2 神经网络

深度学习网络是指隐层大于一层的神经网络，所以了解神经网络的基本原理是学习深度学习必不可少的步骤。

本章首先讨论生物神经元的相关研究，然后介绍常用的神经元，最后以感知机和深度神经网络为例介绍神经网络的基本原理。

2.1 在神经科学中对生物神经元的研究

神经元相互连接组成神经网络。每一个神经元从其他神经元处获得输入信息，少部分神经元也从接收器获得信息；神经元处理这些输入信息，一旦被激活，就会继续发送信号至其他相连的神经元。

机器学习中的神经元以生物神经元为原型，受到了其机制不少的启发和影响；然而，为了可以顺利地完成模型的实现，不可避免的，对机器学习中的神经元进行了不少抽象和简化，甚至有些已经跳出了生物神经元的束缚。下面我们首先了解生物神经元的机制，然后再详细了解机器神经网络的神经元模型。

2.1.1 神经元激活机制

神经生物学家 David Hubel 和 Torsten Wiesel 由于发现了“视觉系统的信息处理”而荣获 1981 年的诺贝尔生理或医学奖。1958 年，他们通过实验证实了位于后脑皮层的神经元与视

觉刺激之间存在某种对应关系。换句话说，一旦视觉受到了某种刺激，后脑皮层的特定部分的神经元就会被激活。他们的实验发现了一种被称为“方向选择性细胞”的神经元，当看到眼前物体的边缘，而且这个边缘指向某一个方向时，这种神经元就会被激活。

神经生物学家认识到，生物神经元是神经系统的重要组成单位之一。随后的深入研究揭示了神经元的基本构造由细胞体和神经突（包括树突、轴突、突触）组成，如图 2-1 所示。树突呈树状分支，为神经元的“信息接收区”，它将受到刺激引起的电位变化向胞体传递；然后会有一个“触发区”负责整合电位，决定是否达到阈值，从而产生神经冲动；细长的轴突为“传导区”，而其末端的突触为“输出区”——神经冲动会导致突触释放出神经传递物质或者电力，从而实现将整合的信息向下一个神经元进行传递的过程。

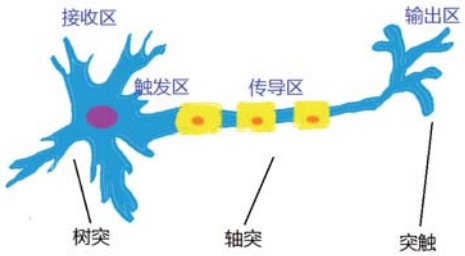


图 2-1　生物神经元模型

机器学习中的神经元也暗合了这几个功能区域：接收区、触发区、传导区、输出区。其中，触发区最重要，不同的触发机制也标志着不同类型的神经元。

2.1.2　神经元的特点

除神经元的基本激活机制以外，科学家发现，大脑不同位置的神经元似乎专门实现各自的功能。尽管如此，但是各种神经元本身的构成却很相似。在研究中甚至发现，早期的大脑损伤，其功能可能是以其他部位的神经元来代替实现的。当然，在生物体中，这需要在非常早期才有可能。有趣的是，在深度学习中也有类似的实现：在一个数据集上训练成型的深度神经网络，在另一个完全不同的数据集上只需稍加训练，就有可能适应和完成那个新的任务。这在机器学习中被称为“迁移学习”（Transfer Learning）。

此外，科学家还发现，神经元具有稀疏激活性，即尽管大脑具有多达五百万亿个神经元，但真正同时被激活的仅有 1%~4%。这种稀疏激活性也影响了机器学习中的神经元的模型设计，比如稍后提到的 ReLU 神经元，对小于 0 的输入都进行了抑制，极大地提高了选择性激活的特征。在 Dropout 及其他剪连接策略中，稀疏性也得到应用。

上面简单了解了生物神经元的机制后，下面来具体了解机器学习中的神经元的代表模型。

2.2 神经元模型

前文提到，生物神经元被以多种形式抽象和简化，但它们都有一个共同的特征，即由输入、激活函数、输出构成。各种神经元的简化模型的不同之处就在于激活函数不一样。图 2-2 列出了几种基本神经元。

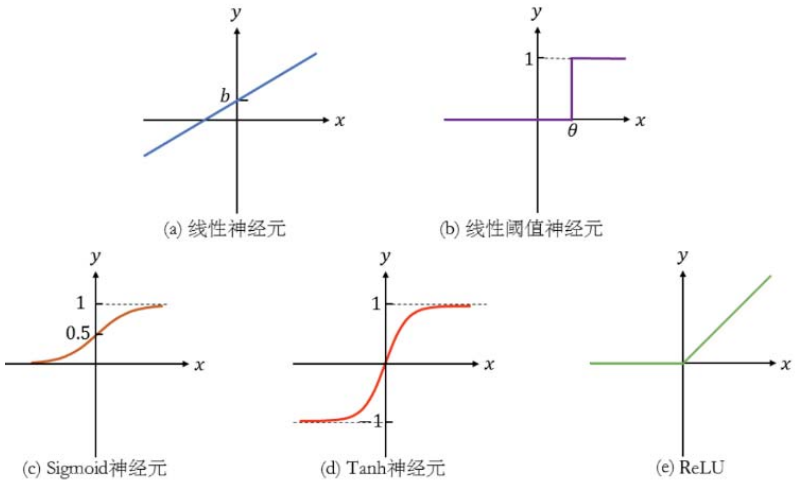


图 2-2 神经元的简化模型

2.2.1 线性神经元

线性神经元（Linear Neuron）是指输出与输入呈线性关系的一种简单模型。其表达式为 $y = wx + b$。如图 2-2（a）所示，它实现的是输入信息的完全传导。在现实中，由于其缺乏对信息的整合而基本不被使用，仅作为一个概念基础。

2.2.2 线性阈值神经元

早在 1943 年，人工神经网络（Artificial Neural Network，ANN）的提出者 Warren Sturgis McCulloch（1898—1969 年）和 Walter Harry Pitts（1923—1969 年）就分析了一种简单的人

工神经元模型，并且指出了它们运行简单逻辑运算的机制。这种简单的神经元采用线性神经元和二值“开/关”相结合，称为线性阈值神经元（Linear Threshold Neuron），也被称为 McCulloch-Pitts 神经元。它具有以下特征。

- 输入和输出都是二值的。
- 每个神经元都具有一个固定的阈值 θ。
- 每个神经元都从带有权重的激活突触接收输入信息。
- 抑制突触对任意激活突触有绝对否决权。
- 每次汇总带权突触的和，如果大于阈值 θ 而且不存在抑制突触输入，则输出为 1，否则为 0。

假定神经元的 n 个输入为 $x_1, x_2, x_3, \cdots, x_n$，输出为 y，那么每次汇总的和为：

$$\text{sum} = \sum_{i=1}^{n} w_i x_i$$

$$y = f(\text{sum}) = \begin{cases} 1, & \text{sum} \geqslant \theta \text{且无抑制突触输入} \\ 0, & \text{其他} \end{cases}$$

其中，w_i 就是权重，sum 是带权和，θ 是阈值，f 是一个与阈值 θ 相关的线性阈值函数（Linear Threshold Function）。抑制突触输入可以理解为一个特权开关，一旦其值为 1，则输出必为 0。

以函数 $y = \bar{x}_1 x_2 + x_2 \bar{x}_3$ 为例，其中 x_1, x_2, x_3 是布尔输入，y 是上帝知道的真实标注（Label），$\hat{y}$ 是神经元的输出。现假定权重向量 $\boldsymbol{w} = [-1, 2, -1]$，阈值 θ 为 $\frac{1}{2}$，且没有抑制突触输入。由之前定义可知，$\text{sum} = \sum_{i=1}^{n} w_i x_i = -x_1 + 2x_2 - x_3$。考虑 x_1, x_2, x_3 的 8 种不同取值，我们可以得到如表 2-1 所示的相关结果。可以看出，这个神经元能完美模拟这个布尔函数。

表 2-1　布尔函数 $y = \bar{x}_1 x_2 + x_2 \bar{x}_3$ 的“开/关”神经元计算表

x_1	x_2	x_3	sum	$\hat{y}$	y
0	0	0	0	0	0
0	0	1	−1	0	0
0	1	0	2	1	1
0	1	1	1	1	1
1	0	0	−1	0	0
1	0	1	−2	0	0

续表

x_1	x_2	x_3	sum	$\hat{y}$	y
1	1	0	1	1	1
1	1	1	0	0	0

2.2.3 Sigmoid 神经元

Sigmoid 神经元可以使输出平滑而连续地限制在 0~1 的范围内，它靠近 0 的区域接近于线性，而远离 0 的区域为非线性。Sigmoid 神经元可以将实数“压缩”至 0~1 的范围内，大的负数趋向于 0，大的正数则趋向于 1。

Sigmoid 神经元的数学表达式为：

$$y = \frac{1}{1+\mathrm{e}^{-x}}$$

虽然它的激活函数看起来比前面的模型要复杂不少，但是它的求导结果很漂亮（在训练神经网络中需要计算激活函数的导数）。具体的求导运算如下：

$$\begin{aligned}\frac{\partial y}{\partial x} &= -\frac{1}{(1+\mathrm{e}^{-x})^2}\cdot \mathrm{e}^{-x}\cdot(-1)\\ &= \frac{\mathrm{e}^{-x}}{(1+\mathrm{e}^{-x})^2}\\ &= \frac{1}{1+\mathrm{e}^{-x}}\cdot\frac{1+\mathrm{e}^{-x}-1}{1+\mathrm{e}^{-x}}\\ &= y\cdot(1-y)\end{aligned}$$

由此可见，Sigmoid 的导数可以直接用它的输出值来计算，非常简单。

Sigmoid 函数在过去被广泛使用，除求导简单外，还源于它很好地阐释了一个神经元的“燃烧率（Firing Rate）”：从一个假定的完全不激活（0）到完全饱和的燃烧（1）。

但 Sigmoid 神经元近年来变得鲜少使用。它的两个主要缺陷如下：

（1）Sigmoid 函数进入饱和区后会造成梯度消失

Sigmoid 神经元的一个非常不受欢迎的属性是在函数两端响应趋向于饱和（接近于 0 或 1），这些区域的梯度几近于 0。在后向传播中，这个（局部）梯度将以乘数的关系进入整个优化过程。这样，如果局部梯度值很小，将很有效地“杀掉”梯度，使得几乎没有信号流过神经元到达它的权重并递归回数据。此外，在初始化 Sigmoid 神经元参数时也需要加倍小心，

以避免函数进入饱和区。例如，如果初始化的参数值过大，大部分神经元工作在饱和区，则网络变得很难学习。

（2）Sigmoid 函数并非以 0 为中心

这一属性同样不受欢迎，因为通过神经元向后传播的网络需要处理非 0 的数据，这将对梯度下降的过程造成影响。因为如果进入一个神经元的数据总是正的（如 $f = w^{\mathrm{T}}x + b, x > 0$），那么在反向传播时参数 w 的梯度要么都是正的，要么都是负的（取决于整个表达式 f 的符号）。这可能导致更新参数 w 时出现恼人的 zig-zag 运动。不过，当这些梯度在一个批处理数据中先进行累加，最后再更新参数时，符号可能会发生变化，这在一定程度上可以降低影响。

2.2.4 Tanh 神经元

Tanh 神经元是 Sigmoid 神经元的一个继承，它将实数“压缩”至 −1~1 的范围内，因此改进了 Sigmoid 变化过于平缓的问题。

Tanh 神经元的数学表达式为：

$$y = \frac{\mathrm{e}^{x} - \mathrm{e}^{-x}}{\mathrm{e}^{x} + \mathrm{e}^{-x}}$$

Tanh 的求导结果如下：

$$\frac{\partial y}{\partial x} = \frac{(\mathrm{e}^{x} + \mathrm{e}^{-x})(\mathrm{e}^{x} + \mathrm{e}^{-x}) - (\mathrm{e}^{x} - \mathrm{e}^{-x})(\mathrm{e}^{x} - \mathrm{e}^{-x})}{(\mathrm{e}^{x} + \mathrm{e}^{-x})^{2}} = 1 - y^{2}$$

2.2.5 ReLU

整流线性单元（Rectified Linear Unit），又称为修正线性单元，一般以其英文缩写 ReLU 来指代。其数学表达式为：

$$y = \begin{cases} x, & x > 0 \\ 0, & \text{其他} \end{cases}$$

该函数等价于 $y = \max(0, x)$。它在阈值以下的输出都被截断成“0”，在阈值以上的输出则线性不变，如图 2-3 左图所示。Krizhevsky 等的实验表明 ReLU 比 Tanh 的收敛速度快 6 倍[1]，如图 2-3 右图所示。

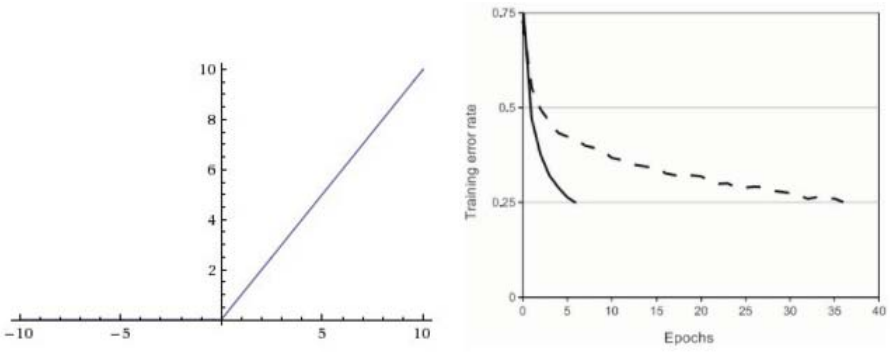


图 2-3 ReLU 函数图像（左）及 ReLU 与 Tanh 的实验对比（右）

ReLU 是分段可导的，其导数形式非常简单：

$$\frac{\partial y}{\partial x}=\begin{cases}1, & x>0\\ 0, & \text{其他}\end{cases}$$

ReLU 在神经网络的实际应用中被广泛采用，因为其既具有非线性的特点，使得信息整合能力大大增强；在一定范围内又具有线性的特点，使得其训练简单、快速。使用 ReLU 有以下优点。

- 相比 Sigmoid 和 Tanh，ReLU 在随机梯度下降过程中能够明显加快收敛速度（有人认为这是由于它具有线性、非饱和的形态）。
- 相比 Sigmoid 和 Tanh 包含复杂算子，ReLU 通过简单的阈值操作就能实现。

然而，ReLU 并不是万能的，在训练过程中可能是脆弱的并且出现“死亡”。例如，流经 ReLU 神经元的大梯度可能导致权重更新到不再被任何数据激活的位置上。如果发生这种情况，流经该神经元的梯度将永远为 0。也就是说，在训练过程中，ReLU 单元会不可逆转地死去。如果学习率设得太高，那么在网络中甚至有高达 40% 的神经元不能被激活。通过调整学习率，能够限制这种情况的发生。

针对 ReLU 的相关缺点，近年来又出现了很多变种，包括 Leaky ReLU[2] 试图解决 ReLU “死亡”单元的问题。当 $x<0$ 时，函数不再直接取 0，而是取 $0.01x$，即：

$$y=\begin{cases}x, & x>0\\ 0.01x, & \text{其他}\end{cases}$$

该函数等价于：

$$f(x) = \max(x, 0.01x)$$

再往后，何恺明等人再次提出了含参数变量 α 的 ReLU——PReLU[3]，其函数形式为：

$$y = \begin{cases} x, & x > 0 \\ \alpha x, & \text{其他} \end{cases}$$

该函数等价于：

$$f(x) = \max(x, \alpha x)$$

从上面描述的 ReLU 相关激活函数来看，ReLU 是一种特殊的 Maxout 函数。

2.2.6 Maxout

Maxout 激活函数[4] 就是通常的最大值函数 max，即多个输入取最大值：

$$y = \max_k a_k = \max(w_1^{\mathrm{T}} x + b_1, w_2^{\mathrm{T}} x + b_2, \cdots, w_n^{\mathrm{T}} x + b_n)$$

其求导非常简单，取最大值的一路有梯度，其他路无梯度：

$$\frac{\partial y}{\partial a_i} = \begin{cases} 1, & a_i \text{为最大值} \\ 0, & \text{其他} \end{cases}$$

Maxout 能够在一定程度上缓解梯度消失的问题，同时又规避了 ReLU “死亡”单元的问题，但是增加了参数和计算量。

2.2.7 Softmax

Softmax 通常应用在多分类问题的输出层，它可以保证所有输出神经元之和为 1，而每个输出对应的 [0, 1] 区间的数值就是该输出的概率，在应用时取概率最大的输出作为最终的预测。Softmax 函数的形式如下：

$$y_j = \frac{\mathrm{e}^{z_j}}{\sum_{t=1}^{k} \mathrm{e}^{z_t}}$$

偏导数分为两种情况：

（1） $i = j$ 时

$$\frac{\partial y_i}{\partial z_i} = \frac{e^{z_i}(\sum_{t=1}^{k} e^{z_t}) - e^{z_i}e^{z_i}}{(\sum_{t=1}^{k} e^{z_t})^2} = y_i - {y_i}^2$$

（2） $i \neq j$ 时

$$\frac{\partial y_j}{\partial z_i} = \frac{0 - e^{z_i}e^{z_j}}{(\sum_{t=1}^{k} e^{z_t})^2} = 1 - y_i y_j$$

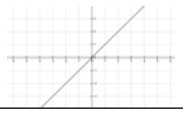
2.2.8 小结

激活函数是深度学习中非常重要的组成部分，也涌现出了非常多的研究成果，其中 ReLU 和 Maxout 的变种就有很多，以下简单罗列一些代表性工作。

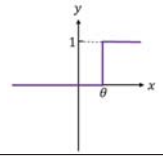
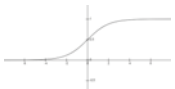
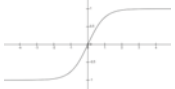
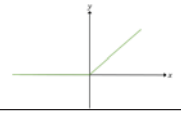
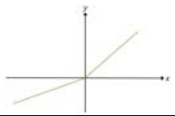
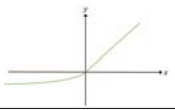
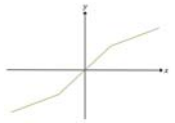
- 将 PReLu 中的 α 作为一定范围内随机变量的随机 Leaky ReLU（Randomized Leaky Rectified Linear Unit，RReLU）[5]；
- 将横坐标左侧变成指数函数的指数线性单元（Exponential Linear Unit，ELU）[6]；
- 自适应的分段线性函数 APL（Adaptive Piecewise Linear）[7]；
- 分段线性的 S 型 ReLU（S-Shaped Rectified Linear Activation Unit，SReLU）[8]。

表 2-2 总结了常见的相关激活函数图像以及对应的公式和偏导数。其中 Maxout 和 Softmax 涉及多路，省略了函数图像。

表 2-2 激活函数汇总[9]

名称	图像	公式	导数
Identity		$f(x) = x$	$f'(x) = 1$
Linear		$f(x) = wx + b$	$f'(x) = w$

续表

名称	图像	公式	导数
Linear Threshold Neuron		$f(x)=\begin{cases}1, & x\geqslant\theta\text{且无抑制突触输入}\\0, & \text{其他}\end{cases}$	$f'(x)=\begin{cases}?, & x=\theta\\0, & \text{其他}\end{cases}$
Sigmoid		$f(x)=\dfrac{1}{1+\mathrm{e}^{-x}}$	$f'(x)=f(x)(1-f(x))$
Tanh		$f(x)=\dfrac{\mathrm{e}^{x}-\mathrm{e}^{-x}}{\mathrm{e}^{x}+\mathrm{e}^{-x}}$	$f'(x)=1-f(x)^2$
ReLU		$f(x)=\begin{cases}x, & x>0\\0, & \text{其他}\end{cases}$	$f'(x)=\begin{cases}1, & x>0\\0, & \text{其他}\end{cases}$
Leaky ReLU		$f(x)=\begin{cases}x, & x>0\\0.01x, & \text{其他}\end{cases}$	$f'(x)=\begin{cases}1, & x>0\\0.01, & \text{其他}\end{cases}$
PReLU		$f(x)=\begin{cases}x, & x>0\\\alpha x, & \text{其他}\end{cases}$	$f'(x)=\begin{cases}1, & x>0\\\alpha, & \text{其他}\end{cases}$
RReLU		$f(x)=\begin{cases}x, & x>0\\\alpha x, & \text{其他}\end{cases}$ $\alpha\sim U(l,u), l<u, l,u\in[0,1)$ U为均匀分布	$f'(x)=\begin{cases}1, & x>0\\\alpha, & \text{其他}\end{cases}$
ELU		$f(x)=\begin{cases}x, & x>0\\\alpha(\mathrm{e}^{x}-1), & \text{其他}\end{cases}$	$f'(x)=\begin{cases}1, & x>0\\\alpha\mathrm{e}^{x}, & \text{其他}\end{cases}$
APL		$f(x)=\max(0,x)+$ $\sum_{s=1}^{S}a_i^s\max(0,-x+b_i^s)$	根据各 max 函数的取值求偏导数
SReLU		$f(x)=\begin{cases}t_l+\alpha_l(x-t_l), & x\leqslant t_l\\x, & t_l<x<t_r\\t_r+\alpha_r(x-t_r), & x\geqslant t_r\end{cases}$	$f'(x)=\begin{cases}\alpha_l, & x\leqslant t_l\\1, & t_l<x<t_r\\\alpha_r, & x\geqslant t_r\end{cases}$

续表

名称	图像	公式	导数
Maxout	—	$f(\boldsymbol{x}) = \max_k a_k$	$\frac{\partial f(\boldsymbol{x})}{\partial a_i} = \begin{cases} 1, & a_i \text{为最大值} \\ 0, & \text{其他} \end{cases}$
Softmax	—	$f(\boldsymbol{x})_j = \frac{e^{z_j}}{\sum_{t=1}^{k} e^{z_t}}$	$\frac{\partial f(\boldsymbol{x})_j}{\partial z_i} = f(\boldsymbol{x})_i(\delta_{ij} - f(\boldsymbol{x})_j)$ 其中：$\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & \text{其他} \end{cases}$

激活函数如此之多，一般在实践中选用哪些呢？针对这个问题网上有很多相关的讨论，但是结论并不统一。一般来说，推荐不用 Sigmoid 的较多，ReLU 要注意检查“死亡”单元的比例，PReLU 和 Maxout 等都是可以尝试的。此外，虽然理论上可以多种激活函数混用，但在实践中较少这样应用。

2.3 感知机

在了解了神经元的基本概念后，我们可以从计算机神经网络长长的源头——感知机开始，详细地了解这门学科及其跌宕起伏的历史。

2.3.1 感知机的提出

一切都要从 20 世纪 60 年代说起，Frank Rosenblatt 出版的《神经动力学原理：感知机和大脑机制的理论》揭开了人工神经网络研究的序幕。书中介绍了多种感知神经元，以及一套简洁却显得十分有效的学习算法。这本书引起了学术界和民众的极大兴趣，在那个科技腾飞的年代，人们对人工神经网络怀有极大的信心和期望。

感知机的结构极其简单。输入层为人们选择的“特征值”（feature），感知机内含一套参数，称为“权重”，特征值和权重相乘后求和，与一个阈值比较后输出为 0 或 1。回顾上一节所提到的神经元，相信读者对它已经不陌生了。以 $X = [x_1, x_2, \cdots, x_n]$ 来表示输入值，以 $W = [w_1, w_2, \cdots, w_n]$ 来表示对应参数，以 b 来表示阈值的相反数，以 y 来表示预测值，以 $\tilde{y}$ 来表示真实值，则感知机模型如图 2-4 所示。

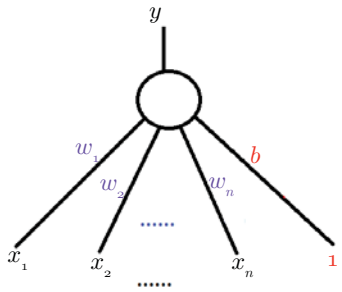


图 2-4　感知机模型

感知机模型的学习算法（即更新规则）也十分简明，如算法 2-1 所示。

算法 2-1　感知机模型的学习算法

随机取一个训练样本 $(X, \tilde{y})$，通过当前感知机模型进行预测。

- 如果预测结果正确，则当前感知机维持不变。
- 如果预测结果错误，则按下述规则更新感知机参数。
  - 如果样本真实输出值 $\tilde{y}=1$，而实际预测值 $y=0$，则将参数与该样本输入的“和”作为新的参数 $W \leftarrow W+X$。
  - 如果样本真实输出值 $\tilde{y}=0$，而实际预测值 $y=1$，则将参数与该样本输入的“差”作为新的参数 $W \leftarrow W-X$。

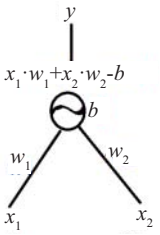
这套学习算法简单明了，并且保证模型不断优化收敛，直至满足所有的训练样本。直到后来人们才发现，这个论断过于草率。然而，在当时，一切还笼罩在虚假的繁荣之中。例如，当时专家宣称已经可以用计算机区分出“坦克”和“卡车”。这在当时是爆炸性的科技进展，但后来才发现，之所以能区分，是因为“坦克”的图片都是在晴天拍摄的，而“卡车”的图片都是在阴天拍摄的，感知机仅仅是计算了图片的平均光强，从而区分出这两类图片，并不是像人们期望的那样，真的对图片中的物体有了感知。这类事例给人工神经网络带来了非常负面的影响。

2.3.2　感知机的困境

1969 年，Minsky 和 Papert 对当时的感知机模型进行了深入的研究，并且出版了一本书，名字就叫《感知机》。书中分析了感知机“能做”和“不能做”的事情，这是对当时感知机理论很好的分析和总结。然而，人们普遍关注的是感知机“不能”完成的方面，并将该论述

扩大化，认为其是神经网络的普适困境，因而悲观地论断神经网络没有出路和研究价值。学术界对神经网络的研究开始冷淡。

具体来说，一方面，此类感知机模型仅能处理线性可分的问题，如果面对的问题并不是线性可分的，那么这种单层感知机是没有可行解的。比如，对于简单的“异或”问题，由于它是非线性问题，因而感知机是无法得到可行解的。读者可参考图 2-5 试试，是不是找不到一组可行解使得模型输出可以满足异或逻辑。另一方面，感知机需要人工提取特征作为输入。换个思路来看，对于非线性问题，感知机只有通过人工提取特定的特征——在这些特征中将非线性的因素包含进来——使得特征仅用线性关系就可判别，才能达到目标。但这意味着什么呢？这意味着很多问题的难点被转移到“特征的设定”上，而这一点又只能通过人工来完成，感知机完全帮不上忙。这个事实引起了人们对感知机实际功用的质疑。对于感知机的研究者来说，如何自动提取这类复杂的特征是摆在面前最迫切的问题。然而，从研究者开始想到解决方法的时候，距离陷入感知机困境的当年，已经过了长达 20 年的时间。



x_1	x_2	y
1	1	0
1	0	1
0	1	1
0	0	0

图 2-5 异或问题

2.4 DNN

解决感知机面临非线性问题的方法，就是将感知机变成多层神经网络，也称为深度神经网络（Deep Neural Network，DNN）。1974 年，Werbos 的博士论文[10] 证明了将感知机叠加起来组成神经网络，并且利用“后向传播”的方法，就可以解决诸如“异或问题”的非线性问题。如图 2-6 所示是其一可行的网络，读者可以验证看看。然而，该论文并没有引起广泛的关注。因为对于大多数研究者来说，多层神经网络相对于当时风头正劲的支持向量机（SVM）而言，其背后缺乏优美的数学理论和解决问题的坚实证明。实际上，这是神经网络一直面临的窘境。即使在其重获关注，并在各个领域获得划时代的成绩后，对于其“成功”的解释依然是一层未揭开的面纱。不过，随着近年来对神经网络的隐层的研究，如“可视化分析”等，人们正逐渐了解它背后神秘的机制。当然，这是后话。现在，就让我们走近神经网络基本模型和基本学习方法吧！

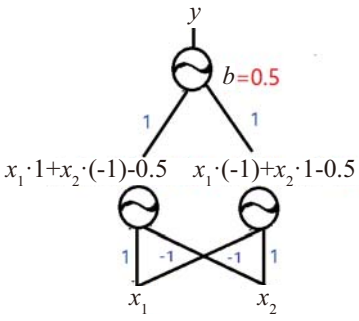


图 2-6　可以解决异或问题的一个多层神经网络

2.4.1　输入层、输出层及隐层

网络结构的第一层为输入层，最后一层为输出层，如果中间有其他层，则被称为“隐层”。如果隐层的数目多于一层，则该神经网络被称为“深度”神经网络。隐层和输出层一般会含有神经元，从而实现非线性。

如图 2-7 所示为一个带有一层隐层的网络模型。如果放大隐层或输出层的神经元，则可以将其分解为输入和参数的线性变换结果 z，该变换结果经过非线性的激活函数 $y = f(z)$ 而得到输出的两部分结构。不过请读者注意，y 与 z 属于同一层，只是为了便于介绍神经网络的训练学习，我们才采取这种分解的结构来表示网络。

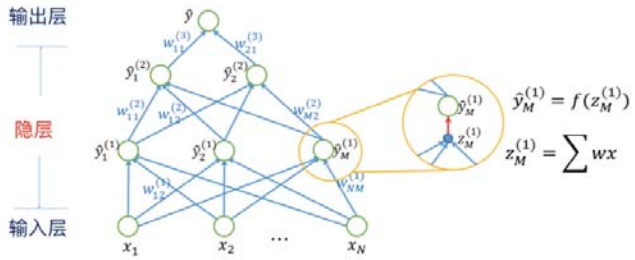


图 2-7　网络模型示例及放大的神经元

2.4.2　目标函数的选取

在讨论神经网络的训练之前，我们首先要明确目标函数。通常，这个目标函数以损失函数（Loss Function）的形式来呈现。例如，常用的均方误差损失函数可以表示为：

$$\text{Loss} = \frac{1}{2N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

其中，N 为样本的数目，y_i 为第 i 个样本的实际标注值，也称为标签（Label），而 $\hat{y}_i$ 为该样本的预测值。由此可见，损失函数值越小越好，当损失函数值为 0 时，则说明模型预测的结果完全无误。

除均方差损失函数外，还有很多不同的损失函数。损失函数的选取一般要根据模型的特点和目标的设立来进行。比如，在一个多分类问题上，最合适的损失函数可能就不是均方差损失函数。下面我们来具体看一下这个问题。

假设这个多分类问题共有 C 个类别，而输出 z_c 也是 C 维的，每一维的输出 z_c 值代表在该类的得分，得分最高的即为最可能的预测类别。对于这样的问题，我们更希望输出是概率形式，因此，在输出层会加一个 Softmax：

$$\hat{y_c} = \frac{\exp(z_c)}{\sum_i \exp(z_i)}$$

这样输出的预测值被转化成了概率值，所有类的概率值之和为 1。对于这样的以 Softmax 为输出层的网络模型，最合适的损失函数就是一种名为交叉熵的损失函数（Cross-entropy Loss Function）：

$$\text{Loss} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

这是因为 $\frac{\partial \text{Loss}}{\partial z_i} = \sum_i \frac{\partial \text{Loss}}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_i} = \hat{y}_i - y_i$。这个偏导数的结果简洁、漂亮。在网络的训练中很重要的就是利用 Loss 对参数的梯度，而此简洁的梯度也让训练更加简单，因此对于输出层为 Softmax 来说，最合适的损失函数为此交叉熵损失函数。对于这个梯度公式的计算过程如下，感兴趣的读者可以自己演算一遍。

$$\begin{aligned}
\frac{\partial \text{Loss}}{\partial z_i} &= \sum_j \frac{\partial \text{Loss}}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial z_i} \\
&= \sum_{j \neq i} \frac{\partial \text{Loss}}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial z_i} + \frac{\partial \text{Loss}}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_i} \\
&= \sum_{j \neq i} -\frac{y_j}{\hat{y}_j} \cdot \frac{-\exp(z_j) \cdot \exp(z_i)}{\left(\sum_k \exp(z_k)\right)^2} + \left(-\frac{y_i}{\hat{y}_i} \cdot \frac{\exp(z_i) \cdot \left(\sum_k \exp(z_k)\right) - \exp(z_i) \cdot \exp(z_i)}{\left(\sum_k \exp(z_k)\right)^2}\right)
\end{aligned}$$

$$= \sum_{j \neq i} \frac{y_j}{\hat{y}_j} \cdot \hat{y}_j \cdot \hat{y}_i + \left(-\frac{y_i}{\hat{y}_i}\right) \hat{y}_i \left(1 - \hat{y}_i\right)$$

$$= \sum_{j \neq i} y_j \cdot \hat{y}_i + y_i \cdot \hat{y}_i - y_i$$

$$= \hat{y}_i - y_i$$

有了目标损失函数以后，我们可以详细地看一下神经网络的训练过程——前向传播（Forward Propagation）与后向传播（Backward Propagation）。

2.4.3 前向传播

为了便于说明，我们以一个简单的神经网络模型为例。如图 2-8 所示为要训练的网络模型，其中输入为 N 维，输出为预测值 $\hat{y}$，目标损失函数采用均方差误差，模型的参数为各层的 w 值，神经元的中间结果用 z 来表示，激活函数采用 Sigmoid。

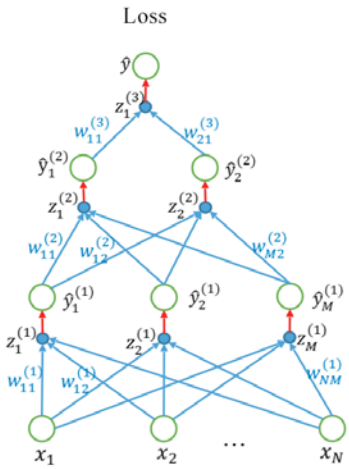


图 2-8　神经网络模型示例

前向传播，就是在当前参数值下，输入值进入网络后，顺序计算，最终得到预测值的过程。在图 2-8 中，则是自下向上沿着箭头方向进行计算的。其中：

$$\begin{cases} z_i^{(1)} = \sum_{j=1}^{N} w_{ji}^{(1)} x_j \\ \hat{y}_i^{(1)} = \dfrac{1}{1 + \mathrm{e}^{-z_i^{(1)}}} \end{cases}$$

为输入层到第一隐层的前向传播计算公式。而第一隐层至第二隐层、第二隐层至输出层的前向传播计算也可以类似推得。最终的损失函数计算为输出值和真实值之间的均方误差或交叉熵误差等。

可以看出，前向传播计算非常简单。但是，所得到的预测值可能和真实值相差较远，其损失函数值也比较大。那么如何训练模型的参数使得损失值下降呢？这个重要的课题在神经网络中是以后向传播来实现的。

2.4.4 后向传播

后向传播，顾名思义，就是从损失值开始，反过来更新网络的参数值，使得更新后的网络的损失值下降的过程。这一过程主要是通过“梯度下降”的方法来实现的。也就是说，基于当前的参数值，能使损失值下降最大的方法可能是向着梯度的反方向更新参数。朝着这个梯度的反方向更新多大的值？更新是否一定能保证损失值减小？这些问题将在下一节中分析。本节我们先研究采取“梯度下降”策略后，如何得到梯度值，从而顺利完成后向传播的过程。

还是以图 2-8 所示的神经网络模型为例。我们先看相邻层之间的梯度计算，然后再看从损失值开始至任意层任意一个参数的梯度的计算方法。

首先，损失函数对输出层的梯度可以很容易求得：

$$\frac{\partial \text{Loss}}{\partial \hat{y}} = -(y - \hat{y})$$

那么，输出层 $\hat{y}$ 对激活函数的输入 $z_1^{(3)}$ 的梯度是什么呢？因为我们选择的激活函数是 Sigmoid，而前面也已推得 Sigmoid 的导数结果，所以现在可以直接得到：

$$\frac{\partial \hat{y}}{\partial z_1^{(3)}} = \hat{y} \cdot (1 - \hat{y})$$

而由于 $z_j^{(3)}$ 是其对应的输入层在相应参数下的线性组合，因此其对参数和输入的偏导都很简单：

$$\begin{cases} \dfrac{\partial z_j^{(3)}}{\partial \hat{y}_i^{(2)}} = w_{ij}^{(3)} \\ \dfrac{\partial z_j^{(3)}}{\partial w_{ij}^{(3)}} = \hat{y}_i^{(2)} \end{cases}$$

而从 $y_i^{(2)}$ 层向下的梯度计算都是类似的：

$$\frac{\partial \hat{y}_i^{(2)}}{\partial z_i^{(2)}} = \hat{y}_i^{(2)} \cdot (1 - \hat{y}_i^{(2)})$$

从 $z_j^{(2)}$ 向下求梯度为：

$$\begin{cases} \frac{\partial z_j^{(2)}}{\partial \hat{y}_i^{(1)}} = w_{ij}^{(2)} \\ \frac{\partial z_j^{(2)}}{\partial w_{ij}^{(2)}} = \hat{y}_i^{(1)} \end{cases}$$

直到隐层向输入层的梯度计算：

$$\frac{\partial \hat{y}_i^{(1)}}{\partial z_i^{(1)}} = \hat{y}_i^{(1)} \cdot (1 - \hat{y}_i^{(1)})$$

一般最后只需要求对参数的梯度（注：生成模型有时也需要对输入的梯度），而不再需要计算对输入值的梯度：

$$\frac{\partial z_j^{(1)}}{\partial w_{ij}^{(1)}} = x_i$$

至此，我们得到了相邻两层梯度的计算结果。

而梯度下降法需要的是任意一个参数相对于损失值的梯度，关于这个梯度的计算，只需要应用梯度的“链式法则”，根据上面求得的结果便可以得到。比如，如果想求得损失值对参数 $w_{M2}^{(2)}$ 的梯度，则只需计算：

$$\begin{aligned} \frac{\partial \text{Loss}}{\partial w_{M2}^{(2)}} &= \frac{\partial \text{Loss}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1^{(3)}} \cdot \frac{\partial z_1^{(3)}}{\partial y_2^{(2)}} \cdot \frac{\partial y_2^{(2)}}{\partial z_2^{(2)}} \cdot \frac{\partial z_2^{(2)}}{\partial w_{M2}^{(2)}} \\ &= -(y - \hat{y}) \cdot \hat{y} \cdot (1 - \hat{y}) \cdot w_{21}^{(3)} \cdot \hat{y}_2^{(2)} \cdot (1 - \hat{y}_2^{(2)}) \cdot y_M^{(1)} \end{aligned}$$

而如果从损失值到变量有多条路径，那么就需要将各条路径上的梯度求出来，然后再加和。比如，如果要求损失值对 $\hat{y}_2^{(1)}$ 的梯度，从前向传播来看，$\hat{y}_2^{(1)}$ 的改变将会沿着两条路径影响到损失值，因此损失值对它的梯度计算应该是：

$$\frac{\partial \text{Loss}}{\partial \hat{y}_2^{(1)}} = \frac{\partial \text{Loss}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1^{(3)}} \cdot \frac{\partial z_1^{(3)}}{\partial \hat{y}_1^{(2)}} \cdot \frac{\partial \hat{y}_1^{(2)}}{\partial z_1^{(2)}} \cdot \frac{\partial z_1^{(2)}}{\partial \hat{y}_2^{(1)}} + \frac{\partial \text{Loss}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1^{(3)}} \cdot \frac{\partial z_1^{(3)}}{\partial \hat{y}_2^{(2)}} \cdot \frac{\partial \hat{y}_2^{(2)}}{\partial z_2^{(2)}} \cdot \frac{\partial z_2^{(2)}}{\partial \hat{y}_2^{(1)}}$$

具体结果不再展开。

至此，后向传播的过程和梯度的具体计算我们已经清楚了。

2.4.5 参数更新

最后，我们再来看一下参数更新。虽然知道了参数更新的方向，并且由后向传播计算出了梯度值，但是沿着这个梯度的反方向更新多少还是一个重要的问题。在实际神经网络的训练中，往往会通过一个重要的参数——学习率（Learning Rate）来控制这个“步长”。也就是说，参数 w 的更新将通过以下表达式：

$$w \Leftarrow w - \eta \cdot \frac{\partial \text{Loss}}{\partial w}$$

其中，“负号”代表与梯度方向相反；η 代表学习率，它作为参数来控制步长；而 $\frac{\partial \text{Loss}}{\partial w}$ 为计算的梯度值。

如图 2-9 所示，当前参数值为 x_t，则下一时刻更新为 $x_{t+1} = x_t - \eta \cdot \partial y/\partial x$。具体来看，当前时刻该位置梯度的方向为 +，大小为 $\Delta y/\Delta x$，那么参数更新的方向将为 −，更新的大小为 $\eta \cdot \Delta y/\Delta x$。由图可见，如果学习率选取得当，更新后对应的 y 值将变小；但是当学习率过大时，也很容易出现 y 值反而增大的现象，发生震荡，如 x'_{t+1} 对应的值；而如果学习率设置得过小，那么虽然不易发生震荡，但收敛速度将会变慢，也会影响训练效果。由此可见，学习率的设置是一个十分重要的问题。在后面的章节中，我们会再详细探讨。

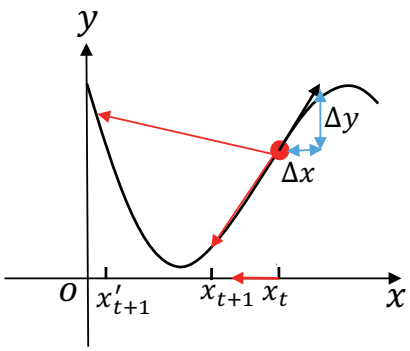


图 2-9 参数更新示例

2.4.6 神经网络的训练步骤

常用的基于梯度下降法的神经网络的训练步骤如算法 2-2 所示。

算法 2-2　神经网络的训练步骤

设计网络结构：输入层与输出层之间由几个隐层组成、各层之间如何连接，以及选择神经元。

选择网络参数的初始值及设立损失函数等。

开始训练：

1. 从训练集随机训练样本 X_i（如果是 batch 模式，则是多个训练样本）。
2. 将 X_i 在当前参数 W 下进行前向传播得到损失值（loss）。
3. 根据链式法则，进行后向传播得到梯度值 $\frac{\partial \text{loss}}{\partial w}$。
4. 更新参数值 $w \Leftarrow w - \eta \cdot \frac{\partial \text{loss}}{\partial w}$。
5. 循环步骤 1~4，直至 loss 满足目标，网络训练完成。

至此，我们大体了解了神经网络的组成和基础训练步骤。后面将会有更深入、更有趣的内容等待我们研究。

参考文献

[1] A Krizhevsky, I Sutskever, GE Hinton. Imagenet classification with deep convolutional neural networks, Advances in neural information processing systems 2012, 1097-1105.

[2] Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng. Rectifier Nonlinearities Improve Neural Network Acoustic Models. ICML 2013.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. IEEE International Conference on Computer Vision (ICCV), 2015.

[4] IJ Goodfellow, D Wardefarley, M Mirza, A Courville, Y Bengio. Maxout Networks. Computer Science, 2013:1319-1327.

[5] Xu B, Wang N, Chen T, et al. Empirical Evaluation of Rectified Activations in Convolutional Network[J]. Computer Science, 2015.

[6] Clevert, Djork-Arné, Unterthiner T, Hochreiter S. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)[J]. Computer Science, 2015.

[7] Agostinelli F, Hoffman M, Sadowski P, et al. Learning Activation Functions to Improve Deep Neural Networks[J]. Computer Science, 2014.

[8] Jin X, Xu C, Feng J, et al. Deep Learning with S-shaped Rectified Linear Activation Units[J]. Computer Science, 2015, 3:1-8.

[9] Activation Function. https://en.wikipedia.org/wiki/Activation_function.

[10] Werbos, P. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University, 1974.