第四章　前馈神经网络

在人工智能领域，**人工神经网络**（Artificial Neural Network，ANN）是指一系列受生物学和神经学启发的数学模型。这些模型主要是通过对人脑的神经元网络进行抽象，构建人工神经元，并按照一定拓扑结构来建立人工神经元之间的连接，来模拟生物神经网络。在人工智能领域，人工神经网络也常常简称为神经网络（Neural Network，NN）或神经模型。

到目前为止，研究者已经提出了多种多样的神经网络模型，比如：感知器、前馈网络、卷积网络、循环网络、自组织映射、Hopfield 网络、Boltzmann 机等。这些神经网络模型的差异主要在于神经元的激活规则、网络连接的拓扑结构以及参数的学习规则等。

4.1　人工神经网络

人工神经网络主要由大量的神经元以及它们之间的有向连接构成。因此考虑三方面：

- 神经元的激活规则：主要是指神经元输入到输出之间的映射关系，一般为非线性函数。
- 网络的拓扑结构：不同神经元之间的连接关系。神经元间的连接也称为权重，即需要学习的参数。
- 学习算法：通过训练数据来学习神经网络的参数。

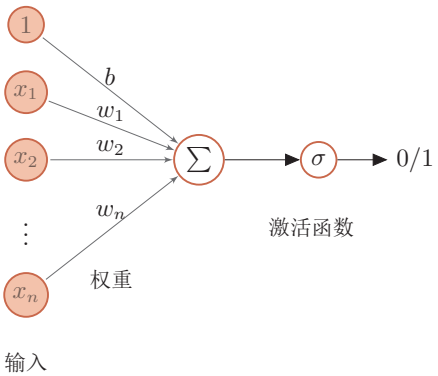


图 4.1: 人工神经元模型

4.1.1 神经元

人工神经元（Artificial Neuron）是构成人工神经网络的基本单元。人工神经元和感知器非常类似，也是模拟生物神经元特性，接受一组输入信号并产出输出。生物神经元有一个阈值，当神经元所获得的输入信号的积累效果超过阈值时，它就处于兴奋状态；否则，应该处于抑制状态。

净输入也叫净活性值（net activation）。

我们用 $\mathbf{x} = (x_1, x_2, \cdots, x_n)$ 来表示人工神经元的一组输入，用**净输入**z 表示一个神经元所获得的输入信号 x 的加权和，

$$z = \sum_{i=1}^{n} w_i x_i + b \tag{4.1}$$

$$= \mathbf{w}^{\mathrm{T}}\mathbf{x} + b, \tag{4.2}$$

其中，$\mathbf{w} = (w_1, w_2, \cdots, w_n)$ 是 n 维的权重向量，b 是偏置。

净输入 z 在经过一个非线性函数后，人工神经元输出它的**活性值**（Activation）a，

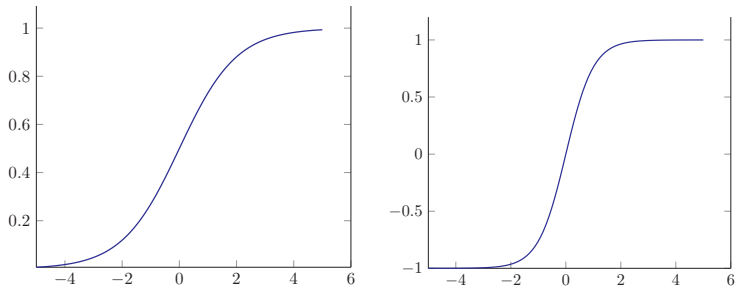
$$a = f(z), \tag{4.3}$$

这里的非线性函数 f 也称为**激活函数**（Activation Function），有 sigmoid 型函数、非线性斜面函数等。

人工神经元的结构如图4.1所示。如果我们设激活函数 f 为 0 或 1 的阶跃函数，人工神经元就是感知器。

数学小知识 | 饱和

对于函数 $f(x)$，若 $x \to -\infty$ 时，其导数 $f'(x) \to 0$，则称其为**左饱和**。若 $x \to +\infty$ 时，其导数 $f'(x) \to 0$，则称其为**右饱和**。当同时满足左、右饱和时，就称其为饱和。



(a) logistic 函数 (b) Tanh 函数

图 4.2: Sigmoid 型激活函数

4.1.2 激活函数

为了增强网络的表达能力，我们需要使用连续非线性**激活函数**（Activation Function）。因为连续非线性激活函数可导，所以可以用最优化的方法来求解。

下面介绍几个在神经网络中常用的激活函数。

Logistic 函数

Sigmoid 型函数是指一类 S 型曲线函数，常用的 sigmoid 型函数有 logistic 函数和 tanh 函数。

Logistic 函数是一种 sigmoid 型函数。其定义为 $\sigma(x)$

$$\sigma(x) = \frac{1}{1+e^{-x}}. \tag{4.4}$$

图4.3a给出了 logistic 函数的形状。Logistic 函数可以看成是一个“挤压”函数，把一个实数域的输入“挤压”到 $(0,1)$。当输入值在 0 附近时，sigmoid 型函数近似为线性函数；当输入值靠近两端时，对输入进行抑制。输入越小，越接

近于0；输入越大，越接近于1。这样的特点也和生物神经元类似，对一些输入会产生兴奋（输出为1），对另一些输入产生抑制（输出为0）。和感知器使用的阶跃激活函数相比，logistic函数是连续可导的，其数学性质更好。

Tanh函数 Tanh函数是也一种sigmoid型函数。其定义为

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4.5}$$

Tanh函数可以看作是放大并平移的logistic函数：

$$\tanh(x) = 2\sigma(2x) - 1. \tag{4.6}$$

图4.3b给出了tanh函数的形状，其值域是$(-1, 1)$。

Hard-Logistic和Hard-Tanh函数

Logistic函数和tanh函数都是sigmoid型函数，具有饱和性，但是计算开销较大。因为这两个函数都是在中间（0附近）近似线性，两端饱和。因此，这两个函数可以通过分段函数来近似。

以logistic函数$\sigma(x)$为例，其导数为$\sigma'(x) = \sigma(x)(1 - \sigma(x))$。logistic函数在0附近的一阶泰勒展开（Taylor expansion）为

$$g_l(x) \approx \sigma(0) + x \times \sigma'(0) \tag{4.7}$$

$$= 0.25x + 0.5. \tag{4.8}$$

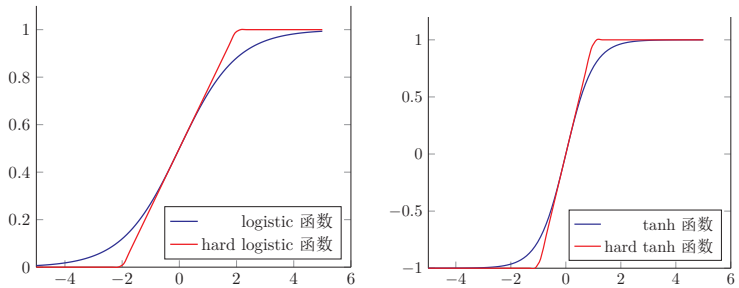
$$\text{hard-logistic}(x) = \begin{cases} 1 & g_l(x) \geq 1 \\ g_l & 0 < g_l(x) < 1 \\ 0 & g_l(x) \leq 0 \end{cases} \tag{4.9}$$

$$= \max(\min(g_l(x), 1), 0) \tag{4.10}$$

$$= \max(\min(0.25x + 0.5, 1), 0). \tag{4.11}$$

同样，tanh函数在0附近的一阶泰勒展开为

$$g_t(x) \approx \tanh(0) + x \times \tanh'(0) \tag{4.12}$$



(a) Hard logistic 函数 (b) Hard tanh 函数

图 4.3: Sigmoid 型激活函数近似

$$= x. \tag{4.13}$$

这样，tanh 函数也可以用分段函数 hard-tanh(x) 来近似。

$$\text{hard-tanh}(x) = \max(\min(g_t(x), 1), -1) \tag{4.14}$$

$$= \max(\min(x, 1), -1). \tag{4.15}$$

图4.3给出了 hard-logistic 和 hard-tanh 函数两种函数的形状。

修正线性单元

修正线性单元（rectified linear unit，ReLU）[Nair and Hinton, 2010]，也叫 rectifier 函数 [Glorot et al., 2011]，是深层神经网络中经常使用的激活函数。ReLU 实际上是一个斜坡（ramp）函数，定义为

$$\text{rectifier}(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{4.16}$$

$$= \max(0, x). \tag{4.17}$$

采用 ReLU 的神经网络只需要进行加、乘和比较的操作，计算上也更加高效。此外，rectifier 函数被认为有生物上的解释性。神经科学家发现神经元具有**单侧抑制、宽兴奋边界、稀疏激活性**等特性。

稀疏激活性 生物神经元只对少数输入信号选择性响应，大量信号被屏蔽了。因为随机初始化的原因，sigmoid系函数同时近乎有50%的神经元被激活，这不符合神经科学的发现。而rectifier函数却具备很好的稀疏性。

Rectifier函数为左饱和函数，在$x > 0$时导数为1，在$x < 0$时导数为0。这样在训练时，如果学习率设置过大，在一次更新参数后，一个采用ReLU的神经元在所有的训练数据上都不能被激活。那么，这个神经元在以后的训练过程中永远不能被激活，这个神经元的梯度就永远都会是0。

需要参见第72页的第4.3节内容

在实际使用中，为了避免上述情况，ReLU有几种被广泛使用的变种。

带泄露的ReLU 带泄露的ReLU（Leaky ReLU）在输入$x < 0$时，保持一个很小的梯度λ。这样当神经元非激活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]。带泄露的ReLU的定义如下：

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \lambda x & \text{if } x \leq 0 \end{cases} \tag{4.18}$$

$$= \max(0, x) + \lambda \min(0, x), \tag{4.19}$$

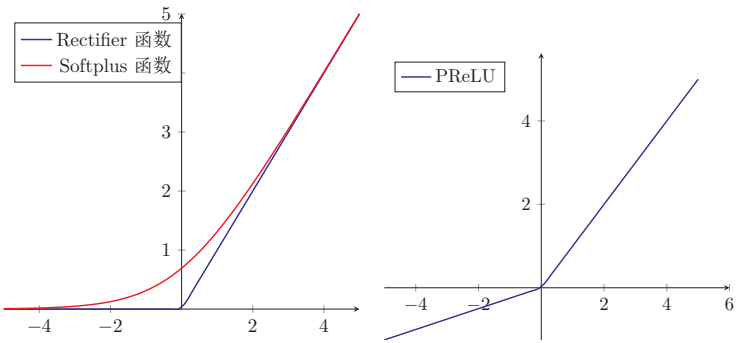
这里，$\lambda \in (0, 1)$是一个很小的常数，比如0.01。

带参数的ReLU 带参数的ReLU（Parametric ReLU，PReLU）引入一个可学习的参数，不同神经元可以有不同的参数[He et al., 2015]。对于第i个神经元，其PReLU的定义为

$$\text{PReLU}_\text{i}(x) = \begin{cases} x & \text{if } x > 0 \\ a_i x & \text{if } x \leq 0 \end{cases} \tag{4.20}$$

$$= \max(0, x) + a_i \min(0, x), \tag{4.21}$$

其中，a_i为$x \leq 0$时函数的斜率。因此，PReLU是非饱和函数。如果$a_i = 0$，那么PReLU就退化为ReLU。如果a_i为一个很小的常数，则PReLU可以看作带泄露的ReLU。PReLU可以允许不同神经元具有不同的参数，也可以一组神经元共享一个参数。



(a) ReLU 和 softplus 函数　　(b) PReLU

图 4.4: ReLU、PReLU 和 softplus 函数

Softplus 函数

Softplus 函数 [Dugas et al., 2001] 可以看作是 rectifier 函数的平滑版本，其定义为

$$\textbf{softplus}(x) = \log(1 + e^x) \tag{4.22}$$

softplus 函数其导数刚好是 logistic 函数。softplus 虽然也有具有单侧抑制、宽兴奋边界的特性，却没有稀疏激活性。

图4.4给出了 ReLU、PReLU 以及 softplus 函数的示例。

Maxout 单元 Maxout 单元 [Goodfellow et al., 2013] 也是一种分段线性函数。Sigmoid 型函数、ReLU 等激活函数的输入是神经元的净输入 z，是一个标量。而 maxout 单元的输入不是经过加权叠加后的净输入，而是神经元的全部原始输入，是一个向量 $\mathbf{x} = [x_1, x_2, \cdots, x_n]^\top$。

每个 maxout 单元有 k 个权重向量 $\mathbf{w}_i \in \mathbb{R}^n$ 和偏置 b_i $(i \in [1, k])$。对于输入 $\mathbf{x}$，可以得到 k 个净输入 z

$$z_i = \mathbf{w}_i \mathbf{x} + b_i \tag{4.23}$$

$$= \sum_{j=1}^{n} w_{i,j} x_j + b_i, \tag{4.24}$$

其中，$\mathbf{w}_i = [w_{i,1}, \cdots, w_{i,n}]^\top$ 为第 i 个权重向量。

激活函数	函数	导数
Logistic 函数	$f(x)=\frac{1}{1+e^{-x}}$	$f'(x)=f(x)(1-f(x))$
Tanh 函数	$f(x)=\frac{2}{1+e^{-2x}}-1$	$f'(x)=1-f(x)^2$
ReLU	$f(x)=\begin{cases}0 & \text{for} \quad x<0\\ x & \text{for} \quad x\geq 0\end{cases}$	$f'(x)=\begin{cases}0 & \text{for} \quad x<0\\ 1 & \text{for} \quad x\geq 0\end{cases}$
SoftPlus 函数	$f(x)=\ln(1+e^x)$	$f'(x)=\frac{1}{1+e^{-x}}$

表 4.1: 常见激活函数及其导数

Maxout 单元的非线性函数定义为

$$\text{maxout}(\mathbf{x})=\max_{i\in[1,k]}(z_i). \tag{4.25}$$

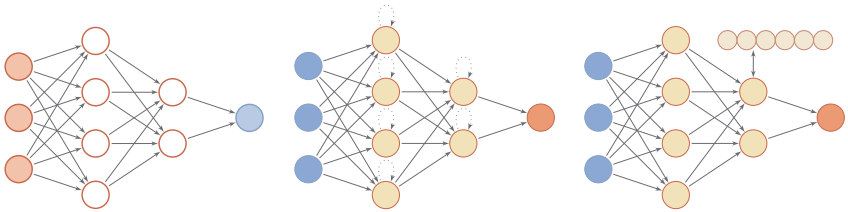
Maxout 单元不单是净输入到输出之间的非线性映射，而是整体学习输入到输出之间的非线性映射关系。Maxout 激活函数可以看作任意凸函数的分段线性近似，并且在有限的点上是不可微的。采用 maxout 单元的神经网络也就做**maxout 网络**。

4.1.3 网络结构

一个生物神经细胞的功能比较简单，而人工神经元只是生物神经细胞的理想化和简单实现，功能更加简单。要想模拟人脑的能力，单一的神经元是远远不够的，需要通过很多神经元一起协作来完成复杂的功能。这样通过一定的连接方式或信息传递方式进行协作的神经元可以看作是一个网络，叫做**人工神经网络**，简称**神经网络**。

到目前为止，研究者已经发明了各种各样的神经网络结构。目前常用的神经网络结构有以下三种：

前馈网络 网络中各个神经元按接受信息的先后分为不同的组。每一组可以看作一个神经层。每一层中的神经元接受前一层神经元的输出，并输出到下一层神经元。整个网络中的信息是朝一个方向传播，没有反向的信息传播。前馈网



(a) 前馈网络 (b) 反馈网络 (c) 记忆网络

图 4.5: 三种不同的网络模型

络可以用一个有向无环路图表示。前馈网络可以看作一个**函数**，通过简单非线性函数的多次复合，实现输入空间到输出空间的复杂映射。这种网络结构简单，易于实现。前馈网络包括全连接前馈网络和卷积神经网络等。

反馈网络 网络中神经元不但可以接收其它神经元的信号，也可以接收自己的反馈信号。和前馈网络相比，反馈网络在不同的时刻具有不同的状态，具有记忆功能，因此反馈网络可以看作一个**程序**，也具有更强的计算能力。反馈神经网络可用一个完备的无向图来表示。

记忆网络 记忆网络在前馈网络或反馈网络的基础上，引入一组记忆单元，用来保存中间状态。同时，根据一定的取址、读写机制，来增强网络能力。和反馈网络相比，忆网络具有更强的记忆功能。

图4.5给出了前馈网络、反馈网络和记忆网络的网络结构示例。

4.2 前馈神经网络

给定一组神经元，我们可以以神经元为节点来构建一个网络。不同的神经网络模型有着不同网络连接的拓扑结构。一种比较直接的拓扑结构是前馈网络。**前馈神经网络**（Feedforward Neural Network）是最早发明的简单人工神经网络。

在前馈神经网络中，各神经元分别属于不同的层。每一层的神经元可以接收前一层神经元的信号，并产生信号输出到下一层。第一层叫**输入层**，最后一

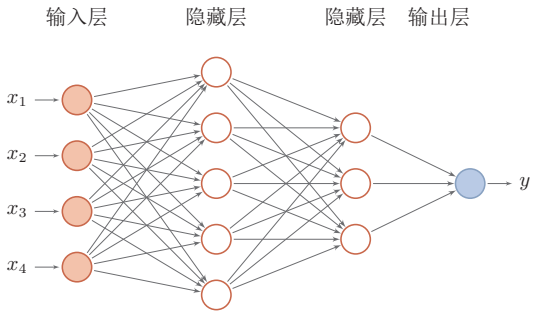


图 4.6: 多层神经网络

层叫**输出层**，其它中间层叫做**隐藏层**。整个网络中无反馈，信号从输入层向输出层单向传播，可用一个有向无环图表示。

前馈神经网络也经常称为**多层感知器**（Multilayer Perceptron，MLP）。但**多层感知器**的叫法并不是十分合理，因为前馈神经网络其实是由多层的logistic回归模型（连续的非线性函数）组成，而不是由多层的感知器（不连续的非线性函数）组成 [Bishop, 2006]。

图4.6给出了前馈神经网络的示例。

4.2.1 前馈计算

给定一个前馈神经网络，我们用下面的记号来描述这样网络。

- L：表示神经网络的层数；
- n^l：表示第 l 层神经元的个数；
- $f_l(\cdot)$：表示 l 层神经元的激活函数；
- $W^{(l)} \in \mathbb{R}^{n^l \times n^{l-1}}$：表示 $l-1$ 层到第 l 层的权重矩阵；
- $\mathbf{b}^{(l)} \in \mathbb{R}^{n^l}$：表示 $l-1$ 层到第 l 层的偏置；
- $\mathbf{z}^{(l)} \in \mathbb{R}^{n^l}$：表示 l 层神经元的净输入（净活性值）；
- $\mathbf{a}^{(l)} \in \mathbb{R}^{n^l}$：表示 l 层神经元的输出（活性值）。

前馈神经网络通过下面公式进行信息传播，

$$\mathbf{z}^{(l)} = W^{(l)} \cdot \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \tag{4.26}$$

$$\mathbf{a}^{(l)} = f_l(\mathbf{z}^{(l)}) \tag{4.27}$$

公式 (4.26) 和 (4.27) 也可以合并写为：

$$\mathbf{z}^{(l)} = W^{(l)} \cdot f_{l-1}(\mathbf{z}^{(l-1)}) + \mathbf{b}^{(l)} \tag{4.28}$$

或者

$$\mathbf{a}^{(l)} = f_l(W^{(l)} \cdot \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}). \tag{4.29}$$

这样，前馈神经网络可以通过逐层的信息传递，得到网络最后的输出 $\mathbf{a}^L$。整个网络可以看作一个复合函数 $f(\mathbf{x}; W, \mathbf{b})$，将输入 x 作为第 1 层的输入 $\mathbf{a}^{(0)}$，将第 L 层的输出 $\mathbf{a}^{(L)}$ 作为整个函数的输出。

$$\mathbf{x} = \mathbf{a}^{(0)} \to \mathbf{z}^{(1)} \to \mathbf{a}^{(1)} \to \mathbf{z}^{(2)} \to \cdots \to \mathbf{a}^{(L-1)} \to \mathbf{z}^{(L)} \to \mathbf{a}^{(L)} = f(\mathbf{x}; W, \mathbf{b})). \tag{4.30}$$

4.2.2 应用到机器学习

在机器学习中，输入样本的特征对分类器的影响很大。以监督学习为例，好的特征可以极大提高分类器的性能。因此，要取得好的分类效果，需要样本的原始特征向量 $\mathbf{x}$ 转换到更有效的特征向量 $\phi(\mathbf{x})$，这个过程叫做特征抽取或特征转换。

多层前馈神经网络可以看作是一个非线性复合函数 $f : \mathbb{R}^n \to \mathbb{R}^m$，将输入 $\mathbf{x} \in \mathbb{R}^n$ 映射到输出 $\mathbf{r} \in \mathbb{R}^m$。因此，多层前馈神经网络也可以看成是一种特征转换方法，其输出 $\mathbf{r}$ 作为分类器的输入进行分类。

给定一个训练样本 $(\mathbf{x}, \mathbf{y})$，先利用多层前馈神经网络将 $\mathbf{x}$ 映射到 $\mathbf{r}$，然后再将 $\mathbf{r}$ 输入到分类器 $g(\cdot)$。

$$\hat{\mathbf{y}} = g(\mathbf{r}, \theta) \tag{4.31}$$

$$= g(f(\mathbf{x}; W, \mathbf{b}), \theta), \tag{4.32}$$

这里，$g(\cdot)$ 为线性或非线性的分类器，θ 为分类器 $g(\cdot)$ 的参数，$\hat{\mathbf{y}}$ 为分类器的输出。

特别地，如果分类器 $g(\cdot)$ 为logistic回归或softmax回归，那么 $g(\cdot)$ 也可以看成是网络的最后一层，即神经网络直接输出不同类别的后验概率。

Logistic回归参考第2.5.2节。

对于两类分类问题 $y \in \{0, 1\}$，logistic回归分类器可以看成神经网络的最后一层。也就是说，网络的最后一层只用一个神经元，并且其激活函数为logistic函数。网络的输出可以直接可以作为两个类别的后验概率。

$$P(y = 1|\mathbf{x}) = f(\mathbf{x}; W, \mathbf{b}) \tag{4.33}$$

其中，$f(\mathbf{x}; W, \mathbf{b}) \in \mathbb{R}$。

Softmax回归参考第2.5.3节。

对于多类分类问题 $y \in \{1, \cdots, C\}$，如果使用softmax回归分类器，相当于网络最后一层设置 C 个神经元，其输出经过softmax函数进行归一化后可以作为每个类的后验概率。

$$\hat{\mathbf{y}} = \text{softmax}(f(\mathbf{x}; W, \mathbf{b})), \tag{4.34}$$

其中，$f(\mathbf{x}; W, \mathbf{b}) \in \mathbb{R}^C$ 为神经网络输出，$\hat{\mathbf{y}} \in \mathbb{R}^C$ 为预测的不同类别后验概率组成的向量。

如果采用交叉熵损失函数，对于样本 $(\mathbf{x}, \mathbf{y})$，其损失函数为

$$\mathcal{L}(y, f(\mathbf{x}; W, \mathbf{b})) = -\mathbf{y}^\top \log \hat{\mathbf{y}} \tag{4.35}$$

这样，给定一组训练样本，网络的参数就可以通过梯度下降法来进行学习。梯度下降法需要计算损失函数对参数的导数，如果通过链式法则逐一进行求导效率比较低。因此在神经网络的训练中经常使用**反向传播算法**来计算梯度。

4.3 反向传播算法

给定一组样本 $(\mathbf{x}^{(i)}, y^{(i)}), 1 \leq i \leq N$，用前馈神经网络的输出为 $f(\mathbf{x}|W, \mathbf{b})$，目标函数为：

$$\mathcal{R}(W, \mathbf{b}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y^{(i)}, f(\mathbf{x}^{(i)}; W, \mathbf{b})) + \frac{1}{2} \lambda \|W\|_F^2, \tag{4.36}$$

$$= \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(W, \mathbf{b}; \mathbf{x}^{(i)}, y^{(i)}) + \frac{1}{2} \lambda \|W\|_F^2, \tag{4.37}$$

这里的正则化项只对权重参数W，而不对偏置$\mathbf{b}$。

这里，W和$\mathbf{b}$包含了每一层的权重矩阵和偏置向量；$\|W\|_F^2$是正则化项，用来防止过拟合；λ是为正数的超参。λ越大，W越接近于0。这里的$\|W\|_F^2$一般使用Frobenius范数：

$$\|W\|_F^2 = \sum_{l=1}^{L}\sum_{i=1}^{n^l}\sum_{j=1}^{n^{l-1}}(W_{ij}^{(l)})^2. \tag{4.38}$$

我们的目标是最小化$\mathcal{R}(W,\mathbf{b};\mathbf{x},y)$。如果采用梯度下降方法，我们可以用如下方法更新参数：

$$W^{(l)} = W^{(l)} - \alpha\frac{\partial \mathcal{R}(W,\mathbf{b})}{\partial W^{(l)}}, \tag{4.39}$$

$$= W^{(l)} - \alpha\left(\frac{1}{N}\sum_{i=1}^{N}(\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x}^{(i)},y^{(i)})}{\partial W^{(l)}}) - \lambda W^{(l)}\right), \tag{4.40}$$

$$\mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \alpha\frac{\partial \mathcal{R}(W,\mathbf{b})}{\partial \mathbf{b}^{(l)}}, \tag{4.41}$$

$$= \mathbf{b}^{(l)} - \alpha\left(\frac{1}{N}\sum_{i=1}^{N}\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x}^{(i)},y^{(i)})}{\partial \mathbf{b}^{(l)}}\right), \tag{4.42}$$

其中，α为学习率。

我们首先来看下$\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial W^{(l)}}$怎么计算。

链式法则参见第231页的公式A.26

根据链式法则，$\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial W_{ij}^{(l)}}$可以写为

$$\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial W_{ij}^{(l)}} = \left(\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial \mathbf{z}^{(l)}}\right)^{\mathrm{T}}\frac{\partial \mathbf{z}^{(l)}}{\partial W_{ij}^{(l)}}. \tag{4.43}$$

公式4.43的第一项是为目标函数关于第l层的神经元$\mathbf{z}^{(l)}$的偏导数，我们称为**误差项**$\delta^{(l)}$：

$$\delta^{(l)} = \frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial \mathbf{z}^{(l)}} \in \mathbb{R}^{n^{(l)}}. \tag{4.44}$$

误差项$\delta^{(l)}$来表示第l层的神经元对最终误差的影响，也反映了最终的输出对第l层的神经元对最终误差的敏感程度。

公式4.43中的第二项是l层的神经元$\mathbf{z}^{(l)}$关于参数$W_{ij}^{(l)}$的偏导数。

我们分别来计算这两个偏导数。

计算误差项 $\delta^{(l)}$ 我们先来看下第 l 层的**误差项** $\delta^{(l)} = \frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial \mathbf{z}^{(l)}}$ 怎么计算。

$$\delta^{(l)} \triangleq \frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial \mathbf{z}^{(l)}} \tag{4.45}$$

$$= \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \cdot \frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial \mathbf{z}^{(l+1)}} \tag{4.46}$$

$$= \mathbf{diag}(f_l'(\mathbf{z}^{(l)})) \cdot (W^{(l+1)})^{\mathrm{T}} \cdot \delta^{(l+1)} \tag{4.47}$$

$$= f_l'(\mathbf{z}^{(l)}) \odot ((W^{(l+1)})^{\mathrm{T}} \delta^{(l+1)}), \tag{4.48}$$

其中 $\odot$ 是向量的点积运算符，表示每个元素相乘。

上述推导中，关键是公式4.46到公式4.47的推导。公式4.46中有三项，第三项根据定义为 $\delta^{(l+1)}$。

第二项因为 $\mathbf{z}^{(l+1)} = W^{(l+1)} \cdot \mathbf{a}^{(l)} + \mathbf{b}^{(l)}$，所以

$$\frac{\partial \mathbf{z}^{(l+1)}}{\partial a^{(l)}} = (W^{(l+1)})^{\mathrm{T}}. \tag{4.49}$$

第一项因为 $\mathbf{a}^{(l)} = f_l(\mathbf{z}^{(l)})$，而 $f_l(\cdot)$ 为按位计算的函数。因此

$$\frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} = \frac{\partial f_l(\mathbf{z}^{(l)})}{\partial \mathbf{z}^{(l)}} \tag{4.50}$$

$$= \mathbf{diag}(f_l'(\mathbf{z}^{(l)})). \tag{4.51}$$

从公式4.48可以看出，第 l 层的误差项可以通过第 $l+1$ 层的误差项计算得到。这就是误差的**反向传播**（Backpropagation，BP）。反向传播算法的含义是：第 l 层的一个神经元的误差项（或敏感性）是所有与该神经元相连的第 $l+1$ 层的神经元的误差项的权重和。然后，再乘上该神经元激活函数的梯度。

计算 $\frac{\partial \mathbf{z}^{(l)}}{\partial W_{ij}^{(l)}}$ 我们先来计算公式4.43中的第二项 $\frac{\partial \mathbf{z}^{(l)}}{\partial W_{ij}^{(l)}}$。

因为 $\mathbf{z}^{(l)} = W^{(l)} \cdot \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$，所以

$$\frac{\partial \mathbf{z}^{(l)}}{\partial W_{ij}^{(l)}} = \frac{\partial (W^{(l)} \cdot \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)})}{\partial W_{ij}^{(l)}} = \begin{bmatrix} 0 \\ \vdots \\ a_j^{(l-1)} \\ \vdots \\ 0 \end{bmatrix}. \tag{4.52}$$

在得到上面两个偏导数之后，公式 (4.43) 可以写为

$$\frac{\partial \mathcal{L}(W, \mathbf{b}; \mathbf{x}, y)}{\partial W_{ij}^{(l)}} = (\delta^{(l)})^{\mathrm{T}} \mathbf{a}^{(l-1)} \tag{4.53}$$

$$= \delta_i^{(l)} a_j^{(l-1)}. \tag{4.54}$$

进一步，$\mathcal{L}(W, \mathbf{b}; \mathbf{x}, y)$ 关于第 l 层权重 $W^{(l)}$ 的梯度为

$$\frac{\partial \mathcal{L}(W, \mathbf{b}; \mathbf{x}, y)}{\partial W^{(l)}} = \delta^{(l)} (\mathbf{a}^{(l-1)})^{\mathrm{T}}. \tag{4.55}$$

同理可得，$\mathcal{L}(W, \mathbf{b}; \mathbf{x}, y)$ 关于第 l 层偏置 $\mathbf{b}^{(l)}$ 的梯度为

$$\frac{\partial \mathcal{L}(W, \mathbf{b}; \mathbf{x}, y)}{\partial \mathbf{b}^{(l)}} = \delta^{(l)}. \tag{4.56}$$

在计算出每一层的误差项之后，我们就可以得到每一层参数的梯度。因此，前馈神经网络的训练过程可以分为以下三步：

1. 前馈计算每一层的状态和激活值，直到最后一层；
2. 反向传播计算每一层的误差项；
3. 计算每一层参数的偏导数，并更新参数。

具体的训练过程如算法4.1所示，也叫**反向传播算法**。

4.4 自动梯度计算

神经网络的参数主要通过梯度下降来进行优化的。当确定了风险函数以及网络结构后，我们就可以手动用链式法则来计算风险函数对每个参数的梯度，并用代码进行实现。但是手动求导并转换为计算机程序的过程非常琐碎并容易出错，导致实现神经网络变得十分低效。目前，几乎所有的主流深度学习框架都包含了自动梯度计算的功能，即我们可以只考虑网络结构并用代码实现，其梯度可以自动进行计算，无需人工干预。这样开发的效率就大大提高了。

← 第 i 行

自动求导的方法可以分为以下三类：

算法 4.1: 反向传播算法

输入: 训练集: $(\mathbf{x}^{(i)}, y^{(i)}), i = 1, \cdots, N$，最大迭代次数：$T$

输出: $W, \mathbf{b}$

初始化 $W, \mathbf{b}$;
for $t = 1 \cdots T$ **do**
  for $i = 1 \cdots N$ **do**
    (1) 前馈计算每一层的状态和激活值，直到最后一层;
    (2) 用公式4.48反向传播计算每一层的误差 $\delta^{(l)}$;
    (3) 用公式4.55和4.56计算每一层参数的导数;
      $\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x}^{(i)},y^{(i)})}{\partial W^{(l)}} = \delta^{(l)}(\mathbf{a}^{(l-1)})^{\mathrm{T}}$;
      $\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x}^{(i)},y^{(i)})}{\partial \mathbf{b}^{(l)}} = \delta^{(l)}$;
    (4) 更新参数;
      $W^{(l)} = W^{(l)} - \alpha \sum_{i=1}^{N} (\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x}^{(i)},y^{(i)})}{\partial W^{(l)}}) - \alpha\lambda W^{(l)}$;
      $\mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \alpha \sum_{i=1}^{N} (\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x}^{(i)},y^{(i)})}{\partial \mathbf{b}^{(l)}})$;
  end
end

数值微分 数值微分（Numerical Differentiation）是用数值方法来计算函数 $f(x)$ 的导数。函数 $f(x)$ 的点 x 的导数定义为

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}. \tag{4.57}$$

舍入误差（Round-off Error）是指数值计算中由于数字舍入造成的近似值和精确值之间的差异，比如用浮点数来表示数。

截断误差（Truncation Error）是数学模型的理论解与数值计算问题的精确解之间的误差。

要计算函数 $f(x)$ 在点 x 的导数，可以对 x 加上一个很少的非零的扰动 Δx，通过上述定义来直接计算函数 $f(x)$ 的梯度。数值微分方法非常容易实现，但是找到一个合适的扰动 Δx 却十分困难。如果 Δx 过小，会引起数值计算问题，比如舍入误差；如果 Δx 过大，会增加截断误差，使得导数计算不准确。因此，数值微分的实用性比较差。在实际应用，经常使用下面公式来计算梯度，可以减少截断误差。

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x}. \tag{4.58}$$

数值微分的另外一个问题是计算复杂度。假设参数数量为 n，则每个参数都需要单独施加扰动，并计算梯度。假设每次正向传播的计算复杂度为 $O(n)$，则计算数值微分的总体时间复杂度为 $O(n^2)$。

符号微分 符号微分（Symbolic Differentiation）是一种基于符号计算的自动求导方法。

符号计算（Symbolic Computation），也叫**代数计算**，是指用计算机来处理带有变量的数学表达式。这里的变量看作是符号（Symbols），一般不需要代入具体的值。符号计算的输入和输出都是数学表达式，一般包括对数学表达式的化简、因式分解、微分、积分、解代数方程、求解常微分方程等运算。

和符号计算相对应的概念是数值计算，即将数值代入数学表示中进行计算。

比如数学表达式的化简：

$$\text{输入：} 3x - x + 2x + 1 \tag{4.59}$$

$$\text{输出：} 4x + 1. \tag{4.60}$$

符号计算一般来讲是对输入的表达式，通过不断使用一些事先定义的规则进行转换。当转换结果不能再继续使用变换规则时，便停止计算。

Theano [Bergstra et al., 2010] 和 Tensorflow [Abadi et al., 2016] 都采用了符号微分的方法进行自动求解梯度。符号微分可以在编译时就计算梯度的数学表示，并进一步利用符号计算方法进行优化。此外，符号计算的一个优点是符号计算和平台无关，可以在CPU或GPU上运行。

符号微分也有一些不足之处。一是编译时间较长，特别是对于循环，需要很长时间进行编译。二是为了进行符号微分，一般需要设计一种专门的语言来表示数学表达式，并且要对变量（符号）进行预先声明。三是程序很难调试。

自动微分 自动微分（Automatic Differentiation）是一种可以对一个（程序）函数进行计算导数的方法。符号微分的处理对象是数学表达式，而自动微分的处理对象是一个函数或一段程序。而自动微分可以直接在原始程序代码进行微分。自动微分的基本原理是所有的数值计算可以分解为一些基本操作，包含 $+, -, \times, /$ 和一些初等函数 $\exp, \log, \sin, \cos$ 等。图4.7给出了符号微分与自动微分的关联。

自动微分也是利用链式法则来自动计算一个复合函数的梯度。以神经网络中常见的复合函数 $f(x; w, b) = \sigma(wx + b)$ 为例，

$$f(x; w, b) = \sigma(wx + b) \tag{4.61}$$

$$= \frac{1}{\exp(-(wx + b)) + 1}. \tag{4.62}$$

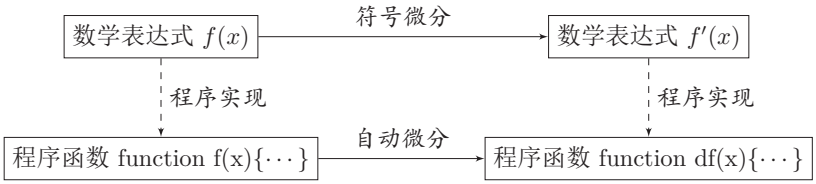


图 4.7: 符号微分与自动微分对比

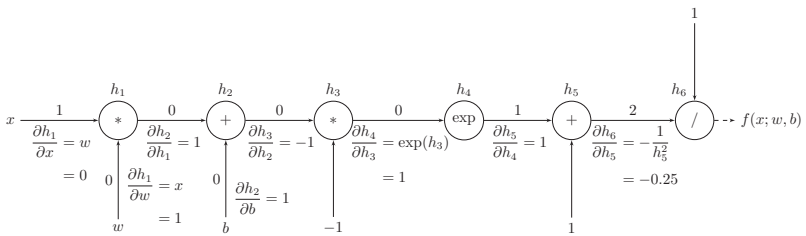


图 4.8: 复合函数 $f(x;w,b)=\sigma(wx+b)$ 的计算图

为了简单起见，我们设复合函数 $f(x;w,b)$ 的输入 x 为标量，参数为权重 w 和偏置 b。我们可以将其分解为一系列的基本操作，并构成一个计算图（Computational Graph），其中每个节点为一个基本操作，如图4.8所示。每一步基本操作的导数都可以通过简单的规则来实现。

计算图是数学运算的图形化表示。计算图中节点表示一个变量或基本操作。

这样函数 $f(x;w,b)$ 关于参数 w 和 b 的导数可以通过计算图上的节点 $f(x;w,b)$ 与参数 w 和 b 之间路径上所有的导数连乘来得到，即

$$\frac{\partial f(x;w,b)}{\partial w}=\frac{\partial f(x;w,b)}{\partial h_6}\frac{\partial h_6}{\partial h_5}\frac{\partial h_5}{\partial h_4}\frac{\partial h_4}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial w}, \tag{4.63}$$

$$\frac{\partial f(x;w,b)}{\partial b}=\frac{\partial f(x;w,b)}{\partial h_6}\frac{\partial h_6}{\partial h_5}\frac{\partial h_5}{\partial h_4}\frac{\partial h_4}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial b}. \tag{4.64}$$

以 $\frac{\partial f(x;w,b)}{\partial w}$ 为例，当 $x=1,w=0,b=0$ 时，可以得到

$$\frac{\partial f(x;w,b)}{\partial w}|_{x=1,w=0,b=0}=\frac{\partial f(x;w,b)}{\partial h_6}\frac{\partial h_6}{\partial h_5}\frac{\partial h_5}{\partial h_4}\frac{\partial h_4}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial w} \tag{4.65}$$

$$=1\times-0.25\times1\times1\times-1\times1\times1 \tag{4.66}$$

$$=0.25. \tag{4.67}$$

如果函数和参数之间有多条路径，可以将这多条路径上的导数再进行相加，得到最终的梯度。

按照计算导数的顺序，自动微分可以分为两种模式：前向模式和反向模式。**前向模式**是按计算图中计算方向的相同方向来递归地计算梯度。以 $\frac{\partial f(x;w,b)}{\partial w}$ 为例，当 $x=1, w=0, b=0$ 时，前向模式的累积计算顺序如下：

$$\frac{\partial h_1}{\partial w} = x = 1 \tag{4.68}$$
$$\frac{\partial h_2}{\partial w} = \frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial w} = 1 \times 1 = 1 \tag{4.69}$$
$$\frac{\partial h_3}{\partial w} = \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial w} = -1 \times 1 \tag{4.70}$$
$$\vdots \qquad \vdots \tag{4.71}$$
$$\frac{\partial h_6}{\partial w} = \frac{\partial h_6}{\partial h_5}\frac{\partial h_5}{\partial w} = -0.25 \times -1 = 0.25 \tag{4.72}$$
$$\frac{\partial f(x;w,b)}{\partial w} = \frac{\partial f(x;w,b)}{\partial h_6}\frac{\partial h_6}{\partial w} = 1 \times 0.25 = 0.25 \tag{4.73}$$

反向模式是按计算图中计算方向的相反方向来递归地计算梯度。以 $\frac{\partial f(x;w,b)}{\partial w}$ 为例，当 $x=1, w=0, b=0$ 时，反向模式的累积计算顺序如下：

$$\frac{\partial f(x;w,b)}{\partial h_6} = 1 \tag{4.74}$$
$$\frac{\partial f(x;w,b)}{\partial h_5} = \frac{\partial f(x;w,b)}{\partial h_6}\frac{\partial h_6}{\partial h_5} = 1 \times -0.25 \tag{4.75}$$
$$\frac{\partial f(x;w,b)}{\partial h_4} = \frac{\partial f(x;w,b)}{\partial h_5}\frac{\partial h_5}{\partial h_4} = -0.25 \times 1 = -0.25 \tag{4.76}$$
$$\vdots \qquad \vdots \tag{4.77}$$
$$\frac{\partial f(x;w,b)}{\partial w} = \frac{\partial f(x;w,b)}{\partial h_1}\frac{\partial h_1}{\partial w} = 0.25 \times 1 = 0.25 \tag{4.78}$$

前向模式和反向模式可以看作是应用链式法则的两种梯度累积损方式。从反向模式的计算顺序可以看出，**反向模式和反向传播的计算梯度的方式相同**。

对于一般的函数形式 $f: \mathbb{R}^n \to \mathbb{R}^m$，前向模式需要对每一个输入变量都进行一遍遍历，共需要 n 遍。而反向模式需要对每一个输出都进行一个遍历，共需要 m 遍。当 $n > m$ 时，反向模式更高效。在前馈神经网络的参数学习中，风险函数为 $f: \mathbb{R}^n \to \mathbb{R}$，输出为标量，因此采用反向模式为最有效的计算方式，只需要一遍计算。

在深度学习框架里，Python语言autograd[1]和Lua语言的Torch-autograd[2]采用了自动微分方法，可以对各自的表达式、程序进行进行自动求导。比如，Autograd可以直接对Python和Numpy代码进行求导。

符号微分和自动微分都利用计算图和链式法则来自动求解导数。符号微分在编译阶段先构造一个复合函数的计算图，通过符号计算得到导数的表达式，还可以对导数表达式进行优化，在程序运行阶段才代入变量的具体数值进行计算导数。而自动微分则无需事先编译，在程序运行阶段边计算边记录计算图，计算图上的局部梯度都直接代入数值进行计算，然后用前向或反向模式来计算最终的梯度。

4.5 梯度消失问题

在神经网络中误差反向传播的迭代公式为

$$\delta^{(l)} = f_l'(\mathbf{z}^{(l)}) \odot ((W^{(l+1)})^{\mathrm{T}}\delta^{(l+1)}, \tag{4.79}$$

其中需要用到激活函数f_l的导数。

误差从输出层反向传播时，在每一层都要乘以该层的激活函数的导数。

当我们使用sigmoid型函数：logistic函数$\sigma(x)$或tanh函数时，其导数为

$$\sigma'(x) = \sigma(x)(1-\sigma(x)) \in [0, 0.25] \tag{4.80}$$

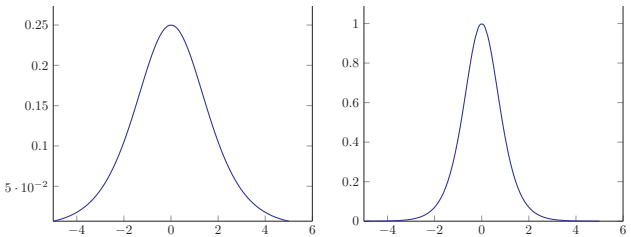
$$\tanh'(x) = 1 - (\tanh(x))^2 \in [0, 1]. \tag{4.81}$$

我们可以看到，sigmoid型函数导数的值域都小于1。并且由于sigmoid型函数的饱和性，饱和区的导数更是接近于0。这样，误差经过每一层传递都会不断衰减。当网络层数很深时，梯度就会不停的衰减，甚至消失，使得整个网络很难训练。这就是所谓的**梯度消失问题**（Vanishing Gradient Problem），也叫**梯度弥散**问题。梯度消失问题在过去的二三十年里一直没有有效地解决，是阻碍神经网络发展的重要原因之一。

在深层神经网络中，减轻梯度消失问题的一个方法就是使用导数比较大的激活函数，比如ReLU（rectifier函数）。这样误差可以很好地传播，训练速度得到了很大的提高。

[1] https://github.com/HIPS/autograd

[2] https://github.com/twitter/torch-autograd



(a) logistic 函数的导数 (b) tanh 函数的导数

图 4.9: 激活函数的导数

4.6 训练方法

随机梯度下降 对于大规模数据,很难使用批量梯度下降。随机梯度下降（Stochastic Gradient Descent，SGD）

随机梯度下降：每次迭代（随机）选择**一个**样本计算梯度，并更新参数。

$$\mathbf{a}_{t+1} = \mathbf{a}_t - \lambda \frac{\partial \mathcal{R}\big(\theta_t; x^{(t)}, y^{(t)}\big)}{\partial \theta}, \tag{4.82}$$

Mini-batch 随机梯度下降 Mini-batch 随机梯度下降：批量梯度下降和随机梯度下降的折中。

每次迭代选取 $m(1 \leq m \leq N)$ 个样本进行参数更新。

$$\mathbf{a}_{t+1} = \mathbf{a}_t - \lambda \frac{1}{m} \sum_{i \in I_t} \frac{\partial \mathcal{R}\big(\theta_t; x^{(i)}, y^{(i)}\big)}{\partial \theta}, \tag{4.83}$$

4.7 总结和深入阅读

多层前馈神经网络作为一种机器学习方法在很多模式识别和机器学习的教材中都有介绍，比如《Pattern Recognition and Machine Learning》[Bishop, 2006]，《Pattern Classification》[Duda et al., 2001] 等。虽然多层前馈网络在 2000 年以前就已被广泛使用，但是基本上都是三层网络（即只有一个隐藏层），

神经元的激活函数基本上都是sigmoid型函数，并且使用的损失函数大多数是平方损失。

参考文献

Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

C.M. Bishop. *Pattern recognition and machine learning*. Springer New York., 2006.

R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley, New York, 2nd edition, 2001. ISBN 0471056693.

Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. Incorporating second-order functional knowledge for better option pricing. *Advances in Neural Information Processing Systems*, pages 472–478, 2001.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C Courville, and Yoshua Bengio. Maxout networks. In *ICML*, volume 28, pages 1319–1327, 2013.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.

Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, 2013.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.