第 5 章

深度学习与神经网络

自从 2016 年基于“深度学习”工作原理的人工智能机器人——阿尔法狗(AlphaGo)第一次在围棋领域击败人类围棋世界冠军以来,深度学习算法声名鹊起,极大地推动了人工智能领域的研究进程,并迅速渗透各行各业,同时带动了一大批新兴产业。本章以深度学习为背景,重点介绍卷积神经网络(CNN)、循环神经网络(RNN)、长短时记忆网络(LSTM)等深度学习模型的原理、结构与计算方法,并给出每个模型详细的理论推导和基于 Python 的代码实现。

5.1 深度学习

5.1.1 深度学习概念

深度学习(Deep Learning,DL)是机器学习(Machine Learning,ML)领域中一个新的研究方向,它被引入机器学习使其更接近于人工智能目标。深度学习的概念来源于人工神经网络(Artificial Neural Network,ANN),所以又称深度神经网络(Deep Neural Networks,DNN)。人工神经网络主要使用计算机的计算单元和存储单元模拟人类大脑神经系统中大量的神经细胞(神经元)通过神经纤维传导并相互协同工作的原理。深度学习在一定程度上等同于多层或者深层神经网络,如图 5-1 所示。深度学习通过算法构造多层神经网络,经过多层处理,逐渐将初始的“低层”特征表示转化为“高层”特征表示,再用自学习模型便可完成复杂的分类等学习任务。因此,可将深度学习理解为深度特征学习或深度表示学习。深度学习使机器模仿视听和思考等人类的活动,解决了很多复杂的模式识别难题,使得人工智能技术前进了一大步。典型的深度学习模型有卷积神经网络(Convolutional Neural Networks,CNN)、循环神经网络(Recurrent Neural Network,RNN)、长短时记忆网络(Long Short-Term Memory,LSTM)、深度置信网络(Deep Belief Network,DBN)模型等。

5.1.2 深度学习原理

相比传统的机器学习,深度学习有更好的特征学习能力。在传统的机器学习算法中需要手工编码特征,相比之下深度学习对特征的识别由算法自动完成,机器学习的这个处理过程不仅耗时,而且还需要较高的专业知识和一定的人工参与才能完成。而深度学习通过大数据技术直接从数据中自动学习各种特征并进行分类或者识别,做到全自动数据分析,如图 5-2 所示。因此,在解决复杂问题时(如目标识别、自然语言处理等),传统机器学习算法通常先把问题分成几块,一个一个地解决好之后再重新组合起来,但是深度学习则是一次性

地端到端(end-to-end)解决。

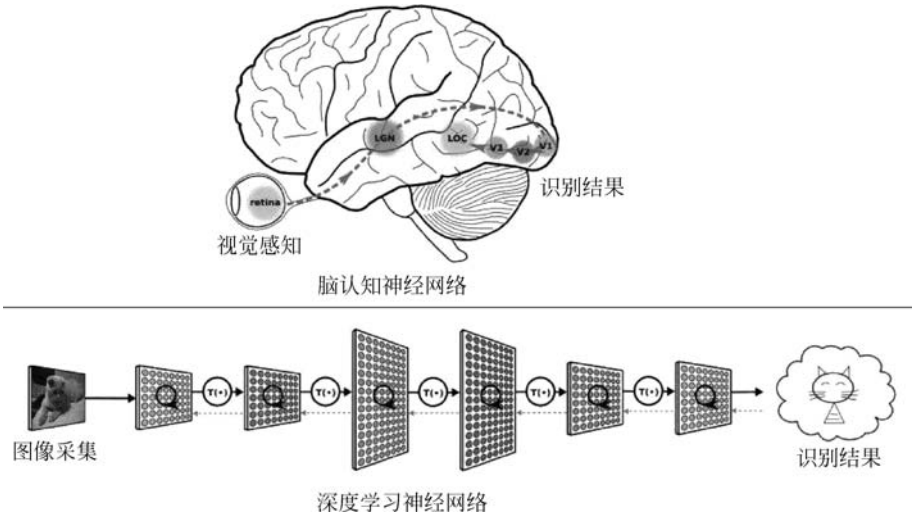


图 5-1　脑认知神经网络与深度学习神经网络

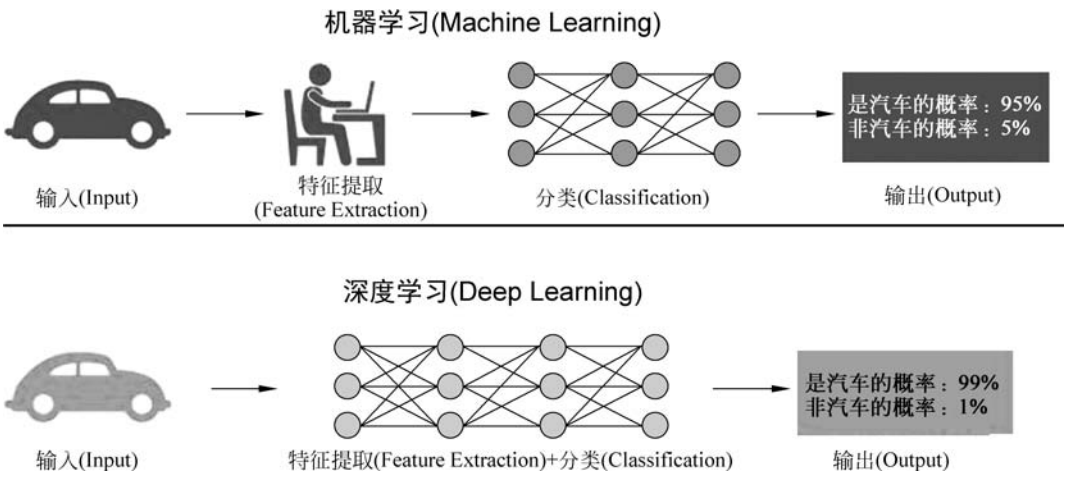


图 5-2　机器学习与深度学习的区别

相较于传统的浅层学习，深度学习的不同之处在于：

(1) 强调了模型结构的深度，通常有 5 层、6 层，甚至十几层的隐层节点，如图 5-3 所示。

(2) 明确了特征学习的重要性，也就是说，通过逐层特征变换，将样本在原空间的特征表示变换到一个新特征空间，从而使分类或预测更容易。与人工规则构造特征的方法相比，利用大数据来学习特征，更能刻画数据丰富的内在信息。

通过设计建立适量的神经元计算节点和多层运算层次结构，选择合适的输入层和输出层，通过网络的学习和调优，建立起从输入到输出的函数关系。虽然不能 100％找到输入与输出的函数关系，但是可以尽可能地逼近现实的关联关系，进而使用训练成功的网络模型实现我们对复杂事务处理的自动化要求。

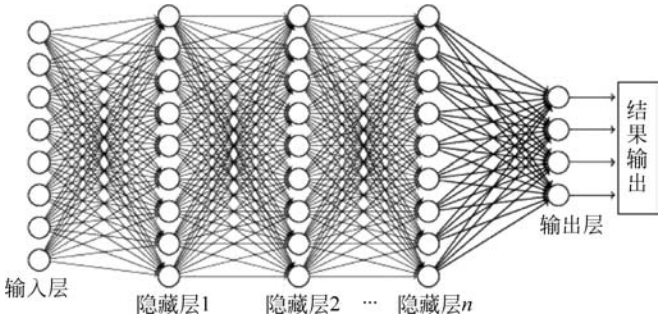


图 5-3　深度神经网络模型结构

5.1.3　深度学习训练

如果对深度学习中的所有层同时训练，时间复杂度会很高。如果每次只训练一层，偏差就会逐层传递。因此在 2006 年，Geoffrey Hinton 提出了在非监督数据上建立多层神经网络的一个有效方法，简单地说就是分两步进行：一是每次训练一层网络；二是调优，使原始表示 X 向上生成的高级表示 R 和该高级表示 R 向下生成的 X' 尽可能一致，方法是：

(1) 首先逐层构建单层神经元，这样每次训练一个单层网络。

(2) 当所有层训练完后，Hinton 使用 Wake-Sleep 算法进行调优。

将除最顶层的其他层间的权重变为双向，这样最顶层仍然是一个单层神经网络，而其他层则变为图模型。向上的权重用于“认知”，向下的权重用于“生成”。然后使用 Wake-Sleep 算法调整所有的权重。让认知和生成达成一致，也就是保证生成的最顶层表示能够尽可能正确地复原底层的节点。例如顶层的一个节点表示人脸，那么所有人脸的图像应该激活这个节点，并且这个结果向下生成的图像应该能够表现为一个大概的人脸图像。

深度学习训练过程主要包含两方面内容，如图 5-4 所示。

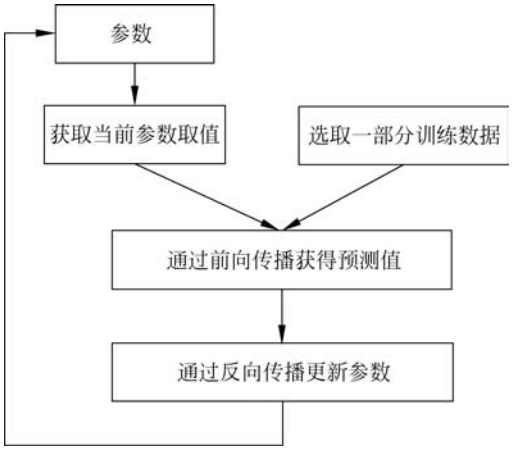


图 5-4　深度学习训练过程

1) 使用自下而上的非监督学习

采用无标定数据或有标定数据分层训练各层参数，这一步可被看作一个无监督训练过程，也可被看作特征学习过程。先用数据训练第 1 层，训练时先学习第 1 层的参数(这一层

可被看作得到一个使输出和输入差别最小的 3 层神经网络的隐藏层)。由于模型容量的限制及稀疏性约束,使得所得到的模型能够学习到数据本身的结构,从而得到比输入更具有表示能力的特征。在学习并得到第 $N-1$ 层后,将 $N-1$ 层的输出作为第 N 层的输入,训练第 N 层,由此分别得到各层的参数。

2) 自顶而下的监督学习

基于第 1 步得到的各层参数来进一步调整多层模型的参数,这一步是一个有监督训练过程。第 1 步类似神经网络的随机初始化初值过程,由于深度学习的第 1 步不是随机初始化,而是通过学习输入数据的结构得到,因而这个初值更接近全局最优,从而能够取得更好的效果,所以深度学习效果好在很大程度上归功于第 1 步的特征学习过程。

5.2　人工神经网络基础

人工神经网络是基于生物学中脑认知神经网络的基本原理,模仿大脑神经系统工作原理所创建的数学模型,它有并行的分布处理能力、高容错性、自我学习等特征。

5.2.1　神经元感知器

人工神经网络中最基本的单元叫神经元,又叫感知器,如图 5-5 所示。它是模拟人脑神经系统的神经元(分析和记忆)、树突(感知)、轴突(传导)的工作原理,借助计算机的快速计算和存储来实现。

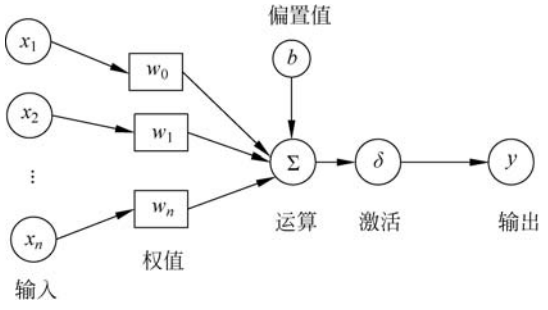


图 5-5　基本神经单元(感知器)示意图

从图 5-5 可以看到,人工神经网络中一个基本的神经元由以下几个部分组成:

- 输入(Input):一个神经元可以接收多个输入$\{x_1, x_2, \cdots, x_n \mid x_i \in \Re\}$。
- 权值(Weight):每个输入都有一个权值 $\omega_i \in \Re$。
- 偏置值(Bias):$b \in \Re$。
- 激活函数(Activate Function):激活函数给神经元引入了非线性因素,使得神经网络可以任意逼近任何非线性函数,这样神经网络就可以应用到众多非线性模型中。
- 输出(Output):神经元输出,该输出可由下面公式计算

$$y = f\left(\sum_{i=0}^{n}(w_i \times x_i)\right) \quad \text{其中 } x_0 = b \tag{5-1}$$

5.2.2 神经网络模型

图 5-6 为神经网络的结构模型图，最左边的层叫作输入层(Input Layer)，最右边的层叫作输出层(Output Layer)。输入层和输出层之间的层叫作隐藏层(Hidden Layer)。含多个隐藏层的神经网络叫作深度神经网络。对于拟合任意一个函数而言，浅层神经网络浅而宽，需要大量的神经元，而深层神经网络深而窄，需要更多的层和较少的神经元。一般深层网络节省资源，但是深层网络并不好训练，需要大量的数据和很好的技巧才能去拟合并训练出好的网络。

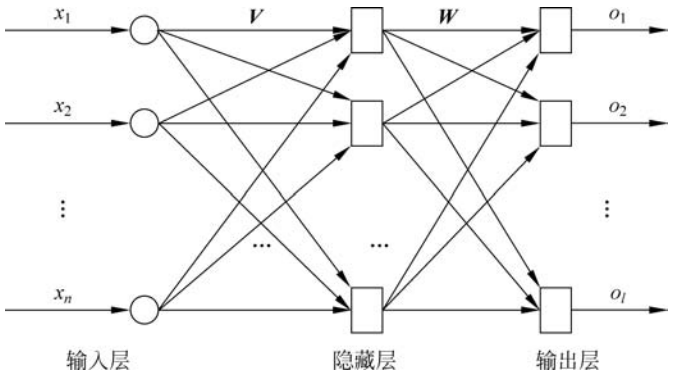


图 5-6 神经网络示意图

5.2.3 学习方式

神经网络的学习方式很多，根据有无数据训练可以将其分为 3 大类。

(1) 有监督学习：将训练样本的数据加入到神经网络的输入端，将期望答案和实际输出做差，可以得到误差信号，通过误差信号来调整权值大小，以此来优化模型输出。

(2) 无监督学习：首先并不给定标准数据样本，而是直接将网络置于环境之中，由自身根据数据特征进行自动学习。

(3) 半监督学习：输入信息介于有监督和无监督之间，不需要给定标准数据样本，但需要对网络的输出做出评判，以此来调整网络参数。

5.2.4 学习规则

学习规则是用来修改神经网络的权值和偏置值的过程和方法，其目的是训练网络，更好地拟合应用的需求，从而完成特殊的任务。常见的学习规则有 Hebb 学习规则、Delta 算法及反向传播算法(Back-propagation，BP)算法，BP 算法是人工神经网络较常采用的学习方法，其基本思想是逐一地由样本集中的样本(X_k，Y_k)计算出实际输出 O_k 和误差测度 E_p，对 w_1，w_2，…，w_n 权值做调整，重复这个循环，直到误差降至最低。用输出层的误差调整输出层权值矩阵，并用此误差估计输出层的直接前导层误差，再用输出层前导层误差估计更前一层的误差，如此获得所有其他各层的误差估计，并用这些估计实现对权矩阵的修改，形成将输出端表现出的误差沿着与输入信号相反的方向逐级向输入端传递的链式求解过程。

BP 算法学习过程应用到深度学习中分为两个子过程：输入数据正向传递子过程和误差数据反向传递子过程，即“正向传播求误差，反向传播求偏导”。完整的学习过程是：对于一个训练样本，将输入正向传播到输出而产生误差。然后将误差信号反向从输出层传递到输入层，

利用该误差信号求出权重修改量 $W_{ji}^{(h)}$（h 表示层数），通过它更新权值 $W_{ji}^{(h)}$，称为一次迭代过程。当误差或者 $W_{ji}^{(h)}$ 仍不满足要求时重复上述操作。

本节以图 5-7 中的三层神经网络模型为例，详细说明 BP 算法的原理及推导求解过程。

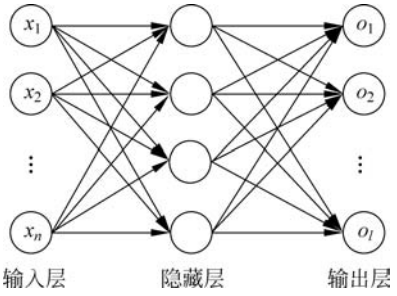


图 5-7　浅层神经网络模型

1. 正向传播求误差

该网络分为三层，设输入层到隐藏层的权值为 $W_{ji}^{(0)}$，隐藏层到输出层的权值为 $W_{ji}^{(1)}$，输入层单元的个数为 n，隐藏层层数为 m，输出层单元个数为 l，采用 Sigmoid 作为激活函数。

输入层的输入向量 $\boldsymbol{X}=(x_1,x_2,\cdots,x_n)$，隐藏层输出向量 $\boldsymbol{H}=(h_1,h_2,\cdots,h_m)$，有

$$\begin{cases}\text{net}_j^{(0)}=\sum_{i=1}^{n}w_{ji}^{(0)}x_i+b_j^{(0)}\\ h_j=f(\text{net}_j^{(0)}-\theta_j^{(0)})=\dfrac{1}{1+\text{e}^{-(\text{net}_j^{(0)}-\theta_j^{(0)})}}\end{cases}\tag{5-2}$$

其中，$\text{net}_j^{(0)}$ 为未激活之前的神经网络计算输出；$w_{ji}^{(0)}$ 为权值；$b_j^{(0)}$ 为节点 h_j 的偏置值；$f()$为激活函数；$\theta_j^{(0)}$ 充当阈值，用来改变单元的活性。解释如下：θ_j 表示该神经元的阈值，根据脑认知生物学知识，只有当神经元接收的信息达到阈值时才会被激活。因此，本书将 net_j 和 θ_j 进行比较，然后通过激活函数处理以产生神经元的输出。

同样，输出层向量 $\boldsymbol{O}=(o_1,o_2,\cdots,o_l)$，有

$$\begin{cases}\text{net}_j^{(1)}=\sum_{i=1}^{m}w_{ji}^{(1)}h_i+b_j^{(1)}\\ o_j=f(\text{net}_j^{(1)}-\theta_j^{(1)})=\dfrac{1}{1+\text{e}^{-(\text{net}_j^{(1)}-\theta_j^{(1)})}}\end{cases}\tag{5-3}$$

其中，$\text{net}_j^{(1)}$ 为未激活之前的神经网络计算输出；$b_j^{(1)}$ 为节点 o_j 的偏置值；$f()$为激活函数；$\theta_j^{(1)}$ 充当阈值，用来改变单元的活性。

至此，完成了从输入层到输出层的数学表达与计算推导。其中，在初始化计算时，即第一次完成从输入经过隐藏层计算后到输出的权值 $\boldsymbol{W}$ 矩阵的初始值，一般根据实际情况采用随机值或者经验值。

2. 反向传播过程求偏导

设 d 为期望输出，o 为实际输出，E 为损失函数（又称误差信号），则损失函数定义为

$$E=\frac{1}{2}(d-o)^2=\frac{1}{2}\sum_{k=1}^{l}(d_k-o_k)^2\tag{5-4}$$

其中，d_k 为输出层第 k 个单元的期望输出；o_k 是样本的第 k 个单元的实际输出。

把损失函数 E 展开到隐藏层，即把式(5-3)代入式(5-4)中，结果如下：

$$\begin{aligned}E&=\frac{1}{2}\sum_{k=1}^{l}(d_k-o_k)^2\\&=\frac{1}{2}\sum_{k=1}^{l}[d_k-f(\text{net}_k^{(1)}-\theta_k^{(1)})]^2\end{aligned}$$

$$=\frac{1}{2}\sum_{k=1}^{l}\left\{d_k-f\left[\left(\sum_{j=1}^{m}w_{kj}^{(1)}h_j+b_k^{(1)}\right)-\theta_k^{(1)}\right]\right\}^2 \tag{5-5}$$

再把式(5-2)代入式(5-5)中,即把损失函数 E 展开到输入层,公式如下:

$$\begin{aligned}E&=\frac{1}{2}\sum_{k=1}^{l}(d_k-o_k)^2\\&=\frac{1}{2}\sum_{k=1}^{l}\left[d_k-f(\mathrm{net}_k^{(1)}-\theta_k^{(1)})\right]^2\\&=\frac{1}{2}\sum_{k=1}^{l}\left\{d_k-f\left[\left(\sum_{j=1}^{m}w_{kj}^{(1)}f(\mathrm{net}_j^{(0)}-\theta_j^{(0)}\right)+b_k^{(1)})-\theta_k^{(1)}\right]\right\}^2\\&=\frac{1}{2}\sum_{k=1}^{l}\left\{d_k-f\left[\left(\sum_{j=1}^{m}w_{kj}^{(1)}f\left(\sum_{i=1}^{n}W_{ji}^{(0)}x_i+b_j^{(0)}-\theta_j^{(0)}\right)+b_k^{(1)}\right)-\theta_k^{(1)}\right]\right\}^2\end{aligned} \tag{5-6}$$

从式(5-6)可以看到,损失函数 E 是一个关于权值的函数,要使损失函数 E 最小,就要沿着梯度的反方向不断修改和调整权值矩阵 $\boldsymbol{W}$。为使 $E(w_{kj}^{(1)})$最小化,可以选择任意初始点 $w_{kj}^{(1)}$,从 $w_{kj}^{(1)}$ 出发沿着梯度下降的方向行进,可使 $E(w_{kj}^{(1)})$下降最快,所以取

$$\Delta w_{kj}^{(1)}=-\eta\frac{\partial E}{\partial w_{kj}^{(1)}},\quad j=1,2,\cdots,m;\ k=1,2,\cdots,l \tag{5-7}$$

其中,η 是一个学习效率,取值 0～1,用于避免陷入求解空间的局部最小。

同理:

$$\Delta w_{kj}^{(0)}=-\eta\frac{\partial E}{\partial w_{kj}^{(0)}},\quad i=1,2,\cdots,n;\ j=1,2,\cdots,m \tag{5-8}$$

对于输出层的 $\Delta w_{kj}^{(1)}$ 有

$$\begin{cases}\Delta w_{kj}^{(1)}=-\eta\dfrac{\partial E}{\partial w_{kj}^{(1)}}=-\eta\dfrac{\partial E}{\partial \mathrm{net}_k^{(1)}}\times\dfrac{\partial \mathrm{net}_k^{(1)}}{\partial w_{kj}^{(1)}}=-\eta\dfrac{\partial E}{\partial \mathrm{net}_k^{(1)}}\times h_j\\ \Delta b_k^{(1)}=-\eta\dfrac{\partial E}{\partial b_k^{(1)}}=-\eta\dfrac{\partial E}{\partial \mathrm{net}_k^{(1)}}\times\dfrac{\partial \mathrm{net}_k^{(1)}}{\partial b_k^{(1)}}=-\eta\dfrac{\partial E}{\partial \mathrm{net}_k^{(1)}}\end{cases} \tag{5-9}$$

对于隐藏层的 $\Delta w_{ji}^{(0)}$:

$$\begin{cases}\Delta w_{ji}^{(0)}=-\eta\dfrac{\partial E}{\partial w_{ji}^{(0)}}\\ \qquad=\displaystyle\sum_{k=1}^{l}-\eta\frac{\partial E}{\partial \mathrm{net}_k^{(1)}}\times\frac{\partial \mathrm{net}_k^{(1)}}{\partial \mathrm{net}_j^{(0)}}\times\frac{\partial \mathrm{net}_j^{(0)}}{\partial w_{ji}^{(0)}}\\ \qquad=\displaystyle\sum_{k=1}^{l}-\eta\frac{\partial E}{\partial \mathrm{net}_k^{(1)}}\times\frac{\partial \mathrm{net}_k^{(1)}}{\partial \mathrm{net}_j^{(0)}}\times x_i\\ \Delta b_j^{(0)}=-\eta\dfrac{\partial E}{\partial b_j^{(0)}}\\ \qquad=\displaystyle\sum_{k=1}^{l}-\eta\frac{\partial E}{\partial \mathrm{net}_k^{(1)}}\times\frac{\partial \mathrm{net}_k^{(1)}}{\partial \mathrm{net}_j^{(0)}}\times\frac{\partial \mathrm{net}_j^{(0)}}{\partial b_j^{(0)}}\\ \qquad=\displaystyle\sum_{k=1}^{l}-\eta\frac{\partial E}{\partial \mathrm{net}_k^{(1)}}\times\frac{\partial \mathrm{net}_k^{(1)}}{\partial b_j^{(0)}}\end{cases} \tag{5-10}$$

对于输出层和隐藏层各定义一个权值误差信号,令

$$\begin{cases}\delta_k^o = -\dfrac{\partial E}{\partial \text{net}_k^{(1)}} \\ \delta_j^y = -\sum\limits_{k=1}^{l} \dfrac{\partial E}{\partial \text{net}_k^{(1)}} \times \dfrac{\partial \text{net}_k^{(1)}}{\partial \text{net}_j^{(0)}}\end{cases} \tag{5-11}$$

则

$$\begin{cases}\Delta w_{kj} = \delta_k^{\text{o}} \eta h_j \\ \Delta v_{ji} = \delta_j^{\text{y}} \eta x_i\end{cases} \tag{5-12}$$

对于输出层和隐藏层，δ_k^{o} 和 δ_j^{y} 可以展开为

$$\begin{cases}\delta_k^{\text{o}} = -\dfrac{\partial E}{\partial \text{net}_k^{(1)}} = -\dfrac{\partial E}{\partial o_k} \times \dfrac{\partial o_k}{\partial \text{net}_k^{(1)}} = -\dfrac{\partial E}{\partial o_k} \times f'(\text{net}_k^{(1)}) \\ \delta_j^{\text{y}} = -\sum\limits_{k=1}^{l} \dfrac{\partial E}{\partial \text{net}_k^{(1)}} \times \dfrac{\partial \text{net}_k^{(1)}}{\partial \text{net}_j^{(0)}}\end{cases} \tag{5-13}$$

$$= -\sum_{k=1}^{l} \frac{\partial E}{\partial o_k} \times \frac{\partial o_k}{\partial \text{net}_k^{(1)}} \times \frac{\partial \text{net}_k^{(1)}}{\partial h_j} \times \frac{\partial h_j}{\partial \text{net}_j^{(0)}}$$

$$= -\sum_{k=1}^{l} \frac{\partial E}{\partial o_k} \times f'(\text{net}_k^{(1)}) \times \frac{\partial \text{net}_k^{(1)}}{\partial h_j} \times f'(\text{net}_j^{(0)}) \tag{5-14}$$

由此，根据式(5-3)，损失函数对 o 和 h 求偏导：

$$\begin{cases}\dfrac{\partial E}{\partial o_k} = -(d_k - o_k) \\ \dfrac{\partial E}{\partial h_j} = -\sum\limits_{k=1}^{l} (d_k - o_k) f'(\text{net}_k^{(1)}) w_{kj}^{(1)}\end{cases} \tag{5-15}$$

其中，由 Sigmoid 函数性质可知，$f'(\text{net}_k^{(1)}) = o_k(1-o_k)$，代入式(5-14)、式(5-15)可得

$$\delta_k^{\text{o}} = -\frac{\partial E}{\partial o_k} \times f'(\text{net}_k^{(1)}) = (d_k - o_k) o_k (1 - o_k) \tag{5-16}$$

$$\delta_j^{\text{y}} = -\frac{\partial E}{\partial y_j} \times f'(\text{net}_j^{(0)}) = \left(\sum_{k=1}^{l} \delta_k^{\text{o}} w_{jk}\right) y_j (1 - y_j) \tag{5-17}$$

所以，BP 算法的权值调节计算式为式(5-16)和式(5-17)。

再考虑各层的偏置设置，隐藏层的净输出为

$$\text{net}_j^{(0)} = \sum_{i=1}^{n} w_{ji}^{(0)} x_i + b_j^{(0)} \tag{5-18}$$

隐藏层偏置的更新为(Δb_j 是偏置 b_j 的改变)：

$$\begin{cases}\Delta b_j = \eta \delta_k^{\text{o}} \\ b_j = b_j + \Delta b_j\end{cases} \tag{5-19}$$

相应地，输出层的净输出为

$$\text{net}_j^{(0)} = \sum_{i=1}^{n} w_{ji}^{(0)} x_i + b_j^{(0)} \tag{5-20}$$

输出层的偏置更新为(Δb_j 是偏置 b_j 的改变)：

$$\Delta b_j = \eta \delta_j^{\text{y}}, \quad b_j = b_j + \Delta b_j \tag{5-21}$$

自此，BP 神经网络的权值矩阵和偏置值计算公式全部推导完毕。

BP 神经网络的部分实现代码如下：

```
#正向传播函数
def _feedForward(self, keep_prob):
""" Forward pass """
z = [];a = []
z.append(np.dot(self.W[0], self.X) + self.B[0])
a.append(PHI[self.ac_funcs[0]](z[-1]))
for l in range(1,len(self.layers)):
z.append(np.dot(self.W[l], a[-1]) + self.B[l])
#a.append(PHI[self.ac_funcs[l]](z[l]))
_a = PHI[self.ac_funcs[l]](z[l])
a.append( ((np.random.rand(_a.shape[0],1) < keep_prob) * _a)/keep_prob )
return z,a

#反向传播函数
def startTraining(self, epochs, alpha, _lambda, keep_prob=0.5, interval=100):
"""
Start training the neural network. It takes the followng parameters :
1. epochs :训练网络的迭代次数
2. alpha :学习率
3. lambda :L2 正则化参数或惩罚参数
4. keep_prob :丢弃正则化参数,意味着部分神经元失活
5. interval :误差和精度更新的间隔
"""
start = time.time()
for i in range(epochs+1) :
z,a = self._feedForward(keep_prob)
delta = self._cost_derivative(a[-1])
for l in range(1,len(z)) :
delta_w = np.dot(delta, a[-l-1].T) + (_lambda) * self.W[-l]
delta_b = np.sum(delta, axis=1, keepdims=True)
delta = np.dot(self.W[-l].T, delta) * PHI_PRIME[self.ac_funcs[-l-1]](z[-l-1])
self.W[-l] = self.W[-l] - (alpha/self.m) * delta_w
self.B[-l] = self.B[-l] - (alpha/self.m) * delta_b
delta_w = np.dot(delta, self.X.T ) + (_lambda) * self.W[0]
delta_b = np.sum(delta, axis=1, keepdims=True)
self.W[0] = self.W[0] - (alpha/self.m) * delta_w
self.B[0] = self.B[0] - (alpha/self.m) * delta_b
if not i%interval :
aa = self.predict(self.X)
if self.loss == 'b_ce':
aa = aa>0.5
self.acc = sum(sum(aa == self.y)) / self.m
cost_val = self._cost_func(a[-1], _lambda)
self.cost.append(cost_val)
elif self.loss == 'c_ce':
aa = np.argmax(aa, axis = 0)
yy = np.argmax(self.y, axis = 0)
self.acc = np.sum(aa==yy)/(self.m)
cost_val = self._cost_func(a[-1], _lambda)
self.cost.append(cost_val)
sys.stdout.write(f'\rEpoch[{i}] : Cost = {cost_val:.2f} ; Acc = {(self.acc*100):.2f}% ;
Time Taken = {(time.time()-start):.2f}s')
print('\n')
return None
```

上述代码中，将 epochs(迭代次数)、alpha(学习率)、_lambda、keep_prob 和 interval 作为函数的参数实现反向传播。从正向传播开始，然后将成本函数的导数计算为 delta。对于每一层计算 delta_w 和 delta_b 来讲，其中包含误差函数对网络的权重和偏差的导数。然后根据各自的公式更新 delta 权重和偏差。在将最后一层的权重和偏差更新到第二层之后，再更新第一层的权重和偏差。这样进行几次迭代，直到权重和偏差的值收敛。

BP 算法虽然是经典的深度学习算法，但对于深层网络仍然有许多不足，主要原因是 Sigmoid 激活函数易出现梯度减小甚至消失，这也是为什么深层卷积神经网络利用 ReLU 函数代替 Sigmoid 激活函数的原因。

5.2.5　激活函数

激活函数又叫激励函数，主要作用是对神经元所获得的输入进行非线性变换，以此反映神经元的非线性特性。常用的激活函数有以下几种类型。

1. 线性激活函数

$$f(x)=\mathrm{k}x+\mathrm{c} \tag{5-22}$$

其中，k 和 c 为常量。线性函数常用在线性神经网络中。

2. 符号激活函数

$$f(x)=\begin{cases}1, & x\geqslant 0\\ 0, & x<0\end{cases} \tag{5-23}$$

3. Sigmoid 激活函数

$$f(x)=\frac{1}{1+\mathrm{e}^{-x}} \tag{5-24}$$

Sigmoid 激活函数又称为 S 形函数，其曲线如图 5-8 所示，是最为常见的激活函数，它将区间$(-\infty,+\infty)$映射到(0,1)的连续区间。特别是 $f(x)$关于 x 处可导，并且有 $f(x)$的导数 $f'(x)=f(x)(1-f(x))$。

4. 双曲正切激活函数

双曲正切激活函数 tanh 的图像如图 5-9 所示。

$$f(x)=\frac{\mathrm{e}^{x}-\mathrm{e}^{-x}}{\mathrm{e}^{x}+\mathrm{e}^{-x}} \tag{5-25}$$

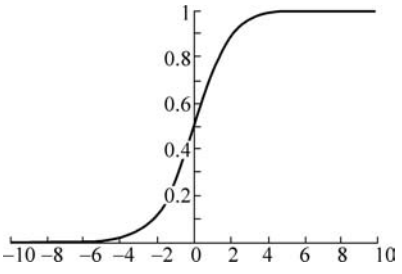


图 5-8　Sigmoid 激活函数图像

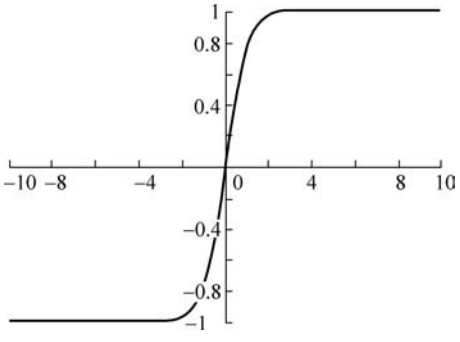


图 5-9　双曲正切激活函数 tanh 的图像

5. 高斯激活函数

$$f(x)=\mathrm{e}^{-\frac{1}{2}\left(\frac{x-\mathrm{c}}{\sigma}\right)^{2}} \tag{5-26}$$

6. ReLU 激活函数

$$f(x)=\begin{cases}x, & x>0\\ 0, & x\leqslant 0\end{cases} \tag{5-27}$$

也可表示为 $f(x)=\max(0,x)$。

在神经网络中，ReLU 激活函数得到广泛应用，如图 5-10 所示，尤其在卷积神经网络中，往往不选择 Sigmoid 或 tanh 函数而选择 ReLU 函数，原因有以下几点：

(1) 与 Sigmoid 函数必须计算指数和导数比较，ReLU 代价小，而速度更快。

(2) 对于梯度计算公式$\nabla=\sigma'\delta x$，其中σ'是 Sigmoid 的导数，在使用 BP 算法求梯度下降的时候，每经过一层 Sigmoid 神经元，都要乘以σ'，但是σ'的最大值为 1/4，所以会导致梯度越来越小，这对于训练深层网络是一个大问题，但是 ReLU 函数的导数为 1，不会出现梯度下降，以及梯度消失问题，从而更易于训练深层网络。

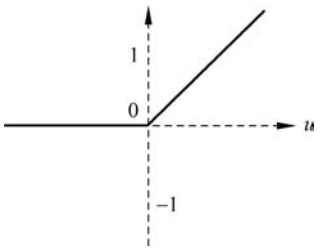


图 5-10　ReLU 函数

(3) 有研究表明，人脑在工作时只有大概 5%的神经元被激活，而 Sigmoid 函数大概有 50%的神经元被激活，而人工神经网络在理想状态时有 15%～30%的激活率，所以 ReLU 函数在小于 0 的时候是完全不激活的，所以可以适应理想网络的激活率要求。

当然，没有一种完美的激活函数，不同的网络有不同的需求函数，需要根据具体的模型选取合适的激活函数。

各个激活函数的 Python 实现代码如下：

```
def sigmoid(z) :
""" Reutrns the element wise sigmoid function. """
return 1./(1 + np.exp(-z))
def sigmoid_prime(z) :
""" Returns the derivative of the sigmoid function. """
return sigmoid(z) * (1-sigmoid(z))
def ReLU(z) :
""" Reutrns the element wise ReLU function. """
return (z * (z > 0))
def ReLU_prime(z) :
""" Returns the derivative of the ReLU function. """
return 1 * (z >= 0)
def lReLU(z) :
""" Reutrns the element wise leaky ReLU function. """
return np.maximum(z/100,z)
def lReLU_prime(z) :
""" Returns the derivative of the leaky ReLU function. """
z = 1 * (z >= 0)
z[z == 0] = 1/100
return z
def tanh(z) :
""" Reutrns the element wise hyperbolic tangent function. """
return np.tanh(z)
def tanh_prime(z) :
""" Returns the derivative of the tanh function. """
return (1-tanh(z) ** 2)
```

5.2.6　梯度下降法

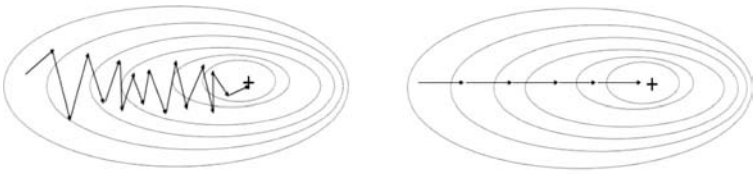
梯度下降法是神经网络模型训练中最常用的优化算法之一。梯度下降法是一种致力于找到函数极值点的算法。前面介绍过，机器学习其实就是不断改进模型参数，以便通过大量训练步骤将损失最小化。有了这个概念，将梯度下降法应用于寻找损失函数(Loss Function)或代价函数(Cost Function)的极值点，便构成依据输入数据的模型进行自我优化的学习过程。

常见的梯度下降法主要有批量梯度下降法、随机梯度下降法和小批量梯度下降法等。

设预测值为 y，要拟合的函数设为 $h_\theta(x)=\theta_0+\theta_1x_1+\theta_2x_2+\cdots+\theta_nx_n$，那么损失函数计算公式为

$$J(\theta)=\frac{1}{2}\sum_{i=1}^{m}h_\theta(x^{(i)}-y^{(i)})^2 \tag{5-28}$$

由上面的 BP 算法推导过程可知，每一次更新权值都需要遍历训练数据中的所有样本，这样的梯度算法就是批梯度下降法(BGD)。假设数据样本非常大，例如达到数以百万计，那么计算量也将非常巨大。因此，深度学习算法一般不采用常规的梯度下降法算法，而是选用随机梯度下降法(SGD)，图 5-11 展示了 SGD 和 BGD 的区别。由图 5-11(a)可以看出，在 SGD 算法中，每次更新权值 $\boldsymbol{w}$ 的迭代，只计算一个样本数据。这样对于一个具有数百万样本的训练数据而言，每完成一次遍历，就会对权值 $\boldsymbol{w}$ 更新数以百万次，这将大大提升运算效率。由于存在样本的噪声和随机性，每次更新权值 $\boldsymbol{w}$ 并不一定会按照损失函数 E 减少的方向行进。尽管算法存在一定随机性，但对于大量的权值 $\boldsymbol{w}$ 更新来说，大体上是沿着梯度减少的方向前进，所以最终也会收敛到最小值的附近。



(a) 随机梯度下降法(SGD)　　(b) 批量梯度下降法(BGD)

图 5-11　SGD 和 BGD 两种梯度下降法比较

从图 5-11(b)可以看出，BGD 算法一直向着函数最小值的最低点前进，而 SGD 明显随机了很多，但从总体上看，仍然是向最低点逼近的。

假设有 30 万个样本，对于 BGD 而言，每次迭代需要计算 30 万个样本才能对参数进行一次更新，需要求得最小值可能需要多次迭代(假设这里是 10)。而对于 SGD，每次更新参数只需一个样本，因此若使用这 30 万个样本进行参数更新，则参数会被更新(迭代)30 万次，而这期间，SGD 保证能够收敛到一个合适的最小值上了。也就是说，在收敛时，BGD 计算了 10×30 万次，而 SGD 只计算了 1×30 万次。

有时候随机性并非坏事，因此 SGD 效率有时候会比较高。如果我们研究的目标函数是一个凸函数，沿着梯度反方向总能找到全局唯一的最小值。但是对于非凸函数来说，存在许多局部最小值。SGD 算法的随机性有助于逃离某些不理想的局部最小值，从而获得一个更好的网络架构模型。

5.2.7 交叉熵损失函数

在神经网络中较常采用交叉熵(Binary Cross Entropy)作为损失函数,设 p 表示真实标记的分布,q 则为训练后的模型的预测标记分布,交叉熵损失函数可以衡量 p 与 q 的相似性。交叉熵作为损失函数还有一个好处,使用 Sigmoid 函数在梯度下降时能避免均方误差损失函数学习速率降低的问题,因为学习速率可以被输出的误差所控制。公式如下:

$$C=\frac{1}{n}\sum_{i=1}^{n}[y_i\ln\sigma(\boldsymbol{w}^{\mathrm{T}}x_i)+(1-y_i)\ln(1-\sigma(\boldsymbol{w}^{\mathrm{T}}x_i))] \tag{5-29}$$

交叉熵损失函数的 Python 实现代码如下:

```
def binary_crossentropy(t,o):
    return -(t*tf.log(o+eps) + (1.0-t)*tf.log(1.0-o+eps))
```

5.2.8 过拟合与欠拟合

机器学习中的一个重要问题便是模型的泛化能力(Generalization Ability),即模型对新鲜样本的适应能力。泛化能力强的模型才是好模型。当模型在训练数据上表现良好但对不可见数据的泛化能力很差时,可能发生过拟合(Overfitting)。若在训练集表现差,则在测试集表现同样会很差,这可能是由欠拟合(Underfitting)导致的,如图 5-12 所示。

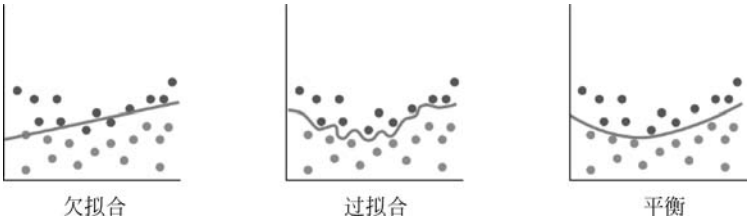


图 5-12 机器学习的过拟合与欠拟合状态

过拟合和欠拟合是所有机器学习算法都要考虑的问题,其中欠拟合的情况比较容易克服,常见解决方法有以下几种。

(1) 增加新特征,可以考虑加入特征组合、高次特征,以此增大假设空间。

(2) 添加多项式特征,这个在机器学习算法里用得很普遍,例如将线性模型通过添加二次项或者三次项使模型泛化能力更强。

(3) 减少正则化参数,正则化的目的是用来防止过拟合,但是模型出现了欠拟合,则需要减少正则化参数。

(4) 使用非线性模型,例如支持向量机、决策树、深度学习等模型。

(5) 调整模型的容量(Capacity),通俗地讲,模型的容量是指其拟合各种函数的能力。

(6) 容量低的模型可能很难拟合训练集;使用集成学习方法,如使用 Bagging,可将多个弱学习器 Bagging。

过拟合常见的解决办法有以下几种。

(1) 在神经网络模型中,可使用权值衰减的方法,即每次迭代过程中以某个小因子降低每个权值。

(2) 选取合适的停止训练标准,使对机器的训练在合适的程度。

(3) 保留验证数据集,对训练成果进行验证。

(4) 获取额外数据进行交叉验证。

(5) 正则化,即在进行目标函数或代价函数优化时,在目标函数或代价函数后面加上一个正则项,一般有 L1 正则与 L2 正则等。

5.3　卷积神经网络

5.3.1　卷积神经网络简介

卷积神经网络(Convolutional Neural Network,CNN)是一类包含卷积计算且具有深度结构的前馈神经网络(Feedforward Neural Network),是深度学习的代表算法之一。卷积神经网络具有表征学习(Representation Learning)能力,能够按其阶层结构对输入信息进行平移不变分类(Shift-invariant Classification),因此也被称为平移不变人工神经网络(Shift-Invariant Artificial Neural Network,SIANN)。

卷积神经网络仿照生物的视知觉(Visual Perception)机制构建,可以进行监督学习和非监督学习,其隐含层内的卷积核参数共享和层间连接的稀疏性使得卷积神经网络能够以较小的计算量对格点化(Grid-like Topology)特征(如像素和声频)进行学习。有稳定的效果且对数据没有额外的特征工程(Feature Engineering)要求。

5.3.2　卷积神经网络结构

典型的卷积神经网络结构主要分为输入层、卷积层、池化层、全连接层、分类层等,如图 5-13 所示。

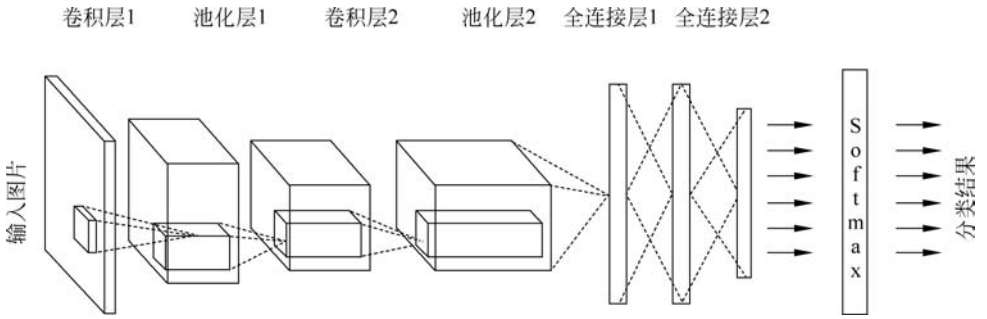


图 5-13　卷积神经网络结构

(1) 输入层(Input Layer)。输入层是整个神经网络的输入,在处理图像的卷积神经网络中,它一般代表了一张图片的像素矩阵。其中三维矩阵的长和宽代表了图像的大小,深度代表了图像的色彩通道(Channel)。例如黑白图的深度为 1,而在 RGB 色彩模式下,图像的深度为 3。从输入层开始,卷积神经网络通过不同的神经网络架构将上一层的三维矩阵转换为下一层的三维矩阵,直到最后的全连接层。

(2) 卷积层(Convolution Layer)。卷积层是一个网络最重要的部分,卷积层试图将神经网络中的每小块进行更加深入地分析从而获得抽象度更高的特征。一般来说,通过卷积

层处理过的节点矩阵会变得更深。

(3) 池化层(Pooling Layer)。池化层神经网络不会改变三维矩阵的深度,但是它可以缩小矩阵的大小。通过池化层可以进一步缩小最后全连接层中节点的个数,从而达到减小整个神经网络参数的目的。

(4) 全连接层(Full Connection Layer)。在经过多轮卷积和池化之后,在卷积神经网络的最后一般会有 1～2 个全连接层给出最后的分类结果。经过几轮卷积和池化之后,可以认定图像中的信息已经被抽象成信息含量更高的特征。我们可以将卷积层和池化层看作自动图像特征提取的过程,在特征提取之后,仍要用全连接层来完成分类问题。

(5) Softmax 层。Softmax 层主要用于分类问题,通过 Softmax 层可以得到当前输出属于不同种类的概率分布情况。该层主要采用 Softmax 函数,又称归一化指数函数,是对数概率回归在 C 个不同值上的推广,公式如下:

$$\text{Softmax}(i)=\frac{e^{-O_i}}{\sum_{j=1}^{C}e^{-O_j}},\quad i=1,2,\cdots,C-1 \tag{5-30}$$

其中,C 表示神经网络输出层的输出数量;i 表示输出层第 i 个输出;O_i 表示第 i 个输出值;e 表示自然常数;$\sum_{j=1}^{C}e^{-O_j}$ 表示所有神经元输出值的对数概率函数之和。

Softmax 函数的 Python 实现代码如下:

```
def softmax(x):
    exp_x = np.exp(x)
    return exp_x / np.sum(exp_x)
```

5.3.3 卷积神经网络计算

1. 卷积层计算

卷积层神经网络结构中最重要的部分就是过滤器(Filter)或者叫作内核(Kernel),图 5-14 显示了这一结构。过滤器可以将当前神经网络的一个子节点矩阵转化为下一层神经网络的一个单位节点矩阵。单位节点矩阵就是长和宽都是 1,但深度不限的节点矩阵。

在一个卷积层中,过滤器所处理的节点矩阵的长和宽都是人为设定的,这个节点矩阵的尺寸也被称为过滤器的尺寸。因为过滤器处理的矩阵深度和当前神经网络节点矩阵的深度是相同的,所以尽管过滤器的节点矩阵是三维的,但是只需给出二维矩阵即可。另外一个需要人为设定的是过滤器的深度,也即输出单位节点矩阵的深度。如图 5-14 所示,左侧小矩阵的尺寸为过滤器的尺寸,而右侧单位矩阵的深度为滤波器的深度。

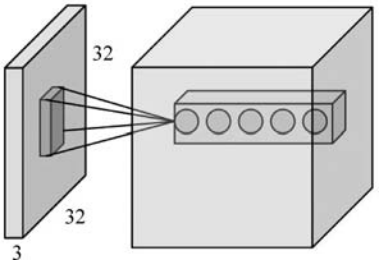


图 5-14 卷积层过滤器结构示意图

为了清楚地解释卷积神经网络的卷积计算,以一张简单的 5×5 图片作为输入,过滤器选取 3×3,步长为 1,得到一个 3×3 的特征图(Feature Map),用 $x_{i,j}$ 表示输入图像的第 i 行第 j 列的像素。用 $w_{m,n}$ 表示过滤器的第 m 行第 n 列的值,w_b 表示权值偏置(Bias),用

$a_{i,j}$ 表示特征图的第 i 行第 j 列的值。激活函数 f 选取 ReLU 函数，则卷积操作可以由下面的公式计算：

$$a_{i,j}=f\left(\sum_{m=0}^{2}\sum_{n=0}^{2}w_{m,n}x_{i+m,j+n}+w_b\right) \tag{5-31}$$

例如，对于图 5-15，特征图左上角的 $a_{0,0}$ 来说，其计算方法为

$$\begin{aligned}a_{0,0}&=f\left(\sum_{m=0}^{2}\sum_{n=0}^{2}w_{m,n}x_{m+0,n+0}+w_b\right)\\&=\text{relu}(w_{0,0}x_{0,0}+w_{0,1}x_{0,1}+w_{0,2}x_{0,2}+w_{1,0}x_{1,0}+w_{1,1}x_{1,1}+\\&\quad w_{1,2}x_{1,2}+w_{2,0}x_{2,0}+w_{2,1}x_{2,1}+w_{2,2}x_{2,2})\\&=\text{relu}(1+0+1+0+1+0+0+0+1+0)\\&=\text{relu}(4)\\&=4\end{aligned}$$

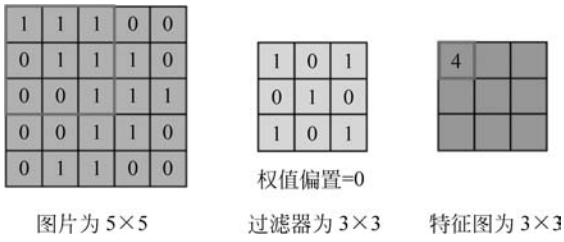


图 5-15　简单图片卷积计算示意图

然后，依次计算出所有值。上例中步长(Stride)为 1，当步长变为 2 时，特征图的尺寸便变成了 2×2，这是由式(5-32)决定的。

$$W_2=(W_1-F+2P)/S+1 \tag{5-32}$$

式中，W_2 表示输出特征图的宽，W_1 表示输入图像的宽，F 表示过滤器的宽，P 是填充零的圈数，S 是步幅。长和宽等价，所以图像的长也可以用式(5-32)计算。

以上就是卷积层卷积的计算，体现了卷积神经网络的局部连接和权值共享特性，通过卷积操作，参数的数量大幅降低。

2. 池化层计算

池化层可以有效地缩小矩阵的尺寸，从而减少最后全连接层的数量。使用池化层既可以加快计算速度也可以有效地防止过拟合问题。

池化的方式很多，最常用的池化方式是最大池化(Max Pooling)和平均池化(Average Pooling)。与卷积层的过滤器类似，池化层的过滤器也需要设置尺寸，唯一不同的是池化层的过滤器只影响一个深度上的节点，即主要减小矩阵的长和宽，不减少矩阵的深度，虽然池化层可以减少矩阵的深度，但是在实际应用中不会这样使用。图 5-16 展示了一个最大池化计算过程的例子。

从图 5-16 可以清晰地看出，池化层只减小了矩阵的长和宽，并未减少矩阵的深度。

3. 全连接层计算

几个卷积和池化层之后通常有一个或多个全连接层(Fully Connected Layers，FCL)，

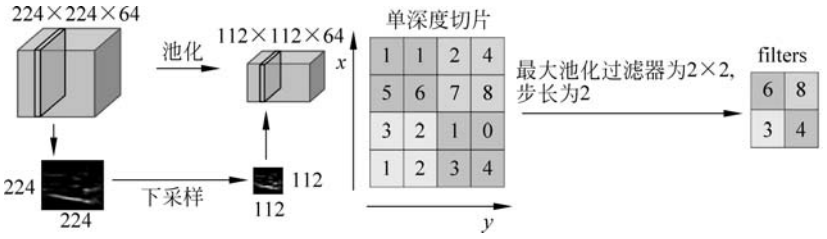


图 5-16　最大池化的计算过程示意图

旨在执行对原始图像的高级抽象。它们将前一层所有的神经元与当前层的每个神经元相连接，即与标准神经网络各层之间的连接相同，在全连接层不保存空间信息。全连接层在整个卷积神经网络中起到“分类器”的作用。如果说卷积层、池化层和激活函数层等操作是将原始数据映射到隐层特征空间，则全连接层起到将学到的“分布式特征表示”映射到样本标记空间的作用。下面举例介绍一下全连接层的计算过程，如图 5-17 所示。

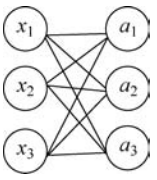


图 5-17　全连接层的计算过程

其中，x_1、x_2、x_3 为全连接层的输入，a_1、a_2、a_3 为输出，则有

$$\begin{cases} a_1 = W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1 \\ a_2 = W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2 \\ a_3 = W_{31}x_1 + W_{32}x_2 + W_{33}x_3 + b_3 \end{cases} \tag{5-33}$$

式(5-33)可以写成矩阵形式：

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ W_{31} & W_{32} & W_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{5-34}$$

以第一个全连接层为例，该层有 50×4×4=800 个输入节点和 500 个输出节点。由于需要对 $\boldsymbol{W}$ 和 b 进行更新，即全连接层的反向传播，还要向前传递梯度，需要计算以下三个偏导数。

1) 对上一层的输出(相当于当前层的输入)求导

若已知传递到该层的梯度 $\frac{\partial \text{loss}}{\partial a}$，则可以通过链式法则求得 loss 对 x 的偏导数。

首先需要求得该层的输出 a_i 对输入 x_j 的偏导数：

$$\frac{\partial a_i}{\partial x_j} = \frac{\sum_{j=1}^{800} w_{ij}x_j}{\partial x_j} = w_{ij} \tag{5-35}$$

再通过链式法则求得 loss 对 x 的偏导数：

$$\frac{\partial \text{loss}}{\partial x_k} = \sum_{j=1}^{500} \frac{\partial \text{loss}}{\partial a_j} \frac{\partial a_j}{\partial x_k} = \sum_{j=1}^{500} \frac{\partial \text{loss}}{\partial a_j} w_{jk} \tag{5-36}$$

在反向传播过程中，若第 x 层的 a 节点通过权值 $\boldsymbol{W}$ 对 $x+1$ 层的 b 节点有贡献，则在反向传播过程中，梯度通过权值 $\boldsymbol{W}$ 从 b 节点传播回 a 节点。例如，如果需要一次训练 16 张图片，即 batch_size=16，则我们可以把计算转化为矩阵形式，如图 5-18 所示。

2) 对权重系数 $\boldsymbol{W}$ 求导

由图 5-18 可知 $\frac{\partial a_i}{\partial w_{ij}} = x_j$，根据计算式(5-35)，有

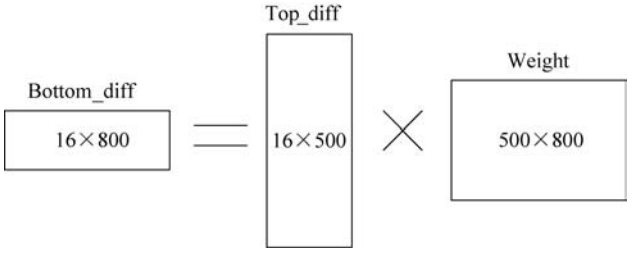


图 5-18　反向传播计算转换示意图

$$\frac{\partial \text{loss}}{\partial w_{kj}}=\frac{\partial \text{loss}}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}}=\frac{\partial \text{loss}}{\partial a_k}\times x_j$$

当 batch_size=16 时，写成矩阵形式，如图 5-19 所示。

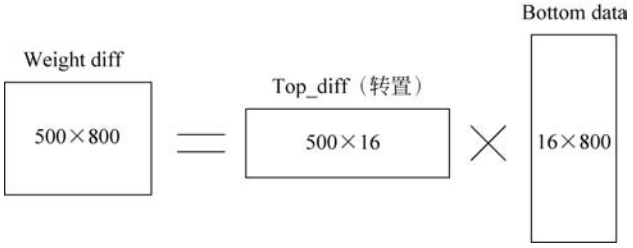


图 5-19　权重求导计算示意图

3）对偏置系数 b 求导

由推导公式可知：$\frac{\partial a_i}{\partial b_i}=1$，即 loss 对偏置系数的偏导数等于对上一层输出的偏导数。当 batch_size=16 时，将不同 batch 对应的相同 b 的偏导相加即可，写成矩阵形式即为乘以一个全 1 的矩阵，如图 5-20 所示。

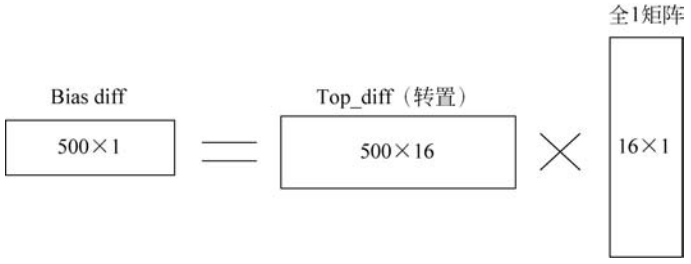


图 5-20　偏置求导计算示意图

由于全连接层参数冗余（仅全连接层参数就可占整个网络参数 80%左右），近期一些性能优异的网络模型如 ResNet 和 GoogLeNet 等采用全局平均池化（Global Average Pooling，GAP）取代 FC 来融合学到的深度特征，最后仍用 Softmax 等损失函数作为网络目标函数来指导学习过程。需要指出的是：用 GAP 替代 FC 的网络通常有较好的预测性能。对于分类任务，Softmax 回归由于其可以生成输出的 Well-formed 概率分布而被普遍使用。给定训练集$\{x(i),y(i);i\in 1,2,\cdots,N;y(i)\in 0,1,\cdots,K-1\}$其中 $x(i)$是第 i 个输入图像块，$y(i)$是它的类标签，第 i 个输入属于第 j 类的预测值 $a_j^{(i)}$ 可以用 Softmax 函数转换：

$p_j^i = \frac{e^{a_j^{(i)}}}{\sum_{k=0}^{K-1} e^{a_k^{(i)}}}$，Softmax 将预测转换为非负值，并进行正则化处理。

5.3.4 典型卷积神经网络

1. AlexNet

该网络 2012 年由 Hinton 学生 Alex 提出，是 Lenet 加宽版，包含 65 万神经元，5 个卷积层，3 个后面带有池化层，最后用了 3 个全连接，如图 5-21 所示。Alex 采用了一系列新技术：首次采用 GPU 硬件加速；成功使用 ReLU 作为 CNN 的激活函数，并验证其效果在较深的网络超过了 Sigmoid，成功解决了 Sigmoid 在网络较深时的梯度弥散问题。训练时使用 Dropout 随机忽略一部分神经元，以避免模型过拟合；使用最大池化，避免平均池化的模糊化效果，让步长比池化核的尺寸小，这样池化层的输出之间会有重叠和覆盖，提升了特征的丰富性；使用 Dropout 有效地防止神经网络的过拟合等。

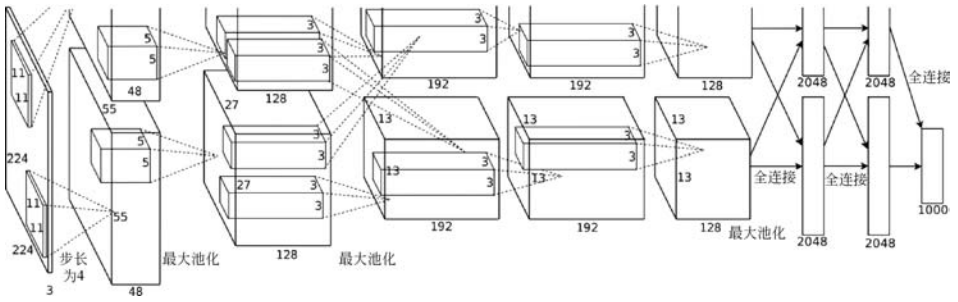


图 5-21 AlexNet 结构

2. VGGNet

VGGNet 由牛津大学的视觉几何组(Visual Geometry Group)和 Google DeepMind 公司的研究员一起研发的深度卷积神经网络，在 ILSVRC 2014 上取得了第二名的成绩，将 Top-5 错误率降到 7.3%。它主要的贡献是展示出网络的深度(depth)是算法优良性能的关键部分。目前使用比较多的网络结构主要有 ResNet(152～1000 层)、GoogLeNet(22 层)、VGGNet(16 层)、VGGNet(19 层)等，如图 5-21 所示。大多数模型基于这几个模型进行改进，采用新的优化算法，以及多模型融合等。到目前为止，VGGNet 依然经常被用来提取图像特征。图 5-22 为 VGGNet-16 的网络结构图。

以下是 VGGNet-16 各层的处理过程：

(1) 输入 224×224×3 的图片，经 64 个 3×3 的卷积核做两次卷积＋ReLU，卷积后的尺寸变为 224×224×64。

(2) 做最大池化，池化单元尺寸为 2×2，效果为图像尺寸减半，池化后的尺寸变为 112×112×64。

(3) 经 128 个 3×3 的卷积核做两次卷积＋ReLU，尺寸变为 112×112×128。

(4) 做 2×2 的最大池化，尺寸变为 56×56×128。

(5) 经 256 个 3×3 的卷积核做 3 次卷积＋ReLU，尺寸变为 56×56×256。

(6) 做 2×2 的最大池化，尺寸变为 28×28×256。

(7) 经 512 个 3×3 的卷积核做 3 次卷积＋ReLU，尺寸变为 28×28×512。

(8) 做 2×2 的最大池化,尺寸变为 14×14×512。

(9) 经 512 个 3×3 的卷积核作 3 次卷积+ReLU,尺寸变为 14×14×512。

(10) 做 2×2 的最大池化,尺寸变为 7×7×512。

(11) 与两层 1×1×4096 和一层 1×1×1000 进行全连接+ReLU(共 3 层)。

(12) 通过 Softmax 输出 1000 个预测结果。

从上面的过程可以看出 VGGNet-16 网络结构比较简洁,都是由小卷积核、小池化核、ReLU 组合而成,简化图如图 5-23 所示。

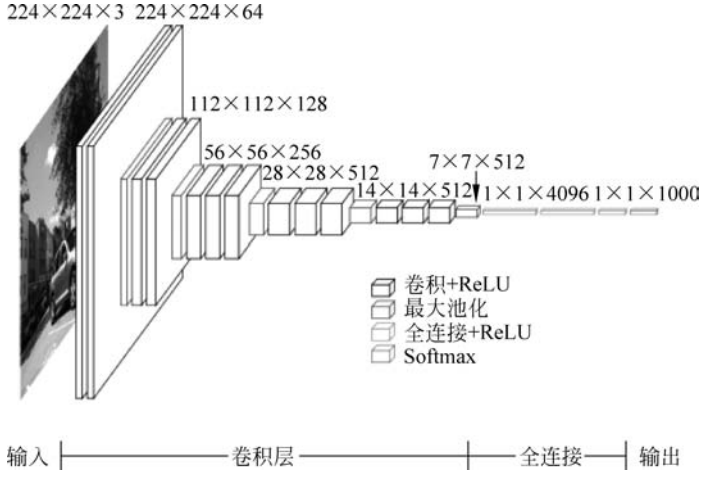


图 5-22　VGGNet-16 网络结构

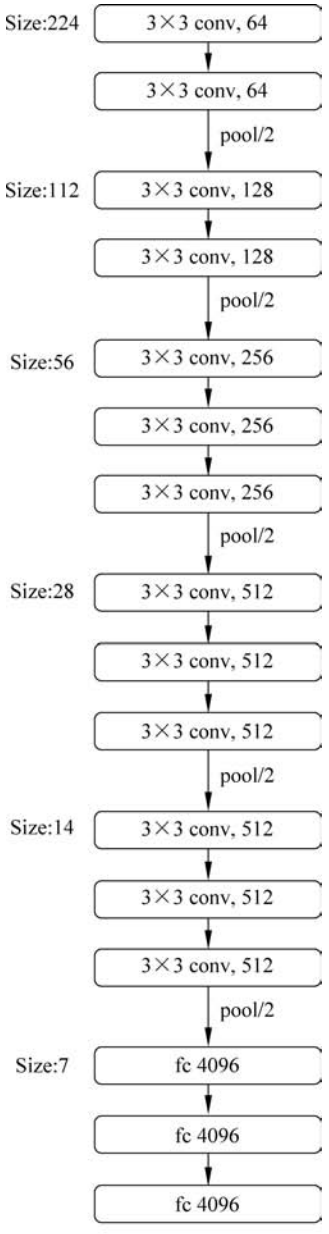


图 5-23　VGGNet-16 的计算过程

5.4 循环神经网络

5.4.1 循环神经网络简介

在5.3节中介绍了卷积神经网络模型，网络结构都是从输入层到隐含层再到输出层，层与层之间是全连接或部分连接的，但每层之间的节点没有连接。考虑这样一个问题，如果要预测一个句子的下一个单词是什么，一般需要用到当前单词及前面的单词，因为句子中前后单词并不是独立的。例如，当前单词是“很”，前一个单词是“天空”，那么下一个单词很大概率是“蓝”。这样的应用CNN并不擅长。由此出现了可以用来处理和预测序列数据的循环神经网络(RNN)。

RNN可以刻画一个序列当前的输出与之前信息的关系，通过不停地将信息循环操作，保证信息的持续存在。从网络结构上，RNN会记忆之前的信息，并利用之前的信息影响后面节点的输出。也就是说，RNN隐藏层之间的节点是有连接的，隐藏层的输入不仅包括输入层的输出，还包括上一时刻隐藏层的输出。RNN的处理方式类似人们对一个问题的思考不会完全从头开始一样，例如在阅读文章的时候，人们会根据之前理解过的信息来理解下面看到的文字。在理解当前文字的时候，人们不会忘记之前看过的文字，从头思考当前文字的含义。传统的神经网络并不能做到这一点，这是在对这种序列信息(如语音)进行预测时的一个缺点。例如对电影中的每个片段进行事件分类，传统的神经网络很难利用前面的事件信息来对后面事件进行分类。而循环神经网络则可以通过连接方式将信息关联起来，从而解决上述问题。随着更加有效的循环神经网络结构被不断提出，循环神经网络挖掘数据中的时序信息及语义信息的深度表达能力被充分利用，并在语音识别、语言模型、机器翻译及时序分析等方面实现了突破。

5.4.2 循环神经网络结构

图5-24展示了一个典型的循环神经网络结构图。循环神经网络的主体结构A的输入除了来自输入层 x_t，还有一个循环的边提供上一时刻的隐藏状态 h_{t-1}。在每一时刻，循环神经网络的模块A在读取了 x_t 和 h_{t-1} 之后，会生成新的隐藏状态 h_t，并产生本时刻的输出 o_t。循环神经网络当前的状态 h_t 是根据上一时刻的状态 h_{t-1} 和当前的输入 x_t 共同来决定的。在时刻 t，状态 h_t 浓缩了前面序列 $x_0, x_1, \cdots, x_{t-1}$ 的信息，用于作为输出 o_t 的参考。

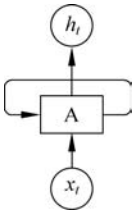


图5-24 循环神经网络结构示意图

由于序列长度可以无限长，维度有限的 h 状态不可能将序列的全部信息都保存下来，因此模型学习只保留与后面任务 o_t，o_{t+1}，…相关的最重要信息。因此，RNN会记忆之前的信息，并利用之前的信息影响后面节点的输出。RNN可以被看作同一神经网络结构被无限复制的结果。正如卷积神经网络在不同的空间位置共享参数，循环神经网络是在不同时间位置共享参数，从而能够使用有限的参数处理任意长度的序列。

如果把图5-24中循环神经网络按时间序列展开，可以得到

图 5-25 所示的结果。

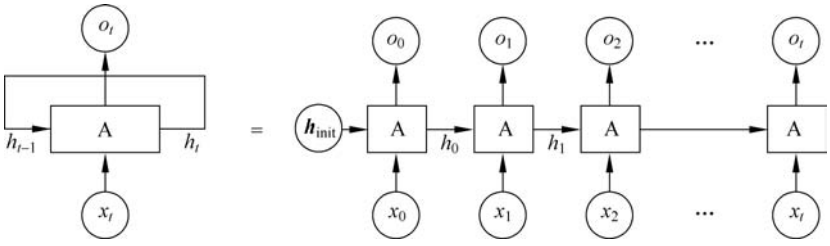


图 5-25　循环神经网络按时间展开后的结构

在图 5-25 中可以更加清楚地看出，RNN 在每一个时刻会有一个输入 x_t，然后根据 RNN 前一时刻的状态 h_{t-1} 计算新的状态 h_t，并输出 O_t。RNN 当前的状态 h_t，是根据上一时刻的状态 h_{t-1} 和当前的输入 x_t 共同决定的。在时刻 t，状态 h_{t-1} 浓缩了前面序列 $x_0, x_1, \cdots, x_{t-1}$ 的信息，用于作为输出 o_t 的参考。由于序列的长度可以无限延长，维度有限的 h 状态不可能将序列的全部信息都保存下来，因此模型必须学习只保留与后面任务 $o_t, o_{t+1}, \cdots$ 相关的最重要的信息。

循环网络的展开在模型训练中有重要意义。从图 5-25 可以看到，RNN 对长度为 N 的序列展开之后，可以视为一个有 N 个中间层的前馈神经网络。这个前馈神经网路没有循环连接，因此可以直接使用反向传播算法进行训练，而不需要任何特别的优化算法。这样的训练方法称为沿时间反向传播(Back-Propagation Through Time)，是训练循环神经网络最常见的方法。

5.4.3　循环神经网络计算

从 RNN 的结构特征可以很容易看出它最擅长解决与时间序列相关的问题。RNN 是处理这类问题时最自然的神经网络结构。对于一个序列数据，可以将这个序列上不同时刻的数据依次传入 RNN 的输入层，而输出可以是对序列中下一个时刻的预测，也可以是对当前时刻信息的处理结果，例如语音识别结果。循环神经网络要求每一个时刻都有一个输入，但是不一定每个时刻都需要输出。

下面以机器翻译为例来介绍 RNN 是如何解决实际问题的。RNN 中每一个时刻的输入为需要翻译的句子中的单词。如图 5-26 所示，如果需要翻译的句子为 ABCD，那么 RNN 第一段每一个时刻的输入就分别是 A、B、C 和 D，然后用“_”作为待翻译句子的结束符。在第一段中，循环神经网络没有输出。从结束符“_”开始，循环神经网络进入翻译阶段。该阶段中每一个时刻的输入是上一个时刻的输出，图 5-26 中虚线所示，而最终得到的输出就是句子 ABCD 翻译的结果。从图中可以看到句子 ABCD 对应的翻译结果就是 XYZ，当网络输出“_”时表示翻译结束。

RNN 可以看作同一神经网络结构(输入层→隐藏层→输出层)在时间序列上被复制多次的结果，这个被复制多次的结构被称为循环体。如何设计循环体的网络结构是循环神经网络解决实际问题的关键。图 5-27 展示了一个最简单的循环体结构，这个循环体中只使用了一个类似全连接层的神经网络结构，图中 tanh 小方框表示一个使用 tanh 作为激活函数的全连接层。下面通过图 5-27 中所示的神经网络来介绍 RNN 前向传播的完整流程。

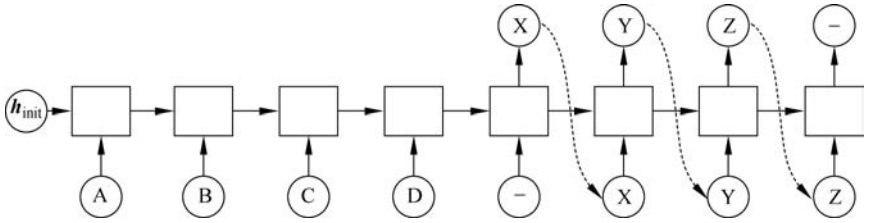


图 5-26　循环神经网络计算示意图

RNN 中的状态是通过一个向量来表示的，这个向量的维度也称为 RNN 隐藏层的大小，假设其为 n。从图 5-27 可以看出，循环体中的神经网络的输入有两部分，一部分为上一时刻的状态，另一部分为当前时刻的输入样本。对于时间序列数据来说（例如不同时刻商品的销量），每一时刻的输入样例可以是当前时刻的数值（例如销量值）；对于语言模型来说，输入样例可以是当前单词对应的单词向量（Word Embedding）。

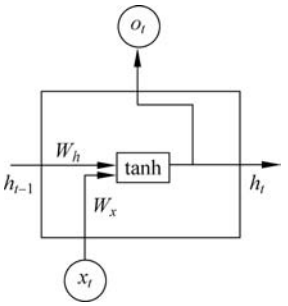


图 5-27　循环神经网络基本循环结构体

假设输入向量的维度为 x，隐藏状态的维度为 n，那么图 5-27 循环体的全连接层神经网络的输入大小为 $n+x$。也就是将上一时刻的状态与当前时刻的输入拼接成一个大的向量作为循环体中神经网络的输入。因为该全连接层的输出为当前时刻的状态，于是输出层的节点个数也为 n，循环体中的参数个数为 $(n+x)\times n+n$ 个。从图中可以看到，循环体中的神经网络输出不但提供给下一时刻作为状态，同时也会提供给当前时刻的输出。注意到循环体状态与最终输出的维度通常不同，因此为了将当前时刻的状态转化为最终的输出，RNN 还需要另外一个全连接神经网络来完成这个过程。这和 CNN 中最后的全连接层的意义是一样的。类似地，不同时刻用于输出的全连接神经网络中的参数也是一致的。

下面举例展示一个 RNN 前向传播的具体计算过程，如图 5-28 所示。

假设状态的维度为 2，输入、输出的维度都为 1，而且循环体中全连接层中的权值为

$$\boldsymbol{w}_{\text{rnn}}=\begin{bmatrix}0.1 & 0.2\\ 0.3 & 0.4\\ 0.5 & 0.6\end{bmatrix}$$

偏置项的大小为 $\boldsymbol{b}_{\text{output}}=[0.1,-0.1]$，用于输出的全连接层权值为

$$\boldsymbol{w}_{\text{output}}=\begin{bmatrix}1.0\\ 2.0\end{bmatrix}$$

偏置项的大小为 $b_{\text{output}}=0.1$。那么在时刻 t_0，因为没有上一时刻，所以将状态初始化为 $\boldsymbol{h}_{\text{init}}=[0,0]$，而当前的输入为 1，所以拼接得到的向量为 $[0,0,1]$，通过循环体中的全连接层神经网络得到的结果为

$$\tanh\left([0,0,1]\times\begin{bmatrix}0.1 & 0.2\\ 0.3 & 0.4\\ 0.5 & 0.6\end{bmatrix}+[0.1,-0.1]\right)=\tanh([0.6,0.5])=[0.537,0.462]$$

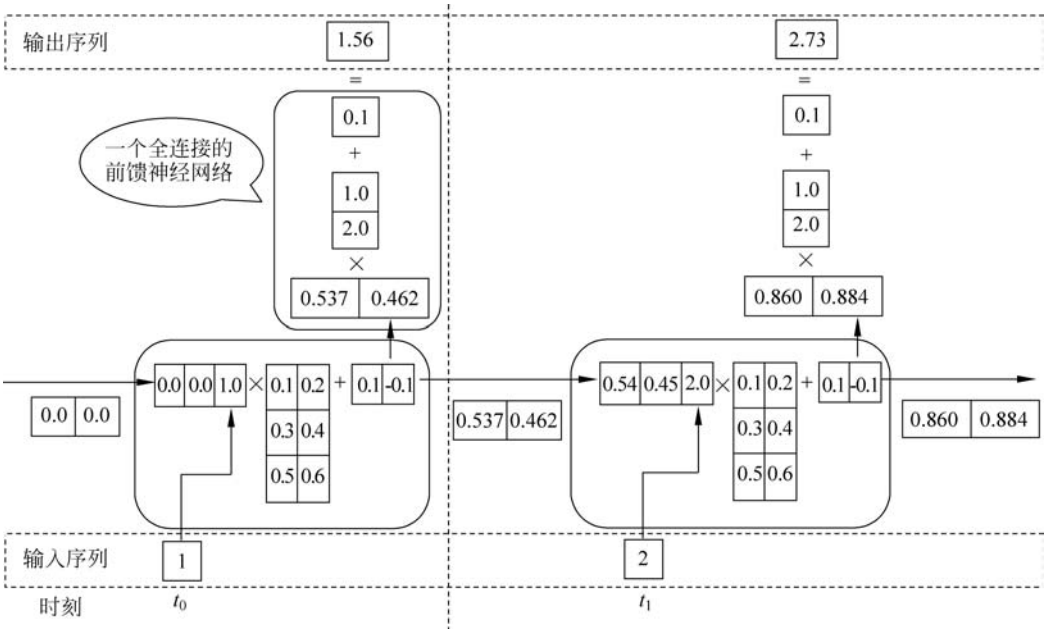


图 5-28　循环神经网络计算举例

使用 t_0 时刻的状态可以类似地推导出 t_1 时刻的状态为[0.860,0.884],而 t_1 时刻的输出为 2.73。在得到 RNN 的前向传播结果之后,可以和其他神经网络类似地定义损失函数。RNN 与其他网络唯一的区别在于它每个时刻都有一个输出,所以 RNN 的总损失为所有时刻或者部分时刻上的损失函数的总和。

下面实现这个简单循环神经网络前向传播的过程,代码如下:

```
import numpy as np

X = [1,2]
state = [0.0, 0.0]
#分开定义不同输入部分的权重以方便操作
w_cell_state = np.asarray([[0.1, 0.2], [0.3, 0.4]])
w_cell_input = np.asarray([0.5, 0.6])
b_cell = np.asarray([0.1, -0.1])

#定义用于输出的全连接层参数
w_output = np.asarray([[1.0], [2.0]])
b_output = 0.1

#执行前向传播过程
for i in range(len(X)):
    before_activation = np.dot(state, w_cell_state) + X[i] * w_cell_input + b_cell
    state = np.tanh(before_activation)
    final_output = np.dot(state, w_output) + b_output
    print("before activation: ", before_activation)
    print("state: ", state)
    print("output: ", final_output)
```

输出结果如下：

```
before activation:  [0.6 0.5]
state:  [0.53704957 0.46211716]
output:  [1.56128388]
before activation:  [1.2923401  1.39225678]
state:  [0.85973818 0.88366641]
output:  [2.72707101]
```

5.5 长短时记忆网络

5.5.1 长短时记忆网络简介

5.4 节中介绍的循环神经网络可以用来连接前面的信息到当前的任务上，例如使用过去的视频段来推测对当前段的理解。但是，当预测位置和相关信息之间的文本间隔不断增大时，循环神经网络有可能会丧失学习到距离非常远的信息的能力，另外在复杂语言场景中，有用信息的间隔有大有小、长短不一，循环神经网络的性能也会受到影响。假设我们用RNN 构建了一个语言模型，可以用来基于先前的词预测下一个词，如预测 the clouds are in the sky 最后的词，这个时候其实并不需要任何其他的上下文，因为下一个词很显然就应该是 sky。在这样的场景中，相关的信息和预测的词位置之间的间隔是非常小的，RNN 可以学会使用先前的信息。但是同样也会有一些更加复杂的语言模型场景，如我们试着去预测 I grew up in China… I speak fluent Chinese 最后的词。当前的信息建议下一个词可能是一种语言的名字，但是如果我们需要弄清楚是什么语言，我们就需要离当前位置很远的 Chinese 的上下文信息。这说明相关信息和当前预测位置之间的间隔变得相当大。理论上，RNN 可以处理这样的长期依赖问题，但在实践中 RNN 不能成功学习到这些知识。随着间隔不断增大，RNN 会彻底丧失学习到连接远信息的能力。

由此，人们提出了长短时记忆网络（Long Short-Term Memory，LSTM），本质上，LSTM 是一种时间循环神经网络。同循环神经网络一样，LSTM 也具有 RNN 重复模块链的形式，如图 5-29 所示。假设输入的句子为 I am from China. I am fluent in ___。为了正确预测出下一个单词 Chinese，LSTM 会更加关注上一句中的 China 并且利用神经元（cell）对其进行记忆。在处理序列时神经元会对获取的信息进行存储，这些信息会用于预测下一个单词。LSTM 网络节点中会单独设计一个“遗忘门”，当遇到句号时，遗忘门会意识到句子中的上下文发生了改变，并忽略当前神经元中存储的状态信息。换句话说，遗忘门的作用是让循环神经网络“遗忘”之前没有用到的信息。

由于独特的设计结构，LSTM 适合于处理和预测时间序列中间隔和延迟非常长的重要事件，它不仅能够解决循环神经网络 RNN 存在的长期依赖问题，还能够解决神经网络中常见的梯度爆炸或梯度消失等问题。作为非线性模型，LSTM 可作为复杂的非线性单元用于构造更大型深度神经网络。

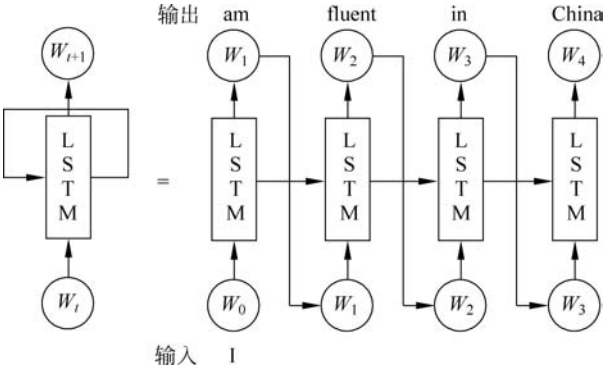


图 5-29　LSTM 链式结构图

5.5.2　长短时记忆网络结构

LSTM 链式结构中每一个计算节点实际上是一种拥有 3 个“门”结构的特殊网络结构，如图 5-30 所示。LSTM 靠一些“门”的结构让信息有选择性地影响循环神经网络中每个时刻的状态。所谓“门”结构就是一个使用 Sigmoid 神经网络和一个按位做乘法计算的操作，这两个操作合在一起就是一个“门”结构。使用 Sigmoid 作为激活函数的全连接神经网络层会输出一个 0～1 之间的数值，描述当前输入有多少信息量可以通过这个结构。当门打开时，如果 Sigmoid 神经网络层输出为 1，则全部信息都可以通过。当门关上时，如果 Sigmoid 神经网络层输出为 0，则任何信息都无法通过。

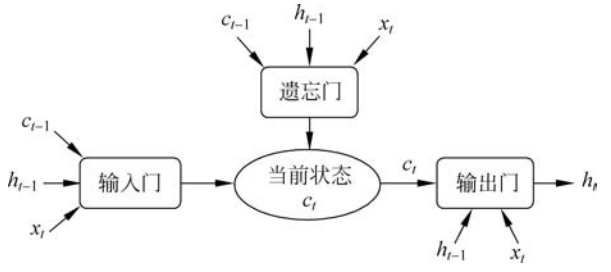


图 5-30　LSTM 单元结构示意图

图 5-30 中，“遗忘门”和“输入门”可以使神经网络更有效地保存需长期记忆的信息。“遗忘门”的作用是让循环神经网络忘记之前没有用的信息。“遗忘门”会根据当前输入 x_t 和上一时刻输出 h_{t-1} 决定哪一部分记忆需要被遗忘。

(1) 假设状态 c 的维度为 n。“遗忘门”会根据当前输入 x_t 和上一时刻输出 h_{t-1} 计算一个维度为 n 的向量 $\boldsymbol{f}=\text{Sigmoid}(W_1\boldsymbol{x}+W_2\boldsymbol{h})$，它的每一维度上的值都在(0,1)范围内，再将上一时刻的状态 $c_{(t-1)}$ 与 $\boldsymbol{f}$ 向量按位相乘，那么 $\boldsymbol{f}$ 取值接近 0 的维度上的信息就会被“忘记”，而 $\boldsymbol{f}$ 取值接近 1 的维度上的信息会被保留。例如模型发现某地原来是绿水蓝天，后来被污染了。于是在看到被污染之后，循环神经网络应该“忘记”之前绿水蓝天的状态。

(2) 在循环网络“忘记”了部分之前的状态后，它还需要从当前的输入补充最新的记忆，这个过程由“输入门”完成。“输入门”会根据 x_t 和 h_{t-1} 决定哪些信息加入到状态 $c_{(t-1)}$ 中

生成新的状态 c_t。例如模型发觉环境被污染之后，需要将这个信息写入新的状态。

(3) LSTM 结构在计算得到新的状态 c_t 后需要产生当前时刻的输出，这个过程由“输出门”完成。“输出门”会根据最新的状态 c_t、上一时刻的输出 h_{t-1} 和当前的输入 x_t 来决定该时刻的输出 h_t。例如当前的状态为被污染，那么“天空的颜色”后面的单词很有可能是“灰色的”。

5.5.3 长短时记忆网络计算

LSTM 的重复单元不同于标准 RNN 里的单元只有一个网络层，它的内部有 4 个网络层，LSTM 的网络结构如图 5-31 所示，

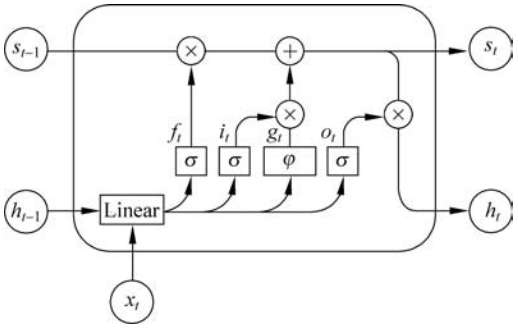


图 5-31 LSTM 网络计算节点标识

在图 5-31 中，$x(t)$表示输入序列，$h(t)$表示输出序列；

激活函数：

$$\varphi(x)=\tanh(x),\quad \sigma(x)=\frac{1}{1+e^{-x}}$$

$f(t)$表示遗忘门，表达的含义是决定从以前状态中丢弃什么信息；

$i(t)$和 $g(t)$构成输入门，决定什么样的新信息被存放在该节点状态中；

$o(t)$所在位置被称作输出门，决定要输出什么值。

节点函数之间的数学关系如式(5-37)所示。

$$\begin{cases} i(t)=\sigma(W_{ix}x(t)+W_{ih}h(t-1))+b_i \\ f(t)=\sigma(W_{fx}x(t)+W_{fh}h(t-1))+b_f \\ g(t)=\varphi(W_{gx}x(t)+W_{gh}h(t-1))+b_g \\ o(t)=\sigma(W_{ox}x(t)+W_{oh}h(t-1))+b_o \\ c(t)=g(t)*i(t)+c(t-1)*f(t) \\ h(t)=c(t)*o(t) \end{cases} \tag{5-37}$$

LSTM 前向推导过程的 Python 实现代码如下：

```
def bottom_data_is(self, x, s_prev = None, h_prev = None):
        # if this is the first lstm node in the network
        if s_prev == None: s_prev = np.zeros_like(self.state.s)
        if h_prev == None: h_prev = np.zeros_like(self.state.h)
```

```
        # save data for use in backprop
        self.s_prev = s_prev
        self.h_prev = h_prev

        # concatenate x(t) and h(t-1)
        xc = np.hstack((x,  h_prev))
        self.state.g = np.tanh(np.dot(self.param.wg, xc) + self.param.bg)
        self.state.i = sigmoid(np.dot(self.param.wi, xc) + self.param.bi)
        self.state.f = sigmoid(np.dot(self.param.wf, xc) + self.param.bf)
        self.state.o = sigmoid(np.dot(self.param.wo, xc) + self.param.bo)
        self.state.s = self.state.g * self.state.i + s_prev * self.state.f
        self.state.h = self.state.s * self.state.o
        self.x = x
        self.xc = xc
```

LSTM 的反向推导过程比较复杂，首先定义一个损失函数 $l(t)$：

$$l(t)=f(h(t),y(t))=\|h(t)-y(t)\|^2 \tag{5-38}$$

$h(t)$与 $y(t)$分别为输出序列与样本标签，下面求解整个时间序列上的 $l(t)$最小化：$L=\sum_{t=1}^{T}l(t)$。其中，T 代表整个时间序列。

通过 L 来计算梯度，假设我们要计算$\frac{\mathrm{d}L}{\mathrm{d}w}$，其中 w 是一个标量（例如可以为矩阵 $\boldsymbol{W}_{gx}$ 的一个元素），由链式法则可以导出：

$$\frac{\mathrm{d}L}{\mathrm{d}w}=\sum_{t=1}^{T}\sum_{i=1}^{M}\frac{\mathrm{d}L}{\mathrm{d}h_i(t)}\frac{\mathrm{d}h_i(t)}{\mathrm{d}w} \tag{5-39}$$

其中 $h_i(t)$是第 i 个单元的输出，M 是 LSTM 单元的个数，网络随着时间 t 前向传播，$h_i(t)$的改变不影响 t 时刻之前的损失值 loss，可以写出：

$$\frac{\mathrm{d}L}{\mathrm{d}h_i(t)}=\sum_{s=1}^{T}\frac{\mathrm{d}l(s)}{\mathrm{d}h_i(t)} \tag{5-40}$$

为了书写方便，可以令$L(t)=\sum_{s=1}^{T}l(s)$来简化书写，这样 $L(t)$就是整个序列的损失，重写式(5-40)，有

$$\frac{\mathrm{d}L}{\mathrm{d}h_i(t)}=\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t)} \tag{5-41}$$

这样就可以将式(5-39)重写为

$$\frac{\mathrm{d}L}{\mathrm{d}w}=\sum_{t=1}^{T}\sum_{i=1}^{M}\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t)}\frac{\mathrm{d}h_i(t)}{\mathrm{d}w} \tag{5-42}$$

我们知道 $L(t)=l(t)+L(t+1)$，那么有

$$\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t)}=\frac{\mathrm{d}l(t)}{\mathrm{d}h_i(t)}+\frac{\mathrm{d}L(t+1)}{\mathrm{d}h_i(t)} \tag{5-43}$$

这说明得到下一时序的导数后可以直接得出当前时序的导数，这样我们就可以计算 T 时刻的导数，然后往后推，在 T 时刻就有

$$\frac{\mathrm{d}L(T)}{\mathrm{d}h_i(T)}=\frac{\mathrm{d}l(T)}{\mathrm{d}h_i(T)}$$

该计算的 Python 实现，代码如下：

```
    def y_list_is(self, y_list, loss_layer):
            """
            Updates diffs by setting target sequence
            with corresponding loss layer.
            Will * NOT * update parameters.  To update parameters,
            call self.lstm_param.apply_diff()
            """
            assert len(y_list) == len(self.x_list)
            idx = len(self.x_list) - 1
            #first node only gets diffs from label ...
            loss = loss_layer.loss(self.lstm_node_list[idx].state.h, y_list[idx])
            diff_h = loss_layer.bottom_diff(self.lstm_node_list[idx].state.h, y_list[idx])
            #here s is not affecting loss due to h(t+1), hence we set equal to zero
            diff_s = np.zeros(self.lstm_param.mem_cell_ct)
            self.lstm_node_list[idx].top_diff_is(diff_h, diff_s)
            idx -= 1
            ###... following nodes also get diffs from next nodes, hence we add diffs to diff_h
            ###we also propagate error along constant error carousel using diff_s
            while idx >= 0:
                loss += loss_layer.loss(self.lstm_node_list[idx].state.h, y_list[idx])
                diff_h = loss_layer.bottom_diff(self.lstm_node_list[idx].state.h, y_list
    [idx])
                diff_h += self.lstm_node_list[idx + 1].state.bottom_diff_h
                diff_s = self.lstm_node_list[idx + 1].state.bottom_diff_s
                self.lstm_node_list[idx].top_diff_is(diff_h, diff_s)
                idx -= 1

            return loss
```

从上面公式可以很容易理解 diff_h 的计算过程，这里的 loss_layer.bottom_diff 定义如下：

```
    def bottom_diff(self, pred, label):
            diff = np.zeros_like(pred)
            diff[0] = 2 * (pred[0] - label)
            return diff
```

下面来推导$\frac{\mathrm{d}L(t)}{\mathrm{d}s(t)}$，结合前面的前向推导公式可以很容易得出：$s(t)$的变化会直接影响$h(t)$和$h(t+1)$，进而影响$L(t)$，即有

$$\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t)}=\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t)}\frac{\mathrm{d}h_i(t)}{\mathrm{d}s_i(t)}+\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t+1)}\frac{\mathrm{d}h_i(t+1)}{\mathrm{d}s_i(t)}$$

因为$h(t+1)$不影响$L(t)$，所以有$\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t+1)}=\frac{\mathrm{d}L(t+1)}{\mathrm{d}h_i(t+1)}$，因此有

$$\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t)}=\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t)}\frac{\mathrm{d}h_i(t)}{\mathrm{d}s_i(t)}+\frac{\mathrm{d}L(t+1)}{\mathrm{d}h_i(t+1)}\frac{\mathrm{d}h_i(t+1)}{\mathrm{d}s_i(t)}$$

$$=\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t)}\frac{\mathrm{d}h_i(t)}{\mathrm{d}s_i(t)}+\frac{\mathrm{d}L(t+1)}{\mathrm{d}s_i(t)}$$

同样，我们可以通过后面的导数逐级反推，得到前面的导数，代码就是 diff_s 的计算过程，下面计算$\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t+1)}\frac{\mathrm{d}h_i(t)}{\mathrm{d}s_i(t+1)}$。

因为 $h(t)=s(t)*o(t)$，那么就有

$$\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t+1)}\frac{\mathrm{d}h_i(t)}{\mathrm{d}s_i(t+1)}=\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t)}o_i(t)=o_i(t)\,[\mathrm{diff\_h}]$$

即

$$\frac{\mathrm{d}L(t)}{\mathrm{d}s_i(t)}=o(t)\,[\mathrm{diff\_h}]_i+[\mathrm{diff\_s}]_i$$

其中，$[\mathrm{diff\_h}]_i$ 和 $[\mathrm{diff\_s}]_i$ 分别表示当前 t 时刻的$\frac{\mathrm{d}L(t)}{\mathrm{d}h_i(t)}$和 $t+1$ 时刻的$\frac{\mathrm{d}L(t)}{\mathrm{d}s_i(t)}$。

下面根据前向过程计算参数偏导如式(5-44)：

$$\begin{cases}\dfrac{\mathrm{d}L(t)}{\mathrm{d}o(t)}=\dfrac{\mathrm{d}L(t)}{\mathrm{d}h(t)}s(t)\\[2ex]\dfrac{\mathrm{d}L(t)}{\mathrm{d}i(t)}=\dfrac{\mathrm{d}L(t)}{\mathrm{d}s(t)}\dfrac{\mathrm{d}s(t)}{\mathrm{d}i(t)}=\dfrac{\mathrm{d}L(t)}{\mathrm{d}s(t)}g(t)\\[2ex]\dfrac{\mathrm{d}L(t)}{\mathrm{d}g(t)}=\dfrac{\mathrm{d}L(t)}{\mathrm{d}s(t)}\dfrac{\mathrm{d}s(t)}{\mathrm{d}g(t)}=\dfrac{\mathrm{d}L(t)}{\mathrm{d}s(t)}i(t)\\[2ex]\dfrac{\mathrm{d}L(t)}{\mathrm{d}f(t)}=\dfrac{\mathrm{d}L(t)}{\mathrm{d}s(t)}\dfrac{\mathrm{d}s(t)}{\mathrm{d}f(t)}=\dfrac{\mathrm{d}L(t)}{\mathrm{d}s(t)}s(t-1)\end{cases}\tag{5-44}$$

该计算的 Python 实现代码如下：

```
def top_diff_is(self, top_diff_h, top_diff_s):
        #notice that top_diff_s is carried along the constant error carousel
        ds = self.state.o * top_diff_h + top_diff_s
        do = self.state.s * top_diff_h
        di = self.state.g * ds
        dg = self.state.i * ds
        df = self.s_prev * ds

        #diffs w.r.t. vector inside sigma / tanh function
        di_input = (1. - self.state.i) * self.state.i * di #sigmoid diff
        df_input = (1. - self.state.f) * self.state.f * df
        do_input = (1. - self.state.o) * self.state.o * do
        dg_input = (1. - self.state.g ** 2) * dg #tanh diff

        #diffs w.r.t. inputs
        self.param.wi_diff += np.outer(di_input, self.xc)
        self.param.wf_diff += np.outer(df_input, self.xc)
        self.param.wo_diff += np.outer(do_input, self.xc)
        self.param.wg_diff += np.outer(dg_input, self.xc)
        self.param.bi_diff += di_input
        self.param.bf_diff += df_input
```

```
        self.param.bo_diff += do_input
        self.param.bg_diff += dg_input

        #compute bottom diff
        dxc = np.zeros_like(self.xc)
        dxc += np.dot(self.param.wi.T, di_input)
        dxc += np.dot(self.param.wf.T, df_input)
        dxc += np.dot(self.param.wo.T, do_input)
        dxc += np.dot(self.param.wg.T, dg_input)

        #save bottom diffs
        self.state.bottom_diff_s = ds * self.state.f
        self.state.bottom_diff_x = dxc[:self.param.x_dim]
        self.state.bottom_diff_h = dxc[self.param.x_dim:]
```

这里的 top_diff_h 和 top_diff_s 分别是上文的 diff_h 和 diff_s。下面介绍 wi_diff 的求解过程，其他变量的求解过程与此类似。

$$\frac{\mathrm{d}L(t)}{\mathrm{d}W_i}=\frac{\mathrm{d}L(t)}{\mathrm{d}i(t)}\frac{\mathrm{d}i(t)}{\mathrm{d}(W_i x_c(t))}\frac{\mathrm{d}(W_i x_c(t))}{\mathrm{d}x_c(t)} \tag{5-45}$$

该计算的 Python 实现代码如下：

```
wi_diff += np.outer((1. - i) * i * di, xc)
wf_diff += np.outer((1. - i) * i * df, xc)
wo_diff += np.outer((1. - i) * i * do, xc)
wg_diff += np.outer((1. - i) * i * dg, xc)
```

本章小结

深度学习是机器学习研究中的一个新领域，其动机在于建立、模拟人脑进行分析学习的神经网络，它模仿人脑的机制来解释数据，例如图像、声音和文本。深度学习的概念源于人工神经网络的研究。含多隐层的多层感知器就是一种深度学习结构。深度学习通过组合低层特征形成更加抽象的高层表示属性类别或特征，以发现数据的分布式特征表示。深度学习使用多层神经网络模仿人脑神经系统的工作原理，在人工智能领域取得了巨大成功。本章首先对深度学习的概念及工作机理进行了介绍，同时结合人工神经网络阐述多层神经网络的理论基础与计算方法。接着对深度神经网络中的概念和方法进行详细介绍。最后结合深度学习的几种典型模型，如卷积神经网络、循环神经网络和长短时记忆网络进行详细介绍，并给出对应的 Python 代码实现。

课后思考题

1. 目前有哪些深度学习开源框架？试分别比较优缺点。
2. 深度学习与神经网络之间有什么关联？

3. 什么是激活函数？为什么神经网络需要激活？
4. 什么是卷积神经网络？目前有哪些卷积神经网络模型？
5. 什么叫全连接神经网络？
6. 什么叫递归神经网络？递归神经网络主要在哪些领域取得了良好的表现？
7. 什么是长短时记忆网络 LSTM？它有什么优点？