第1章

深度学习基础知识

1.1 深度学习的发展历程

深度学习是目前最为火热的研究领域之一，并在很多任务中取得了很大的成功。作为机器学习的一个重要分支，深度学习在取得成功的背后也经历了较为曲折的发展历程。深度学习经历了三次发展浪潮。

20 世纪 40 年代，受生物大脑启发，“M-P 神经元模型”[1] 首次提出用来模仿神经元的结构和工作原理。在这个模型中，一个神经元接收来自 n 个神经元的信号，这些信号以加权方式进行连接组合，在与神经元阈值进行比较后得到当前神经元的输出信号。但是这些信号的连接权重大多采用人工预先设置的方式。在该模型的基础上，由两层神经元模型构成的“感知机”模型[2-3] 被提出，它能根据输入样本学习权重。该模型虽然当时获得了很大关注，但其本质上还是一种线性模型，学习能力有限，无法解决诸如异或问题等线性不可分问题。因此对深度学习的研究逐渐停滞。

20 世纪 80 年代，伴随联结主义思想[4] 和误差反向传播算法[5] 的提出，深度学习转向学习能力更强的多层网络，多层网络能够很好地解决线性不可分问题，因此越来越多地应用于多种模式识别任务中，如手写字体识别[6] 等。但当时计算机的硬件水平有限，多层的网络需要强大的运算能力，同时误差反向传播在网络层数很深时会出现梯度消失问题，因此深度学习再次陷入瓶颈期。

2006 年，多伦多大学的 Geoffrey Hinton[7] 提出深度信念网络，其使用无监督逐层预训练，再用有监督的反向传播误差调优的方式，来解决误差反向传播的梯度消失问题。在此后的 2012 年，Geoffrey Hinton 团队使用 AlexNet[8] 一举夺得了 ImageNet 图像识别大赛的冠军，其识别率显著超越了以往算法的识别率，获得了极大的关注。而此时计算机硬件水平相较以前明显改善，特别是采用图像处理器（graphics processing unit，GPU）极大地提高了运算能力，深度学习再次活跃起来，并迅速促进了各个领域的发展。

1.2 卷积神经网络

卷积神经网络在深度学习发展中发挥了重要的作用。目前的卷积神经网络通常由卷积层、池化层、全连接层及激活函数等组成，本节中对卷积神经网络结构中重要的组件进行介绍。

1.2.1 卷积层

卷积层包含可训练参数 $\boldsymbol{w}$（称作滤波器或者卷积核）和偏置参数 $\boldsymbol{a}$。卷积操作的通用表达式为

$$\boldsymbol{y} = \boldsymbol{w} \otimes \boldsymbol{x} + \boldsymbol{a} \tag{1.1}$$

式中：$\otimes$表示卷积运算。

根据处理数据的维度差异，卷积层可分为一维卷积层、二维卷积层和多维卷积层。一维卷积层通常处理如信号信息或者时间序列等，假设滤波器长度为 m，对于一个信号序列 $x_1, x_2, \cdots$ 的第 t 时间，一维卷积可以表示为

$$\boldsymbol{y}_t = \sum_{k=1}^{m} \boldsymbol{w}_k \cdot \boldsymbol{x}_{t-k+1} + \boldsymbol{a}_t \tag{1.2}$$

二维卷积经常用来处理图像，给定一个输入 $\boldsymbol{x} \in \mathbb{R}^{H \times W}$，卷积滤波器 $\boldsymbol{w} \in \mathbb{R}^{h \times w}$，卷积运算表示为

$$\boldsymbol{y}_{ij} = \sum_{m=1}^{h} \sum_{n=1}^{w} \boldsymbol{w}_{m,n} \cdot \boldsymbol{x}_{i-m+1, j-n+1} + \boldsymbol{a} \tag{1.3}$$

卷积滤波器的尺寸小于输入的尺寸，即 $h \ll H$，$w \ll W$。在卷积计算中，还引入了步长和填充。步长即滤波器在卷积计算时滑动的距离或者时间间隔。填充即在输入的两端填充一个常量（通常为 0），便于输入数据能被完整地计算卷积。图 1.1 展示了二维卷积计算的示例。

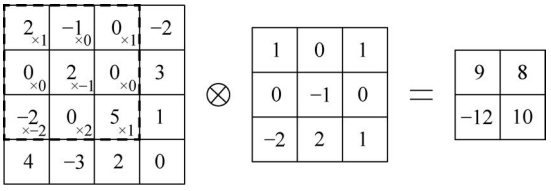

图 1.1 卷积计算示例

卷积层有如下几个重要的性质。

（1）局部连接。感知机模型中，每个输出都要与输入相连，每个连接都需要一个权重参数，假如有 s 个输入和 t 个输出，此时需要的参数量为 $s \times t$，时间复杂度为 $O(s \times t)$。随着输入和输出数量的增加，所需的参数和时间复杂度也急剧增加。

而卷积层中，每个输入只与滤波器相连，即 $k=h\times w$ 个连接，同时 $k\ll s$，这种连接只需要 $k\times t$ 的参数和 $O(k\times t)$ 的时间复杂度。

(2) 权值共享。在卷积层中，滤波器的每个元素作用在输入的每个位置上，不需要输入的每个位置都设置单独的权重值。因此可以大大地减少每层需要保存的参数量，显著地降低了存储需求。

(3) 平移等变性。卷积层的权值共享也使得卷积层具有平移等变性，即不管输入如何改变，其输出值也以同样的方式发生改变。

(4) 空间属性。一般情况下，根据输入神经元个数 n，滤波器大小 m，步长 s，及输入两端填充的常量的个数 p，可以计算出输出神经元个数为 $(n+2p-m)/s+1$。

1.2.2　池化层

池化层是卷积神经网络中常见的网络层，其通常位于两个卷积层之间，用来对特征进行选择，降低特征数量进而减少网络参数。池化操作是将输入划分为多个区域，选出每个区域内的最大输出值或者将输出值进行平均，因此池化操作可以分为最大池化和平均池化。该操作可以形式化地表示为假设输入为 $\boldsymbol{X}\in\mathbb{R}^{H\times W\times C}$，将每个特征通道划分为多个大小为 $m\times n(1\leqslant m\leqslant H,1\leqslant n\leqslant W)$ 的区域 $\boldsymbol{R}$。

(1) 最大池化：选择每个区域中的最大值及其索引：

$$\boldsymbol{Y}_{h,w}=\max_{i\in\boldsymbol{R}_{m,n}}\boldsymbol{X}_i \tag{1.4}$$

$$i'=\operatorname*{argmax}_{i\in\boldsymbol{R}_{m,n}}\boldsymbol{X}_i \tag{1.5}$$

i' 最大值在输入中的位置，记录该位置方便反向传播时的计算梯度。

(2) 平均池化：计算每个区域中输出值的平均值：

$$\boldsymbol{Y}_{h,w}=\frac{1}{C}\sum_{i\in\boldsymbol{R}_{m,n}}\boldsymbol{X}_i \tag{1.6}$$

其中，C 是 X_i 所在区域中的值的个数。如果设置 $m=H,n=W$，则是全局平均池化。

不管什么样的池化函数，当输入进行少量平移时，经过池化函数的大多数输出并不会发生改变，此即为平移不变性。

1.2.3　全连接层

不同于卷积层依靠滤波器进行局部连接，全连接层是将输出神经元与每个输入神经元相连接。一般地，全连接层包含较多的参数和计算量，如果输入神经元有 s 个，输出为 t 个，则该层的参数量为 $s\times t$。对于全连接层的输入，通常需要进行拉平(flatten)操作，而且一旦该层确定后，要求输入维持固定大小。

1.2.4　激活函数

激活函数受启发于生物神经网络，每个神经元与其他神经元相连接，当它“兴

奋”时就会向相连的神经元发送化学物质，从而改变这些神经元内的电位；如果某神经元的电位超过了一个“阈值”(threshold)，那么它就会被激活，即“兴奋”起来，向其他神经元发送化学物质。

1. 阶跃函数

理想中的激活函数是图 1.2 所示的阶跃函数，它将输入值映射为输出值“0”或者“1”。“1”对应于神经元兴奋，“0”对应于神经元抑制。然而阶跃函数具有不连续、不光滑等不太好的性质，因此实际常用 Sigmoid 函数作为激活函数。

2. Sigmoid 函数

典型的 Sigmoid 函数的数学表达式为

$$f(x)=\frac{1}{1+\mathrm{e}^{-x}} \tag{1.7}$$

其图形如图 1.2(a)所示，它把可能在较大范围内变化的输入值挤压到(0,1)输出值范围内。

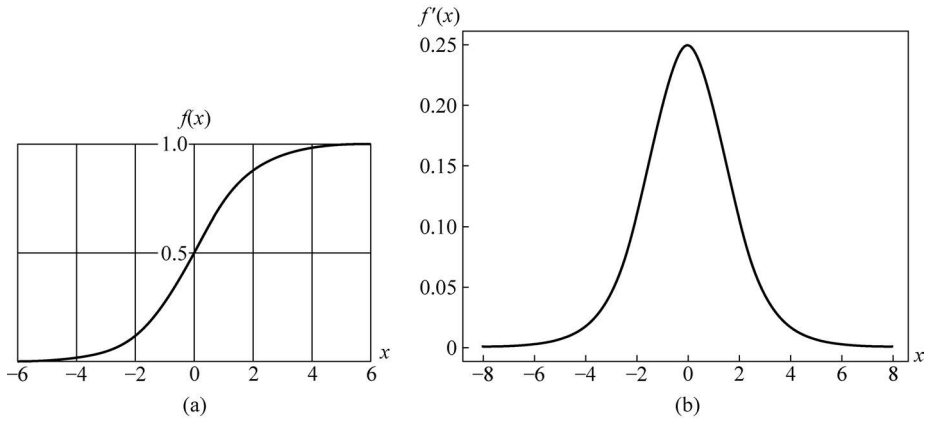

图 1.2 Sigmoid 激活函数

(a) Sigmoid 函数；(b) Sigmoid 函数的导数

但是该激活函数在实际使用中具有如下缺点。

(1) 梯度消失/梯度爆炸。Sigmoid 函数的导数为

$$f'(x)=f(x)\cdot(1-f(x)) \tag{1.8}$$

其导数图形如图 1.2(b)所示。从图中可看出，该函数的导数最大值为 $f'(0)=0.25$。在神经网络层数较深时，如果权重的初始值在[0,1]范围内，则可以计算出反向传播的梯度指数级减少至接近于 0，因此出现了梯度消失的现象；如果权重的初始值在(1,+∞)，则可能出现梯度爆炸的现象。

(2) 输出不是 0 中心的。这意味着后一层神经元的输入也不是 0 中心的。这将产生一个问题，即如果输入 $\boldsymbol{x}>0$，在反向传播梯度时，卷积权重 $\boldsymbol{w}$ 的梯度可能全是正值或者全是负值，权重的更新将是“Z”字形的，最终导致整个网络的更新较慢。

(3) 计算较为耗时。由于其表达式及导数中包含指数运算，计算机求解时较为耗时，特别是神经网络规模较大时，将影响训练的时间。

3. Tanh 激活函数

Tanh 激活函数的表达式为

$$f(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{1.9}$$

根据其表达式，可以很容易地得到其图形及导数图形，如图 1.3 所示。

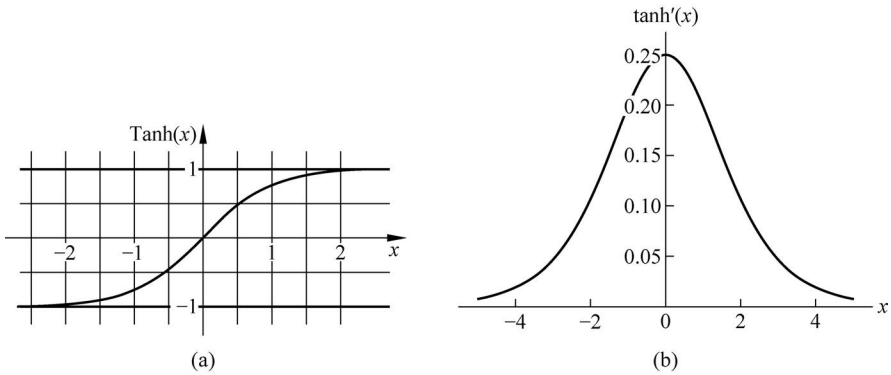

图 1.3　Tanh 激活函数

(a) Tanh 激活函数；(b) Tanh 激活函数的导数

从图 1.3 可以看出，不同于 Sigmoid 函数，Tanh 激活函数将输入值变换到(−1,1)范围内，而且该函数的输出值是 0 中心的。但是其也存在梯度消失及计算耗时的问题。

4. ReLU 函数

修正线性单元(rectified linear unit，ReLU)激活函数的表达式为

$$f(x)=\max(0,x) \tag{1.10}$$

其函数图形如图 1.4 所示。

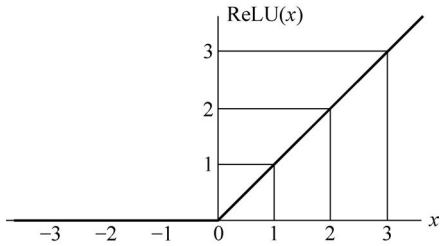

图 1.4　ReLU 函数图像

ReLU 函数能够解决梯度消失的问题，而且该函数仅需要判断输入是否大于 0，因此计算速度很快。其在训练时的收敛速度也很快，因而成为目前流行的激活

函数。然而，ReLU 函数的输出值也不是 0 中心的，而且如果参数的初始化设置不当或者学习率较高会导致某些神经元永远不会被激活，即进入“死亡”状态。

1.2.5 损失函数

损失函数用于评价模型的预测值与真实值不一致的程度。损失函数值越小，模型的表现越好。下面简要介绍常见的损失函数。

1. “0-1”损失函数

“0-1”损失即预测值与真实值不相等则为“1”，否则为“0”：

$$\mathcal{L}(y,p)=\begin{cases}1, & y\neq p\\ 0, & y=p\end{cases} \tag{1.11}$$

式中：y 表示真实值；p 表示模型的预测值；下同。“0-1”损失函数表示分类判断错误的个数，但其是一个非凸函数。

2. 绝对值损失函数

绝对值损失函数是计算预测值与真实值的差的绝对值，又称 L1 损失函数：

$$\mathcal{L}(y,p)=|y-p| \tag{1.12}$$

在预测值 p 等于真实值 y 时，导数不存在，并且损失值随着误差的增大而线性增大。

3. 平方损失函数

平方损失函数是计算预测值与真实值的差的平方值，又称 L2 损失函数：

$$\mathcal{L}(y,p)=(y-p)^2 \tag{1.13}$$

该损失函数在预测值与真实值相等处得到最小值，损失值随着误差的增大而迅速增加，一般用于回归任务中。

4. 对数损失函数

对数损失函数的表达式为

$$\mathcal{L}(y,p(y\mid x))=-\log(p(y\mid x)) \tag{1.14}$$

对数损失函数能很好地预测值的概率分布，多用于分类问题中。

5. 交叉熵损失函数

交叉熵损失函数表达式为

$$\mathcal{L}(y,p)=y\log(p)+(1-y)\log(1-p) \tag{1.15}$$

从概率角度看，该损失函数是假设样本服从伯努利分布时，对真实值的极大似然估计。其是目前较为常用的损失函数之一。

1.2.6 Dropout

Dropout[9]是一种较为常用的正则化方式，即通过限制模型的复杂度以防止模型出现过拟合现象。Dropout 是在训练神经网络特别是层数较多的神经网络时，

随机丢弃一部分神经元。丢弃的方法是设置一个概率 p，对每个神经元以概率 p 来判断其是否需要保留。在训练时，以概率为 p 的伯努利分布生成一个丢弃掩码 $m\in\{0,1\}^d$（d 表示神经元个数）。对于一个神经层 $y=f(x)$，引入一个丢弃函数 $d(y)$，该函数定义为

$$d(y)=\begin{cases} m\odot y, & \text{训练时} \\ py, & \text{测试时} \end{cases} \tag{1.16}$$

p 可以通过验证集来获得，也可以直接设置为某个固定值（通常为 0.5）。由于在训练时丢弃了部分神经元及其输出值，神经元平均数量变为原先的 p 倍；而在测试时，所有神经元均被激活，因此需要在测试时每个神经元的输出值都乘以 p，以避免训练和测试时网络输出值的不一致，或者在训练时将输出值除以 p，而在测试时不做任何处理。

Dropout 为什么能够缓解过拟合进而有利于提高网络泛化性呢？一种普遍的解释是：在训练时按照一定的概率“失活”部分模型计算单元（神经元），训练对象实际上是模型的子结构，其原理上近似基于集成学习降低泛化误差的技术。

1.2.7　数据预处理

通常情况下，参与训练的样本由于来源、质量等不同，其特征的取值范围差异很大。在训练或者计算时，取值范围大的特征往往占据主导作用。特别是在采用上述有界激活函数时（如 Sigmoid、Tanh 函数），特征的取值范围很大，容易出现梯度消失或者爆炸的情况，需要精心地设计参数的初始化。另外，取值范围的差异也会影响梯度下降时的搜索方向进而影响收敛速率。因此有必要对参与训练的样本数据进行预处理。下面简单介绍常用的数据预处理方法。

缩放归一化：缩放归一化是最简单的数据预处理方法，其主要利用样本的最大值和最小值，将样本的每个特征的取值缩放到[0,1]或者[−1,1]之间。对于每一维特征 x，

$$\hat{x}_i=\frac{x_i-\min(x)}{\max(x)-\min(x)} \tag{1.17}$$

其中 $\min(x)$和 $\max(x)$分别表示与 x 同一维度的所有样本上的最小值和最大值。

标准归一化：标准归一化是将每一维的特征都处理为符合标准正态分布（均值为 0，方差为 1）。假设样本个数为 N，每个样本有 D 维特征，对于每一维特征 $\boldsymbol{x}_{ij}$（$i\in\{1,2,\cdots,N\}$，$j\in\{1,2,\cdots,D\}$），首先计算它的均值和方差：

$$\mu_j=\frac{1}{N}\sum_{i=1}^{N}\boldsymbol{x}_{ij} \tag{1.18}$$

$$\sigma_j^2=\frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{x}_{ij}-\mu_j)^2 \tag{1.19}$$

然后，将特征 $\boldsymbol{x}_{ij}$ 减去均值 μ_j 再除以标准差 σ_j 即得到归一化后的特征：

$$\hat{x}_{ij}=\frac{\boldsymbol{x}_{ij}-\mu_j}{\sigma_j} \tag{1.20}$$

白化：白化是用来降低输入数据的冗余信息。输入数据经过白化处理之后，特征之间的相关性降低，并且所有特征具有相同的方差。白化主要通过使用主成分分析的方法去除各个成分之间的相关性。

1.2.8 批归一化

在神经网络中，某一层的输入是其之前网络层的输出。如果前一网络层的参数发生变化，则该层的输入也会随之改变。特别是在使用随机梯度下降算法训练的过程中，每次的参数更新必然导致神经网络中每层的输入分布发生变化。层数越深，分布的变化越明显。分布的差异变化，会影响梯度下降时的搜索方向进而减缓模型的优化速率。通常称某个神经元的输入分布发生改变，其参数需要重新学习的现象称为内部协变量偏移。

为了缓解内部协变量偏移问题，需要将每个网络层进行归一化处理，使每个网络层的输入分布在训练时保持一致。因此 Ioffe 和 Szegedy 在 2015 年提出了批归一化 BN(batch normalization)[10]的方法。该方法的思路就是将每个网络层进行归一化处理，由于层数较多，使用效率更高的标准归一化，将每个网络层的特征 $\boldsymbol{x}^{(l)}$（表示第 l 层网络）都归一化到正态分布，即

$$\hat{\boldsymbol{x}}^{(l)}=\frac{\boldsymbol{x}^{(l)}-\mathbb{E}[\boldsymbol{x}^{(l)}]}{\sqrt{\mathrm{var}(\boldsymbol{x}^{(l)})+\varepsilon}} \tag{1.21}$$

式中：$\mathbb{E}[\boldsymbol{x}^{(l)}]$和 $\mathrm{var}(\boldsymbol{x}^{(l)})$代表在当前参数下，$\boldsymbol{x}^{(l)}$每一维在当前数据集下的期望和方差。由于对全部数据集进行统计存在困难，此处以目前采用的小批量样本集的期望和方差近似估计。

假设小批量样本集包含 m 个样本，第 l 层网络的特征为 $\boldsymbol{x}^{(l)}$，其均值和方差为

$$\mu=\frac{1}{m}\sum_{i=1}^{m}\boldsymbol{x}^{(m,l)} \tag{1.22}$$

$$\sigma^2=\frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{x}^{(m,l)}-\mu)^2 \tag{1.23}$$

$$\hat{\boldsymbol{x}}^{(l)}=\frac{\boldsymbol{x}^{(l)}-\mu}{\sqrt{\sigma^2+\varepsilon}} \tag{1.24}$$

经过上述归一化操作之后，网络层的特征内容发生了变化。如果归一化之后使用 Sigmoid 激活函数，归一化后的数据很可能落入线性变换区间，减弱了神经网络的非线性性质。为解决此问题，引入 $\boldsymbol{\gamma}$ 和 $\boldsymbol{\beta}$ 参数用于缩放和平移，即

$$\boldsymbol{y}^{(l)}=\boldsymbol{\gamma}\odot\hat{\boldsymbol{x}}^{(l)}+\boldsymbol{\beta} \tag{1.25}$$

在特殊情况下，网络学习到的 $\boldsymbol{\gamma}=\boldsymbol{\sigma}$，$\boldsymbol{\beta}=\boldsymbol{\mu}$ 时，模型可将其还原到归一化之前的状态。

由于模型更关注整个数据集上的统计量，因此在得到小批量样本的均值和方差之后，可以使用移动平均的方式计算整个数据集的均值和方差。在测试时就使用计算得到的整个数据集的均值和方差。

1.2.9　优化方法

目前，训练神经网络主要使用梯度下降法来寻找使结构风险最小化的参数。由于训练数据的样本量很大，在训练深层神经网络时，无法在梯度下降的每次迭代过程中计算所有样本的梯度。因此，通常使用小批量梯度下降法训练深层神经网络。

假设 $f(\boldsymbol{x},\theta)$ 为一神经网络，$\boldsymbol{x}$ 为输入数据，θ 为网络参数，在使用小批量梯度下降法时，选取 K 个训练样本 $\mathcal{J}=\{(\boldsymbol{x}^k,\boldsymbol{y}^k)\}_{k=1}^{K}$。在第 t 次迭代时，损失函数关于参数 θ 的偏导数为

$$\frac{\partial \mathcal{L}}{\partial \theta}=\frac{1}{K}\sum_{(\boldsymbol{x}^k,\boldsymbol{y}^k)\in\mathcal{J}}\frac{\partial \mathcal{L}(\boldsymbol{y}^k,f(\boldsymbol{x}^k,\theta))}{\partial \theta} \tag{1.26}$$

第 t 次更新的梯度定义为

$$g_t \triangleq \frac{\partial \mathcal{L}}{\partial \theta_{t-1}} \tag{1.27}$$

使用梯度下降更新参数为

$$\theta_t=\theta_{t-1}-\alpha g_t \tag{1.28}$$

式中：$\alpha>0$，为学习率。

为了能够加快参数的优化速度及优化梯度的更新方向，可以通过学习率衰减和梯度方向优化进行改进。

1. 学习率衰减

学习率在梯度下降算法中至关重要。学习率设置过大，会使得梯度方向越过最优点而反复震荡，最终无法收敛。而学习率过小时，梯度下降较慢而导致收敛速度慢。在实际训练过程中，先设置较大的学习率以快速地优化到最优点附近，然后使用较小的学习率逐渐达到最优点。因此，通常采用学习率随着优化进程逐渐衰减的方式来调整。假设初始学习率为 α_0，第 t 次迭代后的学习率为 α_t，常用的学习率衰减方法有以下几种。

(1) 阶梯衰减。每迭代 n 次，学习率衰减固定的值 β，即

$$\alpha_t=\begin{cases}\alpha_0\times\beta^{t/n}, & t\%n=0\\ \alpha_{t-1}, & t\%n!=0\end{cases} \tag{1.29}$$

(2) 指数衰减。

$$\alpha_t=\alpha_0\times \mathrm{e}^{-t} \tag{1.30}$$

(3) 逆时衰减。

$$\alpha_t=\frac{\alpha_0}{1+k\times t} \tag{1.31}$$

式中：k 为超参数。

在实际使用中，阶梯衰减因其涉及的超参数较少，是目前应用较多的学习率衰减方法。上述介绍的学习率衰减方法，使所有的参数拥有相同的学习率。但是神经网络中每个参数的收敛速度不尽相同，因此一些工作尝试为每个参数自适应地调整学习率。

Adagrad 算法：Adagrad 是由 Duchi 等[11]提出的一种在每次迭代时自适应地调整每个参数的学习率的算法。在第 t 次迭代时，先计算每个参数的梯度平方的累加值：

$$G_t = \sum_{\tau=1}^{t} g_\tau \odot g_\tau \tag{1.32}$$

式中：$\odot$ 表示按元素相乘；g_τ 表示第 τ 次迭代时的梯度。每个参数的学习率则为

$$\alpha_t = \frac{\alpha_0}{\sqrt{G_t + \varepsilon}} \tag{1.33}$$

式中：ε 为一个很小的常数值(一般设置为 $10^{-8} \sim 10^{-4}$)。此时参数更新为

$$\theta_t = \theta_{t-1} - \alpha_t \odot g_t \tag{1.34}$$

根据以上表达式，可以发现，如果某个参数的梯度很大，则其学习率将会很小；相反，如果某个参数的梯度很小或者不被经常更新，则其学习率相对较大。总体上，每个参数的学习率随着迭代次数的增加而逐渐减少。但是该算法的不足之处在于其学习率持续单调下降，如果迭代一定次数没有达到最优点的情况，则由于此时学习率已经很小了，后续迭代也很难再达到最优点。

RMSprop 算法：RMSprop 算法是由 Geoff Hinton[12]提出的一种更有效的自适应学习率衰减算法。它能避免 Adagrad 算法激进地降低学习率以至于出现学习过早停止的问题。RMSprop 使用第 t 次迭代梯度平方的指数衰减移动平均来取代 Adagrad 算法的累加形式，即

$$G_t = \beta G_{t-1} + (1-\beta) g_t \odot g_t \tag{1.35}$$

式中：β 表示衰减率，其通常可以设置为 0.9/0.99。参数的更新差值设置为

$$\Delta\theta_t = -\frac{\alpha_0}{\sqrt{G_t + \varepsilon}} \odot g_t \tag{1.36}$$

RMSprop 算法使得每个参数的学习率并不严格地单调下降。

Adadelta 算法：Adadelta 算法[13]是对 Adagrad 算法的一种改进，类似于 RMSprop 算法，首先计算梯度平方的指数衰减移动平均值。此外，其还需要计算参数更新差值 $\Delta\theta$ 的指数衰减移动平均值，即

$$\Delta x_{t-1}^2 = \rho \Delta x_{t-2}^2 + (1-\rho) \Delta\theta_{t-1} \odot \Delta\theta_{t-1} \tag{1.37}$$

式中：ρ 表示衰减率。参数更新差值则为

$$\Delta\theta_t = -\frac{\sqrt{\Delta x_{t-1}^2 + \varepsilon}}{\sqrt{G_t + \varepsilon}} \odot g_t \tag{1.38}$$