1 深度学习入门

深度学习是深度强化学习的重要构成部分。本章将首先简要介绍深度学习的基础知识，会从简单的单层神经网络开始，逐渐引入更加复杂且学习能力更强的神经网络模型，比如卷积神经网络和循环神经网络的模型。本章在最后将提供一些代码样例，用于介绍深度学习的实现过程。

1.1 简介

如果您已经非常熟悉深度学习，则可以从第 2 章开始阅读。如果您想对深度学习中的部分内容进行深入的学习和了解，推荐您参阅其他相关图书，例如 *Pattern Recognition and Machine Learning* (Bishop, 2006) 和 *Deep Learning* (Goodfellow et al., 2016)。与经典强化学习不同的是，深度强化学习是基于深度学习模型，即深度神经网络，来利用大数据和高性能计算强大优势的。我们可以大致将深度学习模型分为以下两大类。

判别模型用于建模条件概率 $p(y|x)$，其中 x 代表输入数据，而 y 代表输出目标。也就是说，判别模型基于输入数据 x，预测相对应的标签 y。顾名思义，判别模型大多应用于需要进行判断的任务，例如，分类任务和回归任务。具体来说，在分类任务中，模型需要根据输入数据从备选类别中选择正确的目标类别。如果一个任务中仅有两个备选类别且模型只需要从中选取一个正确的目标类别，则为二分类任务，是最为基本的分类任务。例如，在情感分析中 (Maas et al., 2011)，根据文本内容，判断文本表达了正面的情绪还是负面的情绪，即二分类任务。与之相对应的，在多标签分类任务中，备选类别中可能同时有多个正确的目标类别。

在很多情况下，一个分类模型并不直接指定目标类别，而会给每一个备选类别计算一个概率。例如，模型根据某个数据样例，认为它有 80% 的概率来自类别 A，而另有 15% 的概率来自类别 B，5% 的概率来自类别 C。之所以使用这种基于概率的表征，主要是为了便于在训练阶段对模型

进行优化。深度学习已经在很多像图像分类 (Krizhevsky et al., 2009) 和文本分类 (Yang et al., 2019) 的分类任务上取得了巨大的成功。

分类任务的输出均为离散的类别标签，而回归任务则不同。回归任务的输出是连续的数值，如利用过去的交通数据来预测未来一段时间内的车速 (Liao et al., 2018a,b)。只要回归模型是基于条件概率建模的，我们就认为它是判别模型。

生成模型用于建模联合概率 $p(x, y)$。生成模型通常对可观测数据的分布进行建模，从而达到生成可观测数据的目的。生成对抗网络（Generative Adversarial Networks，GANs）(Goodfellow et al., 2014) 就是这样一个例子，它被用于生成图像、重构图像和对图像去噪。然而，类似于 GANs 的深度学习技术与可观测数据的分布并没有显式的关系，因为深度学习技术更关注生成的样本和可观测的真实样本之间的相似程度。与此同时，像朴素贝叶斯（Naive Bayes）的生成模型也用于解决分类任务 (Ng et al., 2002; Rish et al., 2001)。尽管生成模型和判别模型都可以用于解决分类任务，判别模型关注的是哪一个标签更适合可观测数据，而生成模型则尝试建模可观测数据的分布。下面举两个例子来说明它们的不同。朴素贝叶斯对似然概率（Likelihood）$p(x|y)$ 建模，也就是可观测数据在给定标签情况下的条件概率。生成模型先学会创造数据，再去学习如何判别数据，当学习了联合概率分布 $p(x, y)$ 后即可以学会判别，比如给定观测输入 x，输出目标为 1 的概率为 $p(y = 1|x) = \frac{p(x,y=1)}{p(x)}$。

大多数深度神经网络都是判别模型，无论其目的是用于判别类任务还是生成类任务。这是因为很多生成类任务在具体实现中都可以简化为分类或者回归问题。例如，问答系统 (Devlin et al., 2019) 可以简化为根据问题选择文本中相应的段落；自动摘要 (Zhang et al., 2019b) 可以简化为从词表中根据概率选择单词，并组合成摘要。在这两种场景下，它们都在尝试生成文本，但是一个使用了分类的方法，另一个则使用了回归的方法。

具体来说，本章将介绍深度学习相关的基本元素和技术，例如构造深度神经网络必需的神经元、激活函数和优化器等，同时将介绍深度学习相关的应用。本章也将介绍基础的深度神经网络，例如多层感知器（Multilayer Perceptron，MLP）、卷积神经网络（Convolutional Neural Networks，CNNs），以及循环神经网络（Recurrent Neural Networks，RNNs）。最后，1.10 节将基于 TensorFlow 和 TensorLayer 介绍深度神经网络的实现样例。

1.2 感知器

单输出

神经元或节点是深度神经网络最基本的单元。神经元的概念最初是基于大脑中生物神经元提出的，也是生物神经元的一种抽象表示。在大脑中，生物神经元通过树突接受电信号，当生物神经元被激活后，通过轴突将电信号传播给其他附近的生物神经元。在真实的生物系统中，神经元的信息传递并不是在一瞬间发生的，而是需要经过一步一步传递的过程，这个过程可以形象地理

解成激活一个神经网络。当前，深度学习的研究更多地依赖深度神经网络（Deep Neural Networks，简称 DNNs），亦称人造神经网络（Artificial Neural Networks，简称 ANNs）。深度神经网络中的神经元的输入和输出都是数值。一个神经元可以跟下一层的多个神经元同时相连，也可以跟上一层的多个神经元同时相连。具体来说，每个神经元将上一层神经元的输出进行聚合，再通过激活函数决定其最终的输出。如果这些聚集的输入信号足够强，那么这个激活函数将会“激活”（Activate）这个神经元，然后这个神经元会将一个有高数值的信号传递给下一层网络。相对地，如果输入信号不够强，那么一个低数值信号将被传递下去。

一个神经网络可以有任意多个神经元，而这些神经元彼此可以有很多随机的连接。但是为了运算更加容易，神经元往往是层层递进的。一般来说，一个神经网络至少会有两层：输入层和输出层（见图 1.1）。这个网络可以被公式 (1.1) 描述，它可以做一些简单的决定任务，比如帮助几个学生根据天气的情况具体决定他们是否外出踢足球，网络输出的 z 是一个分数，分数越高则代表越可以去踢足球。这个分数取决于三个因素：1）足球场的使用费用 x_1；2）天气 x_2；3）去球场的时间 x_3。如果天气对大家做这个决定比较重要，则其相对应的网络权重 w_2 会有较大的绝对值。同样地，那些对做这个决定影响较小的因素，所对应的网络权重的绝对值就会较小。如果一个权重被设置为零，那么它所对应的输入就对最终的结果完全没有影响。比如，有的学生有钱，不在乎足球场的费用，则 w_1 为 0。我们把具有这样结构的网络叫作单一层网络，也叫作**感知器**（Perceptron）。

$$z = w_1x_1 + w_2x_2 + w_3x_3 \tag{1.1}$$

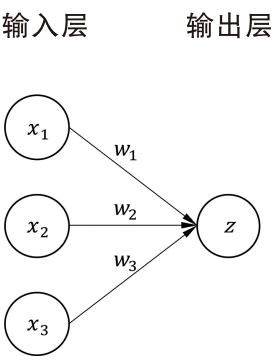

图 1.1　有三个输入神经元和一个输出神经元的神经网络

偏差与决策边界

偏差（Bias）是神经元所附带的一个额外的标量，用来偏移神经网络的输出。图 1.2 所示的一个有偏差 b 的单层神经网络可以用公式 (1.2) 表达：

$$z = w_1x_1 + w_2x_2 + w_3x_3 + b \tag{1.2}$$

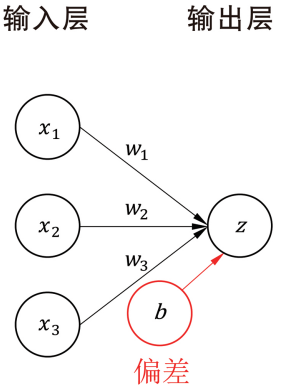

图 1.2 一个有偏差的单层神经网络

偏差可以帮助一个神经网络更好地学习数据。我们不妨定义以下二分类问题：对于输出 z，当且仅当 z 为正数，其所对应的标签 y 为 1，反之为 0：

$$y = \begin{cases} 1 & z > 0 \\ 0 & z \leqslant 0 \end{cases} \tag{1.3}$$

二分类任务的样本数据分布例子如图 1.3 所示。我们现在需要找到最符合这些数据的权重和偏差。我们把这些样本数据分成两个不同的类别的边界定义为决策边界。正式来说，这个边界是 $\{x_1, x_2, x_3 | w_1x_1 + w_2x_2 + w_3x_3 + b = 0\}$。

我们首先把这个问题简化到只有两个输入的情况下，即 $z = w_1x_1 + w_2x_2 + b$。如图 1.3 左所示，如果没有偏差值，也就是说 $b = 0$，那么决策边界必须穿过坐标系的原点（左下的线）。但是，这样很明显不符合数据的分布，因为我们的数据点都是在这个边界的一侧。如果偏差值不是 0，那么决策边界与两个轴的交点就为 $(0, -\frac{b}{w_2})$ 和 $(-\frac{b}{w_1}, 0)$。这样来看，如果我们的权重和偏差值选得好，那么决策边界就能更好地符合数据分布。

进一步来说，当一个神经元有三个输入的时候，$z = w_1x_1 + w_2x_2 + w_3x_3 + b$，此时的边界就会变成如图 1.3 右所示的平面。在一个如单层神经网络（见公式 (1.2)）的线性模型中，这样的一个平面也被称为**超平面**（Hyperplane）。

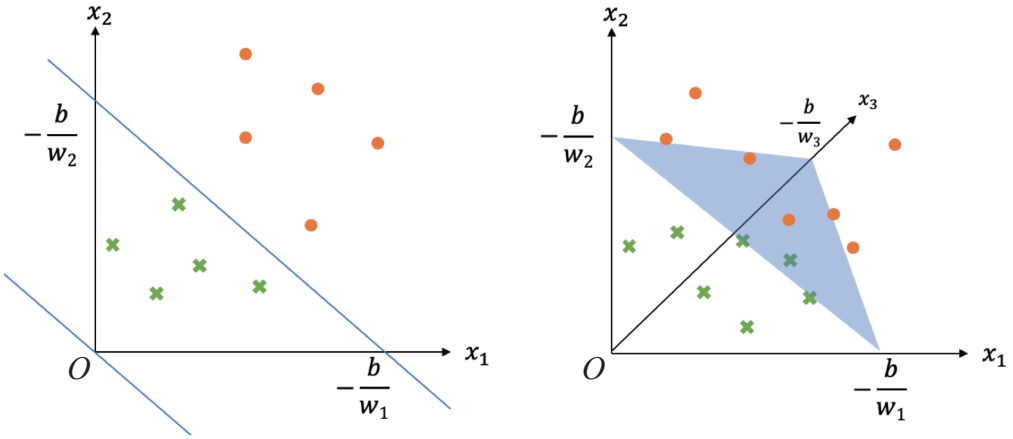

图 1.3　线性模型分别在两个输入和三个输入场景下的决策边界。左：$z = w_1x_1 + w_2x_2 + b$。右：$z = w_1x_1 + w_2x_2 + w_3x_3 + b$。若没有偏差，则决策边界必须经过原点，不能很好地分类

多输出

单层神经网络可以有多个神经元。图 1.4 展示了一个有两个输出神经元的单层网络，由公式 (1.4) 所得。因为每一个输出都和全部输入相连，所以输出层也被称为**密集层**（Dense Layer）或者**全连接层**（Fully-Connected (FC) Layer）：

$$\begin{aligned} z_1 &= w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1 \\ z_2 &= w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2 \end{aligned} \tag{1.4}$$

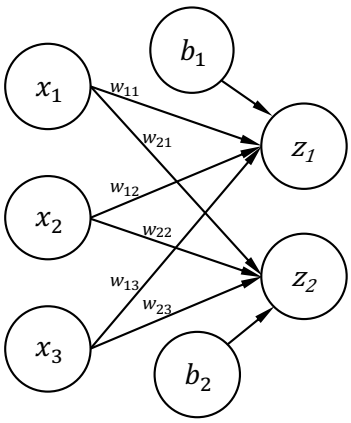

图 1.4　一个有三个输入和两个输出的神经元的神经网络

在实践中，全连接层也可以被矩阵乘法实现：

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \tag{1.5}$$

式中，$\boldsymbol{W} \in \mathbb{R}^{m \times n}$ 是用来表示权重的矩阵，$\boldsymbol{z} \in \mathbb{R}^m, \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{b} \in \mathbb{R}^m$ 分别用来表示输出、输入和偏差的向量。在公式 (1.5) 里的例子中，$m = 2$，$n = 3$，即 $\boldsymbol{W} \in \mathbb{R}^{2 \times 3}$。

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{1.6}$$

1.3 多层感知器

多层感知器（Multi-Layer Perceptron，MLP）(Rosenblatt, 1958; Ruck et al., 1990) 最初指至少有两个全连接层的网络。图 1.5 展现了一个有四个全连接层的多层感知器。那些在输入层和输出层中间的网络层被隐藏（Hidden）了，因为一般来说从网络外面是没有办法直接接触它们的，所以被统称为**隐藏层**（Hidden Layers）。相比只有一个全连接层的网络，MLP 可以从更复杂的数据中学习。从另外一个角度来看，MLP 的学习能力是大于单一层网络的学习能力的。但是拥有更多的隐藏层并不意味着一个网络会有更强的学习能力。通用近似定理说的是：一个有一层隐藏层的神经网络（类似于有一层隐藏层的 MLP）和任何可挤压的激活函数（见后文的 sigmoid 和 tanh）在这一层网络有足够多神经元的情况下，可以估算出任何博莱尔可测函数 (Goodfellow et al., 2016; Hornik et al., 1989; Samuel, 1959)。但是实际上，这样的网络可能会非常难以训练或者容易过拟合（Overfit）（见后文）。因为隐藏层非常大，所以一般的深度神经网络都会有几层隐藏层来降低训练难度。

为什么需要多层网络？为了回答这个问题，我们首先通过逻辑运算的几个例子来展示一个网络是怎么估算一个方程的。我们会考虑的逻辑运算有：与（AND）、或（OR）、同或（XNOR）、异或（XOR）、或非（NOR）、与非（NAND）。这些运算输入都是两个二进制数字，然后输出为 1 或者 0。如与（AND），只有两个输入同时为 1，AND 才会输出 1。这些简单的逻辑计算可以很容易就被感知器学习，就像公式 (1.7) 里展现的那样。

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{如果 } z > 0 \\ 0 & \text{其他情况} \end{cases}, \quad z = w_1 x_1 + w_2 x_2 + b \tag{1.7}$$

图 1.6 展示了被感知器定义的决策边界可以很轻松地把 AND、OR、NOR 和 NAND 运算的 0

和 1 分开出来，但是，XOR 或 XNOR 的决策边界是不可能被找到的。

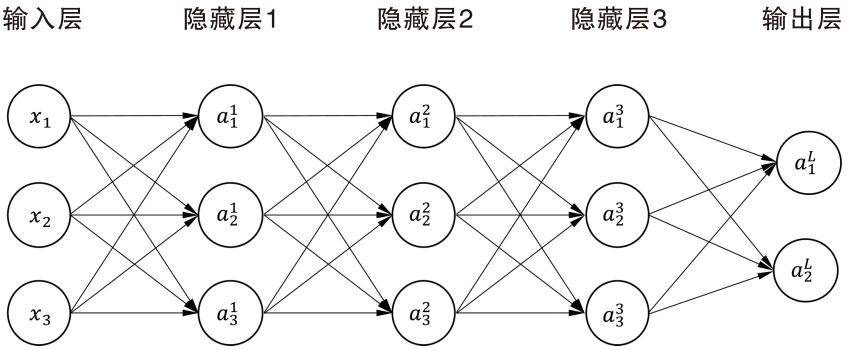

图 1.5　一个具有三个隐藏层和一个输出层的多层感知器。图中使用 a_i^l 表示神经元，其中 l 代表层的索引，i 代表输出的索引

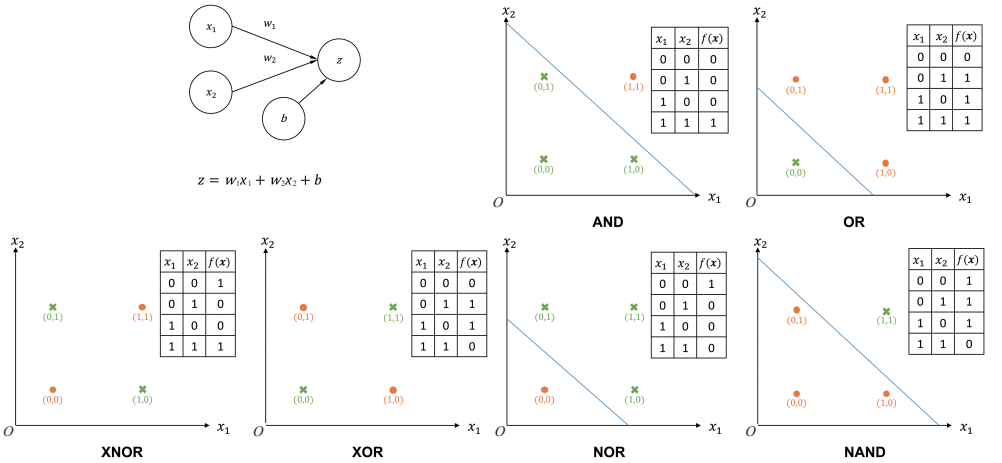

图 1.6　左上：有两个输入和一个输出的感知器。剩下的是：不同的用来把 0（×）和 1（●）分开的决策边界。在这个单层感知器中，能找到 AND、OR、NOR 和 NAND 的决策边界，但找不到可以实现 XOR 和 XNOR 的决策边界

因为我们不能用一个线性模型像单个感知器那样直接估算 XOR，所以必须要转化输入。图 1.7 展现了一个用有一层隐藏层的 MLP 去估算 XOR，这个 MLP 首先将通过估计 OR 和 NAND 运算把 x_1, x_2 转换到了一个新的空间，然后在这个转换过的空间里，这些点就可以被一条估算 AND 的平面分开了。这个被转换过后的空间也被称为特征空间。这个例子说明了怎么通过特征的学习来改善一个模型的学习能力。

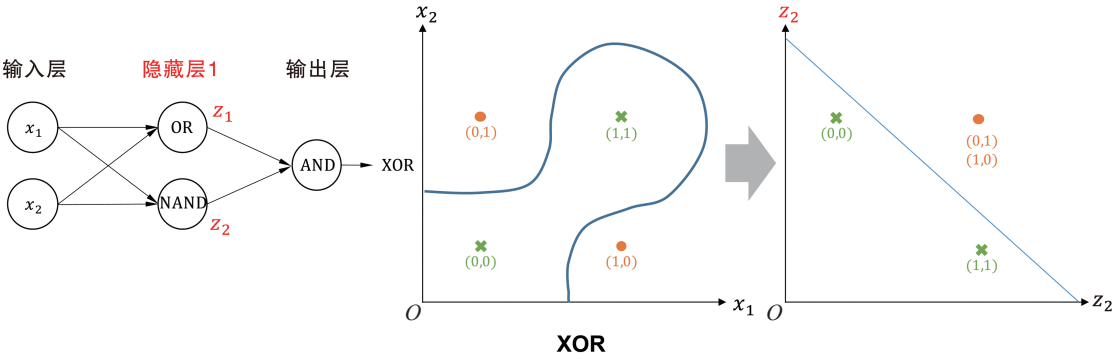

图 1.7 左：一个可以估算 XOR 的 MLP。中和右：把原始数据点转化到特征空间，从而使得这些数据点变得线性可分离

1.4 激活函数

矩阵的加减和乘除运算都是线性运算符，但是一个线性模型的学习能力还是相对有限的。举例来说，线性模型不能轻易地估算一个余弦函数。因为大多数深度神经网络解决的真实问题都不可能被简单地映射到一个线性转换，所以非线性在深度神经网络里至关重要。

实际上，深度学习网络的非线性是通过激活函数来介入的。这些激活函数都是针对每一个元素（Element-Wise）运算的。我们需要这些激活函数来帮助模型获得有任意数值的概率向量。激活函数的选择要根据具体的运用场景来考虑。虽然有一些激活函数在大多数的情况下效果都是不错的，但是在具体的实际运用中，可能还有更好的选择。所以激活函数的设计至今都还是一个活跃的研究方向。本节主要介绍四种非常常见的激活函数：sigmoid、tanh、ReLU 和 softmax。

逻辑函数 **sigmoid** 在作为激活函数时，将输入控制在了 0 和 1 之间，如公式 (1.8) 所示。sigmoid 方程可以在网络的最后一层，使用来做一些分类的任务，以代表 0%～100% 的概率。比如说，一个二维的分类器可以把 sigmoid 方程放在最后一层，来把其数值局限在 0 和 1 之中，然后我们可以用一个简单的临界值决定最终输出的标签是什么（0 或 1）。

$$f(z) = \frac{1}{1 + \mathrm{e}^{-z}} \tag{1.8}$$

与 sigmoid 函数类似的是，**hyperbolic tangent** （**tanh**）把输出值控制到了 -1 和 1 之间，就如公式 (1.9) 所定义那样。tanh 函数可以在隐藏层中使用来提高非线性 (Glorot et al., 2011)。它也可以在输出层中使用，比如网络可以输出像素数值在 -1 和 1 的图像。

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

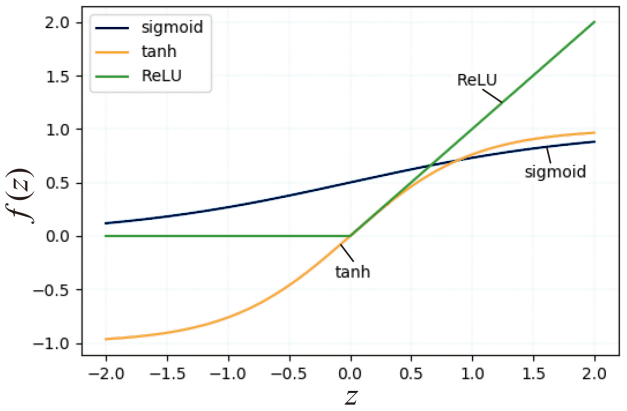

图 1.8　展现三个元素单位运用的方程：sigmoid、tanh 和 ReLU。sigmoid 把数值限制在了 0 和 1 之间，而 tanh 则把数值限制在了 −1 和 1 之间。当输入是负数时，ReLU 则输出 0，但当输入是正数时，其输出等于输入

在公式 (1.10) 中，我们定义了**整流线性单元**（Rectified Linear Unit，ReLU）函数，也叫作 rectifier。ReLU 被广泛地使用于不同的研究当中 (Cao et al., 2017; He et al., 2016; Noh et al., 2015)，在很多层的网络中 ReLU 通常会比 sigmoid 和 tanh 性能更好 (Glorot et al., 2011)。

$$f(z) = \begin{cases} 0 & \text{当 } z \leqslant 0 \\ z & \text{当 } z > 0 \end{cases} \tag{1.10}$$

在实际运用中，ReLU 有以下优势。

- 更易实现和计算：在实现 ReLU 的过程中，首先我们只需要把其数值和 0 做对比，然后根据结果来设定输出是 0 还是 z。而我们在实现 sigmoid 和 tanh 的过程当中，指数函数在大型网络中会更难以计算。
- 网络更好优化：ReLU 接近于线性，因为它是由两个线性函数组成的。这种性质就使得它更容易被优化，我们在本章后面讲解优化细节时再讨论。

然而 ReLU 把负数变成 0，可能会导致输出中信息的丧失。这可能是因为一个不合适的学习速率或者负的偏差而导致的。带泄漏的（Leaky）ReLU 则解决了这个问题 (Xu et al., 2015)。我们在公式 (1.11) 中对它进行了定义。标量 α 是一个较小的正数来控制斜率，使得来自负区间的信息也可以被保留下来。

$$f(z) = \begin{cases} \alpha z & \text{当 } z \leqslant 0 \\ z & \text{当 } z > 0 \end{cases} \tag{1.11}$$

有参数的 ReLU（PReLU）(He et al., 2015) 和 Leaky ReLU 很近似，它把 α 看作一个可以训练的参数。目前我们还没有具体的证据表明 ReLU、Leaky ReLU 或 PReLU 哪个是最好的，它们在不同应用中往往有不同的效果。

不像上述的其他激活函数，在公式 (1.12) 中定义的 **softmax** 函数会根据前一层网络的输出提供归一化。softmax 函数首先计算指数函数 $\mathrm{e}^{\boldsymbol{z}}$，然后每一项都除以这个值进行归一。

$$f(\boldsymbol{z})_i = \frac{\mathrm{e}^{z_i}}{\sum_{k=1}^{K} \mathrm{e}^{z_k}} \tag{1.12}$$

在实际运用当中，softmax 函数只在最后的输出层用来归一输出向量 $\boldsymbol{z}$，使其变成一个概率向量。这个概率向量的每一个值都为非负数，然后它们的和最终会为 1。所以，softmax 函数在多分类任务中被广泛使用，用以输出不同类别的概率。

1.5 损失函数

到目前为止，我们了解了神经网络结构的基础知识，那么网络的参数是怎么自动学习出来的呢？这需要**损失函数**（Loss Function）来引导。具体来说，损失函数通常被定义为一种计算误差的量化方式，也就是计算网络输出的预测值和目标值之间的损失值或者代价大小。损失值被用来作为优化神经网络参数的目标，我们优化的参数包括权重和偏差等。在本节里，我们会介绍一些基本的损失函数，1.6 节会介绍如何使用损失函数优化网络参数。

交叉熵损失

在介绍交叉熵损失之前，首先来看一个类似的概念：Kullback-Leibler (KL) 散度，其作用是衡量两个分布 $P(x)$ 和 $Q(x)$ 的相似度：

$$D_{\mathrm{KL}}(P\|Q) = \mathbb{E}_{x\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{x\sim P}[\log P(x) - \log Q(x)] \tag{1.13}$$

KL 散度是一个非负的指标，并且只有在 P 和 Q 两个分布一样时才取值为 0。因为 KL 散度的第一个项和 Q 没有关系，我们引入交叉熵的概念并把公式的第一项移除。

$$H(P,Q) = -\mathbb{E}_{x\sim P} \log Q(x) \tag{1.14}$$

因此，通过 Q 来最小化交叉熵就等同于最小化 KL 散度。在多类别分类任务中，深度神经网络通过 softmax 函数输出的是不同类别概率的分布，而不是直接输出一个样本属于的类别。所以，我们可以用交叉熵来测量预测分布有多好，从而训练网络。

以一个二分类任务为例。在二分类中，每一个数据样本 x_i 都有一个对应的标签 y_i（0 或 1）。

一个模型需要预测样本是 0 或者 1 的概率，用 $\hat{y}_{i,1}$，$\hat{y}_{i,2}$ 来表示。因为 $\hat{y}_{i,1}+\hat{y}_{i,2}=1$，可以把它们改写为 $\hat{y}_i$ 和 $1-\hat{y}_i$。前者可以代表一个类别的概率，后者可以代表另外一个类别的概率。因此，一个二分类的神经网络可以只有一个输出，且最后一层使用 sigmoid。根据交叉熵的定义，我们有：

$$\mathcal{L}=-\frac{1}{N}\sum_{i=1}^{N}\Big(y_i\log\hat{y}_i+(1-y_i)\log(1-\hat{y}_i)\Big) \tag{1.15}$$

式中，N 代表了总数据样本的大小。因为 y_i 是一个 1 或者 0 的值，因此在 $y_i\log\hat{y}_i$ 和 $(1-y_i)\log(1-\hat{y}_i)$ 中，对于每一个新样本，两个表达式的值只有一个不为零。若 $\forall i, y_i=\hat{y}_i$，则交叉熵就为 0。

在多类别分类任务中，每一个样本 x_i 都会被分到 3 个或者更多的类别中的一个。这时，一个模型需预测每一个类别的概率 $\{\hat{y}_{i,1},\hat{y}_{i,2},\cdots,\hat{y}_{i,M}\}$，且符合条件 $M\geqslant 3$ 和 $\sum_{j=1}^{M}\hat{y}_{i,j}=1$。在这里，每一个样本的目标写作 c_i，它的值域为 $[1,M]$。同时，它也可以被转换成为一个独热编码 $\boldsymbol{y}_i=[y_{i,1},y_{i,2},\cdots,y_{i,M}]$，其中只有 $y_{i,c_i}=1$，其他的都是 0。我们现在就可以把多类别分类的交叉熵写成以下形式：

$$\begin{aligned}\mathcal{L}&=-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}y_{i,j}\log\hat{y}_{i,j}=-\frac{1}{N}\sum_{i=1}^{N}(0+\cdots+y_{i,c_i}\log\hat{y}_{i,c_i}+\cdots+0)\\&=-\frac{1}{N}\sum_{i=1}^{N}\log\hat{y}_{i,c_i}\end{aligned} \tag{1.16}$$

$\mathcal{L}_p$ 范式

向量 $\boldsymbol{x}$ 的 p-范式用来测量其数值幅度大小：如果一个向量的值更大，它的 p-范式也会有一个更大的值。p 是一个大于或等于 1 的值，p-范式定义为

$$\begin{aligned}\|\boldsymbol{x}\|_p&=\left(\sum_{i=1}^{N}|x_i|^p\right)^{1/p}\\\text{i.e., }\|\boldsymbol{x}\|_p^p&=\sum_{i=1}^{N}|x_i|^p\end{aligned} \tag{1.17}$$

p-范式在深度学习中往往用来测量两个向量的差别大小，写作 $\mathcal{L}_p$，如在公式 (1.18) 一样，其中 $\boldsymbol{y}$ 为目标值向量，$\hat{\boldsymbol{y}}$ 为预测值向量。

$$\mathcal{L}_p = \|\boldsymbol{y} - \hat{\boldsymbol{y}}\|_p^p = \sum_{i=1}^{N} |y_i - \hat{y}_i|^p \tag{1.18}$$

均方误差

均方误差 (Mean Squared Error，MSE) 是由公式 (1.19) 所定义的 $\mathcal{L}_2$ 范式的平均值。均方误差可以在网络输出是连续值的回归问题中使用。比如说，两个不同图像在像素上的区别就可以用 MSE 来测量：

$$\mathcal{L} = \frac{1}{N}\|\boldsymbol{y} - \hat{\boldsymbol{y}}\|_2^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{1.19}$$

其中 N 是样本数据的大小，$\boldsymbol{y}$ 和 $\hat{\boldsymbol{y}}$ 分别为目标值向量和预测值向量。

平均绝对误差

与均方误差类似，平均绝对误差 (Mean Absolute Error，MAE) 也可以被用来做回归任务，它被定义为 $\mathcal{L}_1$ 范式的平均。

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{1.20}$$

均方误差和平均绝对误差都可衡量 $\boldsymbol{y}$ 和 $\hat{\boldsymbol{y}}$ 的误差，用以优化网络模型。其中，均方误差提供了更好的数学性质，从而让我们能更简便地计算梯度下降所需要的偏导数。而在平均绝对误差中，当 $y_i = \hat{y}_i$ 时，我们注意到上面公式中的绝对值项无法求导，这对平均绝对误差来说是一个无法解决且需要规避的问题。另外，当 y_i 和 $\hat{y}_i$ 的绝对差大于 1 时，均方误差相对平均绝对误差来说误差值更大。显然地，当 $(y_i - \hat{y}_i) > 1$ 时，$(y_i - \hat{y}_i)^2 > |y_i - \hat{y}_i|$。

1.6 优化

在这一小节里，我们将描述深度神经网络的优化，即深度神经网络参数训练。本节包含了反向传播算法、梯度下降、随机梯度下降和超参数的选择等内容。

1.6.1 梯度下降和误差的反向传播

如果我们有一个神经网络和一个损失函数，那么对于这个网络的训练的意义是通过学习它的 $\boldsymbol{\theta}$ 使得损失值 $\mathcal{L}$ 最小化。最暴力的方法是通过寻找一组参数 $\boldsymbol{\theta}$，使它满足 $\bigtriangledown_{\boldsymbol{\theta}}\mathcal{L} = 0$，以找到损失值的最小值。但这种方法在实际中很难实现，因为通常深度神经网络参数很多、非常复杂。所以

我们需要考虑一种叫作**梯度下降**（Gradient Descent）的方法，它是通过逐步优化来一步一步地寻找更好的参数来降低损失值的。

图 1.9 展示了两个梯度下降的例子。梯度下降的学习过程从一个随机指定的参数开始，其损失值 $\mathcal{L}$ 随参数的更新而逐步下降，其过程如箭头所示。具体来说，在神经网络中，参数通过偏导数 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ 被逐步优化，优化过程为 $\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$，其中 α 为学习率，用以控制步长幅度。可见，梯度下降法的关键是计算出偏导数 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$。

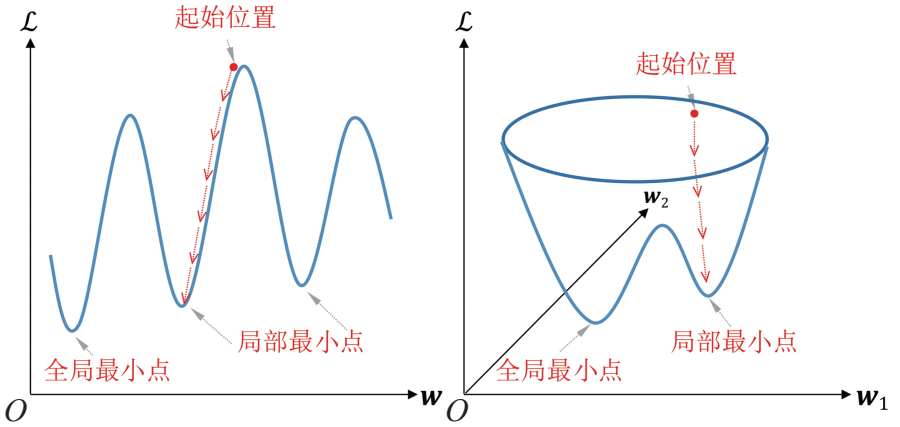

图 1.9　梯度下降的示例：在左图中，我们有一个可以训练的参数 $\boldsymbol{\theta} = w$；在右图中，我们有两个可以训练的参数 $\boldsymbol{\theta} = [w_1, w_2]$。在梯度下降里，整个学习过程的初始化参数是随机的。在每一步对参数调整之后，损失 $\mathcal{L}$ 会慢慢地减少，但无法保证最后能找到全局最小的损失值，在大多数情况下，我们能找到的都是局部最小值

反向传播（Back-Propagation）(LeCun et al., 2015; Rumelhart et al., 1986) 是一种计算神经网络中偏导数 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ 的方法。为了使得表示对 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ 的计算更加清晰，这种方法引入一个中间量 $\boldsymbol{\delta} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}$，用来表示损失函数 $\mathcal{L}$ 对于神经网络输出 $\boldsymbol{z}$ 的偏导数。因此，这种方法可以通过中间量 $\boldsymbol{\delta}$ 来计算损失函数 $\mathcal{L}$ 对于每个参数的偏导数，并最终共同组成 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$。

网络层的序号为 $l = 1, 2, \cdots, L$，其中输出层的序号为 L。对于每个网络层，我们有输出 $\boldsymbol{z}^l$，中间值 $\boldsymbol{\delta}^l = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l}$ 和一个激活值输出 $\boldsymbol{a}^l = f(\boldsymbol{z}^l)$（其中 f 为激活函数）。下面是一个使用均方误差和 sigmoid 激活函数的多层感知器的例子：已知 $\boldsymbol{z}^l = \boldsymbol{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$, $\boldsymbol{a}^l = f(\boldsymbol{z}^l) = \frac{1}{1+\mathrm{e}^{-\boldsymbol{z}^l}}$ 和 $\mathcal{L} = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{a}^L\|_2^2$，可以得出激活值输出对于原先输出的偏导数 $\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} = f'(\boldsymbol{z}^l) = f(\boldsymbol{z}^l)(1 - f(\boldsymbol{z}^l)) = \boldsymbol{a}^l(1 - \boldsymbol{a}^l)$，以及损失函数对于激活值输出的偏导数 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^L} = (\boldsymbol{a}^L - \boldsymbol{y})$。然后，为了计算损失函数对于输出层的偏导数，可以使用链式法则，具体如下：

从输出层开始向后传播误差，先计算输出层的中间量：

- $\boldsymbol{\delta}^L = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^L} \frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} = (\boldsymbol{a}^L - \boldsymbol{y}) \odot (\boldsymbol{a}^L(1 - \boldsymbol{a}^L))$

然后计算损失函数对于后一层输出的偏导数，如（$l = 1, 2, \cdots, L-1$）：

- 已知 $\boldsymbol{z}^{l+1} = \boldsymbol{W}^{l+1}\boldsymbol{a}^l + \boldsymbol{b}^{l+1}$，则 $\frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{a}^l} = \boldsymbol{W}^{l+1}$；且 $\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} = \boldsymbol{a}^l(1 - \boldsymbol{a}^l)$

- 那么 $\boldsymbol{\delta}^l = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{l+1}} \frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{a}^l} \frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} = (\boldsymbol{W}^{l+1})^{\mathrm{T}} \boldsymbol{\delta}^{l+1} \odot (\boldsymbol{a}^l(1-\boldsymbol{a}^l))$

从输出层开始向后传播，计算出所有层的中间值 $\boldsymbol{\delta}^l$ 后，反向传播算法的第二步是在中间值 $\boldsymbol{\delta}^l$ 的基础上计算损失函数对于每层参数 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l}$ 和 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^l}$ 的偏导数。

- 若有 $\boldsymbol{z}^l = \boldsymbol{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$，我们有 $\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{W}^l} = \boldsymbol{a}^{l-1}$ 和 $\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{b}^l} = 1$
- 那么 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l} \frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{W}^l} = \boldsymbol{\delta}^l (\boldsymbol{a}^{l-1})^{\mathrm{T}}$，$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l} \frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{b}^l} = \boldsymbol{\delta}^l$

最后，我们用 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l}$ 和 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^l}$ 及梯度下降更新 $\boldsymbol{W}^l$ 和 $\boldsymbol{b}^l$：

- $\boldsymbol{W}^l := \boldsymbol{W}^l - \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l}$
- $\boldsymbol{b}^l := \boldsymbol{b}^l - \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^l}$

可见，有了偏导数 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = [\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l}, \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^l}]$，我们可以使用梯度下降来对参数进行迭代，直到其收敛到了损失函数中的一个最小值，如图 1.9 所示。在实践中，我们最终得到的最小值往往是一个局部最小值，而不是全局最小值。但是，因为深度神经网络往往可以提供一个很强的表示能力，这些局部最小值通常会很接近全局最小值 (Goodfellow et al., 2016)，使得损失值足够小。

这里额外介绍 sigmoid 的问题，当使用 sigmoid 时，$\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} = \boldsymbol{a}^l(1-\boldsymbol{a}^l)$，当 $\boldsymbol{a}$ 接近于 0 或者 1 时，$\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l}$ 会非常小，从而导致 $\boldsymbol{\delta}^l$ 非常小。在网络很深的情况下，反向传播时 $\boldsymbol{\delta}$ 会越来越小，出现**梯度消失**（Vanishing Gradient）问题，导致模型靠近输入部分的参数很难被更新，模型无法训练起来。而 ReLU 的 $\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l}$ 在 $\boldsymbol{a}$ 大于 0 时衡为 1，就不会有这个问题，这也是现在的深度模型往往在隐藏层中使用 ReLU 而不再使用 sigmoid 的原因。

在梯度下降中，如果数据集的大小（即数据样本的数量）N 较大，则在每个迭代中计算损失函数 $\mathcal{L}$ 的计算开销可能会较高。拿之前的均方误差举例，我们可以把上式展开成

$$\mathcal{L} = \frac{1}{2} \|\boldsymbol{y} - \boldsymbol{a}^L\|_2^2 = \frac{1}{2} \sum_{i=1}^{N} (y_i - a_i^L)^2 \tag{1.21}$$

在实践中，数据集很有可能会很大，梯度下降因需要计算 $\mathcal{L}$ 而变得十分低效。随机梯度下降应运而生，其他对于 $\mathcal{L}$ 的计算只包含少量的数据样本。

1.6.2 随机梯度下降和自适应学习率

与其是在每个迭代中对全部训练数据计算损失函数 $\mathcal{L}$，**随机梯度下降**（Stochastic Gradient Descent, SGD）(Bottou et al., 2007) 计算损失值时随机选取一小部分的训练样本。这些小样本被称为**小批量** (Mini-batch)，而在这些小批量的具体大小被称为**批大小** (Batch Size) B。然后，我们就可以用批大小 B 和 $B \ll N$ 重写公式 (1.21)，得到公式 (1.22)，以改进计算 $\mathcal{L}$ 的效率：

$$\mathcal{L} = \frac{1}{2} \|\boldsymbol{y} - \boldsymbol{a}^L\|_2^2 = \frac{1}{2} \sum_{i=1}^{B} (y_i - a_i^L)^2 \tag{1.22}$$

随机梯度下降的训练过程请见算法 1.1。如果参数在算法 1.1 中更新了足够多的次数，那么小批量可以覆盖整个训练集。

算法 1.1 随机梯度下降的训练过程

Input: 参数 $\boldsymbol{\theta}$，学习率 α，训练步数/迭代次数 S

1: **for** $i = 0$ **to** S **do**
2: 　计算一个小批量的 $\mathcal{L}$
3: 　通过反向传播计算 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$
4: 　$\nabla \boldsymbol{\theta} \leftarrow -\alpha \cdot \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$;
5: 　$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla \boldsymbol{\theta}$ 更新参数
6: **end for**
7: **return** $\boldsymbol{\theta}$；返回训练好的参数

学习率 (Learning Rate) 控制了随机梯度下降中每次更新的步长。如果学习率过大，随机梯度下降可能无法找到最小值，如图 1.10 所示。另一方面，如果学习率过小，随机梯度下降的收敛速率将会变得十分缓慢。如何决定学习率是一个很困难的过程。为了解决这个问题，需要使用自适应学习率算法，如 Adam (Kingma et al., 2014)、RMSProp (Tieleman et al., 2017) 和 Adagrad (Duchi et al., 2011) 等。其作用为通过自动、自适应的方法来调整学习率，从而加速训练算法的收敛速度。这些算法的原理在于，当参数收到了一个较小的梯度时，算法会转到一个更大的步长；反之，如果梯度过大，算法就会给出一个较小的步长。其中，Adam 是最常见的自适应学习率算法。与其直接用梯度更新参数，Adam 首先会计算梯度的滑动平均和二阶动量。然后，如算法 1.2 所示，这些新计算的数值会被用来更新我们想要训练的参数。算法 1.2 中的 β_1 和 β_2 为梯度的遗忘因子，或者分别是其动量和二阶动量。在默认设置下，β_1 和 β_2 的值分别是 0.9 和 0.999 (Kingma et al., 2014)。

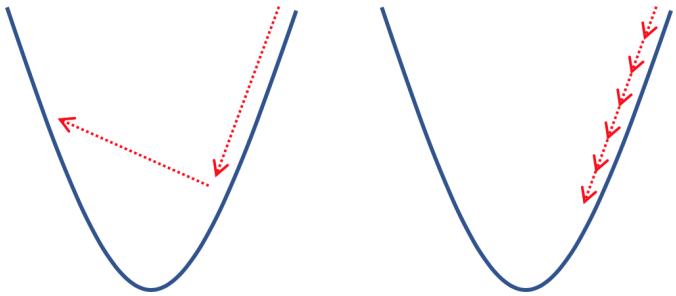

图 1.10　一个很大的学习率可能会加速训练过程，但会导致模型很难训练至一个理想的参数。如左图所示，因为其学习率较右图更大，其损失函数有可能在参数更新后增加，因此更难以接近最小值。同样地，右图的优化有一个更小的学习率，能更好地找到低点，但训练速度较慢

算法 1.2 Adam 优化器的训练过程

Input: 参数 $\boldsymbol{\theta}$，学习率 α，训练步数/迭代次数 S，$\beta_1 = 0.9$，$\beta_2 = 0.999$，$\epsilon = 10^{-8}$

$\boldsymbol{m}_0 \leftarrow 0$; 初始化一阶动量
$\boldsymbol{v}_0 \leftarrow 0$; 初始化二阶动量
for $t = 1$ **to** S **do**
  $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$; 用一个随机的小批量计算梯度
  $\boldsymbol{m}_t \leftarrow \beta_1 \cdot \boldsymbol{m}_{t-1} + (1-\beta_1) \cdot \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$; 更新一阶动量
  $\boldsymbol{v}_t \leftarrow \beta_2 \cdot \boldsymbol{v}_{t-1} + (1-\beta_2) \cdot (\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}})^2$; 更新二阶动量
  $\hat{\boldsymbol{m}}_t \leftarrow \frac{\boldsymbol{m}_t}{1-\beta_1^t}$; 计算一阶动量的滑动平均
  $\hat{\boldsymbol{v}}_t \leftarrow \frac{\boldsymbol{v}_t}{1-\beta_2^t}$; 计算二阶动量的滑动平均
  $\nabla \boldsymbol{\theta} \leftarrow -\alpha \cdot \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t + \epsilon}}$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla \boldsymbol{\theta}$; 更新参数
end for
return $\boldsymbol{\theta}$; 返回训练好的参数

1.6.3 超参数筛选

在深度学习中，**超参数**（Hyper-Parameters）指和设置相关的参数，比如层的数量，以及训练过程的设置参数，如更新步的数量、批大小和学习率。这些设置参数会在很大程度上影响模型的表现，因此它们是组成一个理想模型的重要因素。

为了衡量不同超参数对于模型表现的影响，我们通常将数据集划分为训练集（Training Set）、验证集（Validation Set）和测试集（Testing Set）。不同的超参数设置分别用训练集训练出不同的模型，然后在验证集上进行性能评估。最后，我们用在验证集上表现最好的超参数在测试集上做最后的性能评估。在这里需要注意的是，我们不能用测试集调整超参数，不然就是已知考卷试题的作弊行为。

交叉验证

在一个小数据集上，把数据集分为训练集、验证集和测试集的做法会浪费宝贵的数据。具体来说，如果训练集分得过小，可能会因为训练数据不足而让训练出来的模型表现不佳。从另一方面来说，如果训练集分得过多、验证集过小，模型也不能在一个小数据集上被充分地评估。为了解决这个问题，可使用**交叉验证**（Cross Validation），所有数据都能被用来训练模型，不再需要验证集，以充分利用数据。

在一个 k 折交叉验证策略中，一个数据集将会被分成 k 个互相不重复的子集，并且每个子集包含同样数量的数据。我们将重复训练模型 k 次，其中每次训练时，一个子集将会被选为测试集，而剩下的数据将会被用来训练模型。最后用来评估的结果则是：k 次训练后，模型输出性能（如准确度）的平均值。图 1.11 展示了一个四折交叉验证示例。

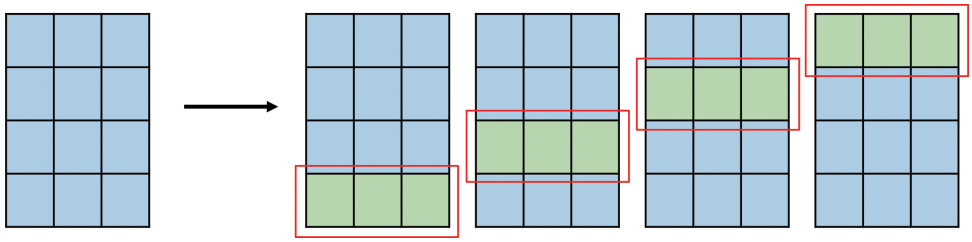

图 1.11　四折交叉验证（Four-Fold Cross-Validation）示例。数据集被划分为四个子集（为了展示目的，每一行为一个子集）。在每次训练中，而加框的子集被当作测试数据，其他被当作训练数据。最后模型评估的结果则是四次训练预测的平均

1.7　正则化

我们把那些用来使得一个模型在训练集和测试集都有很好效果的方法叫作正则化办法。本节主要介绍过拟合和一些不同的正则化方法，如权重衰减、Dropout 和批标准化。

1.7.1　过拟合

一个机器学习的模型为了减少训练集上的损失而进行的优化，并不能保证它在测试集上的效果良好。一个被过度优化了的模型会有很小的训练集误差，但有很大的测试集误差，这种现象为**过拟合**（Overfitting）。

图 1.12 中，虚线代表的多项式模型就存在过拟合的问题。这个模型在训练集上过度一致，而在测试集上就不太符合。当使用一个这样过拟合的模型在现实应用中应用新的数据时，是不可靠的。相反地，由实线代表的线性模型虽有很少的参数，但是却更符合测试数据的趋势。

和过拟合相对的是**欠拟合**（Underfitting），即模型在训练集和测试集上都有了很大的误差。但是在现实中，欠拟合很容易解决，比如可以用一个更大的模型来解决（更多网络层及更多的参数等），而解决过拟合会更加棘手。最简单的一个方法就是使用更多的训练数据，但这不是一个万能药，因为数据的获取和标记都需要代价。

1.7.2　权重衰减

权重衰减（Weight Decay）是一种简单却有效的用于解决过拟合的正则化方法。它用了一个正则项作为惩戒，使得 $\boldsymbol{\theta}$ 有更小的绝对值。以图 1.12 为例，如果多项式模型从 c 到 h 的参数有更小的绝对值，那这个模型的上下摇摆幅度就会减小，能更好地拟合数据。用参数范式作为惩戒的损失函数的定义为

$$\mathcal{L}_{\text{total}} = \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) + \lambda \Omega(\boldsymbol{\theta}) \tag{1.23}$$

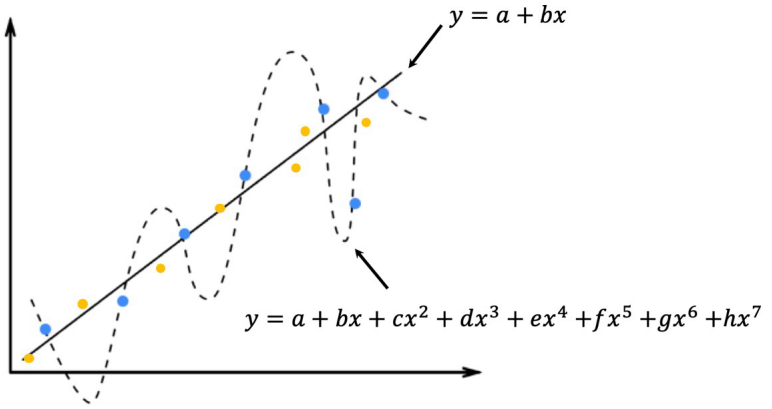

图 1.12 一个过拟合的例子：深色点代表了训练集，浅色点代表了测试集。虽然由实线代表的线性模型在训练集上有一个更大的损失值，但实线的模型比虚线代表的多项式模型在测试集上误差更小。我们可以说这个多项式模型对训练集过拟合了

其中 $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ 是从使用目标 $\boldsymbol{y}$ 和预测 $\hat{\boldsymbol{y}}$ 来计算的损失函数，Ω 是模型的参数范式惩戒函数，λ 是有比较小的值，以控制参数范式惩戒函数的幅度。

两种最常见的参数范式惩戒函数是 $\mathcal{L}_1 = \|\boldsymbol{W}\|$ 和 $\mathcal{L}_2 = \|\boldsymbol{W}\|_2^2$。深度神经网络的参数的绝对值通常小于 1，所以 $\mathcal{L}_1$ 会比 $\mathcal{L}_2$ 输出一个更大的惩戒，因为当 $|w| < 1$ 时，$|w| > w^2$。可见，$\mathcal{L}_1$ 函数用来作为参数范式惩戒函数时，会让参数偏向于更小的值甚至为 0。这是模型隐性地选择特征的方法，把那些不重要特征的相对应参数设为一个很小的值或者是 0。

我们可以进一步通过几何方法来看看 $\mathcal{L}_1$ 和 $\mathcal{L}_2$ 的区别。由图 1.13 所示的坐标系里，有两个模型参数 w_1, w_2。${w_1}^2 + {w_2}^2 = r^2$ 是一个半径为 r 的圆（图 1.13 左）而 $|w_1| + |w_2| = r$ 是一个对

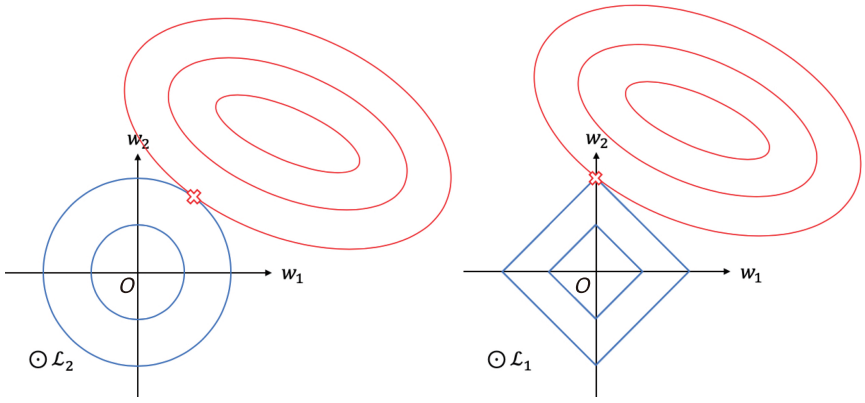

图 1.13 左图：原始损失值的轮廓线（红色）还有 $\mathcal{L}_2$ 损失值（蓝色）。右图：原始损失值的轮廓线（红色）还有 $\mathcal{L}_1$ 损失值（蓝色）。从红色轮廓线和蓝色轮廓线交接的地方可见，$\mathcal{L}_1$ 更有可能使得参数为 0

角线长为 $2r$ 的正方形（图 1.13 右）。它们两个都被蓝色轮廓线表示。在图中，红色的线代表的是初始的损失 $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$。初始损失和参数范式惩戒的交点用“叉”标记了出来。$\mathcal{L}_1$ 更有可能使得参数为 0，两个轮廓的交接位于正方形的顶点上。

1.7.3 Dropout

Dropout 是另一个很受欢迎的用来解决过拟合问题方法 (Hinton et al., 2012; Srivastava et al., 2014)。当神经元数量非常多时，网络会出现共适应的问题，从而会有过拟合的现象。神经元的共适应指神经元之间会互相依赖。故而，造成一旦有一个神经元失效了，就有可能所有依赖它的神经元都会失效，以至于整个网络瘫痪的局面。为了避免参数过多导致的共适应，Dropout 在训练的过程中，将隐藏层的输出按比例随机设为 0。就像图 1.14 中所示一样，每一层会有几个神经元随机地失去和其他层的连接。

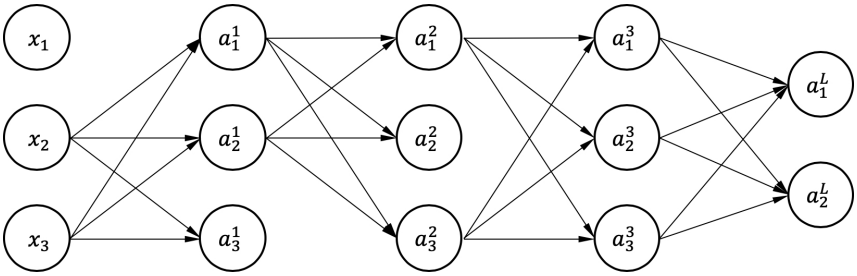

图 1.14　训练过程中对一个神经网络使用 Dropout，让它的某些连接消失

在反向传播当中，如果有的输出 $\boldsymbol{a}^l$ 为 0，那么其相对应的那一层的偏导数 $\boldsymbol{\delta}^l$ 也是 0。只有还有连接的神经元会被更新。所以 Dropout 法其实是在训练很多不同的小的网络，且共用参数 (Hinton et al., 2012)。在测试过程当中，Dropout 就不能被使用了，没有输出会被设为 0。这就意味着是所有网络一起来预测最终的结果。**集成学习**（Ensemble Learning）就是这样一个例子 (Hara et al., 2016)，它用很多模型学会做同一个任务，然后测试的时候使用所有模型输出的结果来提高准确性。关于 Dropout 的理论证明在原始的论文里是没有的 (Hinton et al., 2012)，但是近期有了些新的结果，比如说 (Hara et al., 2016) 就证明了它在集成学习里的有效性，以及 (Gal et al., 2016) 证明了它在贝叶斯里的有效性。

1.7.4 批标准化

批标准化（Batch Normalization）(Ioffe et al., 2015) 层标准化了网络的输出，也就是让输出的平均值变为 0，方差变为 1。这样做的目的是提高训练的稳定性。在训练的过程中，批标准化层会用一个移动平均的办法来计算每一批输入的平均值和方差，以估计整个训练集的平均值和方差。

每一批输入的平均值和方差会被用来标准化这一批输入。在模型测试的过程当中，我们会保持移动平均值和方差不变来标准化输入。

除了提高性能和稳定性，批标准化也可以提升正则化的作用。和 Dropout 里对隐藏层加一个不确定性一样，移动平均值和方差也同样地引入了一定的随机性，因为在每一个回合当中，它都是根据具体的那一批的随机样本来决定更新的。因此，在训练中有了这样一个变化的神经网络会变得更加鲁棒。

1.7.5 其他缓和过拟合的方法

我们有很多其他方法来预防过拟合，比如说，**早停法**（Early Stopping）或者**数据增强**（Data Augmentation）。早停法会当网络在满足一定的实际条件时停止训练，比如说在验证集上有了足够高的精确度。图 1.16 描述了损失在训练过程可能会慢慢增加，也就是过拟合的开始，不过我们可以用早停法在过拟合开始前的那个点停止训练。

图 1.15 一个图像数据增强的例子。左上角的是原图，其他图片是通过随机的反转、平移、缩近等运算得到的

数据增强即增加现有训练数据的大小，如运用反转、旋转、移动和放缩等运算合理生成数据，以减少过拟合，从而提高网络性能 (Dong et al., 2017; He et al., 2016; Howard et al., 2017; Simonyan et al., 2015)。和图像数据一样，音频数据也一样可以通过增加噪声或者其他改变来增强。最近研究表明，通过改变音频速度来增强，可以提高语音识别算法的性能 (Ko et al., 2015)。

但是我们不能把同样的方法运用在字符信息上面，因为字符的大小和排序有它特定的意思。比如说，“人类喜欢狗狗”和“狗狗喜欢人类”的意思是不一样的。一个可以增强字符数据的现实方法是用规定的同义词来复述句子 (Zhang et al., 2015)，也可以不增强原始数据，文献 (Reed et al., 2016) 利用两个随机句子的向量表征的内插来进行数据增强。

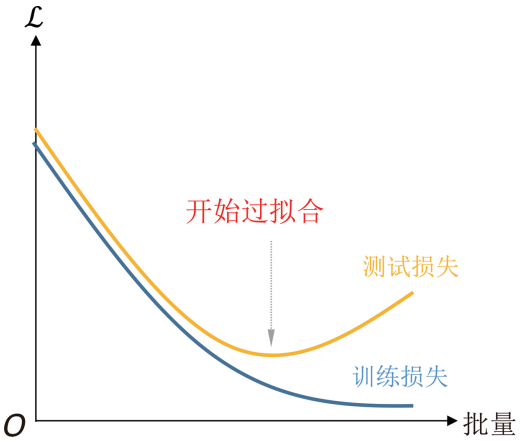

图 1.16　过拟合的训练曲线。我们可以用早停法来让训练过程在开始过拟合之时就停止

1.8　卷积神经网络

卷积神经网络（Convolutional Neural Network，CNN）(LeCun et al., 1989) 是前向神经网络的一种，它在很多不同的领域里都有很大的作用，如计算机视觉 (He et al., 2016; Krizhevsky et al., 2012; Simonyan et al., 2015)、时序预测 (van den Oord et al., 2016)、自然语言处理 (Yin et al., 2017; Zhang et al., 2019a) 和强化学习 (James et al., 2019; Rusu et al., 2016)。很多已经在现实世界落地的机器学习系统都是基于 CNN 之上的。本节介绍两种网络层：卷积层和池化层，它们都是 CNN 结构的一部分。

卷积层可能是 CNN 最有识别度的一个特征。其主要思想来自对人脑中并排处理视觉输入的学习。和图 1.17 所示的一样，卷积输出使用了四个不同的神经元来处理同样的输入图像区间。不同的神经元可能负责的处理任务不一样，如处理边缘、颜色和角度等任务。在卷积层的神经元只是和局部有连接，并不是和前一层的所有单元都有连接。卷积层可以被层层地叠加在一起，也就是说，一个卷积层的输出可以作为另外一个卷积层的输入。卷积层最大的优点是，相对于全连接层，它需要的参数会少得多，能更快地被训练出来。图 1.17 展示了在卷积层的每一个神经元有关于局部输入的所有通道的信息。如果说一个 RGB 图片是输入，那么一个在卷积层的神经元就能知道卷积核运算之后的一个局部区域的所有 RGB 通道。

在卷积层里的卷积运算使用了不同的卷积核来提取各种各样重要的特征。当其中一层网络的输入是高/宽为 W 的向量，并且我们使用一个大小为 F 的卷积核，卷积运算将输入的向量切分为若干小区间，然后每个小区间依次和卷积核进行点乘计算。其中步长 S 规定了每个小区间之间的距离。若步长为 2 ($S = 2$)，卷积核则会跟每个距离为 2 的小区间进行点乘运算。如果要确保边缘的数值也被很好地考虑在内的话，那么就需要在边缘填充零（Zero Padding）。若使填充的大小为

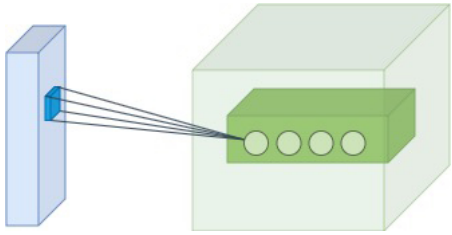

图 1.17 从样本图片计算卷积层的方法：蓝色的是输入层，绿色为卷积输出层。假如输入是一个有三个通道的 RGB 图片，那么卷积层输出也有不同通道。和全连接层不同的是，卷积层的神经元只和输入层的部分区域所连接，而不是和所有输入都有连接。图中展示了卷积层的不同通道是怎么和输入层有局部连接的（见彩插）

P，则一个卷积层的输出层大小就可以用公式 (1.24) 来进行计算。

$$\left\lfloor \frac{W-F+2P}{S}+1 \right\rfloor \tag{1.24}$$

输出层的深度（输出通道的数量）和卷积核的数量是一致的。图 1.18 具体地展示了卷积运算的流程。在图 1.18 中，有一个大小为 4×4（高 $\times$ 宽）的 RBG 图片、一个大小为 $3\times 3\times 3$（高 $\times$ 宽 $\times$ 输入通道）的卷积核，步长 $S=1$，边缘填充 $P=0$。根据公式 (1.24)，输出值的高/宽为 $(4-3+0)/1+1=2$。输出的深度（卷积核的通道数）是 1（因为只有一个卷积核）。为了计算在每一个通道左上角的那个数值，首先计算输入图片和卷积核的点乘，得到三个值，这三个值的和就是左上角的数值。卷积运算所得到的输出可以通过一层激活函数来引入非线性。

池化层利用了图片相邻像素类似的性质来进行下取样。我们认为，合适的像素只留取一个区域里的最大值或者平均值的下取样，会在建模当中有很多益处。通常有两种池化方法来减少数据大小：最大值池化和平均值池化。在图 1.19 中，在一个 4×4 的输入上和在步长是 2 的情况下，演示了最大值池化和平均值池化的例子。池化层可以很明显地减少输出大小，提高之后层的计算效率。比如说，在一个卷积层以后会有数以百计的通道，在输出被传递给全连接层之前，使用池化层来减小输出大小会减小计算量。

通常来说，卷积层、池化层和全连接层是 CNN 的核心构建部分。图 1.20 展示了一个有两个卷积层、一个最大值池化层和一个全连接层的网络。这里需要注意的是激活函数可以同样地用在卷积层上。

和前向神经网络不同的是，CNN 借用了**参数共享的**概念。在模型的不同部分使用参数共享，让整个模型更加高效（更少的参数和内存需求），然后它也可以用来处理不同的数据形式（不同大小或者长度）。回想一下，在一个全连接层中有一个权重矩阵，里面的元素 w_{ij} 代表着前一层第 i 神经元和当前层第 j 神经元连接。但在一个卷积层里，卷积核其实就是权重，它们在运算输出的时候是被重复使用的。对卷积核的重复使用就减少了在卷积网络里对参数的需求，这也就是为什么在输入和输出大小类似的情况下，卷积层比全连接层所需要的参数更少。

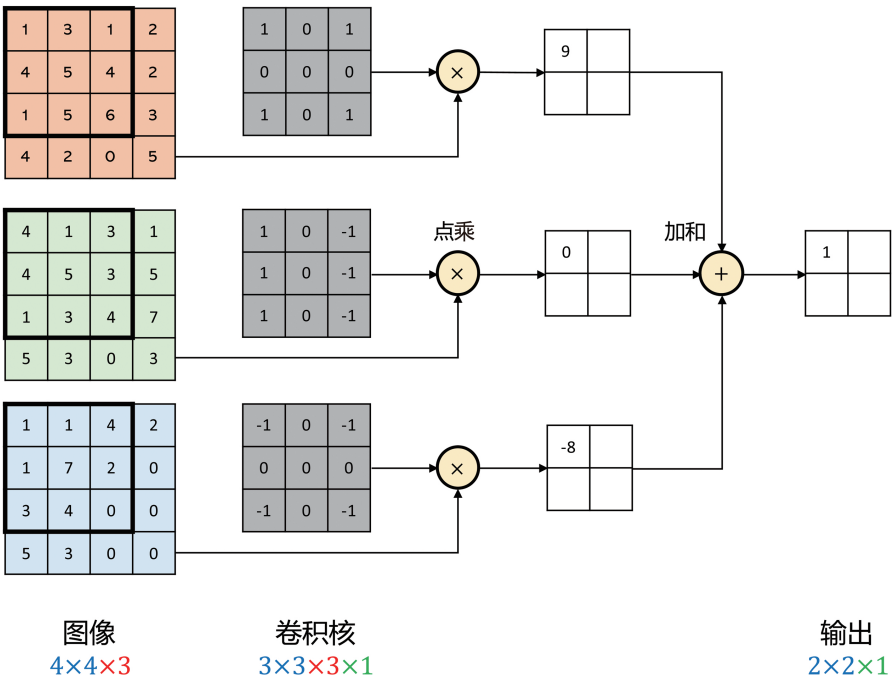

图 1.18　卷积运算的示意图，在这个例子里有一个大小为 $3 \times 3 \times 3 \times 1$ 的卷积核（Filter，也称为 Kernel）（尺寸为：高 × 宽 × 输入通道数 × 输出通道数）被用到了一个大小为 4×4（高 × 宽）的有 3 个输入通道的 RBG 图片上。图片和卷积核的点乘在不同的通道上都会应用。点乘所获得的值最终会被求和，然后得到输出的左上角的那个值

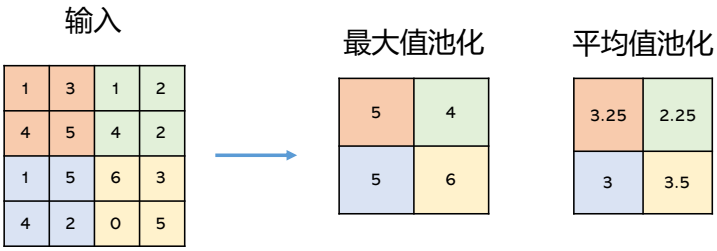

图 1.19　2×2 最大值池化和平均值池化的例子，它们的步长为 2，输入大小是 4×4

我们可以进一步地通过批标准化（批标准化层），即内部的样例迁移，来提高 CNN 的训练效率 (Ioffe et al., 2015)。我们之前提过，一个批标准化层是通过一个平均值和一个方差来进行标准化且独立于其他层的。也就是说，批标准化简化了在梯度更新的时候不同层之间的关系，从而可以用更大的学习速率来加快学习过程。

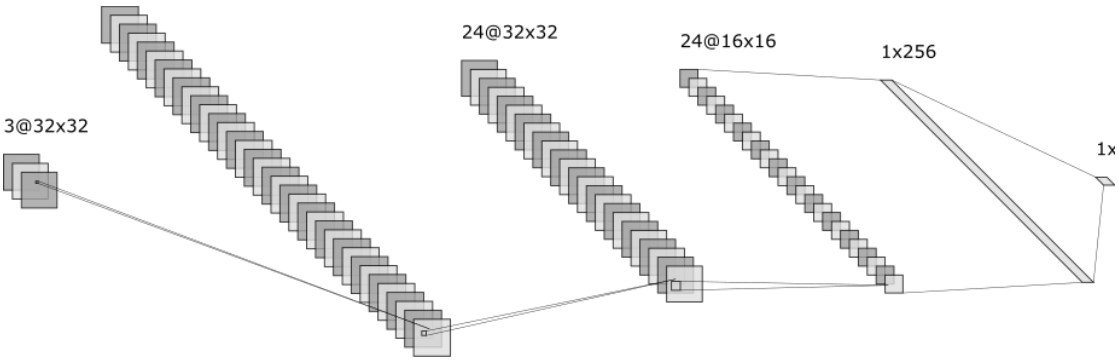

图 1.20 一个有两个卷积层、一个池化层和一个全连接层的网络。图片使用 NN-SVG 构造

1.9 循环神经网络

循环神经网络（Recurrent Neural Networks，RNN）(Rumelhart et al., 1986) 是另一种深度学习模型结构，主要用于处理序列数据。图像数据可以用网格加数值来表示，而序列数据作为另一种常见的数据类型，则被定义为一串元素 $\{x_1, x_2, \cdots, x_n\}$。例如，文本是由一串单词组成的，而股票的价格也可以用一串交易金额来表示。

序列数据的一个重要特点是，这一串元素之间有互相影响。例如，人们可以轻松地根据文章的开头，大致推测出文章接下来的内容。然而，针对这种元素之间的影响进行建模是相当具有挑战性的，尤其是当这一串序列非常长的时候。因此，循环神经网络需要能够有效地积累序列信息，并且考虑前序信息和后序信息之间的影响。

与卷积神经网络类似，循环神经网络同样使用了参数共享。参数共享得以让循环神经网络对序列上不同位置的元素重复使用同一组权重。我们来一起看一个例子，卷积神经网络需要能够学到“深度学习从 2010 年开始受到追捧”和“从 2010 年开始，深度学习受到追捧”这两句话其实表达的是相同的意思，尽管两句话的语序并不相同。同样，当卷积神经网络对猫的图片进行分类的时候，猫在图片中的位置也不应该影响模型做出正确的判断。

循环神经网络可以处理任意长度的输入序列，这一点与卷积神经网络可以处理任意长宽的输入图像相似。之所以如此，是因为循环神经网络使用了循环单元（Cell）作为基本的计算单元。针对输入序列中的每个数据元素，循环单元会被依次反复调用并进行计算。因此循环单元中会维护一个隐状态（Hidden State），用于记录序列中的信息。循环单元的计算包含两个输入，分别是序列中的当前数据及循环单元之前的隐状态。循环单元根据两个输入计算新的隐状态作为输出，新的隐状态也将用在下一轮计算当中，如图 1.21 所示。最简单的循环单元使用线性变换（公式 (1.25)）：

$$\boldsymbol{h}_t = \boldsymbol{W}[\boldsymbol{x}_t; \boldsymbol{h}_{t-1}] + \boldsymbol{b} \tag{1.25}$$

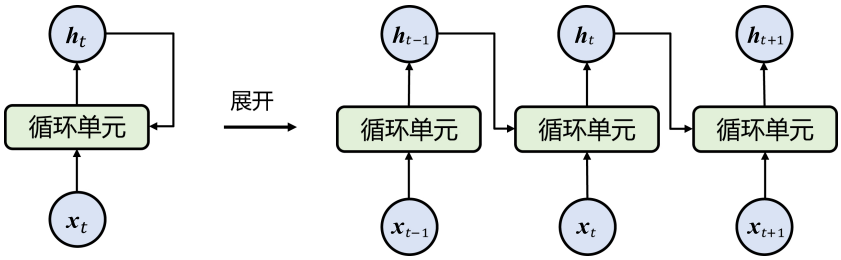

图 1.21　循环神经网络结构示意图。循环单元（cell）接收数值 $\boldsymbol{x}_t$ 和前序信息的隐状态 $\boldsymbol{h}_{t-1}$，然后输出新的隐状态 $\boldsymbol{h}_t$.

公式 (1.25) 中，隐状态 $\boldsymbol{h}_{t-1}$ 与输入数据 $\boldsymbol{x}_t$ 组合在一起，然后与线性核 $\boldsymbol{W}$ 做矩阵乘法，同时偏置 $\boldsymbol{b}$ 也可以加入新的隐状态当中。由于线性核 $\boldsymbol{W}$ 会被反复计算，循环神经网络实际上构建了一个深度计算图，而深度较大的计算图可能导致梯度爆炸或者梯度消失。当 $\boldsymbol{W}$ 的特征值幅度大于 1 时，可能导致梯度爆炸，而梯度爆炸会让学习过程完全失效。与之相反，若 $\boldsymbol{W}$ 的特征值幅度小于 1，则将可能导致梯度消失，梯度消失会让模型无法有效地根据学习目标进行优化。如果输入序列很长，那么使用简单循环单元的循环神经网络将有可能遇到这两种梯度问题的其中之一。

简单循环单元有严重的遗忘问题，当给定句子“我是中国人，我的母语是 ____”，简单循环单元会很容易预测出结果是“中文”，但是当句子很长时，如“我是中国人，我去英国读书，后来在法国工作，我的母语是 ____”，隐状态被多次更新后，简单循环单元很可能无法预测出正确的结果。**长短期记忆**（Long Short-Term Memory，LSTM）(Hochreiter et al., 1997) 是一种更加先进的循环单元，并常用于处理长序列中元素之间的影响。使用 LSTM 作为循环单元的循环神经网络亦常被简称为 LSTM。

与简单循环单元不同，LSTM 循环单元有两个状态量：单元状态（Cell State），记为 $\boldsymbol{C}_t$；隐状态（Hidden State），记为 $\boldsymbol{h}_t$。计算单元状态的过程实际上构建了一条信息高速路（如图 1.22 所示），这条信息高速路贯穿整个序列并且只使用了简单的计算过程。由于这条信息高速路让信息流可以较为便捷地穿越整个序列，因此 LSTM 可以较好地考虑长序列当中两个距离较远的元素之间的影响，即长期记忆。与此同时，LSTM 基于门（Gate）的机制计算隐状态。这种基于门的计算机制利用 sigmoid 激活函数来控制信息的遗忘或者叠加，因为 sigmoid 函数的值域介于 0 和 1 之间。也就是说，当 sigmoid 函数输出为 1 时，相对应的信息会被完整地保留。与之相反，当 sigmoid 函数输出为 0 时，相对应的信息会完全丢失。

在 LSTM 当中，一共有三个基于门的计算机制，分别是遗忘门（Forget Gate）、输入门（Input Gate）和输出门（Output Gate）。首先，遗忘门根据新的输入来决定单元状态当中是否有部分信息应该被遗忘。其次，输入门决定哪些输入信息应该被加入单元状态中，目的是长期存储这部分信息并取代被遗忘的信息。最后的输入门根据最新的单元状态，决定 LSTM 循环单元的输出。这三

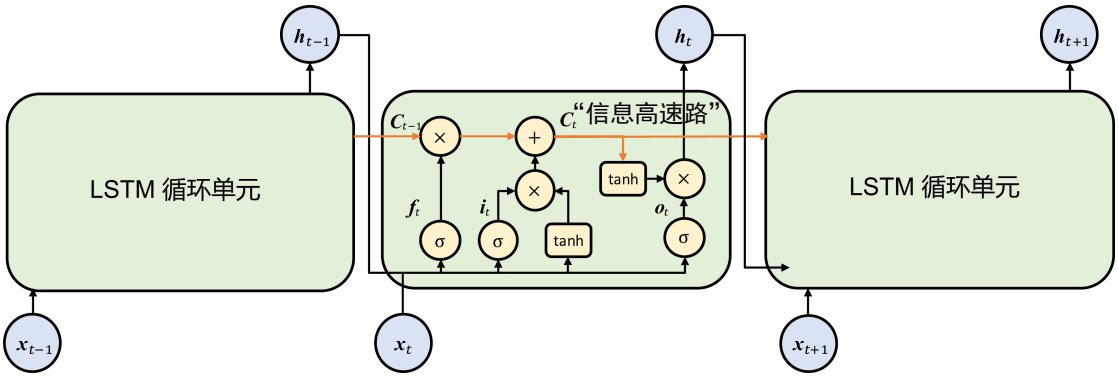

图 1.22 使用 LSTM 循环单元的循环神经网络示意图。LSTM 循环单元包括两个状态，即单元状态（Cell State）$\boldsymbol{C}_t$ 和隐状态（Hidden State）$\boldsymbol{h}_t$。除此之外，还有三个门（Gate）用于控制信息的取舍。本图依据文献 (Olah, 2015) 重新绘制

个基于门的计算机制可以用方程 (1.26) 定义，其中 σ 代表 simoid 函数。

$$
\begin{aligned}
\text{遗忘门：} \quad & \boldsymbol{f}_t = \sigma(\boldsymbol{W}_f[\boldsymbol{h}_{t-1}; \boldsymbol{x}_t] + \boldsymbol{b}_f) \\
\text{输入门：} \quad & \boldsymbol{i}_t = \sigma(\boldsymbol{W}_i[\boldsymbol{h}_{t-1}; \boldsymbol{x}_t] + \boldsymbol{b}_i) \\
\text{输出门：} \quad & \boldsymbol{o}_t = \sigma(\boldsymbol{W}_o[\boldsymbol{h}_{t-1}; \boldsymbol{x}_t] + \boldsymbol{b}_o) \\
\text{更新单元状态：} \quad & \boldsymbol{C}_t = \boldsymbol{f}_t \times \boldsymbol{C}_{t-1} + \boldsymbol{i}_t \times \tanh(\boldsymbol{W}_C[\boldsymbol{h}_{t-1}; \boldsymbol{x}_t] + \boldsymbol{b}_C) \\
\text{更新隐状态：} \quad & \boldsymbol{h}_t = \boldsymbol{o}_t \times \tanh(\boldsymbol{C}_t)
\end{aligned} \tag{1.26}
$$

循环神经网络有很多种，而 LSTM 是其中之一，还有 GRU（Gated Recurrent Units）。最近的研究工作尝试对比了不同结构的循环神经网络，但是关于哪一种结构更优，目前尚无定论 (Cho et al., 2014; Jozefowicz et al., 2015)。

在深度学习中，循环神经网络主要用于处理序列数据，如自然语言和时间序列 (Chung et al., 2014; Liao et al., 2018b; Mikolov et al., 2010)，同时也会用于处理强化学习的问题 (Peng et al., 2018; Wierstra et al., 2010)。根据输入和输出的关系，循环神经网络的结构在不同的场景也会有些许变化。例如，在文本分类的问题中，循环神经网络的输入是一串单词序列，而输出是单个代表类别的标签 (Lee et al., 2016; Zhang et al., 2019a)。在机器翻译 (Bahdanau et al., 2015; Luong et al., 2015; Sutskever et al., 2014) 或者自动摘要 (Nallapati et al., 2017) 的任务中，循环神经网络的输入和输出均是一串单词序列。关于更多的细节，感兴趣的读者可以查看我们其他的讲义，链接见读者服务。

1.10　深度学习的实现样例

本节将介绍深度学习的实现样例，其中模型的代码将基于 Python 3、TensorFlow 2.0 和 TensorLayer 2.0。

1.10.1　张量和梯度

张量（Tensor）是 TensorFlow 中最基本的计算单元，特指计算函数的输出，由计算函数生成，如 `tf.constant`，`tf.matmul` 等。这些张量本身并不存储计算结果，而是为获取 TensorFlow session 中产生该结果的计算过程提供便利。在 TensorFlow 2.0 中，无须手动运行会话（Session），因为在 Eager execution 设计思想下，运算图和会话的运行细节仅在后端可见。比如，在下面的矩阵乘法示例中，我们可以通过 `tf.constant` 创建矩阵，并通过 `tf.matmul` 计算输出为另一个矩阵的乘法。

代码 1.1　TensorFlow 中基于张量的矩阵乘法

```
>>> import tensorflow as tf
>>> a = tf.constant([[1, 2], [1, 2]])
# tf.Tensor(
# [[1 2]
#  [1 2]], shape=(2, 2), dtype=int32)
>>> b = tf.constant([[1], [2]])
# tf.Tensor(
# [[1]
#  [2]], shape=(2, 1), dtype=int32)
>>> c = tf.matmul(a, b)
# tf.Tensor(
# [[5]
#  [5]], shape=(2, 1), dtype=int32)
```

在深度神经网络的前向传播中，Tensors 实例会自动相互连接，从而形成一个运算图。因此，我们可以通过 TensorFlow 自带的自动差分和运算图相关功能，在反向传播时计算梯度。TensorFlow 2.0 更是提供了 `tf.GradientTape` 方法，用于计算输入变量对被记录操作的梯度。

神经网络的前向传播和损失函数的计算应当在 `tf.GradientTape` 作用域之内，而反向传播和权重更新则可以在作用域之外。`tf.GradientTape` 将所有在作用域内执行的运算都记录到 `Tape` 中，然后通过反向自动差分机制，计算每个运算符和输入变量相对应的梯度。直到 `tape.gradient()` 被调用后，`tf.GradientTape` 所占用的资源才会被释放。

代码 1.2 TensorFlow 和 TensorLayer 中的梯度计算

```
import tensorflow as tf
import tensorlayer as tl
def train(model, dataset, optimizer):
    # 给定一个 TensorLayer 模型
    # 遍历数据，其中 x 为输入，y 为输出
    for x, y in dataset:
        # 构建 tf.GradientTape 的作用域
        with tf.GradientTape() as tape:
            prediction = model(x) # 前向传播
            loss = loss_fn(prediction, y) # 损失函数
        # 反向传播并计算梯度
        # 然后释放 tf.GradientTape 所占用的资源
        gradients = tape.gradient(loss, model.trainable_weights)
        # 根据梯度，利用优化器更新权重
        optimizer.apply_gradients(zip(gradients, model.trainable_weights))
```

1.10.2 定义模型

在 TensorLayer 2.0 中，模型（`Model`）是一个包含多个 `Layer` 的实体，并且定义了 `Layer` 之间传播运算。TensorLayer 2.0 提供了两套定义模型的接口，其中静态模型接口让用户可以更加流畅地定义模型，而动态模型接口让前向传播更加灵活。静态模型需要用户手动构建运算图并编译，模型一旦编译后，前向传播将不能修改。与之不同的是，动态模型可以像普通 Python 代码一样即刻执行（Eager Execution），而且前向传播是可以修改的。

如下面的实现样例所示，我们可以将静态模型和动态模型的差别总结成两个方面。首先，静态模型中的 Layer 在声明的同时也会定义与其他 Layer 的连接关系（即前向传播）。根据 Layer 之间的连接关系，TensorLayer 可以自动推断每个 Layer 输入变量的大小，并相应构建权重。因此，当 `Model` 初始化的时候，只需要明确模型的输入和输出即可，而 TensorLayer 将自动根据 Layer 之间的连接构建运算图。然而，动态模型则不同，前向传播的顺序（即 Layer 之间的连接关系）在动态模型初始化时是不需要明确的，因为动态模型的前向传播直到前向函数 `forward` 被实际调用的时候才能确定。因此，动态模型无法自动推断每个 Layer 输入变量的大小，必须通过输入参数 `in_channels` 显式地明确 Layer。

其次，静态模型的前向传播一旦定义即固定，因此更加易于加速计算过程。TensorFlow 2.0 提供了一个新的功能，即 `tf.function`，可作为装饰器套在函数上，加速函数内的计算。与静态模型不同的是，动态模型的前向传播更加灵活。例如，用户可以根据不同的输入和参数来控制前向传播，同时也可以根据需要选择执行或者跳过部分 Layer 的计算。

代码 1.3　静态模型样例：多层感知器（MLP）

```
import tensorflow as tf
from tensorlayer.layers import Input, Dense
from tensorlayer.models import Model

# 包含了三个全连接层的多层感知器模型
def get_mlp_model(inputs_shape):
    ni = Input(inputs_shape)
    # 因为明确定义了 Layer 之间的连接关系
    # 可以自动推断每个 Layer 的 in_channels
    nn = Dense(n_units=800, act=tf.nn.relu)(ni)
    nn = Dense(n_units=800, act=tf.nn.relu)(nn)
    nn = Dense(n_units=10, act=tf.nn.relu)(nn)
    # 根据连接关系自动构建模型
    M = Model(inputs=ni, outputs=nn)
    return M

MLP = get_mlp_model([None, 784])
# 开启 eval 模式
MLP.eval()
# 给定输入数据
# 该计算过程可以通过 TensorFlow 2.0 中的 @tf.function 加速
outputs = MLP(data)
```

代码 1.4　动态模型样例：多层感知器（MLP）

```
import tensorflow as tf
from tensorlayer.layers import Input, Dense
from tensorlayer.models import Model

class MLPModel(Model):
    def __init__(self):
        super(MLPModel, self).__init__()
        # 因为无法明确 Layer 之间的连接关系，必须手动提供 in_channels
        # 给定输入数据的大小，即 784
        self.dense1 = Dense(n_units=800, act=tf.nn.relu, in_channels=784)
        self.dense2 = Dense(n_units=800, act=tf.nn.relu, in_channels=800)
        self.dense3 = Dense(n_units=10, act=tf.nn.relu, in_channels=800)
```

```
    def forward(self, x, foo=False):
        # 定义前向传播
        z = self.dense1(z)
        z = self.dense2(z)
        out = self.dense3(z)
        # 灵活控制前向传播
        if foo:
            out = tf.nn.softmax(out)
        return out

MLP = MLPModel()
# 开启 eval 模式
MLP.eval()
# 给定输入数据
# 通过参数 foo 控制前向传播
outputs_1 = MLP(data, foo=True) # 使用 softmax
outputs_2 = MLP(data, foo=False) # 不使用 softmax
```

1.10.3 自定义层

TensorLayer 2.0 为用户提供了大量的神经网络层，也支持 Lambda Layer 以方便用户创造高度自定义的层。如下所示，最简单的例子是把一个 lambda 表达式直接传入 Lambda Layer。用户可以通过一个自定义输入参数的函数和 fn_args 选项来初始化或者调用 Lambda Layer。

```
import tensorlayer as tl
x = tl.layers.Input([8, 3], name='input')
y = tl.layers.Lambda(lambda x: 2*x)(x) # 没有可训练的权重

def customize_fn(input, foo): # 参数可以通过 Lambda Layer 的 fn_args 定义
    return foo * input
z = tl.layers.Lambda(customize_fn, fn_args={'foo': 42})(x) # this layer has no weights.
```

Lambda Layer 拥有可训练的权重。下面的示例可以展示如何在自定义函数外定义权重，并通过 fn_weights 选项传入 Lambda Layer。

```
import tensorflow as tf
import tensorlayer as tl
a = tf.Variable(1.0) # 自定义函数作用域之外的权重
def customize_fn(x):
    return x + a
x = tl.layers.Input([8, 3], name='input')
y = tl.layers.Lambda(customize_fn, fn_weights=[a])(x) # 通过 fn_weights 传递权重
```

此外，Lambda Layer 还可以使 Keras 与 TensorLayer 兼容。用户可以定义一个 Keras 模型，并将其以一个函数的形式传入 Lambda Layer，因为 Keras 的模型是可被调用的。同时，为了让自定义模型和 Keras 模型一起被训练，Keras 模型中可被训练的权重需要被手动提取，然后传入 Lambda Layer 中。

```
import tensorflow as tf
import tensorlayer as tl
# 定义一个 Keras 模型
layers = [
    tf.keras.layers.Dense(10, activation=tf.nn.relu),
    tf.keras.layers.Dense(5, activation=tf.nn.sigmoid),
    tf.keras.layers.Dense(1, activation=tf.identity)
]
perceptron = tf.keras.Sequential(layers)
# 获得 Keras 模型的可被训练的权重
_ = perceptron(np.random.random([100, 5]).astype(np.float32))

class CustomizeModel(tl.models.Model):
    def __init__(self):
        super(CustomizeModel, self).__init__()
        self.dense = tl.layers.Dense(in_channels=1, n_units=5)
        self.lambdalayer = tl.layers.Lambda(perceptron, perceptron.trainable_variables)
        # 将可以训练的权重传递给 Lambda Layer

    def forward(self, x):
        z = self.dense(x)
        z = self.lambdalayer(z)
        return z
```

1.10.4 多层感知器：MNIST 数据集上的图像分类

用户可以通过 TensorLayer 2.0 中提供的 `Model`、`Layer` 和其他支持性的 API 来灵活、直观地设计和实现自己的深度学习模型。为了帮助读者更好地了解如何用 TensorLayer 实现一个深度学习模型，这里首先介绍一个利用多层感知器在 MNIST 数据集 (LeCun et al., 1998) 上分类图片的示例。该数据集包含了 70,000 张手写数字的图片。一个深度学习模型的建立通常会包含五个步骤，分别是数据加载、模型定义、训练、测试和模型存储。

TensorLayer 在 `tl.files` 中提供了多个常用数据集的 API，包括 MNIST、CIFAR10、PTB、CelebA 等。比如说，我们可以用 `tl.files.load_mnist_dataset` 和一个具体的 shape 加载 MNIST 数据集。通常来说，数据集会被划分为三个子集：训练集、验证集和测试集。

```
# 通过 TensorLayer 加载 MNIST 数据集
X_train, y_train, X_val, y_val, X_test, y_test = tl.files.load_mnist_dataset(shape=(-1,
    784)) # 每个 MNIST 图像的原始尺寸为 28 * 28，即一共有 784 个像素点
```

就像在 1.10.2 节里提到的一样，在 TensorLayer 2.0 中，一个多层感知器模型可以通过静态模型或者动态模型两种方法来实现。在这个例子中，我们的模型有三个 `Dense` 层，且为静态模型，同时，用 `Dropout` 来防止过拟合现象的产生。

```
# 构建模型
ni = tl.layers.Input([None, 784]) # 根据输入数据定义尺寸
# 多层感知器
nn = tl.layers.Dropout(keep=0.8)(ni)
nn = tl.layers.Dense(n_units=800, act=tf.nn.relu)(nn)
nn = tl.layers.Dropout(keep=0.5)(nn)
nn = tl.layers.Dense(n_units=800, act=tf.nn.relu)(nn)
nn = tl.layers.Dropout(keep=0.5)(nn)
nn = tl.layers.Dense(n_units=10, act=None)(nn)
# 给定输入和输出，构建模型
network = tl.models.Model(inputs=ni, outputs=nn, name="mlp")
```

多层感知器在 MNIST 数据集上的训练是指对其权重的学习。用户可以通过调用 `tl.utils.fit` 函数来触发训练过程。除此之外，我们还需要通过 `tl.utils.test` 函数来验证模型的性能。

```
# 定义一个函数来评估模型的准确度
# 与损失函数不同，这个函数不用于更新模型
def acc(_logits, y_batch):
    return tf.reduce_mean(
        tf.cast(
```

```
            tf.equal(
                tf.argmax(_logits, 1),
                tf.convert_to_tensor(y_batch, tf.int64)),
            tf.float32),
        name='accuracy'
    )

# 训练
tl.utils.fit(
    network, # 模型
    train_op=tf.optimizers.Adam(learning_rate=0.0001), # 优化器
    cost=tl.cost.cross_entropy, # 损失函数
    X_train=X_train, y_train=y_train, # 训练集
    acc=acc, # 评估指标
    batch_size=256, # 批样本数量
    n_epoch=20, # 训练轮数
    X_val=X_val, y_val=y_val, eval_train=True, # 验证集
)

# 测试
tl.utils.test(
    network, # 训练好的模型
    acc=acc, # 评估指标
    X_test=X_test, y_test=y_test, # 测试集
    batch_size=None, # 批样本数量，如果为 None 则将测试集一起输入模型，因此当且仅当测试集
    # 很小的时候可以将此设置为 None
    cost=tl.cost.cross_entropy # 损失函数
)
```

最后，多层感知器模型的权重可以保存至本地的一个文件中，使得我们可以在后面需要的时候恢复模型参数，用于推理，该多层感知器示例的完整实现代码链接见读者服务。

```
# 将模型权重保存到文件中
network.save_weights('model.h5')
```

1.10.5 卷积神经网络：CIFAR-10 数据集上的图像分类

CIFAR-10 数据集 (Krizhevsky et al., 2009) 是一个通用的、具有一定挑战性的图像分类基准测试。此数据集一共包含 10 类数据，其中每类分别有 6000 张 32×32 RGB 图片，且每张图片只专注于描述单个物体，如狗、飞机、船舶等。使用 TensorLayer 2.0 中的 `Dataset` 和 `Dataloader` APIs，我们可以很简单地加载 CIFAR-10 并对其做数据增强。

```
# 定义数据增强
def _fn_train(img, target):
    # 1. 随机切割长宽均为 24 的一小块图片
    img = tl.prepro.crop(img, 24, 24, False)
    # 2. 随机水平翻转图片
    img = tl.prepro.flip_axis(img, is_random=True)
    # 3. 正则化：减去像素点的平均值并除以方差
    img = tl.prepro.samplewise_norm(img)
    target = np.reshape(target, ())
    return img, target

# 加载训练集
train_ds = tl.data.CIFAR10(train_or_test='train', shape=(-1, 32, 32, 3))
# dataloader 加载数据集和数据增强算法
train_dl = tl.data.Dataloader(train_ds, transforms=[_fn_train], shuffle=True,
    batch_size=batch_size, output_types=(np.float32, np.int32))

# 加载测试集
test_ds = tl.data.CIFAR10(train_or_test='test', shape=(-1, 32, 32, 3))
# dataloader 加载测试集
test_dl = tl.data.Dataloader(test_ds, batch_size=batch_size)

# 遍历数据集
for X_batch, y_batch in train_dl:
# 训练、测试模型的代码
```

在这个示例里，我们将使用带有批标准化 (Ioffe et al., 2015) 的卷积神经网络来对 CIFAR-10 中的图片进行分类。该模型有两个卷积模块，其中每个模块含有一个批标准化层。模型的最后包含了三个全连接层。该卷积网络示例的完整实现代码链接请见读者服务。

```
# 包含了 BatchNorm 的卷积神经网络
def get_model_batchnorm(inputs_shape):
```

```
    # 自定义权重初始化
    W_init = tl.initializers.truncated_normal(stddev=5e-2)
    W_init2 = tl.initializers.truncated_normal(stddev=0.04)
    b_init2 = tl.initializers.constant(value=0.1)

    # 输入层
    ni = Input(inputs_shape)

    # 第一个卷积层 Conv2d，以及 BatchNorm 和池化层 MaxPool
    nn = Conv2d(64, (5, 5), (1, 1), padding='SAME', W_init=W_init, b_init=None)(ni)
    nn = BatchNorm2d(decay=0.99, act=tf.nn.relu)(nn)
    nn = MaxPool2d((3, 3), (2, 2), padding='SAME')(nn)

    # 第二个卷积层 Conv2d，以及 BatchNorm 和池化层 MaxPool
    nn = Conv2d(64, (5, 5), (1, 1), padding='SAME', W_init=W_init, b_init=None)(nn)
    nn = BatchNorm2d(decay=0.99, act=tf.nn.relu)(nn)
    nn = MaxPool2d((3, 3), (2, 2), padding='SAME')(nn)

    # 卷积层的输出传递给三个全连接层
    nn = Flatten()(nn)
    nn = Dense(384, act=tf.nn.relu, W_init=W_init2, b_init=b_init2)(nn)
    nn = Dense(192, act=tf.nn.relu, W_init=W_init2, b_init=b_init2)(nn)
    nn = Dense(10, act=None, W_init=W_init2)(nn)

    # 给定输入和输出，构建模型
    M = Model(inputs=ni, outputs=nn, name='cnn')
    return M
```

1.10.6　序列到序列模型：聊天机器人

聊天机器人（Chatbot）的设计通常涵盖了语音和文字对话的应用。在这个示例中，我们将简化这一设计，并考虑文字输入和反馈的情形。因此，序列到序列模型（Seq2seq）(Sutskever et al., 2014) 是实现聊天机器人的一个很好的选择。该模型需要序列作为输入和输出，因此，我们可以在此把聊天机器人的输入和输出定义为句子，又可被理解为是文字的序列。seq2seq 模型会被训练去对输入句子以另一句话的形式做适当的回应。虽然 seq2seq 模型在提出的时候主要应用于机器翻译，但在其他序列-序列应用场景中同样具有良好的应用前景，如交通预测 (Liao et al., 2018a,b)、文本自动摘要 (Liu et al., 2018; Zhang et al., 2019b) 等。

在实践中，一个 seq2seq 模型由两个 RNN 组成，其一为编码 RNN，其二为解码 RNN。编码 RNN 会学习一个对于输入语句的表示，然后解码 RNN 便可尝试生成一个针对输入的回应。TensorLayer 库提供的 API 可以在一行以内生成一个 Seq2seq 模型。

```
# Seq2seq 模型
model_ = Seq2seq(
    decoder_seq_length=decoder_seq_length, # 解码的最大长度
    cell_enc=tf.keras.layers.GRUCell, # 编码 RNN 的循环单元
    cell_dec=tf.keras.layers.GRUCell, # 解码 RNN 的循环单元
    n_layer=3, # 编码 RNN 和解码 RNN 的层数
    n_units=256, # RNN 的隐状态大小
    embedding_layer=tl.layers.Embedding(vocabulary_size=vocabulary_size,
        embedding_size=emb_dim), # 编码 RNN 的嵌入层
)
```

下面展示了一些基于 Seq2seq 的聊天机器人模型的结果，聊天机器人的完整实现代码链接请见读者服务。该模型可以在获取一个输入句子后输出多种可能的结果。

```
Query > happy birthday have a nice day
 > thank you so much
 > thank babe
 > thank bro
 > thanks so much
 > thank babe i appreciate it
```

参考文献

BAHDANAU D, CHO K, BENGIO Y, 2015. Neural machine translation by jointly learning to align and translate[C]//Proceedings of the International Conference on Learning Representations (ICLR).

BISHOP C M, 2006. Pattern recognition and machine learning[M]. springer.

BOTTOU L, BOUSQUET O, 2007. The Tradeoffs of Large Scale Learning.[C]//Proceedings of the Neural Information Processing Systems (Advances in Neural Information Processing Systems) Conference: volume 20. 161-168.

CAO Z, SIMON Z, WEI S E, et al., 2017. Realtime multi-person 2d pose estimation using part affinity fields[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

CHO K, VAN MERRIËNBOER B, GULCEHRE C, et al., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation[C]//Proceedings of the Empirical Methods in Natural Language Processing (EMNLP) Conference.

CHUNG J, GULCEHRE C, CHO K, et al., 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling[J]. arXiv preprint arXiv:1412.3555.

DEVLIN J, CHANG M W, LEE K, et al., 2019. BERT: Pre-training of deep bidirectional transformers for language understanding[C/OL]//Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). Minneapolis, Minnesota: Association for Computational Linguistics: 4171-4186. DOI: 10.18653/v1/N19-1423.

DONG H, ZHANG J, MCILWRAITH D, et al., 2017. I2t2i: Learning text to image synthesis with textual data augmentation[C]//Proceedings of the IEEE International Conference on Image Processing (ICIP).

DUCHI J, HAZAN E, SINGER Y, 2011. Adaptive subgradient methods for online learning and stochastic optimization[J]. Journal of Machine Learning Research (JMLR), 12(Jul): 2121-2159.

GAL Y, GHAHRAMANI Z, 2016. Dropout as a bayesian approximation: Representing model uncertainty in deep learning[C]//Proceedings of the International Conference on Machine Learning (ICML). 1050-1059.

GLOROT X, BORDES A, BENGIO Y, 2011. Deep sparse rectifier neural networks[C]//Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS). 315-323.

GOODFELLOW I, POUGET-ABADIE J, MIRZA M, et al., 2014. Generative Adversarial Nets[C]// Proceedings of the Neural Information Processing Systems (Advances in Neural Information Processing Systems) Conference.

GOODFELLOW I, BENGIO Y, COURVILLE A, 2016. Deep learning[M]. MIT Press.

HARA K, SAITOH D, SHOUNO H, 2016. Analysis of dropout learning regarded as ensemble learning[C]//Proceedings of the International Conference on Artificial Neural Networks (ICANN). Springer: 72-79.

HE K, ZHANG X, REN S, et al., 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification[C]//Proceedings of the IEEE international conference on computer vision. 1026-1034.

HE K, ZHANG X, REN S, et al., 2016. Deep Residual Learning for Image Recognition[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

HINTON G E, SRIVASTAVA N, KRIZHEVSKY A, et al., 2012. Improving neural networks by preventing co-adaptation of feature detectors[J]. arXiv preprint arXiv:1207.0580.

HOCHREITER S, HOCHREITER S, SCHMIDHUBER J, et al., 1997. Long Short-Term Memory.[J]. Neural Computation, 9(8): 1735-80.

HORNIK K, STINCHCOMBE M, WHITE H, 1989. Multilayer feedforward networks are universal approximators[J]. Neural networks, 2(5): 359-366.

HOWARD A G, ZHU M, CHEN B, et al., 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications[J]. Computing Research Repository (CoRR).

IOFFE S, SZEGEDY C, 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift[J]. arXiv preprint arXiv:1502.03167.

JAMES S, WOHLHART P, KALAKRISHNAN M, et al., 2019. Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 12627-12637.

JOZEFOWICZ R, ZAREMBA W, SUTSKEVER I, 2015. An empirical exploration of recurrent network architectures[C]//International Conference on Machine Learning. 2342-2350.

KINGMA D, BA J, 2014. Adam: A method for stochastic optimization[C]//Proceedings of the International Conference on Learning Representations (ICLR).

KO T, PEDDINTI V, POVEY D, et al., 2015. Audio augmentation for speech recognition[C]//Annual Conference of the International Speech Communication Association.

KRIZHEVSKY A, HINTON G, et al., 2009. Learning multiple layers of features from tiny images[R]. Citeseer.

KRIZHEVSKY A, SUTSKEVER I, HINTON G E, 2012. Imagenet classification with deep convolutional neural networks[C]//Advances in Neural Information Processing Systems. 1097-1105.

LECUN Y, BOSER B, DENKER J S, et al., 1989. Backpropagation applied to handwritten zip code recognition[J]. Neural computation, 1(4): 541-551.

LECUN Y, BOTTOU L, BENGIO Y, et al., 1998. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 86(11): 2278-2324.

LECUN Y, BENGIO Y, HINTON G, 2015. Deep learning[J]. Nature, 521(7553): 436.

LEE J Y, DERNONCOURT F, 2016. Sequential short-text classification with recurrent and convolutional neural networks[C/OL]//Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. San Diego, California: Association for Computational Linguistics: 515-520. DOI: 10.18653/v1/N16-1062.

LIAO B, ZHANG J, CAI M, et al., 2018a. Dest-ResNet: A deep spatiotemporal residual network for hotspot traffic speed prediction[C]//2018 ACM Multimedia Conference on Multimedia Conference. ACM: 1883-1891.

LIAO B, ZHANG J, WU C, et al., 2018b. Deep sequence learning with auxiliary information for traffic prediction[C]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. ACM: 537-546.

LIU P J, SALEH M, POT E, et al., 2018. Generating wikipedia by summarizing long sequences[C]// International Conference on Learning Representations.

LUONG T, PHAM H, MANNING C D, 2015. Effective approaches to attention-based neural machine translation[C/OL]//Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. Lisbon, Portugal: Association for Computational Linguistics: 1412-1421. DOI: 10.18653/v1/D15-1166.

MAAS A L, DALY R E, PHAM P T, et al., 2011. Learning word vectors for sentiment analysis[C]// HLT '11: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1. Stroudsburg, PA, USA: Association for Computational Linguistics: 142-150.

MIKOLOV T, KARAFIÁT M, BURGET L, et al., 2010. Recurrent neural network based language model[C]//Interspeech.

NALLAPATI R, ZHAI F, ZHOU B, 2017. Summarunner: A recurrent neural network based sequence model for extractive summarization of documents[C]//AAAI'17: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence. San Francisco, California, USA: AAAI Press: 3075–3081.

NG A Y, JORDAN M I, 2002. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes[C]//Proceedings of the Neural Information Processing Systems (Advances in Neural Information Processing Systems) Conference. 841-848.

NOH H, HONG S, HAN B, 2015. Learning deconvolution network for semantic segmentation[C]// Proceedings of the International Conference on Computer Vision (ICCV). 1520-1528.

OLAH C, 2015. Understanding lstm networks[Z].

PENG X B, ANDRYCHOWICZ M, ZAREMBA W, et al., 2018. Sim-to-real transfer of robotic control with dynamics randomization[C]//2018 IEEE International Conference on Robotics and Automation (ICRA). IEEE: 1-8.

REED S, AKATA Z, YAN X, et al., 2016. Generative Adversarial Text to Image Synthesis[C]//Proceedings of the International Conference on Machine Learning (ICML).

RISH I, et al., 2001. An empirical study of the naive bayes classifier[C]//International Joint Conference on Artificial Intelligence 2001 workshop on empirical methods in artificial intelligence: volume 3. 41-46.

ROSENBLATT F, 1958. The perceptron: a probabilistic model for information storage and organization in the brain.[J]. Psychological Review, 65(6): 386.

RUCK D W, ROGERS S K, KABRISKY M, et al., 1990. The multilayer perceptron as an approximation to a bayes optimal discriminant function[J]. IEEE Transactions on Neural Networks, 1(4): 296-298.

RUMELHART D E, HINTON G E, WILLIAMS R J, 1986. Learning representations by back-propagating errors[J]. Nature, 323(6088): 533.

RUSU A A, RABINOWITZ N C, DESJARDINS G, et al., 2016. Progressive neural networks[J]. arXiv preprint arXiv:1606.04671.

SAMUEL A, 1959. Some studies in machine learning using the game of checkers[C]//IBM Journal of Research and Development.

SIMONYAN K, ZISSERMAN A, 2015. Very deep convolutional networks for large-scale image recognition[C]//Proceedings of the International Conference on Learning Representations (ICLR).

SRIVASTAVA N, HINTON G, KRIZHEVSKY A, et al., 2014. Dropout: A simple way to prevent neural networks from overfitting[J]. Journal of Machine Learning Research (JMLR), 15(1): 1929-1958.

SUTSKEVER I, VINYALS O, LE Q V, 2014. Sequence to sequence learning with neural networks[C]// Proceedings of the Neural Information Processing Systems (Advances in Neural Information Processing Systems) Conference. 3104-3112.

TIELEMAN T, HINTON G, 2017. Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning[R]. Technical Report.

VAN DEN OORD A, DIELEMAN S, ZEN H, et al., 2016. WaveNet: A generative model for raw audio[C]//Arxiv.

WIERSTRA D, FÖRSTER A, PETERS J, et al., 2010. Recurrent policy gradients[J]. Logic Journal of the IGPL, 18(5): 620-634.

XU B, WANG N, CHEN T, et al., 2015. Empirical evaluation of rectified activations in convolutional network[C]//Proceedings of the International Conference on Machine Learning (ICML) Workshop.

YANG Z, DAI Z, YANG Y, et al., 2019. Xlnet: Generalized autoregressive pretraining for language understanding[C]//Advances in Neural Information Processing Systems. 5754-5764.

YIN W, KANN K, YU M, et al., 2017. Comparative study of cnn and rnn for natural language processing[J]. arXiv preprint arXiv:1702.01923.

ZHANG J, LERTVITTAYAKUMJORN P, GUO Y, 2019a. Integrating semantic knowledge to tackle zero-shot text classification[C/OL]//Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). Minneapolis, Minnesota: Association for Computational Linguistics: 1031-1040. DOI: 10.18653/v1/N19-1108.

ZHANG J, ZHAO Y, SALEH M, et al., 2019b. PEGASUS: Pre-training with extracted gap-sentences for abstractive summarization[J]. arXiv preprint arXiv:1912.08777.

ZHANG X, ZHAO J, LECUN Y, 2015. Character-level convolutional networks for text classification[C]// Advances in Neural Information Processing Systems. 649-657.