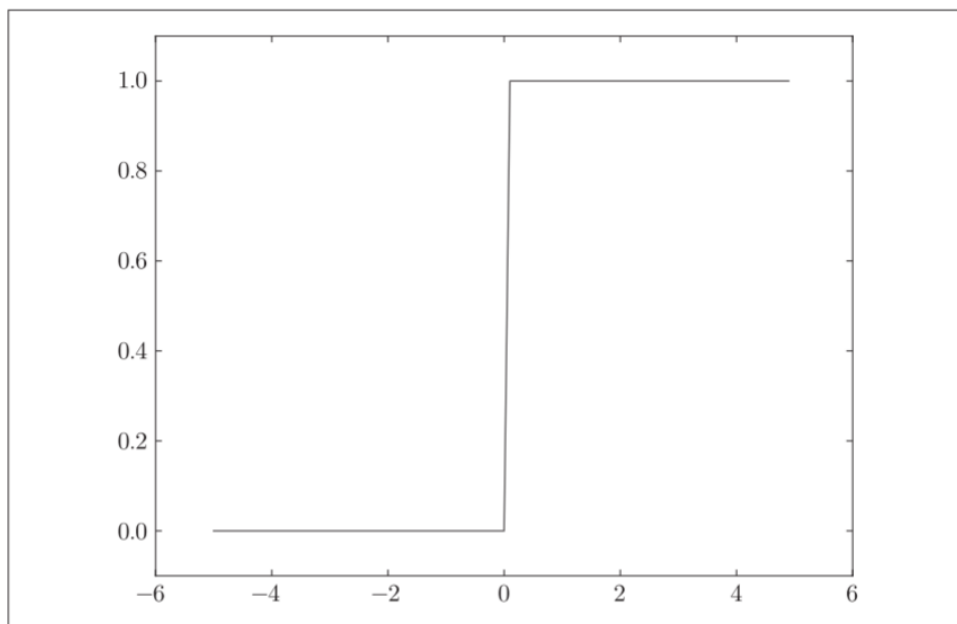


图 3-6 阶跃函数的图形

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。

## sigmoid函数的实现

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

- `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。
- 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。

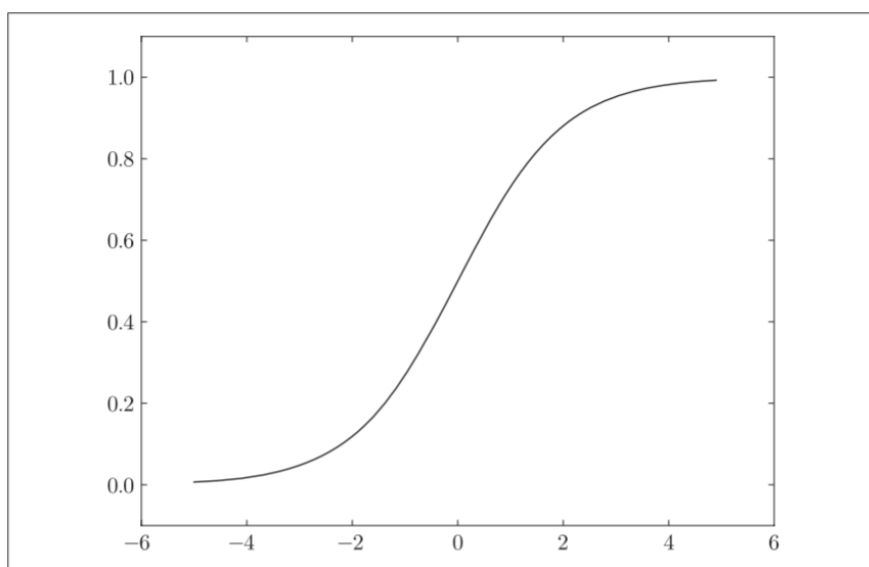


图 3-7 sigmoid 函数的图形

## sigmoid函数和阶跃函数的比较

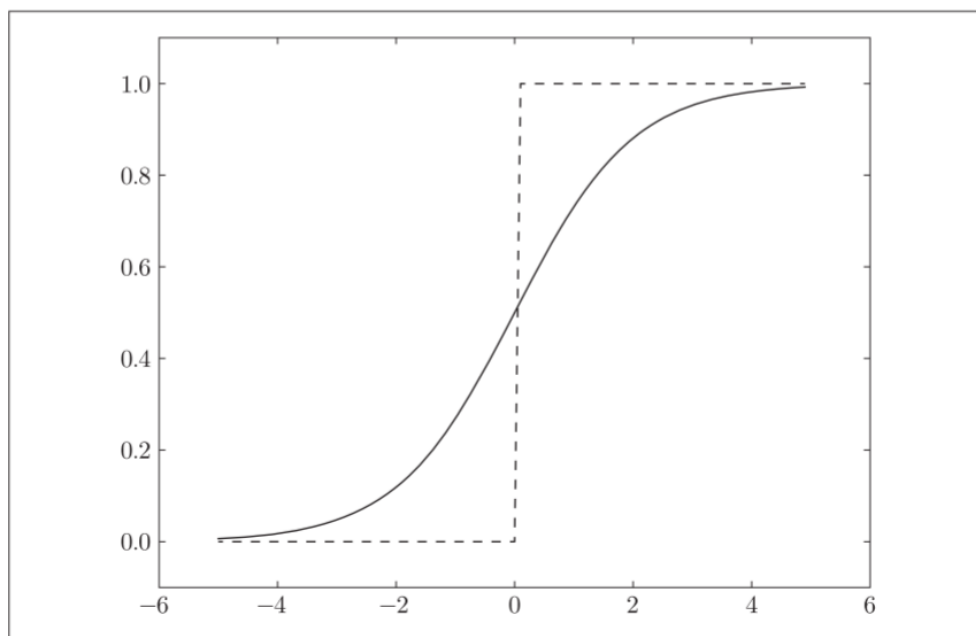


图 3-8 阶跃函数与 sigmoid 函数(虚线是阶跃函数)

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。

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。

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。

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。

## 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。

🏆 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。

**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”。

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A。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。

## ReLU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）。

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

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

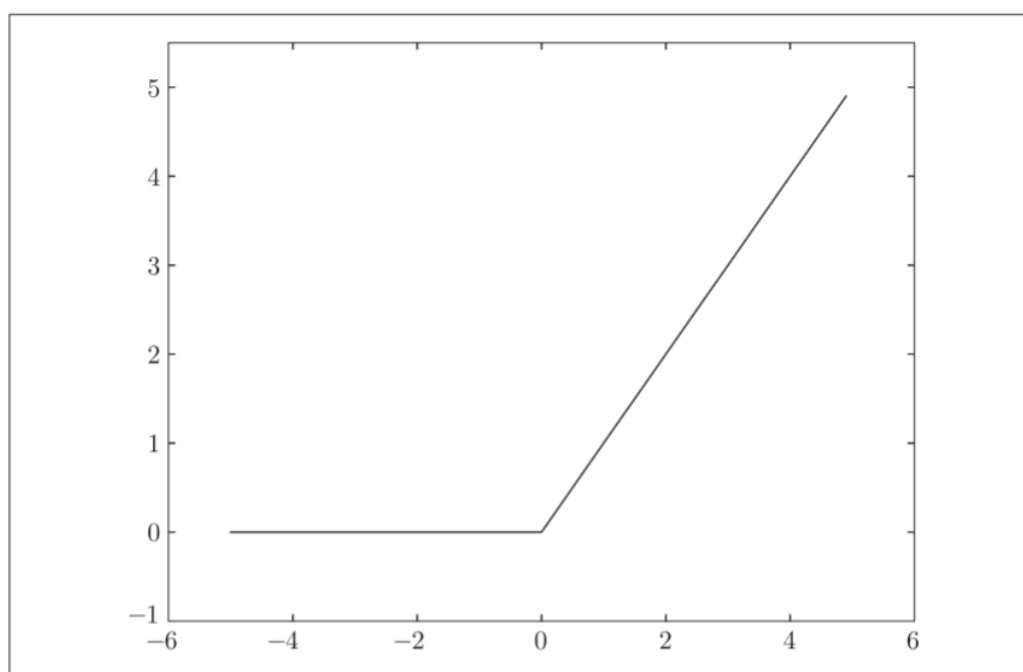


图3-9 ReLU函数

## 多维数组的运算

## 多维数组

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。

注意，这里的一维数组shape的结果也是个元组（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。

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

## 矩阵乘法

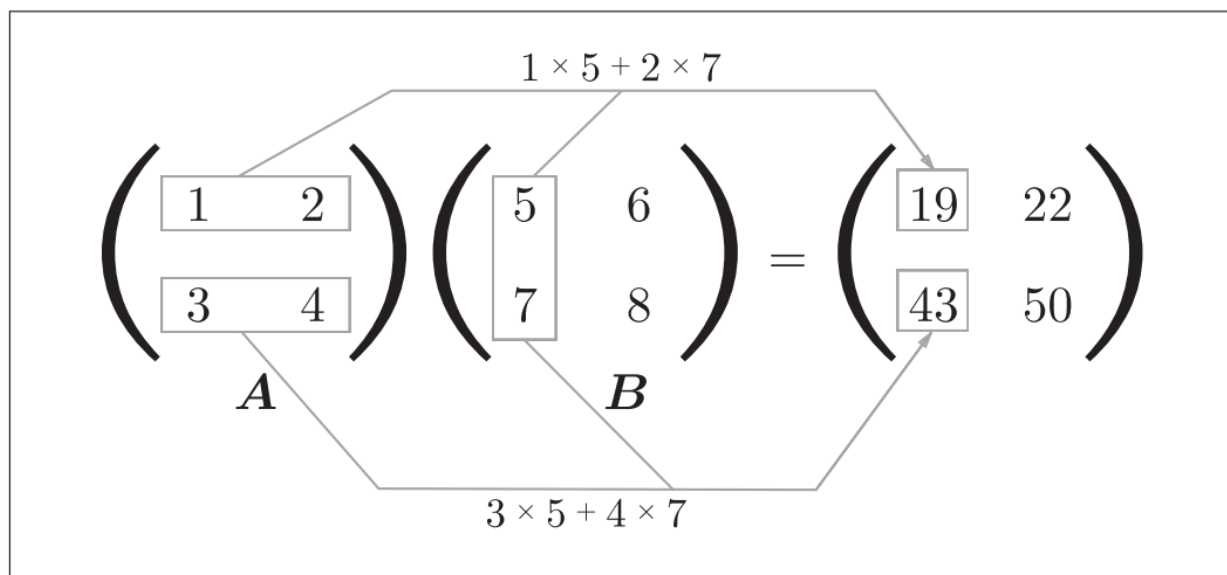


图 3-11 矩阵的乘积的计算方法

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。

在本书的数学标记中，矩阵将用黑斜体表示（比如，矩阵A），以区别于单个元素的标量（比如，a或b）。



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

A和B都是 $2 \times 2$ 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。

矩阵的乘积运算中，操作数（A、B）的顺序不同，结果也会不同。

矩阵A的第1维的元素个数（列数）必须和矩阵B的第0维的元素个数（行数）相等。另外，当A是二维矩阵、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。

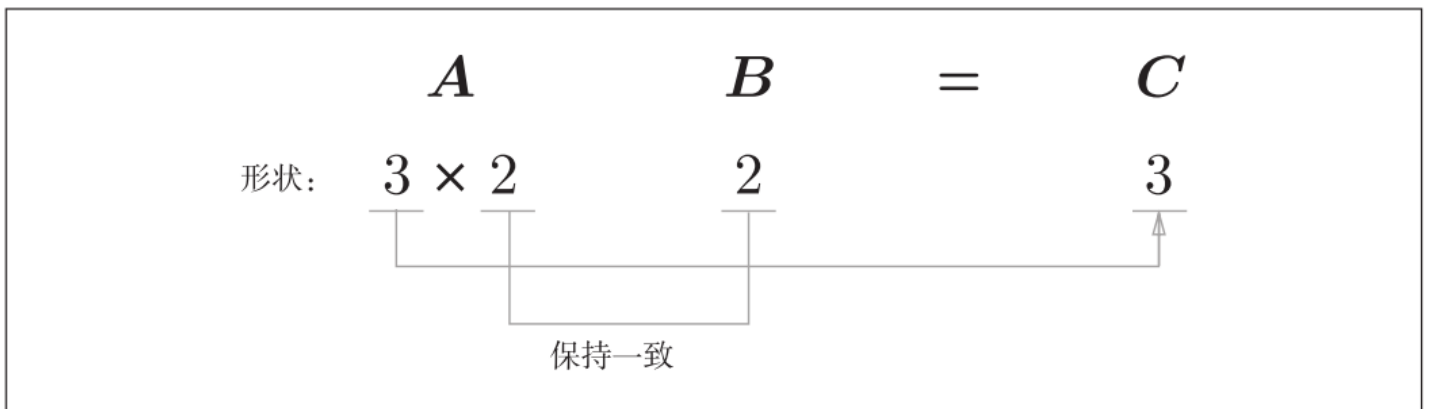


图3-13 A是二维矩阵、B是一维数组时，也要保持对应维度的元素个数一致

## 神经网络的内积

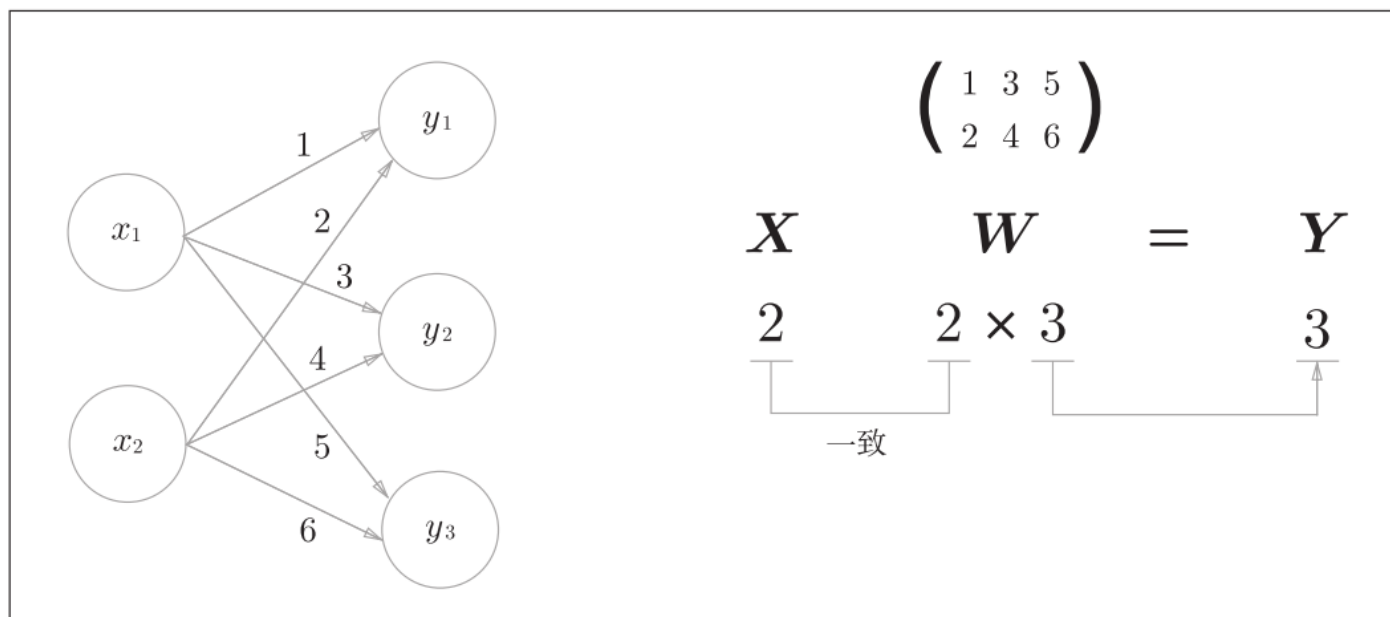


图 3-14 通过矩阵的乘积进行神经网络的运算

```
>>> 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]
```

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层神经网络的实现

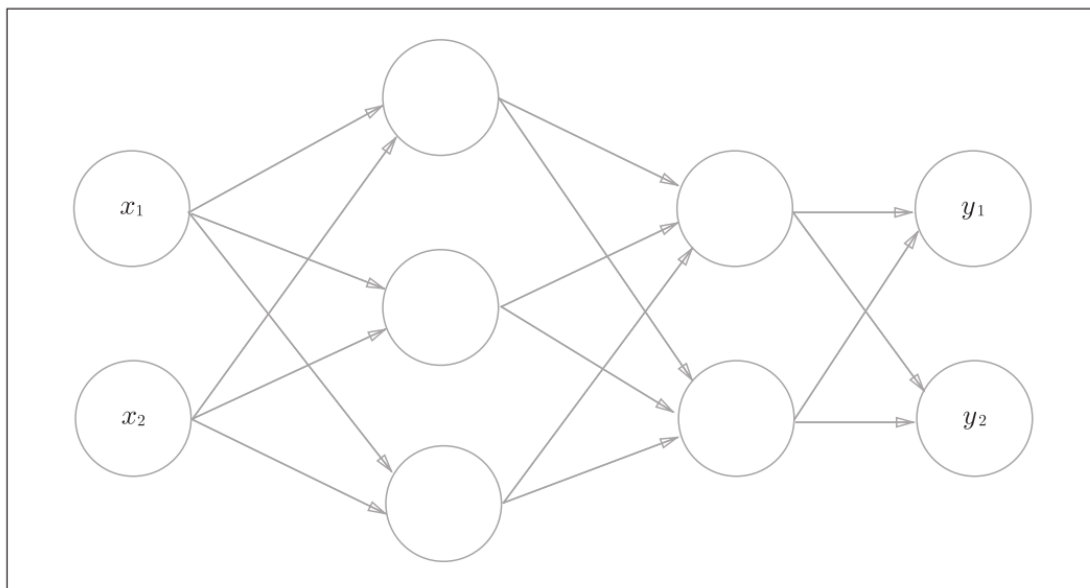


图 3-15 3层神经网络：输入层(第0层)有2个神经元，第1个隐藏层(第1层)有3个神经元，第2个隐藏层(第2层)有2个神经元，输出层(第3层)有2个神经元

## 符号确认

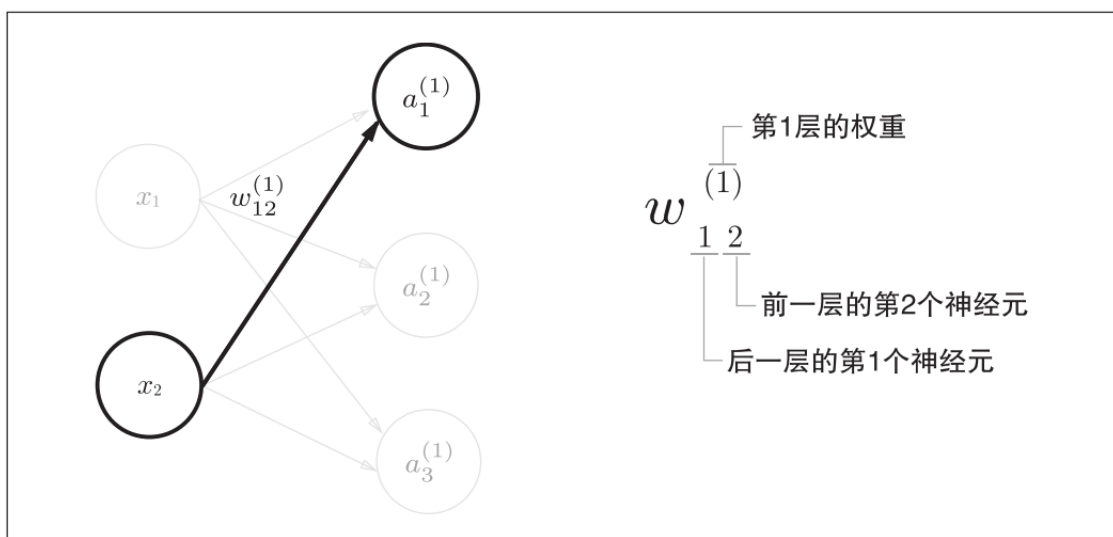


图 3-16 权重的符号

## 各层间信号传递的实现

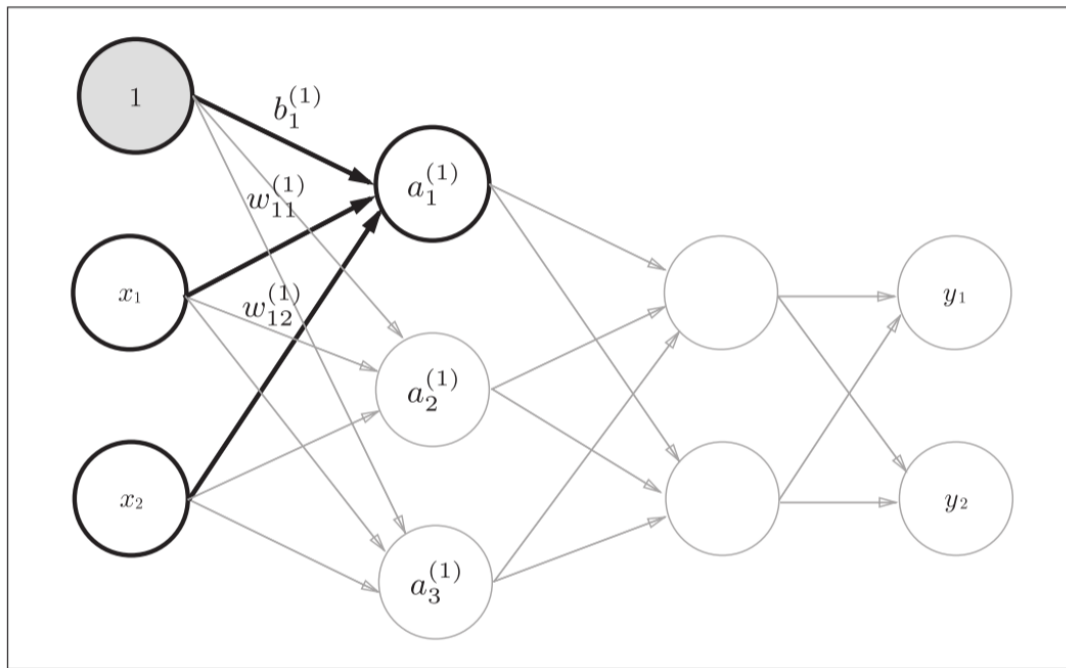


图 3-17 从输入层到第 1 层的信号传递

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。

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”）。

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

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)。

$$\mathbf{A}^{(1)} = \mathbf{XW}^{(1)} + \mathbf{B}^{(1)} \quad (3.9)$$

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

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

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

```
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
```

```
Z1 = sigmoid(A1)
```

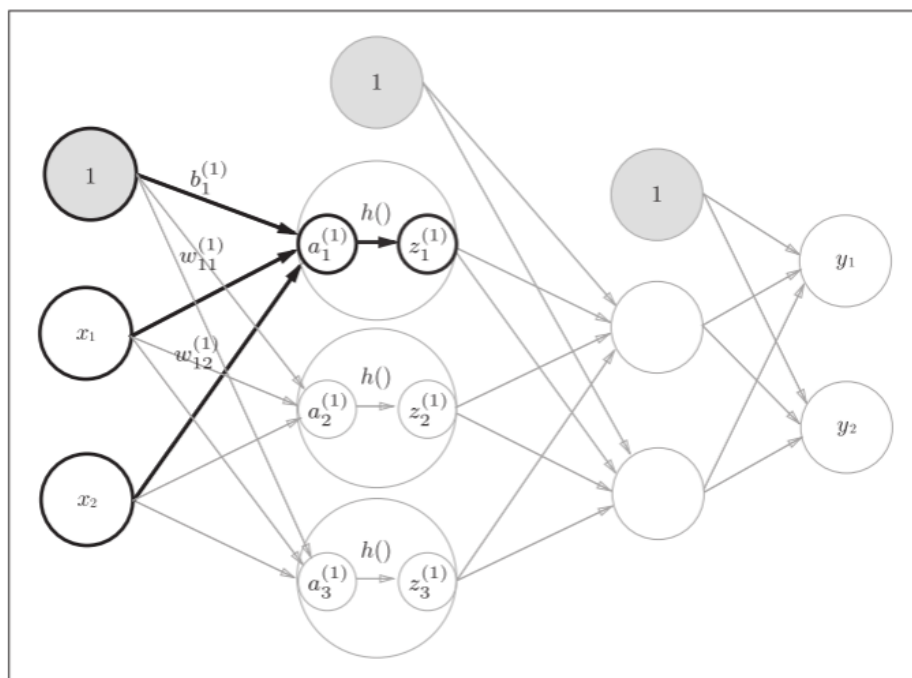


图3-18 从输入层到第1层的信号传递

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一致。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()（ $\sigma$ 读作sigma）。



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。

## 代码实现小结

```
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。

## 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）。

## 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。

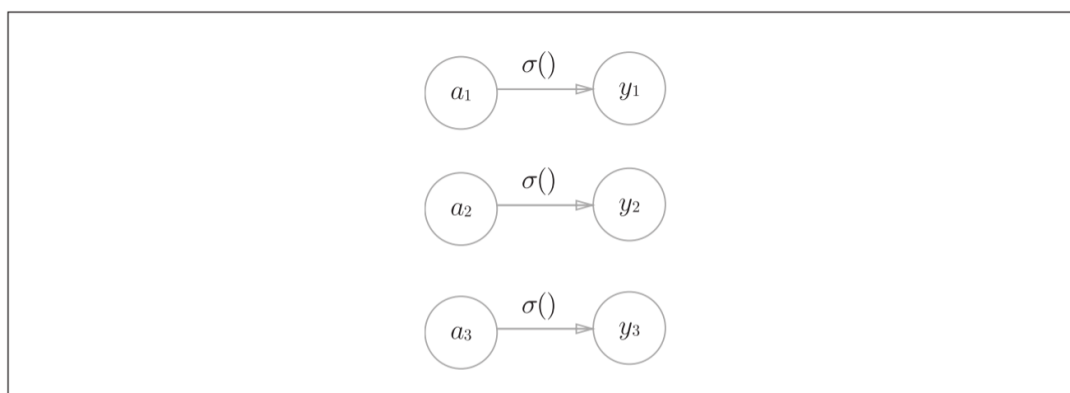


图 3-21 恒等函数

### 分类问题中使用的softmax函数

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

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  $y_k$ 。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。

softmax 函数的输出通过箭头与所有的输入信号相连。输出层的各个神经元都受到**所有输入信号**的影响。

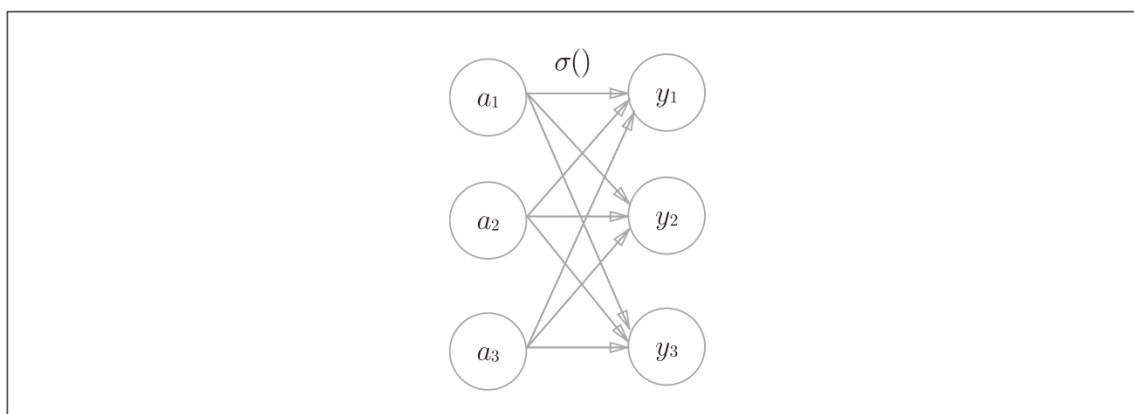


图 3-22 softmax 函数




```

>>> 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]

```

## 实现softmax函数时的注意事项

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。

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

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**。

```

>>> 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。

```

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

```

## softmax函数的特征

- softmax函数的输出是0.0到1.0之间的实数。
- softmax函数的输出值的总和是1。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”
- 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”A和“推理”两个阶段。首先，在学习阶段进行模型的学习B，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）。

# 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。

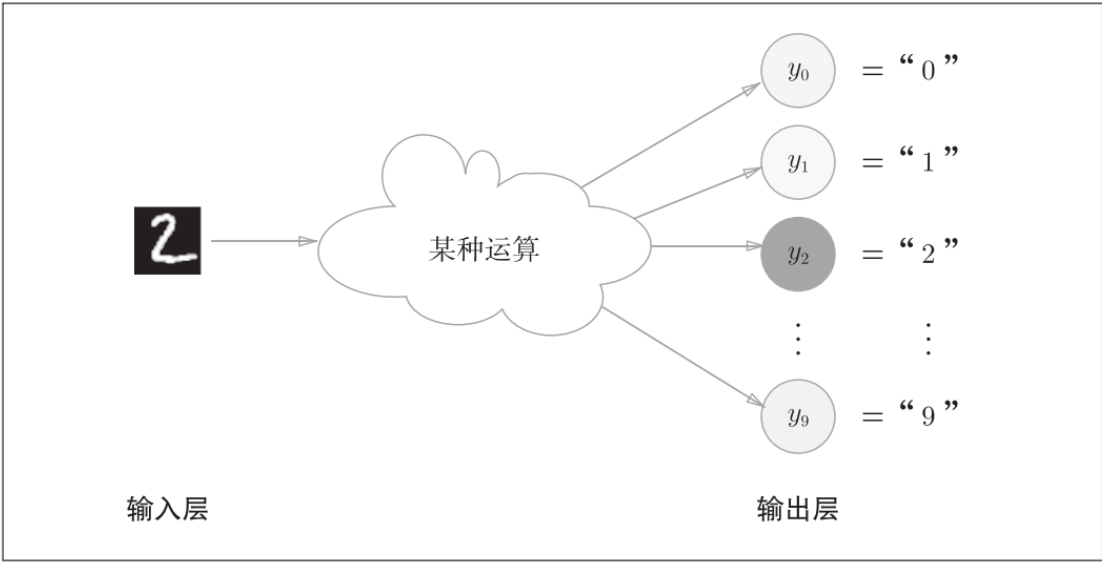


图 3-23 输出层的神经元对应各个数字

## 手写数字识别

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**。

## 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  $28 \text{ 像素} \times 28 \text{ 像素}$  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）。

`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  $1 \times 28 \times 28$  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]` 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。

 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。

```

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)

```

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。

## 神经网络的推理处理

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A。这个文件中以字典变量的形式保存了权重和偏置参数。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。

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)))

```

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。

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。

## 批处理



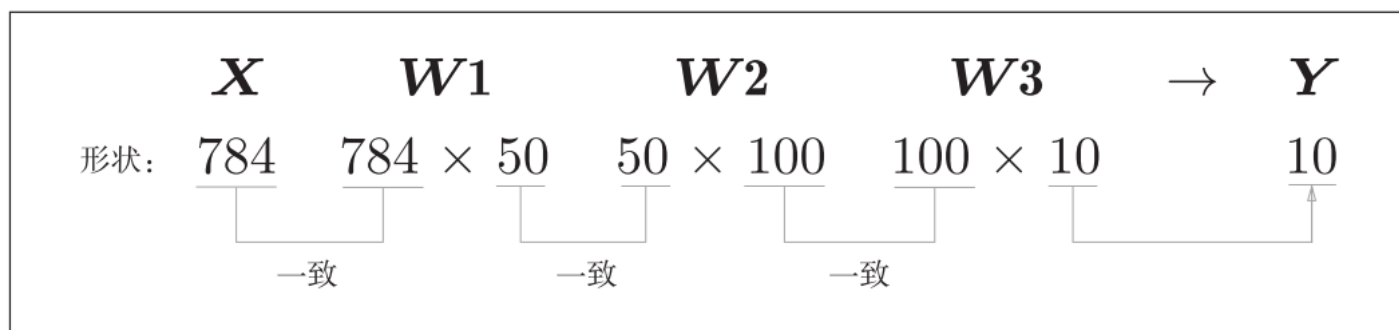


图 3-26 数组形状的变化

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x的形状改为100 × 784，将100张图像打包作为输入数据。比如，x[0]和y[0]中保存了第0张图像及其推理结果，等等。

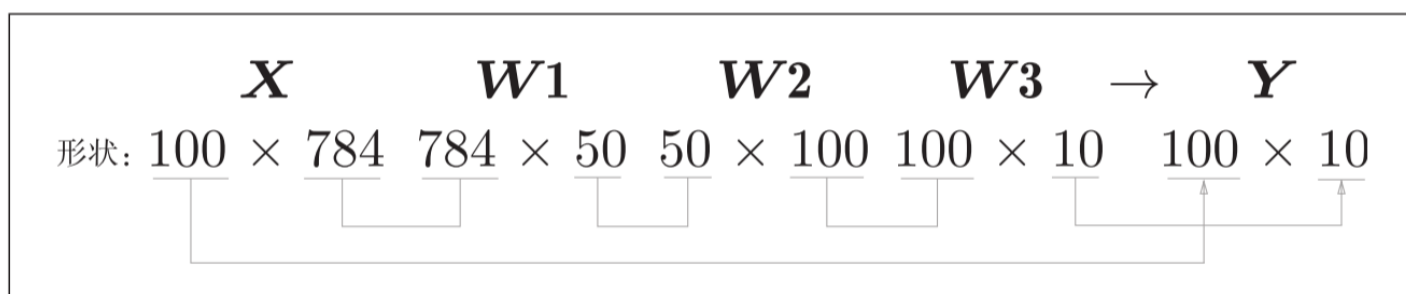



图 3-27 批处理中数组形状的变化

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。

```
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)))
```

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 $100 \times 10$ 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）。

矩阵的第0维是列方向，第1维是行方向。

## 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。