第 15 章 人工神经网络

人类很早就对大脑如何运作感兴趣。人类大脑是由大约 1 千亿个“神经元”(neurons)相连而构成的“神经网络”(neural network)。

McCulloch and Pitts (1943)首先提出神经元的简化数学模型。

Rosenblatt(1958)提出“感知机”(perceptron)，通过引入“学习”(learning)，使得感知机具备将事物分类的能力。

但这种单层的神经网络无法得到非线性的决策边界，直到出现“多层感知机”(multilayer perceptron)，即“多层神经网络”(multilayer neural network)。随着算法日益改进，演化成炙手可热的“深度学习”(deep learning)。

15.1 人工神经网络的思想

在人工智能(Artificial Intelligence，简记 AI)领域，主要有两大派系。

一个派系为“符号主义”(Symbolicism)，又称逻辑主义，主张用公理和逻辑体系搭建一套人工智能系统。

另一派系则是“连接主义”(Connectionism)，也称仿生学派，主张模仿人类的神经元，用神经网络的连接机制实现人工智能。

符号主义者认为，人工智能应模仿人类的逻辑方式获取知识。

连接主义者则奉行大数据和学习来获得知识。

人工神经网络(Artificial Neural Networks，简记 ANN)是连接主义的代表作。

人类大脑是由大量“神经细胞”(neural cells)为基本单位而组成的神经网络。

神经细胞也称“神经元”(neurons)。图 15.1 为神经元的基本结构示意图。

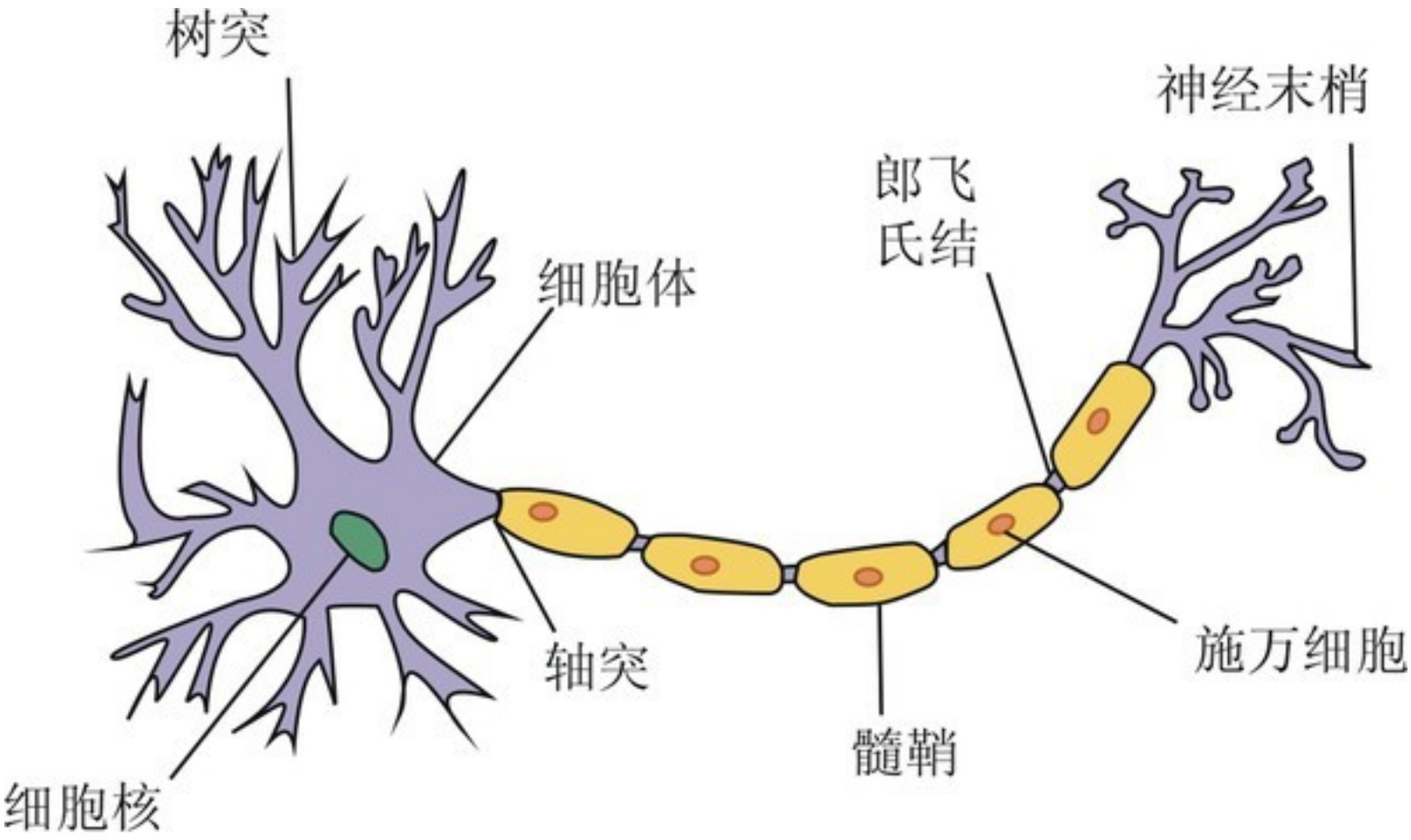

图 15.1 神经元结构示意图

在图 15.1 中，一个神经元通过左边的“树突”(dendrite)从其他神经元的“轴突”(axon)及“轴突末梢”(axon terminal)获取电子或化学信号(未在图中显示其他神经元)。

两个神经元之间的连接部位(junction)，称为“神经突触”(synapse)(在图中标为“神经末梢”)。

连接在一起的神经元，可以共同兴奋，即所谓“neurons wired together, fire together”。

从树突(dendrites)获得不同的信号后，神经元的“细胞体”(cell body)将这些信号进行加总处理。

如果这些信号的总量超过某个阈值，则神经元会兴奋起来，并通过轴突向外传输信号，经过神经突触(synapses)，而为其他神经元的树突所接收。

1943 年，美国神经生理学家 Warren McCulloch 与数学家 Walter Pitts 将生物神经元简化为一个数学模型 (McCulloch and Pitts, 1943)，简称 **M-P 神经元模型**，参见图 15.2。

从图 15.2 可见，M-P 神经元模型与生物神经元在形式上类似。

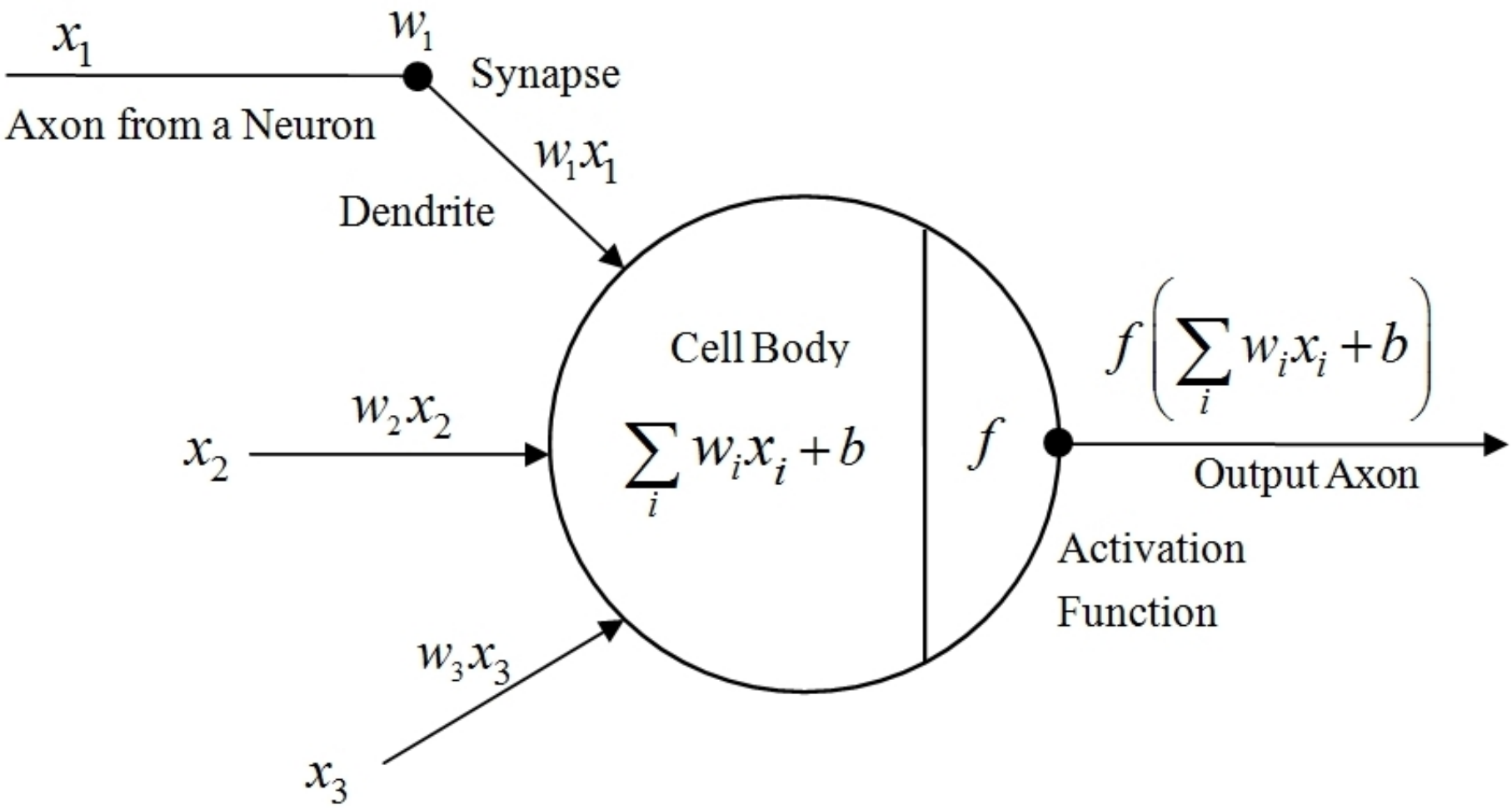

图 15.2 M-P 神经元的数学模型

将神经元视为一个计算单位，它首先从树突(dendrites)输入信号 $\mathbf{x} \equiv (x_1 \cdots x_p)'$，在细胞体(cell body)进行加权求和 $\sum_{i=1}^{p} w_i x_i$，其中 $\mathbf{w} \equiv (w_1 \cdots w_p)'$为权重(不同信号的重要性不同)。

如果求和之后的总数，超过某个阈值(比如，$-b$)，则神经元兴奋起来，通过轴突(axon)向外传递信号；反之，则神经元处于抑制状态：

$$I\left(\sum_{i=1}^{p} w_i x_i + b > 0\right) = \begin{cases} 1 & if \sum_{i=1}^{p} w_i x_i > -b \\ 0 & if \sum_{i=1}^{p} w_i x_i \le -b \end{cases} \tag{15.1}$$

其中，$I(\cdot)$为示性函数；参数b表示阈值(门槛值)，称为**偏置**(bias)。此处的示性函数$I(\cdot)$称为**激活函数**(activation function)。

M-P 神经元模型本质上只是一个纯数学模型(尽管也通过电阻得到物理实现)，其中的参数$\mathbf{w}$与b需要人为指定，而无法通过训练样本进行学习。

15.2 感知机

Rosenblatt(1958)提出**感知机**(Perceptron)，使得 M-P 神经元模型具备学习能力，成为神经网络模型的先驱。

对于二分类问题，考虑使用分离超平面“$b+\mathbf{w}'\mathbf{x}=0$”进行分类，而响应变量$y\in\{1,-1\}$。

如果$b+\mathbf{w}'\mathbf{x}>0$，则预测$y=1$。

如果$b+\mathbf{w}'\mathbf{x}<0$，则预测$y=-1$。

如果$b+\mathbf{w}'\mathbf{x}=0$，可随意预测。

正确分类要求$y_i(b+\mathbf{w}'\mathbf{x}_i)>0$。

反之，如果$y_i(b+\mathbf{w}'\mathbf{x}_i)<0$，则为错误分类。

从某个初始值$(\mathbf{w}_0,b_0)$出发，感知机希望通过调整参数$(\mathbf{w},b)$，使得模型的错误分类最少。

感知机的目标函数为最小化所有分类错误观测值的“错误程度”之和：

$$\min_{\mathbf{w},b} \; L(\mathbf{w},b) = -\sum_{i\in\mathcal{M}} y_i(b+\mathbf{w}'\mathbf{x}_i) \qquad (15.2)$$

其中，$\mathcal{M}$为所有错误分类(misclassified)的个体下标之集合。

假定$\mathcal{M}$不变(如果$\mathcal{M}$有变，在迭代过程中更新即可)，则此目标函数的梯度向量为

$$\frac{\partial L(\mathbf{w},b)}{\partial \mathbf{w}} = -\sum_{i\in\mathcal{M}} y_i \mathbf{x}_i \qquad (15.3)$$

$$\frac{\partial L(\mathbf{w},b)}{\partial b} = -\sum_{i\in\mathcal{M}} y_i \qquad (15.4)$$

使用梯度下降法，沿着负梯度方向更新，则参数的更新规则为

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \sum_{i\in\mathcal{M}} y_i \mathbf{x}_i \qquad (15.5)$$

$$b \leftarrow b + \eta \sum_{i\in\mathcal{M}} y_i \qquad (15.6)$$

其中，η为“学习率”(learning rate)，也称“步长”(step size)。

通过迭代，可使损失函数$L(\mathbf{w},b)$不断减小，直到变为 0 为止。

感知机算法的直观解释为，当一个样本点被错误分类，即出现在分离超平面“$b+\mathbf{w}'\mathbf{x}=0$”的错误一侧时，则调整参数$(\mathbf{w},b)$，使得分离超平面向该误分类点的一侧移动，以减少此误分类点与超平面的距离，直至正确分类为止。

可以证明，对于线性可分的数据，感知机一定会收敛。

只要给予足够的数据，感知机具备学得参数$(\mathbf{w},b)$的能力，仿佛拥有“感知”世界的能力(比如，自动将事物分类)，故名“感知机”。

对于线性可分的数据，感知机虽然一定会收敛，但从不同的初始值出发，一般会得到不同的分离超平面，无法得到唯一解。

由于所得超平面未必是“最优分离超平面” (optimal separating hyperplane，参见第 14 章)，故感知机的泛化能力也没有保证。

如果数据为线性不可分，则感知机的算法不会收敛。

感知机更严重的缺陷是，它的决策边界依然为线性函数。可将感知机的预测函数写为

$$f(\mathbf{x}) = sign\left(\mathbf{w}'\mathbf{x} + b\right) \quad (15.7)$$

其中，$sign(\cdot)$为符号函数，满足

$$sign(z)=\begin{cases}1 & if\ z\geq 0\\ -1 & if\ z<0\end{cases} \tag{15.8}$$

虽然符号函数 $sign(\cdot)$ 为非线性，但感知机的决策边界为 $\mathbf{w}'\mathbf{x}+b=0$，即分离超平面，依然为线性函数。

感知机无法适用于决策边界为非线性的数据。

例 感知机无法识别“异或函数”。在逻辑学中，有几个常见的逻辑运算，包括“与”(AND)、“或”(OR)、“与非”(NOT AND)、“异或”(Exclusive Or，简记 XOR)。

“异或”是一种排他性(exclusive)的“或”，即当二者取值不同时为“真”(TRUE)，而当二者取值相同时即为“假”(FALSE)。

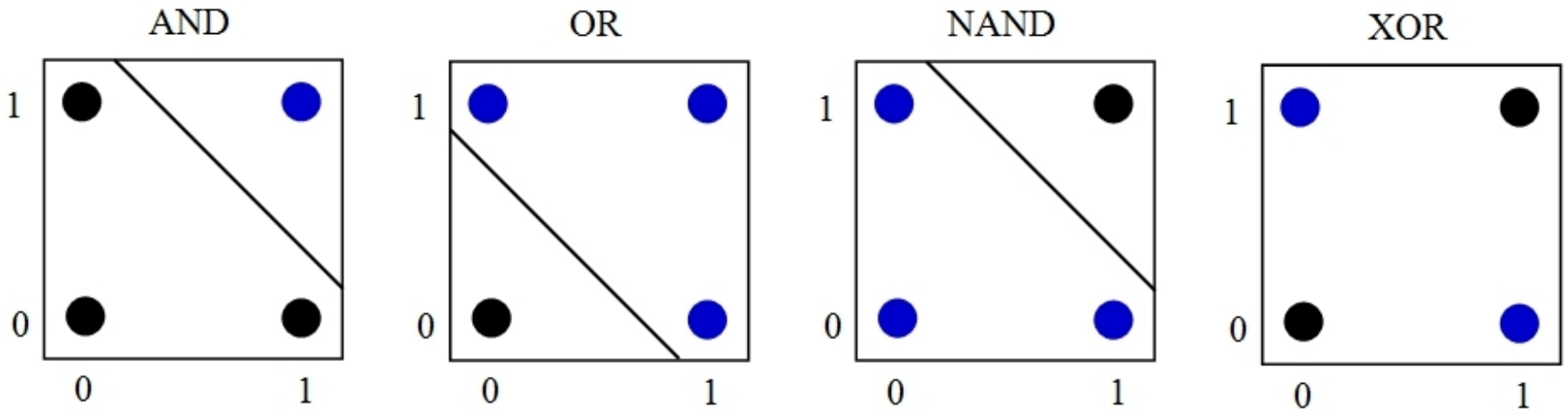

图 15.3 异或函数的非线性决策边界

逻辑判断 TRUE 记为 1(以蓝点表示)，而 FALSE 记为 0(以黑点表示)。

以图 15.3 中最左边的 AND 运算为例，只有当输入值都是 1(TRUE)时，经过 AND 运算后才是 1(TRUE)，以右上角的蓝点表示；在这种情况下，存在线性的决策边界。

对于 OR 与 NAND 的运算，也存在线性的决策边界。

对于 XOR 的运算，由于 TRUE 与 FALSE 分别分布在两个对角上，故无法找到线性的决策边界，存在非线性的决策边界。

1969 年，Marvin Minsky 与 Seymour Papert 在专著 *Perceptrons* 指出，感知机连基本的异或函数都无法区分，功能十分有限。

当时学界普遍认为感知机无发展前途，使得人工神经网络研究陷入低谷。

15.3 神经网络的模型

事实上，在感知机的基础上，并不难得到非线性的决策边界。

只要引入多层神经元，经过两个及以上的非线性激活函数迭代之后，即可得到非线性的决策边界。

非线性的激活函数是关键：如果使用线性的激活函数，则无论叠加或嵌套多少次(相当于微积分的复合函数)，所得结果一定还是线性函数。

首先，考虑具有多个输出结果(multi output)的感知机，如图 15.4 所示。

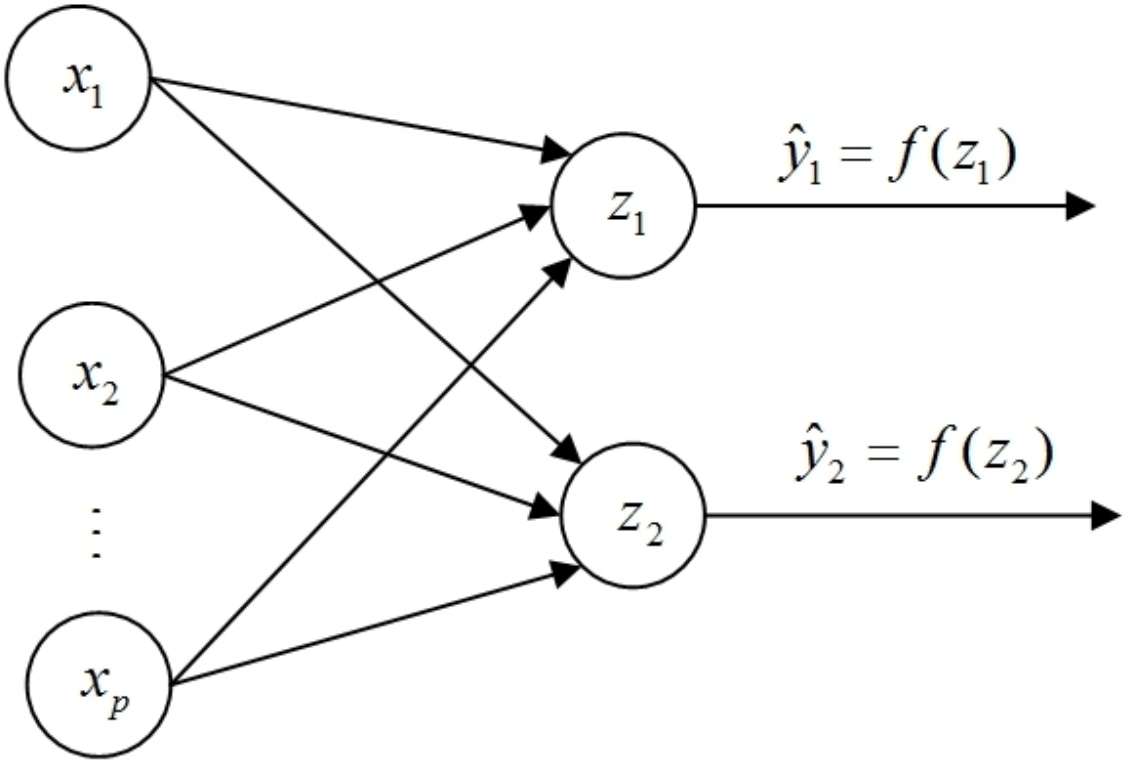

图 15.4 多输出的感知机

在图 15.4 中，共有两个输出(响应)变量，$\hat{y}_1$与$\hat{y}_2$。

其中，$z_1 \equiv b_1 + \sum_{i=1}^{p} w_{i1} x_i$与$z_2 \equiv b_2 + \sum_{i=1}^{p} w_{i2} x_i$，均为在施加激活函数之前的加总值；而$f(\cdot)$为激活函数。

其次，图 15.4 中的多个输出结果，可重新作为输入变量，经过加权求和后，再次施以激活函数，参见图 15.5。

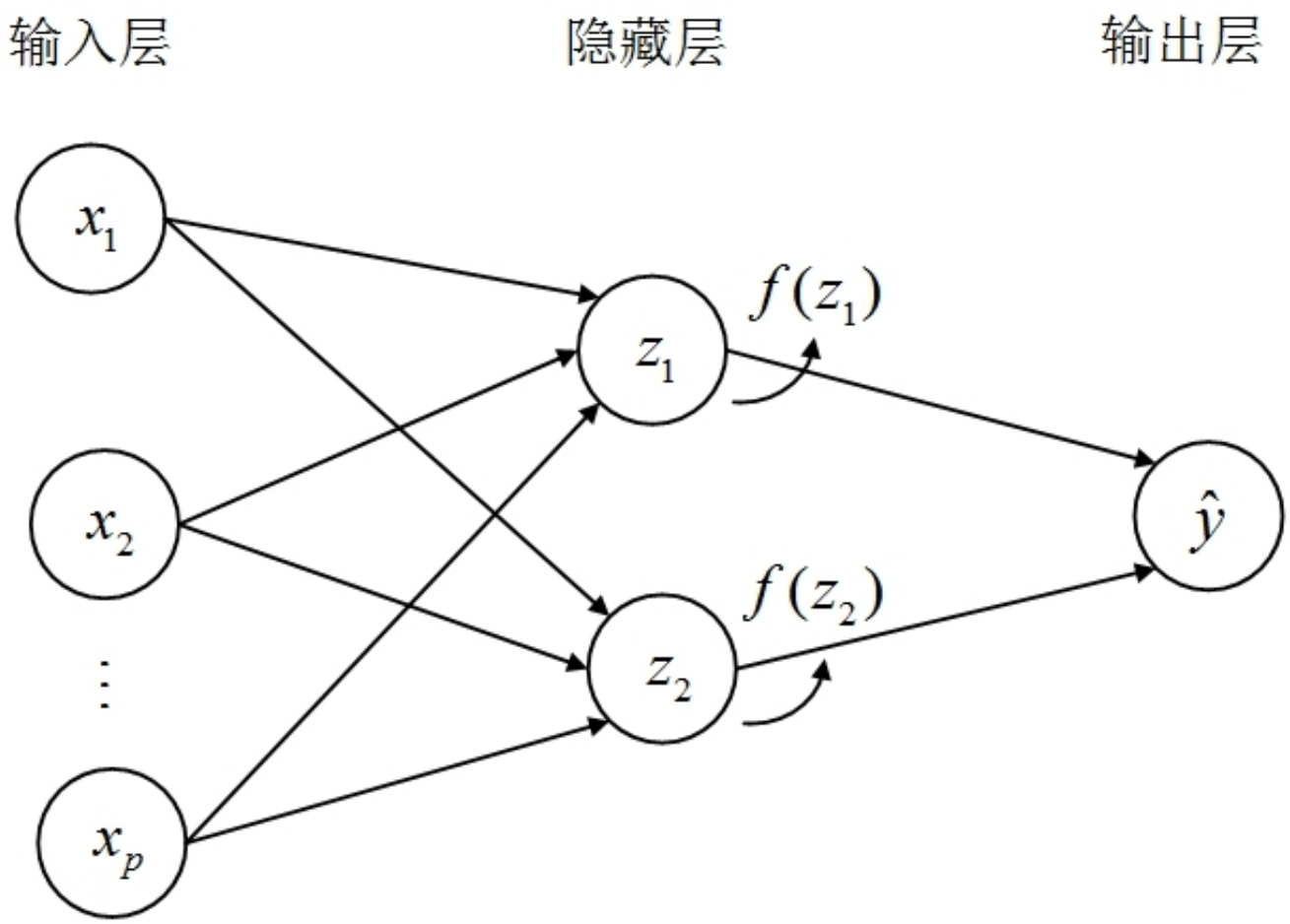

图 15.5 多层感知机

在图 15.5 中，最终输出结果为

$$\hat{y} = f\left(b^{(2)} + w_1^{(2)} f(z_1) + w_2^{(2)} f(z_2)\right) \qquad (15.9)$$

即对 $f(z_1)$与 $f(z_2)$再次加权求和，然后再施加激活函数 $f(\cdot)$。

函数(15.9)所对应的决策边界为非线性的。

在图 15.5 中，最左边为**输入层**(input layer)，中间为**隐藏层**(hidden layer)，而最右边为**输出层**(output layer)。

隐藏层可有更多的神经元，而输出层也可有多个输出结果，参见图 15.6。

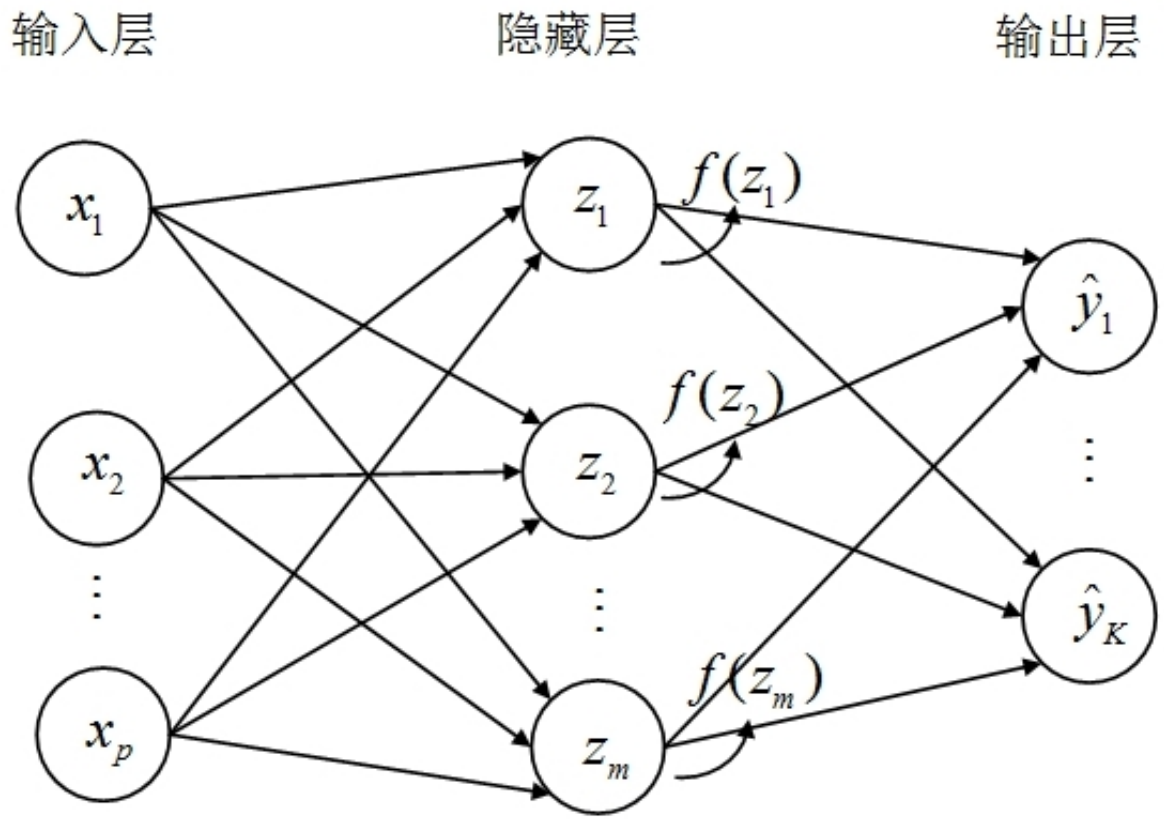

图 15.6 单隐藏层的神经网络

更一般地，神经网络模型可以有多个隐藏层，参见图 15.7。

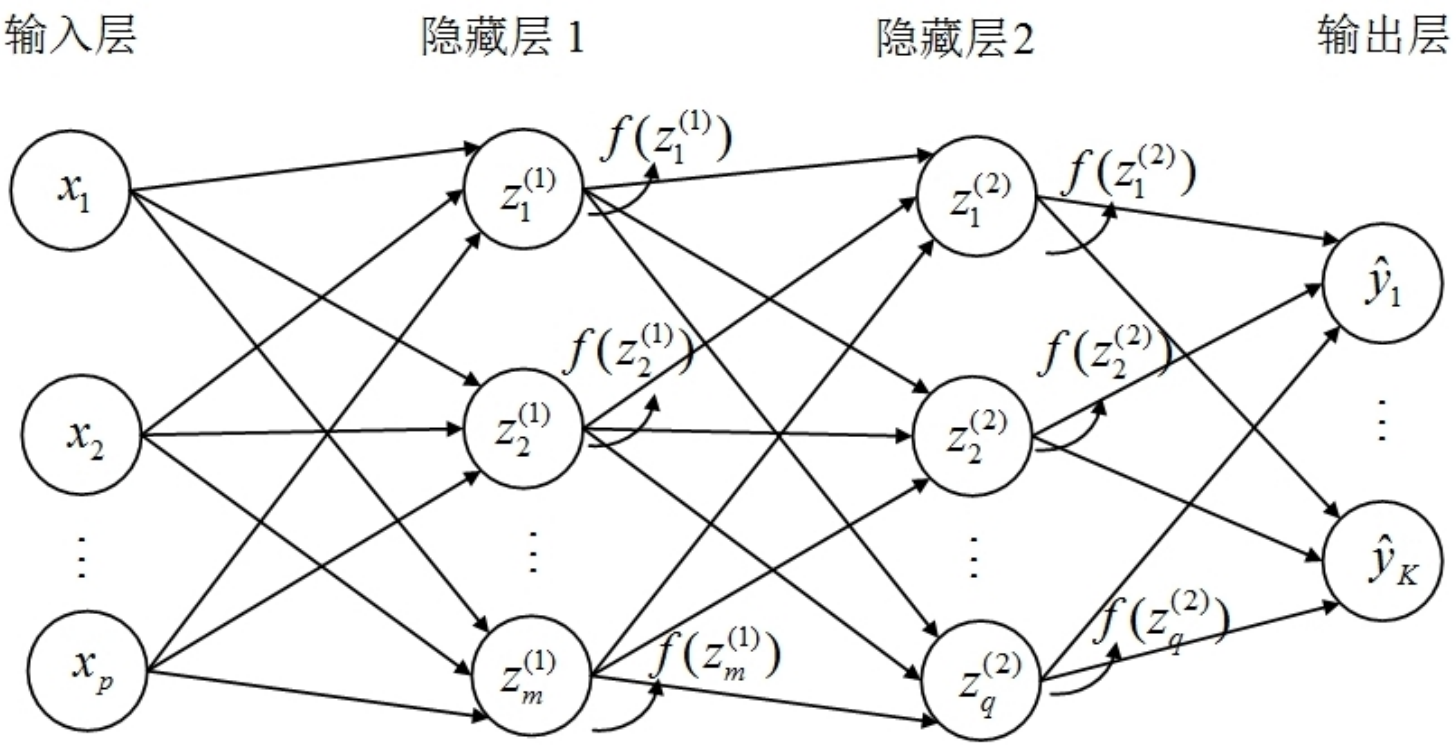

图 15.7 双隐藏层的神经网络

这种标准的神经网络，称为**前馈神经网络**(feedforward neural network)，因为输入从左向右不断前馈。

也称为**全连接神经网络**(fully-connected neural network)，因为相邻层的所有神经元都相互连接。

针对特殊的数据类型，可能还需要特别的网络结构，比如卷积神经网络(适用于图像识别)、循环神经网络(适用于自然语言等时间序列)等。

如果神经网络的隐藏层很多，则称为**深度神经网络**(deep neural networks)，简称**深度学习**(deep learning)。

15.4 神经网络的激活函数

感知机使用符号函数(15.8)作为激活函数，但这是一个不连续的“阶梯函数”(step function)，不便于进行最优化。

激活函数必须为非线性函数，因为这个世界本质上是非线性的。神经网络模型中常用的激活函数包括：

(1) S 型函数(sigmoid function)，参见图 15.8 左图。狭义的 S 型函数就是逻辑分布的累积分布函数，其表达式为

$$\Lambda(z) \equiv \frac{1}{1+e^{-z}} \qquad (15.10)$$

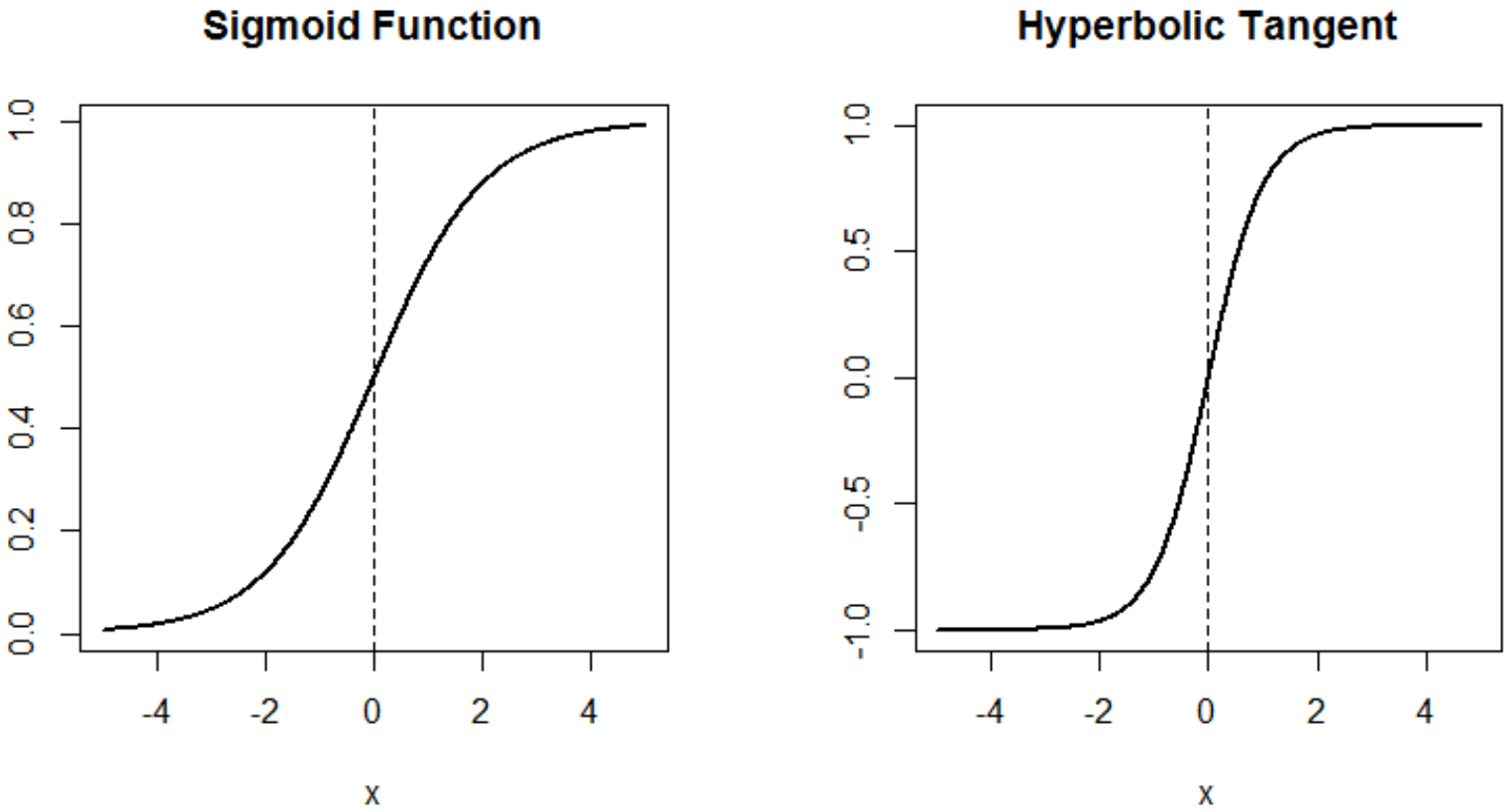

图 15.8 两种 S 型激活函数

此函数也用于逻辑回归(Logit regression)，在机器学习中有时记为$\sigma(z)$，也称为“逻辑函数”(logistic function)。

S 型函数可视为一种“挤压函数”(squashing function)，即把输入的任何实数都挤压到$(0,1)$区间。

当输入值z在 0 附近时，S 型函数近似为线性函数；而当输入值z靠近两端$(\pm\infty)$时，则对输入进行抑制。

输入越小，则输出越接近于 0；而输入越大，则输出越接近于 1；此特点与生物神经元类似。

由于 S 型函数的输出值介于 0 与 1 之间，故可将其解释为概率分布。

与感知机所用的阶梯激活函数相比，sigmoid 函数为连续可导(continuously differentiable，即导函数存在且连续)，其数学性质更好。

但当输入靠近两端($|z|$很大)时，S 型函数的导数$\Lambda'(z)$趋向于 0，故在训练神经网络时，可能导致“梯度消失”(vanishing gradient)的问题，使得梯度下降法失效。S 型函数的导数为(参见第 5 章)：

$$\Lambda'(z)=\Lambda(z)\left[1-\Lambda(z)\right] \qquad (15.11)$$

当$z\to-\infty$或$z\to\infty$时，$\lim\Lambda'(z)=0$。这种情形称为“两端饱和”。

(2) 双曲正切函数(hyperbolic tangent function)，参见图 15.8 右图。双曲正切函数是一种广义的 S 型函数，因为它的形状也类似于拉长的英文大写字母 S，其表达式为

$$\tanh(z) \equiv \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \qquad (15.12)$$

Tanh 函数可看作是将 Logistic 函数进一步拉伸到$(-1,1)$区间。容易证明，二者有如下关系：

$$\tanh(z) = 2\Lambda(2z) - 1 \qquad (15.13)$$

其中，上式右边可写为

$$2\Lambda(2z)-1=2\cdot\frac{1}{1+e^{-2z}}-1=\frac{1-e^{-2z}}{1+e^{-2z}}$$
$$=\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}=\tanh(z) \tag{15.14}$$

Tanh 函数的输出是零中心化的(zero-centered)，而 Logistic 函数的输出一定大于 0。非零中心化的输出(例如 Logistic 函数)，会使得下一层的神经元输入发生“偏置偏移”(bias shift)，使得梯度下降的收敛速度变慢。

Tanh 函数也是两端饱和的，即当输入靠近两端时，其导数趋向于 0，依然可能发生梯度消失的问题。

(3) 修正线性单元(Rectified Linear Unit，简记 ReLU)，也称“线性整流函数”，参见图 15.9 左图。

为解决 Logistic 函数与 Tanh 函数的两端饱和问题，Nair and Hinton(2010)提出如下 ReLU 函数，成为目前深度神经网络中经常使用的激活函数：

$$\mathrm{ReLU}(z) \equiv \max(0, z) = \begin{cases} z & if \ z \geq 0 \\ 0 & if \ z < 0 \end{cases} \qquad (15.15)$$

ReLU 实际上是一个斜坡(ramp)函数。当输入$z \geq 0$，其输出也是z，即所谓“线性单元”(linear unit)；而当输入$z < 0$时，则将输出“修正”(rectified)为 0。

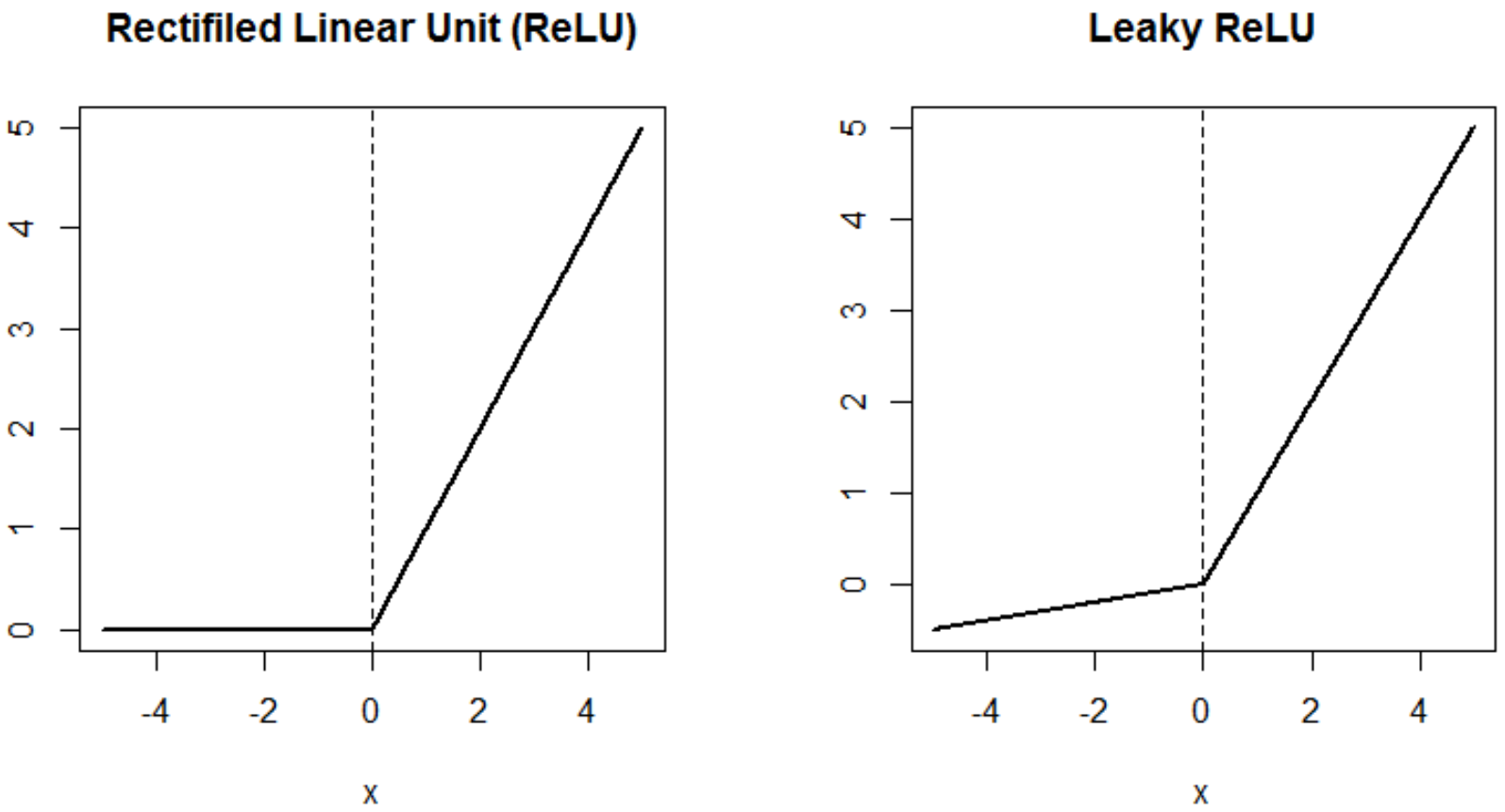

图 15.9 ReLU 与泄露 ReLU 激活函数

以 ReLU 函数作为激活函数，计算非常方便。

相比于 S 型函数的两端饱和，ReLU 函数为“左饱和函数”，即当 $z \to -\infty$ 时，ReLU 函数的导数趋向于 0。

当 $z > 0$ 时，ReLU 函数的导数恒等于 1，这可在一定程度上缓解神经网络训练中的梯度消失问题，加快梯度下降的收敛速度。

ReLU 函数被认为具有生物学上的解释，比如单侧抑制、宽兴奋边界(即兴奋度可以很高)。

在生物神经网络中，同时处于兴奋状态的神经元一般非常稀疏，而 ReLU 激活函数则可导致较好的稀疏性。

ReLU 函数也是非零中心化的，导致下一层出现“偏置偏移”，影响梯度下降的效率。

更严重的问题是，由于当$z<0$时，ReLU 函数的导数恒等于 0，这导致神经元在训练时可能“死亡”，称为“死亡 ReLU 问题”(dying ReLU problem)。

所谓“神经元死亡”，就是无论该神经元的输入是什么，其输出永远是 0，故无法更新其输入的权重。

(4) 泄露 ReLU(Leaky ReLU，简记 LReLU)，参见图 15.9 右图。

解决“死亡 ReLU 问题”的一种方式是，当输入$z<0$时，依然保持一个很小的梯度$\gamma>0$。这样使得当神经元处于非激活状态时，也能有一个非零的梯度可以更新参数，避免永远不能被激活(Maas et al., 2013)。泄露 ReLU 函数的定义为

$$\text{LReLU}(z)\equiv\begin{cases} z & if\ z\geq 0 \\ \gamma z & if\ z<0 \end{cases} \qquad (15.16)$$

其中，$\gamma>0$是一个很小的正数，比如 0.01。当$\gamma<1$时，泄露 ReLU 可写为

$$\text{LReLU}(z)=\max(z,\gamma z) \qquad (15.17)$$

(5) 软加函数(Softplus Function)，参见图 15.10。

ReLU 函数并不光滑，而且在$z<0$时，导数一直为 0。

软加函数可视为 ReLU 函数的光滑版本，正好弥补 ReLU 的这些缺点。

Softplus 函数的定义为

$$\text{Softplus}(z)\equiv\ln(1+e^{z}) \qquad (15.18)$$

从图 15.10 可见，Softplus 函数也具有单侧抑制、宽兴奋边界的特性，但没有 ReLU 函数的稀疏激活性(因为 Softplus 函数的导数永远为正)。

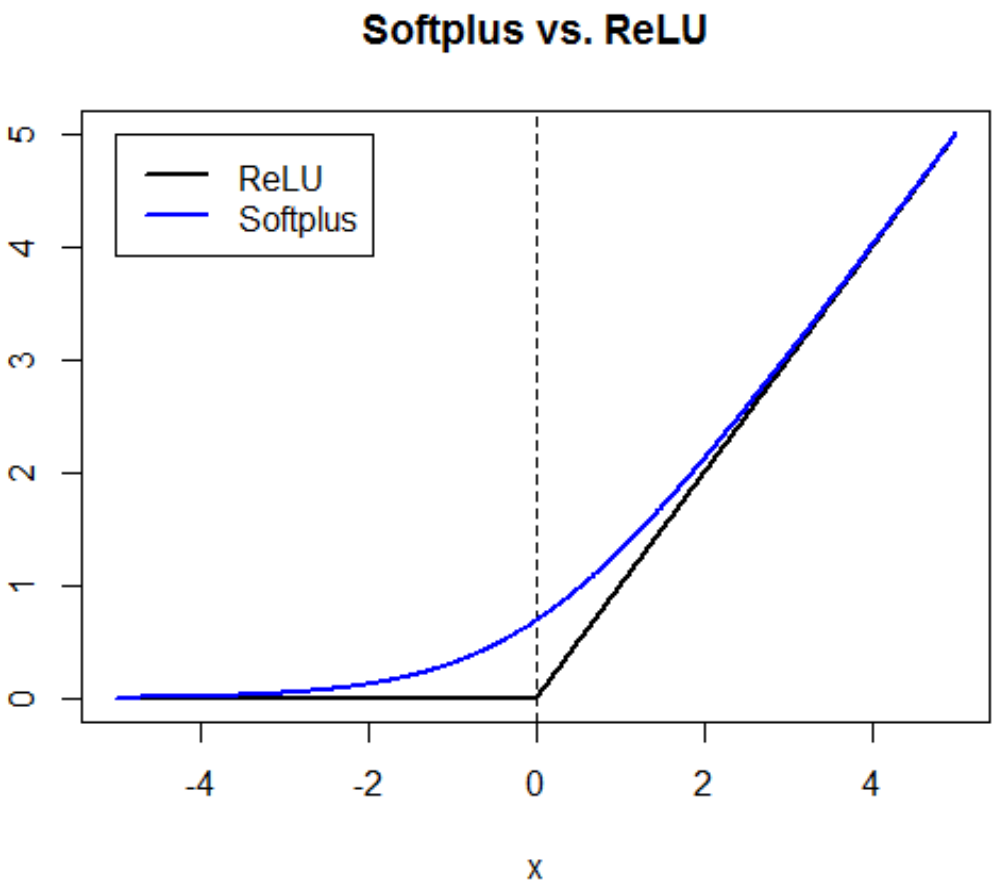

图 15.10 软加函数与 ReLU 函数

15.5 通用函数近似器

前馈神经网络具有很强的函数拟合能力。

在一定意义上，神经网络可作为一种“通用近似器”(universal approximator)来使用。

Cybenko(1988)与 Hornik, Stinchcombe and White(1989)证明了神经网络的“通用近似定理”(Universal Approximation Theorem)。

主要结论：包含单一隐藏层的前馈神经网络模型，只要其神经元数目足够多，则可以任意精度逼近任何一个在有界闭集上定义的连续函数。

首先，包含单隐藏层的前馈神经网络所代表的函数可写为

$$G(\mathbf{x})=\sum_{i=1}^{m}\alpha_i f(\mathbf{w}_i'\mathbf{x}+b_i) \qquad (15.19)$$

其中，$(\mathbf{w}_i, b_i)$为第i个神经元的权重与偏置参数，$f(\cdot)$为激活函数，α_i为连接隐藏层与输出层的参数，而m为神经元的数目。

通用近似定理表明，形如(15.19)的函数在定义于有界闭集上的连续函数之集合中是“稠密的”(dense)。

这意味着，对于任意有界闭集上的连续函数，都可找到形如(15.19)的函数(即单隐层的前馈神经网络)，使二者的距离任意接近。

命题 15.1（通用近似定理） 令$f(\cdot)$为一个合适的激活函数(详见下文)，$\mathcal{I}_p$是一个p维的单位超立方体(unit hypercube)$[0,1]^p$，而$C(\mathcal{I}_p)$是定义在$\mathcal{I}_p$上的所有连续函数之集合。

对于任意一个函数$g \in C(\mathcal{I}_p)$，给定任意小的正数$\varepsilon > 0$，则存在一个正整数m(即神经元数目)，一组实数(α_i, b_i)，以及实数向量$\mathbf{w}_i \in \mathbb{R}^p$，$i = 1, \cdots, m$，使得方程(15.19)所定义的函数$G(\mathbf{x})$，可以任意地接近$g(\mathbf{x})$，即

$$\left|G(\mathbf{x}) - g(\mathbf{x})\right| < \varepsilon, \quad \forall \mathbf{x} \in \mathcal{I}_p \qquad (15.20)$$

在上述定理中，假设定义域为p维单位超立方体$[0,1]^p$，只是为了叙述方便。通用近似定理在任意p维实数空间$\mathbb{R}^p$的有界闭集上依然成立。

在文献中，通用近似定理的激活函数可采取不同形式的非线性函数，既包括非常数(nonconstant)、有界(bounded)且单调递增的连续函数(例如 S 型函数、双曲正切函数)，也包括无界(unbounded)且单调递增的连续函数(例如 ReLU)，甚至允许不连续函数(例如阶梯函数)。

通用近似定理表明，神经网络可作为“万能”函数来使用。

但通用近似定理只是说明，对于任意有界闭集上的连续函数，都存在与它非常接近的单隐层前馈神经网络。

但并未给出找到此神经网络的方法，也不知道究竟需要多少个神经元，才能达到既定的接近程度ε。

在实际应用中，一般并不知道真实函数$g(\mathbf{x})$，而我们更关心神经网络$G(\mathbf{x})$的泛化能力。

由于神经网络的强大拟合能力，反而容易在训练集上过拟合，故需要避免过拟合，以降低测试误差。

15.6 神经网络的损失函数

未经训练的神经网络就像空白的大脑，并不具备预测与分类的能力。

“训练”意味着估计神经网络模型的诸多参数。对于神经网络而言，知识就储存在这些参数中。

神经网络的通常训练方法为，在参数空间使用梯度下降法，使损失函数最小化。神经网络的损失函数之一般形式可写为

$$\mathbf{W}^{*}=\underset{\mathbf{w}}{\operatorname{argmin}}\ \frac{1}{n}\sum_{i=1}^{n}L\left(y_i,G(\mathbf{x}_i;\mathbf{W})\right) \qquad (15.21)$$

其中，参数矩阵$\mathbf{W}$(也可排为参数向量)包含神经网络模型的所有参数(包括偏置)，其每一列对应于神经网络每一层的参数；

$\mathbf{W}^*$为$\mathbf{W}$的最优值，$G(\mathbf{x}_i;\mathbf{W})$为神经网络对观测值$\mathbf{x}_i$所作的预测(即$\hat{y}_i$)，而$L\left(y_i,\hat{y}_i\right)$为损失函数。

整个样本的损失函数为每个观测值之损失$L\left(y_i,G(\mathbf{x}_i;\mathbf{W})\right)$的平均值。

对于响应变量为连续的回归问题，一般使用平方损失函数(squared loss function)，最小化训练集的均方误差：

$$\mathbf{W}^*=\underset{\mathbf{w}}{\operatorname{argmin}}\ \frac{1}{n}\sum_{i=1}^{n}\left(y_i-G(\mathbf{x}_i;\mathbf{W})\right)^2 \qquad (15.22)$$

对于响应变量为离散的分类问题，则一般使用“交叉熵损失函数”(cross-entropy loss function)，即多项逻辑回归(Multinomial Logit)的对数似然函数之负数。

特别地，对于二分类问题，一般使用“二值交叉熵损失函数”(binary cross-entropy loss function)，即逻辑回归的对数似然函数之负数：

$$\mathbf{W}^{*}=\underset{\mathbf{w}}{\operatorname{argmin}}\ -\frac{1}{n}\sum_{i=1}^{n}\Big[y_i\ln G(\mathbf{x}_i;\mathbf{W})+(1-y_i)\ln\big(1-G(\mathbf{x}_i;\mathbf{W})\big)\Big] \tag{15.23}$$

15.7 神经网络的算法

由于神经网络通常包含很多参数，而且涉及较多非线性的激活函数，故一般不便于求二阶导数(黑塞矩阵)，无法使用牛顿法。

常使用梯度下降法训练神经网络，故需计算神经网络$G(\mathbf{x}_i;\mathbf{W})$的梯度向量。

对于神经网络，最常用的计算梯度向量方法为**反向传播算法**(Back Propagation，简记 BP)。

BP 算法最早由 Werbos (1974)提出，但时值 AI 寒冬，未引入重视；此后由 Rumelhart, Hinton and Williams(1986)重新发明。

对于多层的神经网络，越靠近网络右边(后端)的参数，其导数越容易计算，因为它们离输出层更近。

反向传播算法就是使用微积分的“链式法则”(chain rule)，将靠左边(前端)的参数之导数，递归地表示为靠右边(后端)的参数之导数的函数。

具体来说，记第l个隐层的第i个神经元的输出值，也称**激活值**(activation)为$a_i^{(l)}$，则

$$a_i^{(l)} = f\underbrace{\left(\sum_j w_{ji}^{(l)} a_j^{(l-1)}\right)}_{\equiv z_i^{(l)}} \equiv f(z_i^{(l)}) \qquad (15.24)$$

其中，$w_{ji}^{(l)}$为第$(l-1)$隐层第 j 个神经元的激活值$a_j^{(l-1)}$对$a_i^{(l)}$的作用权重，而 $f(\cdot)$为激活函数。

在施加激活函数之前，记**净输入**(net input)为

$$z_i^{(l)} \equiv \sum_j w_{ji}^{(l)} a_j^{(l-1)} \qquad (15.25)$$

其中，为简化符号，将偏置$b_i^{(l)}$也视为权重，对应于恒等于 1 的特征变量(即常数项)；参见图 15.11。

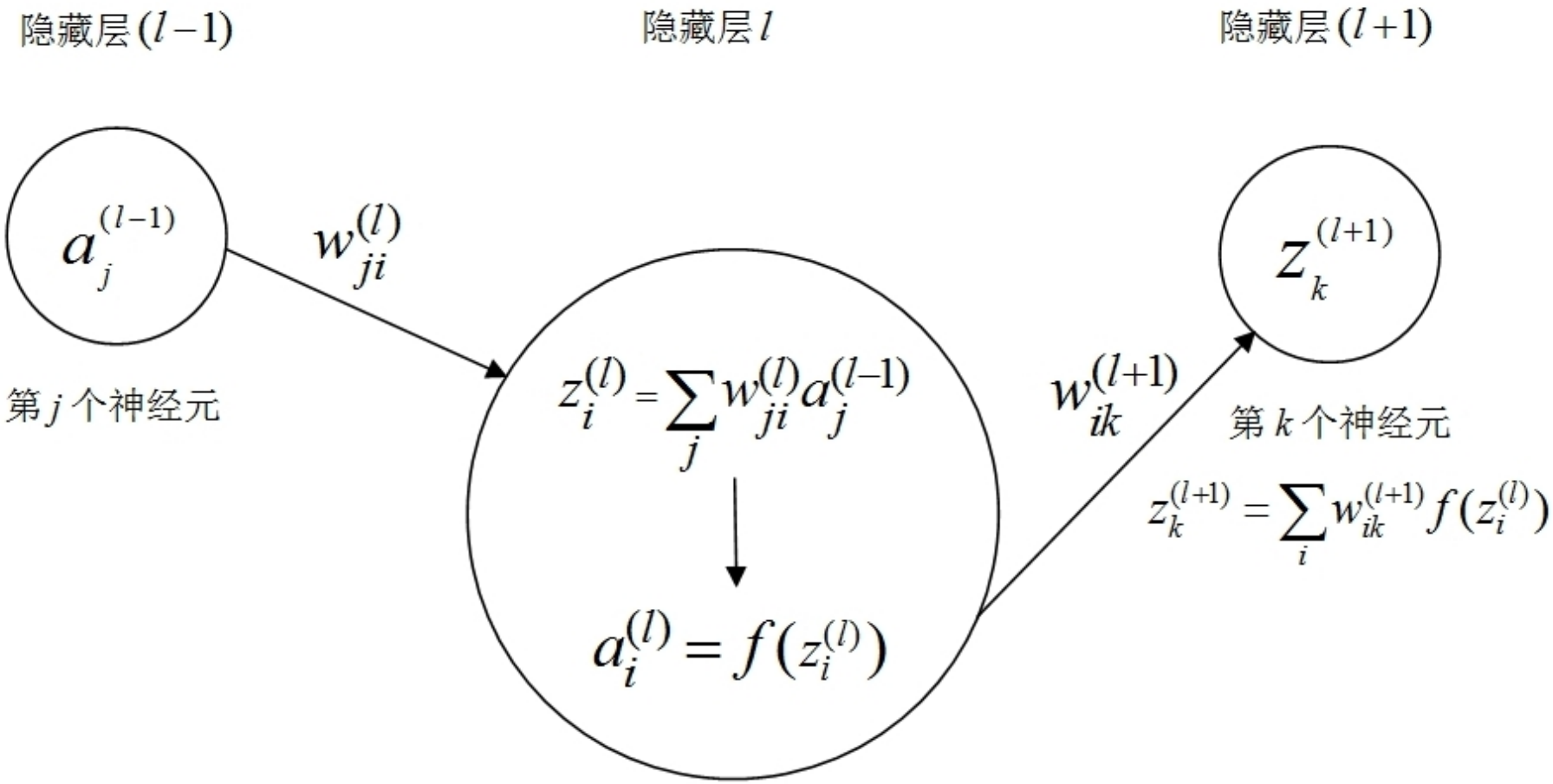

图 15.11 误差反向传播的求导示意图

记神经网络的损失函数为L。考虑将损失函数L对参数$w_{ji}^{(l)}$求导。由于$w_{ji}^{(l)}$仅通过影响净输入$z_i^{(l)}$而作用于L，故根据链式法则可得：

$$\frac{\partial L}{\partial w_{ji}^{(l)}}=\frac{\partial L}{\partial z_i^{(l)}}\cdot\frac{\partial z_i^{(l)}}{\partial w_{ji}^{(l)}}=\underbrace{\frac{\partial L}{\partial z_i^{(l)}}}_{\equiv\delta_i^{(l)}}\cdot a_j^{(l-1)}\equiv\delta_i^{(l)}a_j^{(l-1)} \qquad (15.26)$$

其中，$\delta_i^{(l)}\equiv\frac{\partial L}{\partial z_i^{(l)}}$称为**误差**(error)；因为如果已达到局部最小值，则$\delta_i^{(l)}=0$，无须再更新参数$w_{ji}^{(l)}$。

$z_i^{(l)}$影响损失函数L的途径为，通过第$(l+1)$层所有神经元的净输入$z_k^{(l+1)}$。故再次使用链式法则，即可得到$\delta_i^{(l)}$的表达式：

$$\delta_i^{(l)} \equiv \frac{\partial L}{\partial z_i^{(l)}} = \sum_k \frac{\partial L}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} = \sum_k \delta_k^{(l+1)} \cdot \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} \tag{15.27}$$

其中，根据定义，$\delta_k^{(l+1)} \equiv \frac{\partial L}{\partial z_k^{(l+1)}}$。

进一步。由于$z_k^{(l+1)}=\sum_i w_{ik}^{(l+1)} f(z_i^{(l)})$，故

$$\frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}}=w_{ik}^{(l+1)} f'(z_i^{(l)}) \qquad (15.28)$$

将上式代入方程(15.27)可得：

$$\delta_i^{(l)}=\sum_k \delta_k^{(l+1)}\cdot w_{ik}^{(l+1)} f'(z_i^{(l)})= f'(z_i^{(l)})\sum_k \delta_k^{(l+1)}\cdot w_{ik}^{(l+1)} \quad (15.29)$$

上式将第l隐层的误差$\delta_i^{(l)}$表示为第$(l+1)$隐层的误差$\delta_k^{(l+1)}$之函数，这是一种反向的递推公式。

可以用递归(recursive)的方法计算误差$\delta_i^{(l)}$，然后代入方程(15.26)，即可得到偏导数$\frac{\partial L}{\partial w_{ji}^{(l)}}$。

在计算误差$\delta_i^{(l)} \equiv \frac{\partial L}{\partial z_i^{(l)}}$时，先算最后 1 个隐层的误差，再算倒数第 2 个隐层的误差，以此类推。

这种算法称为**误差反向传播**(error back propagation 或 backward pass)，简称 **BP 算法**(back-propagation algorithm)。

在计算梯度向量时，依然需要知道每一层所有神经元的净输入$z_j^{(l)}$与激活值$a_j^{(l)}$。

故首先需要将每个观测值$(\mathbf{x}_i, y_i)$输入神经网络，从左到右进行**正向传播**(forward propagation 或 forward pass)，得到每一层所有神经元的$z_j^{(l)}$与$a_j^{(l)}$。

然后通过反向传播，计算每一层的误差$\delta_j^{(l)}$；再根据方程(15.26)计算每一层参数的偏导数，并通过梯度下降法更新参数。

在训练神经网络之前，一般建议将全部特征变量**归一化**(normalization，即最小值变为 0 而最大值变为 1)或**标准化**(standardization，即均值变为 0 而标准差变为 1)。

这是因为，如果特征变量的取值范围差别较大，则会影响神经网络的权重参数，不利于神经网络的训练。

对于回归问题，若对特征变量进行归一化处理，则所有特征变量 $x \in [0,1]$。

此时，建议也将响应变量作归一化处理，便于模型的训练与预测。

在选择参数矩阵$\mathbf{W}$的初始值$\mathbf{W}_0$时，一般并不将其所有元素都设为相同的取值(比如，都设为 0 或 1)。

通常从标准正态分布$N(0,1)$或取值介于$[-0.7, 0.7]$的均匀分布中随机抽样，这样有利于不同神经元之间的分化，避免趋同。

15.8 神经网络的小批量训练

对于神经网络的训练，考虑最小化如下损失函数：

$$\min_{\mathbf{w}} \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, G(\mathbf{x}_i;\mathbf{W})\right) \qquad (15.30)$$

其中，$G(\mathbf{x}_i;\mathbf{W})$为一个前馈神经网络模型。由于$G(\mathbf{x}_i;\mathbf{W})$通常是一个高度非线性的函数，故上式中的求和式无法进一步简化。

如果样本容量为 100 万，则目标函数中共有 100 万项相加(对应于 100 万个观测值的损失之和)。

在求损失函数的梯度向量时，需要对每个观测值的损失$L\left(y_i, G(\mathbf{x}_i;\mathbf{W})\right)$分别求梯度向量，然后再将这 100 万个梯度向量加总。

如果样本容量很大，则通常的梯度下降法过于费时，并不可行。

一种解决方法是，每次无放回地(without replacement)随机抽取一个观测值$(\mathbf{x}_i, y_i)$，计算该观测值的梯度向量$\dfrac{\partial L\left(y_i, G(\mathbf{x}_i;\mathbf{W})\right)}{\partial \mathbf{W}}$，然后沿着此负梯度方向，使用合适的学习率$\eta$，进行参数更新：

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial L\left(y_i, G(\mathbf{x}_i;\mathbf{W})\right)}{\partial \mathbf{W}} \qquad (15.31)$$

这种方法称为**随机梯度下降**(Stochastic Gradient Descent，简记 SGD)，最早由 Robbins and Monro(1951)与 Kiefer and Wolfowitz(1952)提出。

SGD 的计算速度大大加快，因为每次仅需计算一个观测值的梯度向量。

但单个观测值的负梯度方向并不一定与整个样本的负梯度方向一致或类似，这导致随机梯度下降的过程充满噪音(noisy)，有时反而会使损失函数上升。

当然，经过不断迭代后，SGD 的长期趋势依然指向损失函数的最小值，参见图 15.12。

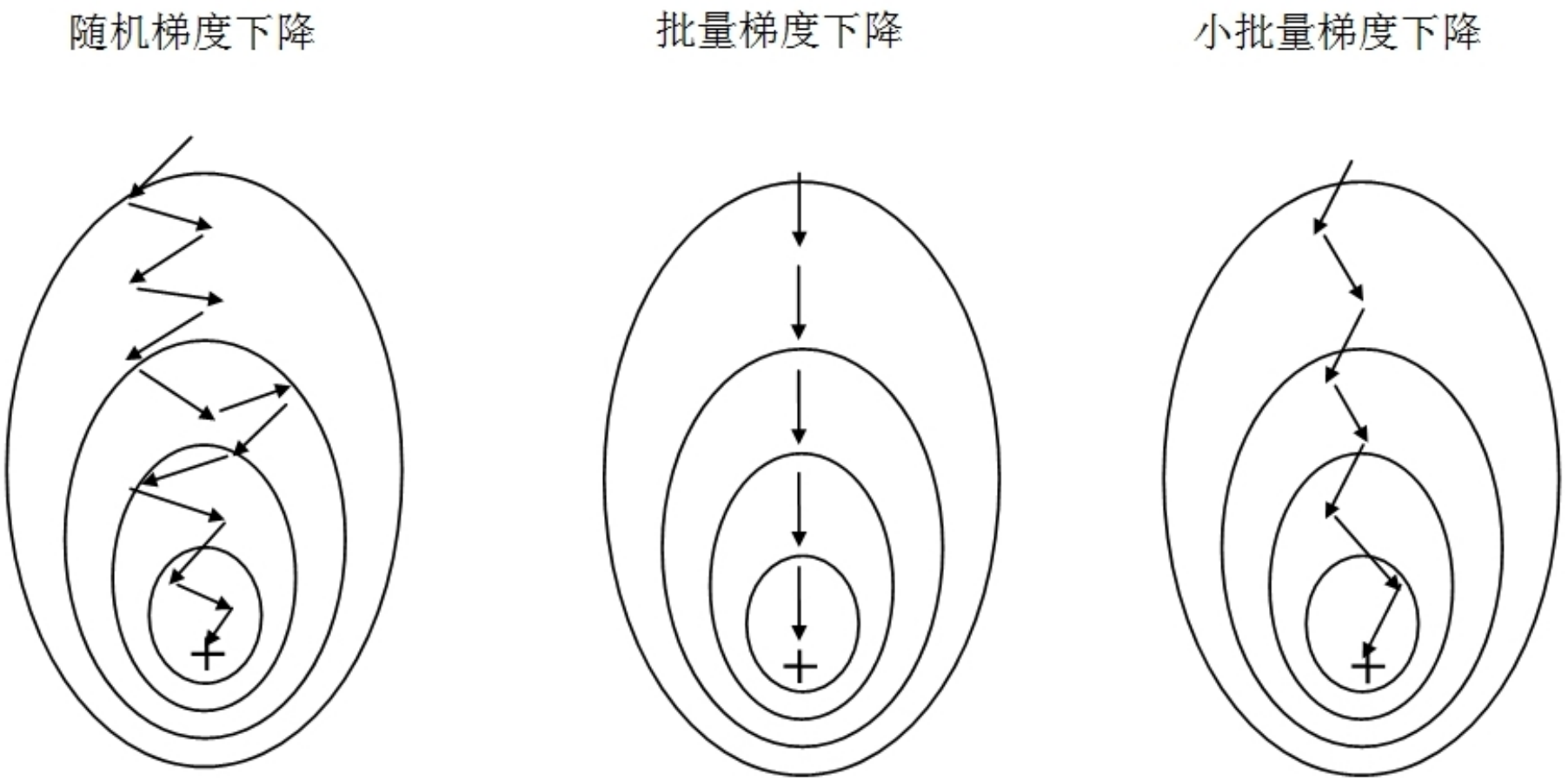

图 15.12 三种梯度下降法的示意图

为克服随机梯度下降的不稳定与噪音，一种折衷方法应运而生。

每次无放回地(without replacement)随机抽取部分观测值，比如B个观测值(例如$B=32$)，计算这B个观测值的梯度向量，再作平均，然后进行参数更新：

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{B} \sum_{i=1}^{B} \frac{\partial L\left(y_i, G(\mathbf{x}_i; \mathbf{W})\right)}{\partial \mathbf{W}} \qquad (15.32)$$

这种方法称为**小批量梯度下降**(Mini-batch Gradient Descent)。

由于B通常不大，故小批量梯度下降依然计算较快。

经过对B个观测值平均之后，可得到对于全样本的真实梯度向量更为准确的估计，故小批量梯度下降的过程更为稳定；尽管目标函数在下降过程中依然会波动。

相对而言，传统的梯度下降法，在计算梯度向量时，同时考虑所有观测值，故称为**批量梯度下降**(Batch Gradient Descent)。

以上三种梯度下降的方法，主要区别在于其**批量规模**(batch size)。

对于随机梯度下降，批量规模$B=1$。

对于批量梯度下降，批量规模就是样本容量，即$B=n$。

对于小批量梯度下降，则$1<B<n$；常见的B选择包括 32，64，128 或 256(设为 2 的指数次方，以适应二进制的 CPU 或显卡 GPU 的内存)。

对于样本容量较大的数据，一般使用小批量梯度下降训练神经网络模型。

另一相关概念为**轮**(epoch)。

在训练模型时，将所有样本数据都用了一遍，即称为“一轮”(one epoch)。经过一轮之后，所有观测值都有机会影响参数更新。

对于批量梯度下降，每次迭代(iteration)都用全部样本计算梯度向量，故一次迭代就是一轮。

对于随机梯度下降，每次仅用一个观测值计算梯度向量，故n次迭代才算一轮(n为样本容量)。

对于小批量梯度下降，由于每次无放回地使用B个观测值计算梯度向量，故n/B(假设可整除)次迭代后，才算一轮。

15.9 神经网络的正则化

包含多个隐藏层的深度神经网络是表达能力很强的模型(very expressive models)，可以学习输入与输出之间非常复杂的函数关系。

如果进行很多轮(epoch)的训练，则容易导致过拟合。

需要对神经网络模型进行“正则化”(regularization)处理。常见的正则化方法包括：

(1) **早停**(Early Stopping)：提前停止训练，而不必等到神经网络达到损失函数或训练误差的最小值。

如何知道在何时停止训练呢？

一般建议将全样本随机地一分为三，即训练集(training set)、验证集(validation set)与测试集(test set)。

首先，在训练集中进行训练，并同时将学得的神经网络模型同步地在验证集中作预测，并计算“验证误差”(validation error)。

其次，当验证误差开始上升时，即停止训练。

最后，将所得的最终模型在测试集中进行预测，并计算“测试误差”(test error)，参见图 15.13。

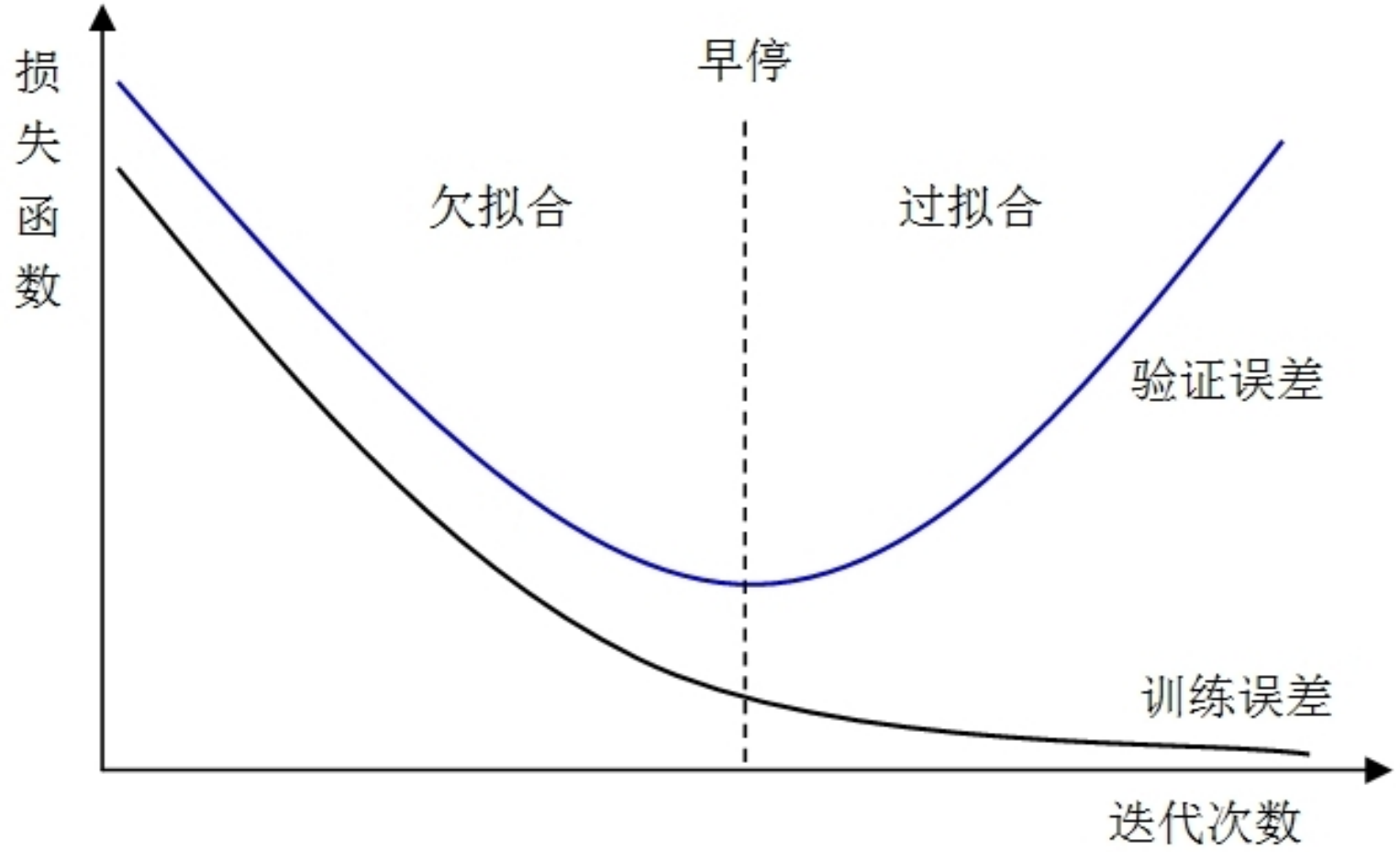

图 15.13 以早停防止过拟合

(2) 丢包(Dropout)。

为避免过拟合，Geoffrey Hinton 的团队(Srivastava et al., 2014)提出，在训练样本时，随机地让某些神经元的激活值取值为 0，即让某些神经元“死亡”，而不再影响神经网络。

通常随机地丢弃 50%的神经元(以及它们在网络中的连接)，这样可以迫使神经网络不过分依赖于某些神经元而导致过拟合，参见图 15.14。

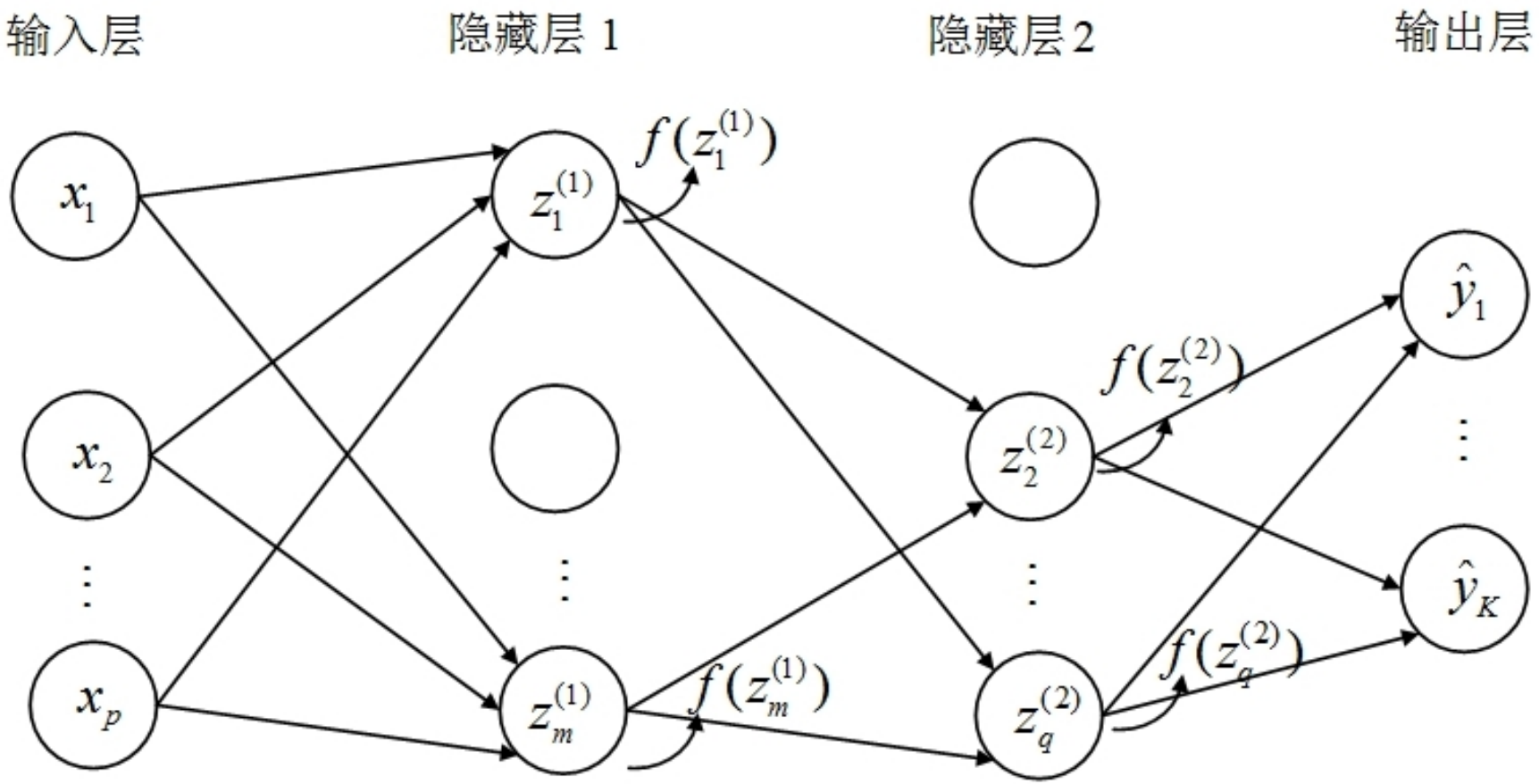

图 15.14 以丢包防止过拟合

(3) **惩罚**(Penalization)。在神经网络模型的目标函数中，可引入类似于“岭回归”(ridge regression)的L_2惩罚项，以进行正则化：

$$\min_{\mathbf{W}} \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, G(\mathbf{x}_i;\mathbf{W})\right)+\lambda\left\|\mathbf{W}\right\|_2^2 \qquad (15.33)$$

其中，$\left\|\mathbf{W}\right\|_2$为矩阵$\mathbf{W}$的“弗罗宾尼斯范数”(Frobenius norm)，即矩阵$\mathbf{W}$所有元素的平方和之开根号；$\lambda>0$为调节参数，可通过交叉验证或验证集法确定。

这是一个“收缩估计量”(shrinkage estimator)，但在神经网络的文献中称为**权重衰减**(weight decay)。

15.10 卷积神经网络

在“计算机视觉”(computer vision)领域，如果使用前馈神经网络进行图像识别，则会导致参数太多、丢失空间信息等问题：

(1) 参数太多。假设输入图像大小为$100\times100\times3$的张量(tensor)，即高度为 100 像素，宽度为 100 像素，且包含 3 个颜色通道(RGB)。可将此张量排列成一个 30,000 维的长向量。

第 1 个隐藏层的每个神经元，到输入层都有 3 万个相互独立的连接。

随着隐藏层神经元的数目增加，参数的规模也急剧上升。这导致整个神经网络的训练效率很低，且易出现过拟合。

(2) 丢失空间信息。

在上述例子中，在将$100\times100\times3$的张量扁平化为 3 万维长向量时，事实上丢失了原来图像中空间相邻的信息。

对于自然图像中的物体，一般都有“局部不变性”(local invariance)的特点，比如在平移、旋转与尺度缩放等操作下不影响其内在信息。

由于全连接网络失去空间相邻信息，故很难提取这种局部不变特征，导致神经网络的预测能力下降。

卷积神经网络(Convolutional Neural Network，简记 CNN 或 ConvNet)是受到生物学中“感受野”(receptive field)的机制启发而提出。

所谓“感受野”，主要是指视觉、听觉等神经系统中一些神经元的特性，即神经元只接受其所支配的刺激区域之信号。

相邻神经元的感受野有部分重叠。

如果神经元的感受野不受限制，则每个神经元都会“太忙”，这相当于全连接网络中的每个隐藏层神经元均需接收巨量的输入信息。

在数学上，对于感受野机制的抽象，就是所谓**卷积运算**(convolution)。使用卷积运算的神经网络，称为“卷积神经网络”。

卷积神经网络的雏形为日本学者福岛邦彦(Fukushima, 1980)提出“新认知机”(Neocognitron)，但使用非监督学习(unsupervised learning)进行训练。

LeCun et al. (1990)将监督学习的反向传播算法用于训练 CNN，并在手写数字的识别上取得成功。

作为卷积运算的示例，假设图像为简单的5×5像素矩阵，我们想用一个3×3的“过滤器”(filter)，也称**卷积核**(convolution kernel)，对此图像进行卷积运算(以$\otimes$表示)，参见图 15.15。

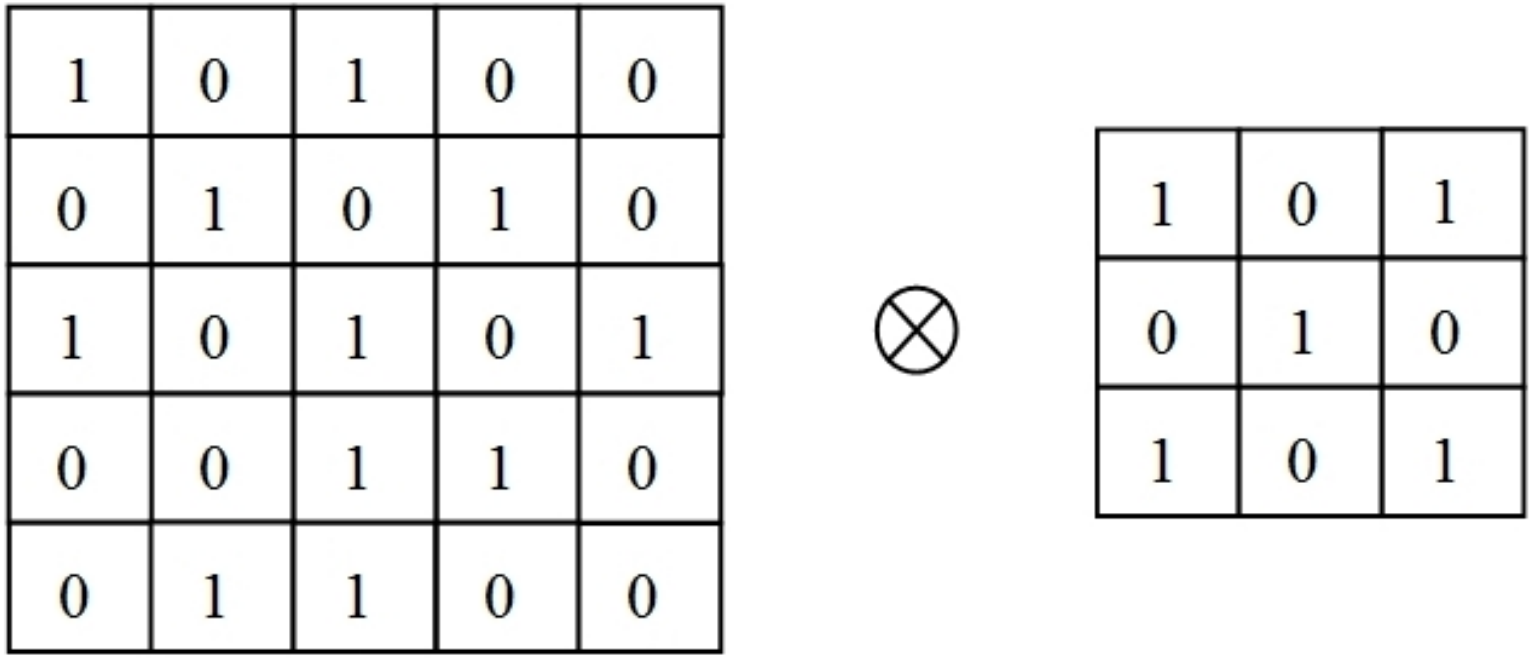

图 15.15 图像与卷积核

将卷积核在输入图像上滑动(slide the filter over the input image)，并将相应元素相乘(element-wise multiply)，再把结果加总(add the outputs)。

先将过滤器与图像左上角的3×3矩阵按元素相乘，然后再加总，结果为 5，参见图 15.16。

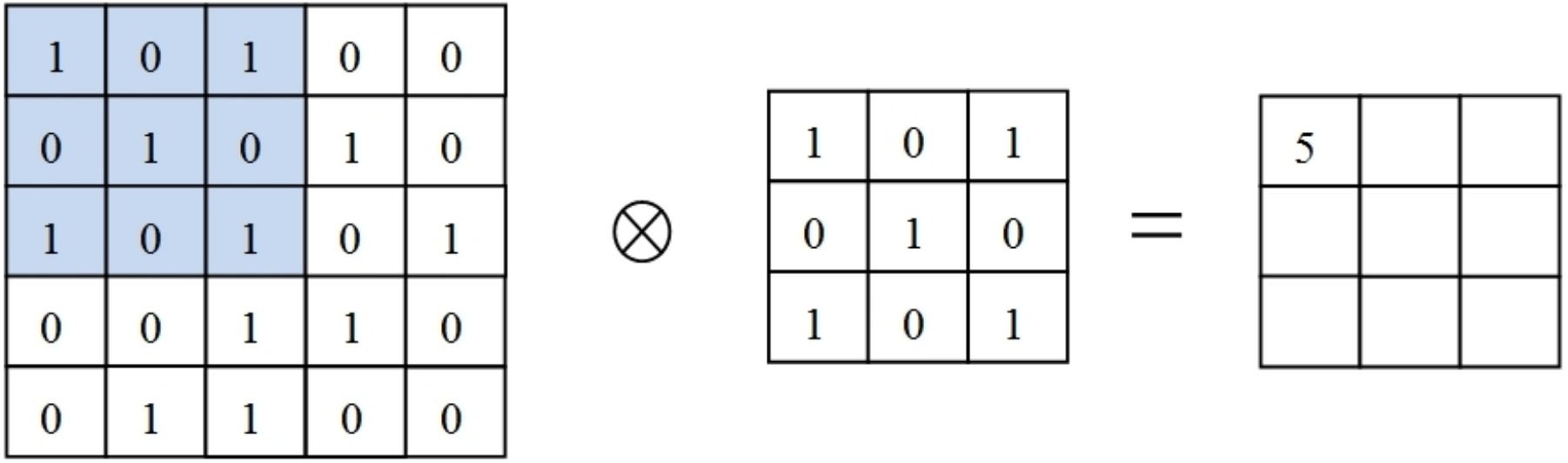

图 15.16 卷积运算的第一步

作为卷积运算的第二步，将过滤器与图像上方中间的3×3矩阵按元素相乘(element-wise multiplication)，然后再加总，结果为 0，参见图 15.16。

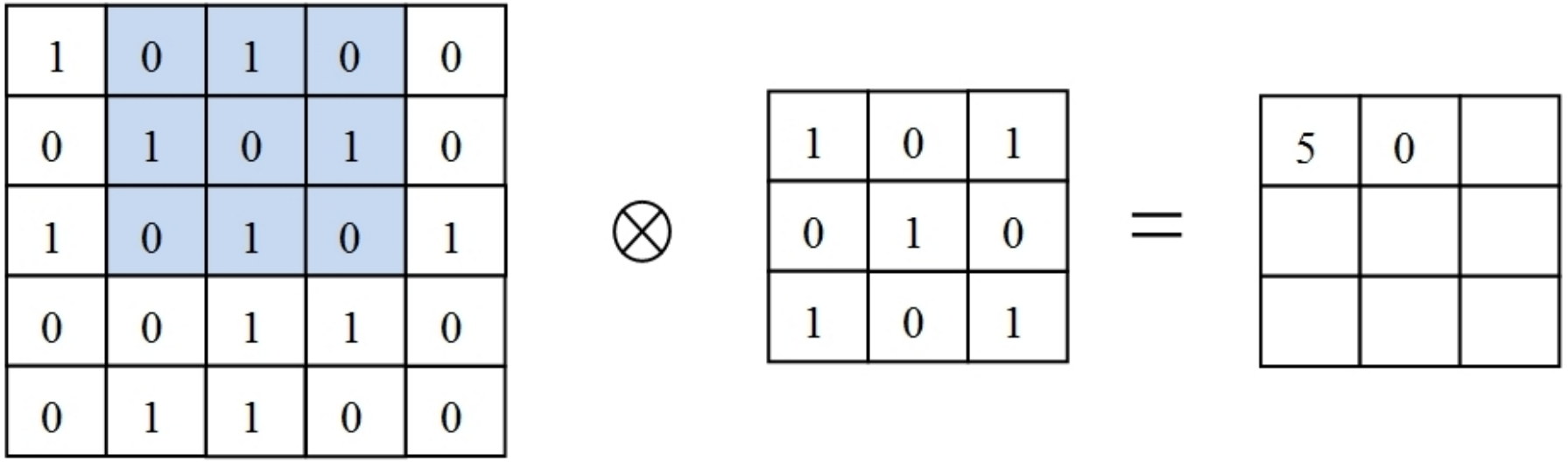

图 15.17 卷积运算的第二步

重复以上步骤，直至将过滤器在图像上全部滑动一遍，所得结果参见图 15.18。

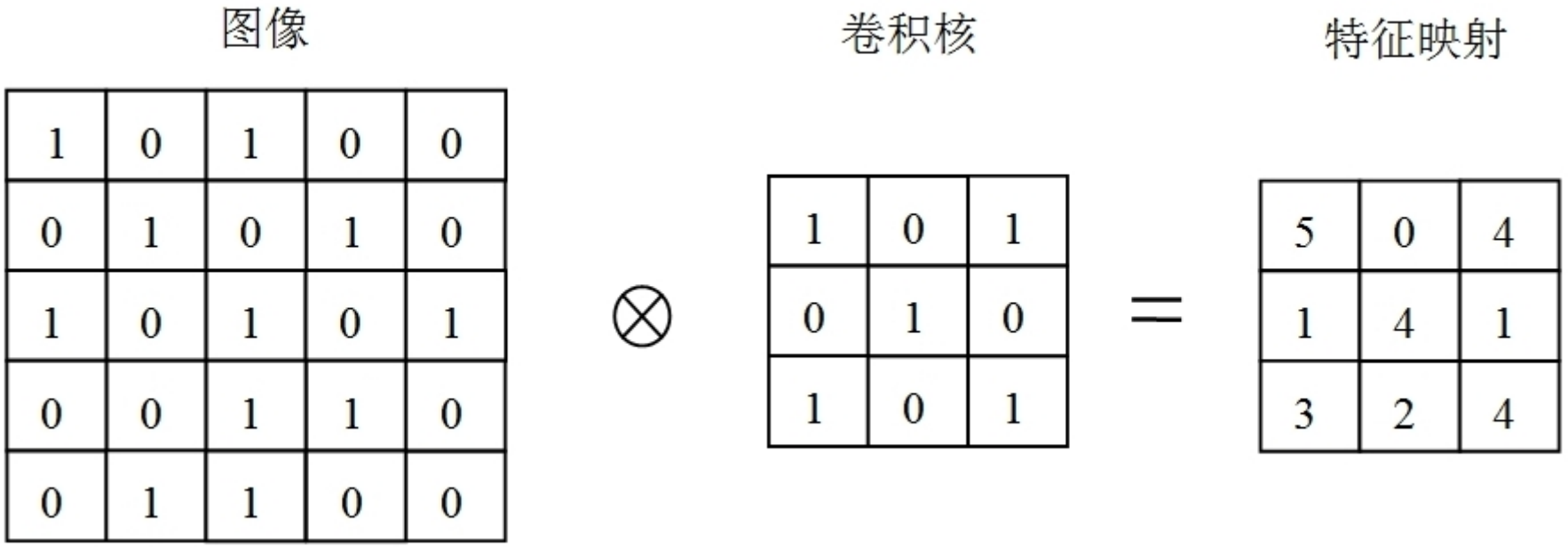

图 15.18 卷积运算的最终结果

卷积运算的意义何在？

在此例中，过滤器的形状类似于大写英文字母X。

过滤器与图像左上角的卷积运算结果为最高值 5，这表明图像左上角的局部区域与过滤器所代表的X形状最为相似(完全相同)。

过滤器与图像上方中间的卷积运算结果为最低值 0，这说明图像上方中间的局部区域与过滤器的X形状差别最大(完全相反，毫无共同之处)。

其他区域的卷积运算结果则介于二者之间，说明这些区域与过滤器的X形状有着不同的相似度。

卷积运算的结果相当于度量了图像中每个局部区域与过滤器的相关程度，故卷积运算也称为**互相关**(cross-correlation)。

卷积运算可以起到从图像中抽取特征(feature extraction)的作用，而卷积运算的结果则称为**特征映射**(feature map)。

在使用一个过滤器与整个图像进行卷积运算时，过滤器的权重参数并不改变，故卷积神经网络具有**权值共享**(weight sharing)的特点，可节省参数。

过滤器中的参数本身，也是神经网络学习的结果。

由于一个卷积核只能提取一种局部特征(local feature)，故需要使用多个卷积核来抽取图像的不同特征，由此构成**卷积层**(convolutional layer)。

然而，经过卷积层运算之后，所得的特征映射(feature map)之维度依然很高(参见图 15.18)。

为此，需要通过一个**汇聚层**(pooling layer)，也称“池化层”，进一步汇聚与压缩信息。

所谓“汇聚”(pooling)，也称为“下采样”(down sampling)或“子采样”(subsampling)，一般有以下两种常用的实现方式：

(1) 最大汇聚(maximum pooling，简称 maxpooling)，即在特征映射中，抽取每个局部区域的最大值，参见图 15.19。

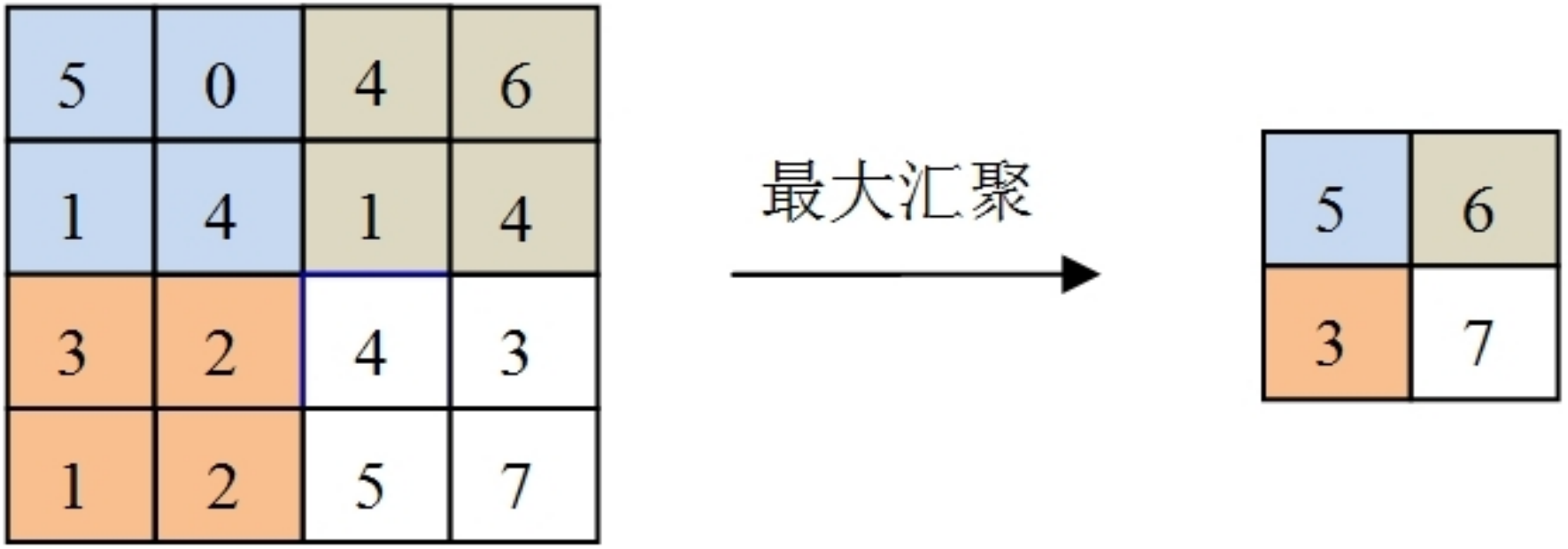

图 15.19 最大汇聚

(2) 平均汇聚(mean pooling)，即在特征映射中，抽取每个局部区域的平均值，参见图 15.20。

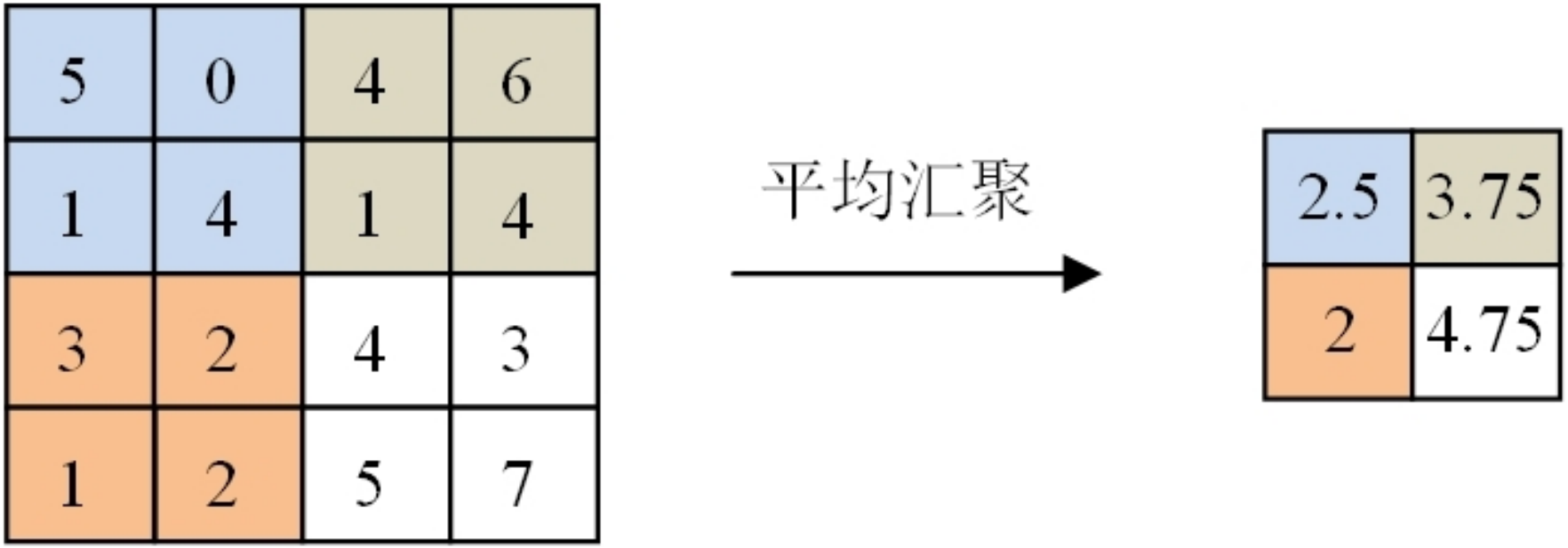

图 15.20 平均汇聚

经过汇聚层之后，图像的局部特征已被抽取并压缩，得到为数较少的特征变量，然后可连接到通常的(全连接)前馈神经网络。

对于分类问题，卷积神经网络的最后一层一般使用多项逻辑(Multinomial Logit)的**软极值函数**(softmax function)作为激活函数，其输出结果就是各类别的条件概率，并以条件概率最大者作为预测类别。

使用卷积神经网络进行图像识别的基本网络结构，参见图 15.21。

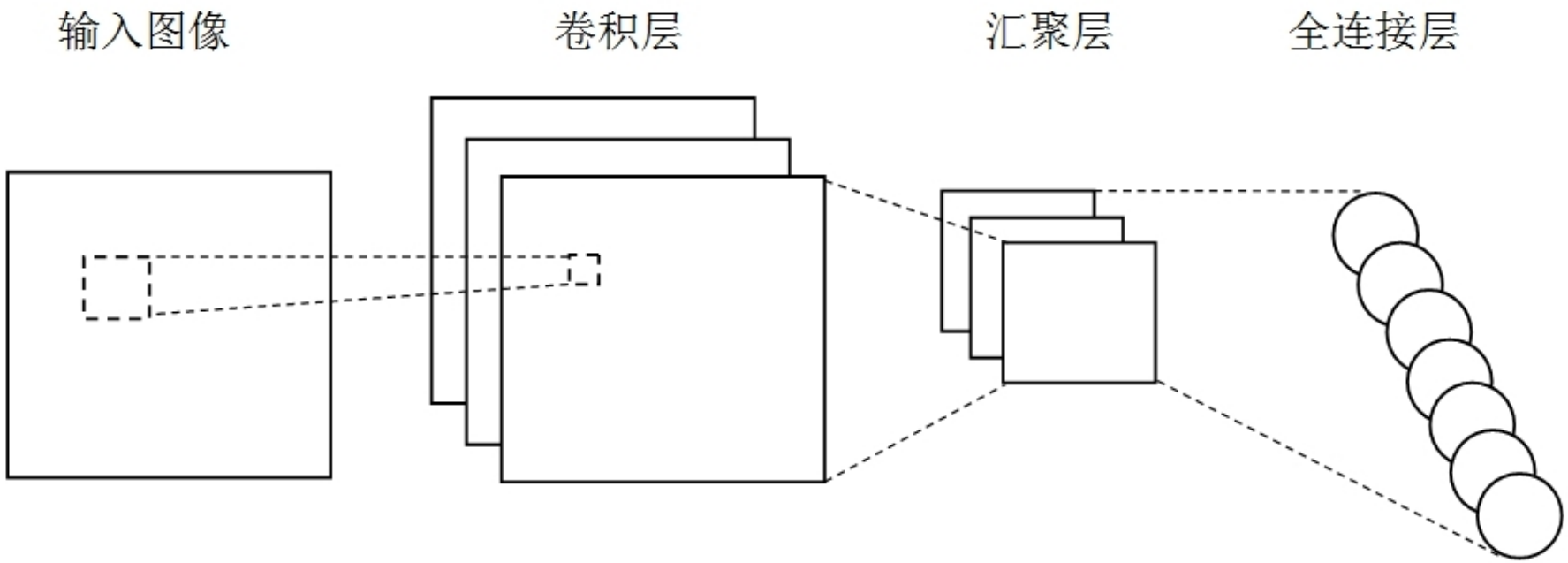

图 15.21 卷积神经网络的基本结构

在图 15.21 的最左边为输入层，之后为卷积层，接着是汇聚层，然后将汇聚层的输出结果，接入全连接层进行分类识别。

对于比较复杂的图像识别任务(比如人脸识别)，仅使用一个卷积层与一个汇聚层还不够，因为这只是提取了图像中的“初级特征”(low level features)，比如边缘(edges)与黑点(dark spots)。

在此之后，再加上一个卷积层与一个汇聚层，以提取“中级特征”(mid level features)，比如眼、耳、鼻。

最后，还需再加上一个卷积层与一个汇聚层，才能提取“高级特征”(high level features)，比如面部结构。

对于卷积神经网络的训练，依然可使用反向传播算法。

一般来说，汇聚层中并没有需要估计的参数(只是求局部最大值或平均值的运算而已)。

对于卷积层来说，本质上可视为部分连接的隐藏层，而未连接部分的权重参数则预先设为零。

LeCun et al. (1990)揭开了现代 CNN 的研究序幕。LeCun 将自己的卷积神经网络命名为 LeNet。图 15.22 为 LeNet-5 的网络结构，共有 7 层(不包括输入层)。

基于 LeNet-5 的手写数字识别系统在 1990 年代被美国很多银行采用，用来识别支票上的手写数字。

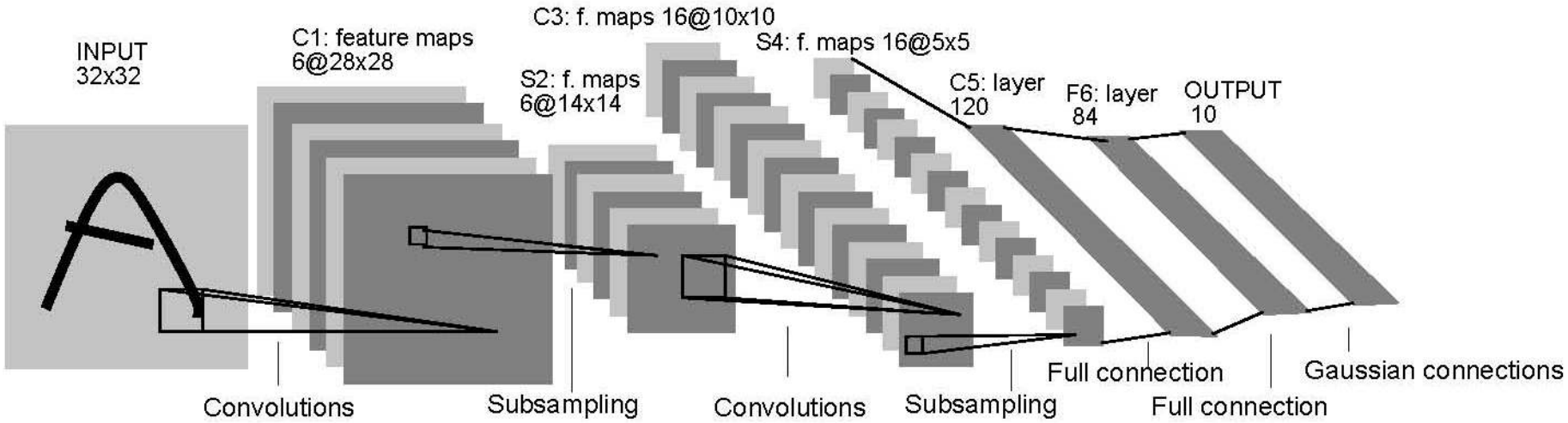

图 15.22 LeNet-5 的网络结构(LeCun et al., 1998)

2012 年出现的 AlexNet (Krizhevsky, Sutskever and Hinton, 2012)则是第一个现代深度卷积网络模型，参见图 15.23。

它首次采用了很多现代深度神经网络的技术方法，比如使用显卡(Graphic Processing Unit，简记 GPU)进行并行训练(图 15.23 的上下两层各用一个 GPU)，采用 ReLU 作为激活函数，使用“丢包”(dropout)防止过拟合，以及使用“数据增强”(data augmentation)提高模型准确率等。

AlexNet 包含 5 个卷积层、3 个全连接层与 1 个使用 softmax 激活函数的输出层，共有 6 千万个参数、65 万个神经元，对 1000 种图像进行分类。

AlexNet 赢得了 2012 年 ImageNet 图像分类竞赛的冠军。

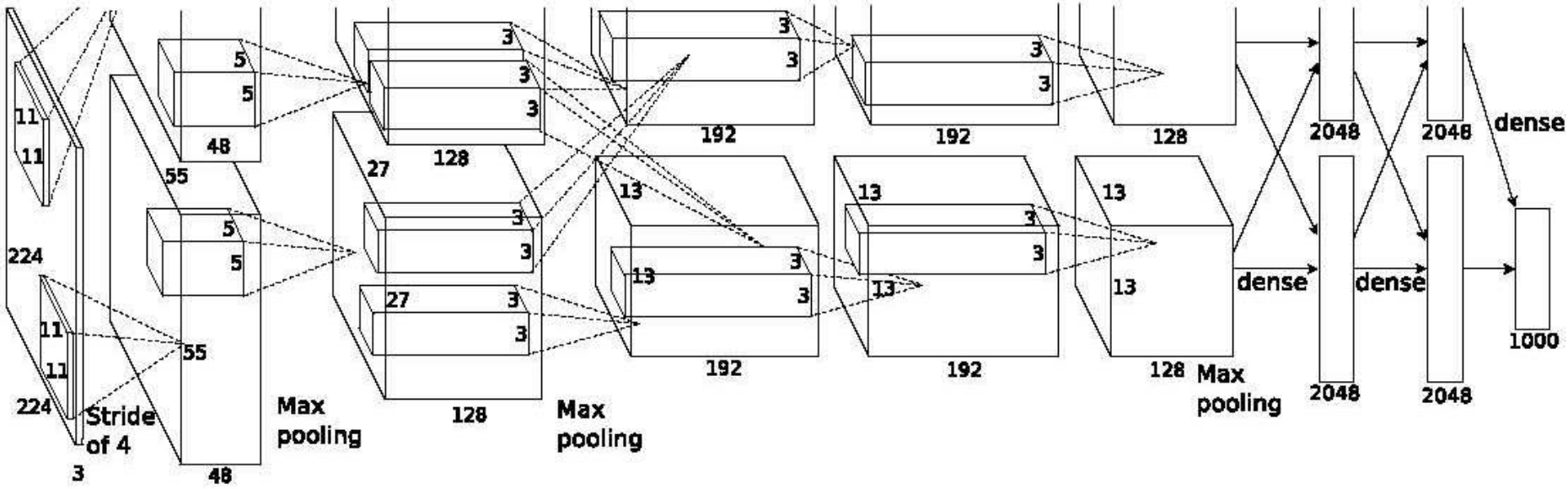

图 15.23 AlexNet 的网络结构(Krizhevsky et al., 2012)

此后，深度神经网络变得越来越深，参数也越来越多。

2014 年谷歌的 GoogLeNet 包含 22 层，为 2014 年 ImageNet 图像分类竞赛冠军(Szegedy et al., 2015)。

2015 年微软的“残差网络”(Residual Network，简记 ResNet)使用多达 152 层的神经网络(He et al. 2016)，使图像分类错误率降到 3.57%，比人类的错误率 5.1%还更低。

此处的“残差”指的是真实函数减去“恒等函数”(identity function)。假设要估计的真实函数为 $f(\mathbf{x})$，则残差函数(residual function)为 $f(\mathbf{x})-\mathbf{x}$。如果$\mathbf{x}$的维度与 $f(\mathbf{x})$不同，可通过线性投影(linear projection)使二者的维度相等，即 $f(\mathbf{x})-\mathbf{W}\mathbf{x}$，其中$\mathbf{W}$为投影矩阵。

残差网络的优势在于，估计残差 $f(\mathbf{x})-\mathbf{x}$通常比估计 $f(\mathbf{x})$更容易。

2018 年，Yoshua Bengio，Geoffrey Hinton 与 Yann LeCun 因对深度神经网络的开创性贡献而获得图灵奖(计算机领域的最高奖)。

15.11 使用 sklearn 估计回归问题的神经网络

以波士顿房价数据 `boston` 为例(参见第 4 章)，演示回归问题的神经网络模型。

15.12 使用 sklearn 估计分类问题的神经网络

作为分类问题的神经网络案例，使用垃圾邮件数据 spam (参见第 8 章)，以及月亮形状的模拟数据来演示。

15.13 使用 Keras 估计回归问题的神经网络

Sci-Kit Learn 虽是流行的机器学习模块，但并非专业的深度学习框架(deep learning framework)。

首先，sklearn 只能估计前馈神经网络，而无法估计卷积神经网络、循环神经网络等其他神经网络模型。

其次，sklearn 只能使用 CPU，无法使用“图像处理器”(Graphics Processing Unit，简记 GPU，也称“显卡”)进行加速，故不适用于大数据的场景。

从本节开始使用 Keras 进行深度学习。

Keras 是一个容易上手的高层深度学习平台。

Keras 可调用一些低层的深度学习框架，包括谷歌的 TensorFlow，微软的 Cognitive Toolkit(CNTK)，以及加拿大蒙特利尔大学(University of Montreal)的 Theano，并以这些深度学习框架作为其“后端引擎”(backend engine)。

一般建议使用 TensorFlow 作为后端引擎。

由于 TensorFlow 模块比较复杂，故建议在 Python 设立一个专门运行 TensorFlow 与 Keras 的“环境”(environment)，以避免与其他 Python 模块冲突。

在 Anaconda 提示符(Anaconda Prompt)之下，可按照以下步骤，创建 TensorFlow 环境，并在其中安装 TensorFlow，Spyder，Keras 以及其他相关模块：

第一步、在 Anaconda Prompt(提示符)依次输入以下命令：

```
conda create -n tensorflow_env tensorflow
conda activate tensorflow_env
conda install spyder-kernels
```

```
conda install keras
conda install scikit-learn
conda install pandas
conda install matplotlib
conda install seaborn
python -c "import sys; print(sys.executable)"
```

其中，第 1 个命令创建一个名为“tensorflow_env”(可自行命名)的环境，并在其中安装 TensorFlow。第 2 个命令进入此 tensorflow_env 环境。

第 3-8 个命令在 tensorflow_env 环境中安装 Spyder，Keras，Sci-Kit Learn，pandas，Matplotlib 与 Seaborn。最后 1 个命令打印 Python 程序所在的位置。

第二步、复制第一步最后 1 个命令所打印的 Python 程序所在位置。该位置的结尾应为 python，pythonw，python.exe 或 pythonw.exe，取决于电脑的操作系统。比如，
“C:\ProgramData\Anaconda3\envs\tensorflow_env\python.exe”。

第三步、启动 Spyder，点击菜单“Tools”→ “Preferences”→ “Python Interpreter”→ “Use the following Python interpreter”，然后粘贴第二步的 Python 程序所在位置，点击 “Apply”与 “OK”。重启 Spyder，即可使用 Keras 与 TensorFlow。

若不再使用 Keras 与 TensorFlow，在 Spyder 中点击菜单“Tools”→“Preferences”→“Python Interpreter”→“Default”，然后点击“Apply”与“OK”，再重启 Spyder 即可。

另外，在 Anaconda Prompt 之下，若要退出 `tensorflow_env` 环境，可输入命令：`conda deactivate`

反之，退出 `tensorflow_env` 环境之后，若要再次进入，可输入命令：`conda activate tensorflow_env`

以波士顿房价数据演示回归问题的神经网络模型。

* 详见教材，以及配套 Python 程序（现场演示）。

15.14 使用 Keras 估计二分类问题的神经网络

对于二分类问题，使用 Keras 估计神经网络模型的流程与上述回归问题类似。

主要差别在于损失函数应设为“`binary_crossentropy`”(二值交叉熵)。

我们使用垃圾邮件数据 `spam` 来演示(详见第 8 章)。

* 详见教材，以及配套 Python 程序（现场演示）。

15.15 使用 Keras 估计多分类问题的神经网络

对于多分类问题，使用 Keras 估计神经网络模型的流程与上述二分类问题类似。

主要差别在于损失函数应设为“`categorical_crossentropy`”。

使用路透社新闻数据 `reuters` 进行演示。

此数据常作为文本分类(text classification)的案例，它将新闻报道分为 46 种不同的主题(topics)。

* 详见教材，以及配套 Python 程序（现场演示）。

15.16 使用 Keras 估计卷积神经网络

以著名的手写数字数据集 MNIST 为例，演示用 Keras 估计卷积神经网络模型。

MNIST 表示“Modified National Institute of Standards and Technology database”。

MNIST 数据包含 60000 个手写数字的训练图像(training images)与 10000 个测试图像(test images)。每个手写数字图像均由28×28的像素矩阵构成(取值介于 0 与 255)。

* 详见教材，以及配套 Python 程序（现场演示）。