第5章 深度学习与 PyTorch

CHAPTER 5

作为后续深度强化学习章节的准备，本章主要介绍深度学习的核心内容和 PyTorch 深度学习软件包。

近年来，深度学习在计算机视觉、语音识别、自然语言处理等诸多领域取得了突破性进展，极大地促进了人工智能的发展。尤其是近年推出的深度学习软件框架（如 TensorFlow、PyTorch、Caffe、MXNet 等），显著降低了深度学习的学习门槛，提升了深度学习的应用范围。硬件平台（如 GPU、TPU、APU、DPU 等）的成熟和算力的提升，更进一步推动了深度学习的发展和落地。

深度学习具有极强的特征表征能力，这正是经典强化学习所需要的。事实上，深度强化学习正是以此为出发点，通过有机融合深度学习和强化学习，使智能体同时具备极强的感知能力和决策能力。

本章首先介绍深度神经网络的基本单元——感知机；然后介绍深度神经网络的拓扑结构，如前向传播机制、误差反向传播机制、训练原理、基本组成要素等；最后介绍本书使用的深度学习编程框架 PyTorch 及一些案例。

5.1 从感知机到神经网络

5.1.1 感知机模型

感知机（Perceptron）的概念于 1957 年由 Rosenblatt 提出，是构成神经网络的最小结构单元，用于模拟人类大脑神经网络的最小构成单元——神经元的工作机制，所以也称人工神经元或神经元（Neuron）。

感知机模型是二分类的线性分类模型，其输入为实例的特征向量，输出为实例的类别，一般取+1 或−1。感知机模型实际上相当于输入空间（特征空间）中将实例划分为正负两类的分离超平面，在机器学习中属于判别模型。感知机模型主要由连接、求和节点和激活函数组成，如图 5-1 所示。

感知机的输入是 n 维特征向量 $\boldsymbol{X}=(x_1,x_2,\cdots,x_n)^{\mathrm{T}}\in\mathbf{R}^n$，$w_1,w_2,\cdots,w_n$ 分别是各个特征的权重，b 是偏置参数，$\sum$ 是求和节点，即

$$z=\sum_{i=1}^{n}w_ix_i+b \tag{5-1}$$

σ 为激活函数，对于输出为+1 或−1 的二分类问题，如果 $z>0$，则处于激活状态，如果 $z\leqslant 0$，则处于抑制状态。设输出为 y，则

$$y=\sigma(z)=\begin{cases}1, & z>0\\ -1, & z\leqslant 0\end{cases} \tag{5-2}$$

综合求和节点和激活函数，感知机模型的形式化表达为

$$y=\sigma(\boldsymbol{W}^{\mathrm{T}}\boldsymbol{X}+b)=\sigma\Big(\sum_{i=1}^{n}w_ix_i+b\Big) \tag{5-3}$$

其中，$\boldsymbol{W}=(w_1,w_2,\cdots,w_n)^{\mathrm{T}}$。由式(5-3)可知，一个感知机的基本功能为对输入特征向量 $\boldsymbol{X}$ 与权值向量 $\boldsymbol{W}$ 内积求和后加上偏置参数 b，并经过非线性激活函数 σ，得到输出结果 y，最终实现了对输入特征向量的二项分类。

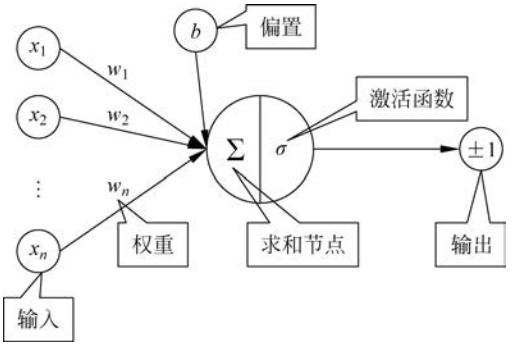

图 5-1　感知机模型示意图

感知机模型的几何解释如图 5-2 所示。求和节点代表一个分离超平面，它将“+”和“−”类分开，因此感知机模型也是支持向量机的基础。

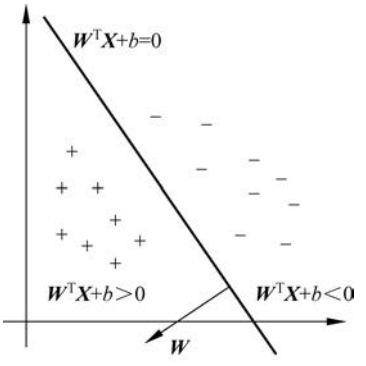

图 5-2　感知机模型几何解释

5.1.2　感知机和布尔运算

运用感知机模型可以表达常见的原子布尔运算。以与运算(AND)为例,AND 的运算法则见表 5-1,将 $\boldsymbol{X}=(x_1,x_2)^{\mathrm{T}}$ 看作输入向量,x_1 AND x_2 有 0 和 1 两种结果,这可以看成两个类别,用“－”来表示“0”类,用“＋”来表示“1”类。输入$(x_1,x_2)^{\mathrm{T}}$ 和 x_1 AND x_2 的位置关系如图 5-3 所示。从图 5-3 可以显然得出至少存在一个分离超平面将“＋”类和“－”类完全分离。也就是说,布尔运算 AND 可以用感知机来表示。同样的结论可以推广到或(OR)和非(NOT)两种运算。

表 5-1　AND 运算法则

x_1	x_2	x_1 AND x_2
1	1	1
0	1	0
1	0	0
0	0	0

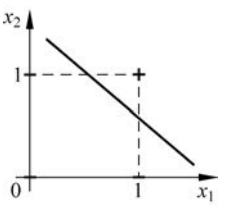

图 5-3　用感知机表示 AND 运算

但并不是所有原子布尔运算都可以用一个感知机来表示。异或运算(XOR)的运算法则见表 5-2,其输入和输出的位置关系如图 5-4 所示。显然,将两类结果完全分开的分离超平面是不存在的。

表 5-2　XOR 运算法则

x_1	x_2	x_1 XOR x_2
1	1	0
0	1	1
1	0	1
0	0	0

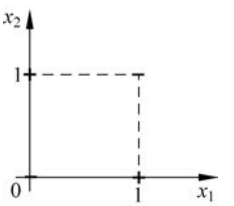

图 5-4　用感知机表示 XOR 运算

我们尝试用两个感知机所组成的网络来解决异或运算的表示问题,由两个感知机组成的一个网络拓扑,如图 5-5 所示。输入$(x_1,x_2)^{\mathrm{T}}$ 分别经过两个感知机,其结果再汇聚到输出节点。取激活函数为 $\max(0,u)$,令输入特征矩阵、第 1 层权值矩阵、第 1 层偏置向量、第 2 层权值向量分别为

$$\boldsymbol{X}=\begin{pmatrix}0&0\\0&1\\1&0\\1&1\end{pmatrix},\quad \boldsymbol{W}=\begin{pmatrix}1&1\\1&1\end{pmatrix},\quad \boldsymbol{C}=\begin{pmatrix}0\\-1\end{pmatrix},\quad \boldsymbol{V}=\begin{pmatrix}1\\-2\end{pmatrix} \tag{5-4}$$

则第 1 层求和节点为

$$\boldsymbol{U}=\boldsymbol{XW}+\boldsymbol{C}=\begin{pmatrix}0 & -1\\ 1 & 0\\ 1 & 0\\ 2 & 1\end{pmatrix} \tag{5-5}$$

经 max(0,u)函数激活后为

$$\boldsymbol{Z}=\max\left(0,\begin{pmatrix}0 & -1\\ 1 & 0\\ 1 & 0\\ 2 & 1\end{pmatrix}\right)=\begin{pmatrix}0 & 0\\ 1 & 0\\ 1 & 0\\ 2 & 1\end{pmatrix} \tag{5-6}$$

继续向前传播到第 2 层求和节点为

$$\boldsymbol{Y}=\boldsymbol{ZW}=\begin{pmatrix}0 & 0\\ 1 & 0\\ 1 & 0\\ 2 & 1\end{pmatrix}\begin{pmatrix}1\\ -2\end{pmatrix}=\begin{pmatrix}0\\ 1\\ 1\\ 0\end{pmatrix} \tag{5-7}$$

这正好是异或运算的结果。也就是说,在图 5-5 的网络拓扑下,按照式(5-4)选择权重和偏置就可以用感知机组成的网络表示异或运算,整个运算过程如图 5-6 所示。

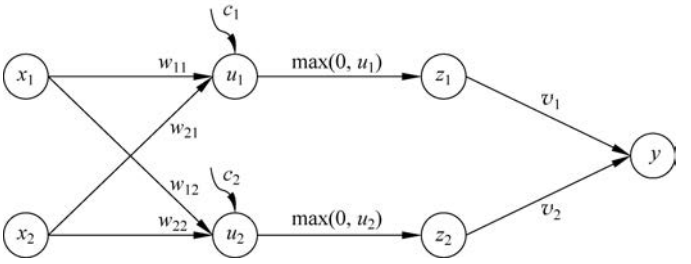

图 5-5　表达异或(XOR)运算的网络拓扑

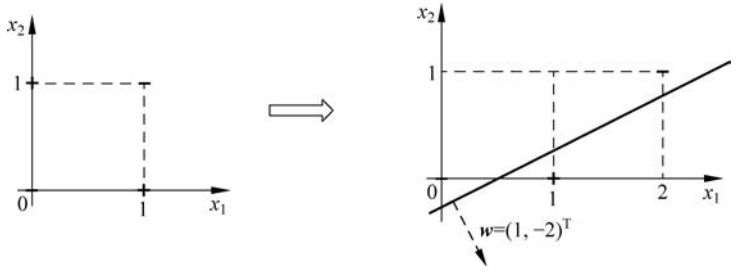

图 5-6　异或运算的几何表达

事实上,可以证明,用感知机组成的网络可以表示所有原子布尔运算。这一结论是重要的,因为所有布尔函数都可以表示为原子布尔函数的互连单元的网络,也就是说,所有布尔

函数都可以用感知机组成的网络来表示，这正是神经网络的理论基础。事实上，已经证明，任意复杂的函数都可以用适当规模的神经网络模型来表示。

5.2　深度神经网络

深度神经网络是深度学习的核心，而深度神经网络有各种各样的结构，主要分为前馈式神经网络、卷积神经网络、循环神经网络等。其中最早研究，并且结构最简单的深度神经网络是前馈式深度神经网络，本节就以此为例介绍深度神经网络的网络拓扑、前向传播、训练模型和误差反向传播。

5.2.1　网络拓扑

一个完整的深度神经网络包括一个输入层、一个或多个隐含层和一个输出层，如图 5-7 所示。

假设某个问题的带标签的训练数据为 $\{\boldsymbol{X}^i,\boldsymbol{Y}^i\}_{i=1}^N$，其中 $\boldsymbol{X}^i=(x_1^i,x_2^i,\cdots,x_n^i)^{\mathrm{T}}\in\mathbf{R}^n$ 称为观测或输入，$\boldsymbol{Y}^i=(y_1^i,y_2^i,\cdots,y_m^i)^{\mathrm{T}}\in\mathbf{R}^m$ 称为输出，N 为训练数据的总条数，$\boldsymbol{Y}^i$ 通常表达了 $\boldsymbol{X}^i$ 所属的类别或 $\boldsymbol{X}^i$ 经过某种映射后的值，所以 $\boldsymbol{Y}^i$ 也称为 $\boldsymbol{X}^i$ 的标签。在图 5-7 的网络拓扑中，输入层的作用是将观测 $\boldsymbol{X}^i$ 输入网络中，输出层的作用是预测 $\boldsymbol{X}^i$ 的标签，一般用 $\hat{\boldsymbol{Y}}^i$ 表示。预测标签 $\hat{\boldsymbol{Y}}^i$ 可能等于真实标签 $\boldsymbol{Y}^i$，也可能不等，隐含层的作用就是找到一组合适的参数 $\boldsymbol{W}$，使输入 $\boldsymbol{X}^i$ 经过网络映射后得到的预测标签尽量等于真实标签 $\boldsymbol{Y}^i$，这也是深度神经网络的终极目标。

显然，输入层和输出层节点数一般要等于训练数据的输入和输出的维度，即 n 和 m，但隐含层的节点数可以任意取，只要能保证各层节点的逻辑连接合理即可。

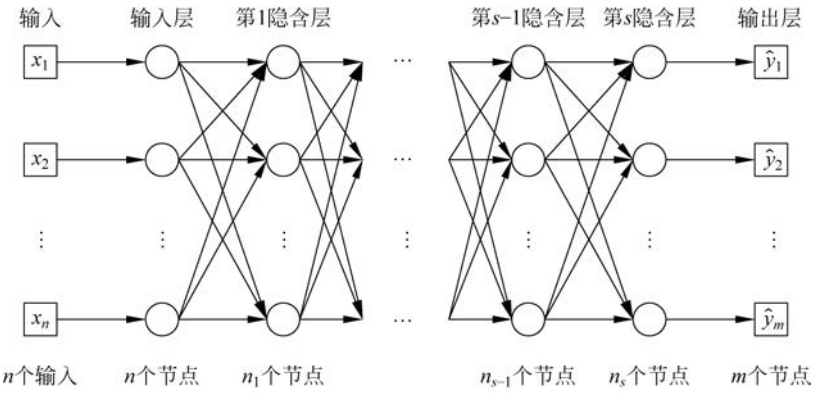

图 5-7　深度神经网络示意图

5.2.2 前向传播

前向传播是指数据从输入层进入神经网络，逐层流经神经网络的各层，并被各层的参数和激活函数映射，最后从输出层流出的过程。

作为示例，仅从第 $l-1$ 隐含层到第 l 隐含层的传播过程为例，如图 5-8 所示，第 $l-1$ 隐含层的最后输出是 $\boldsymbol{Y}^{(l-1)}=(y_1^{(l-1)},y_2^{(l-1)},\cdots,y_{n_{l-1}}^{(l-1)})^{\mathrm{T}}\in\mathbf{R}^{n_{l-1}}$，从第 $l-1$ 隐含层到第 l 隐含层的权重 $\boldsymbol{W}^{(l)}=(w_{ij}^{(l)})_{n_{l-1}\times n_l}$ 为一个 $n_{l-1}\times n_l$ 矩阵，其中 n_{l-1} 和 n_l 分别为第 $l-1$ 隐含层和第 l 隐含层的节点数，数据从第 $l-1$ 隐含层传递到第 l 隐含层要经过两个步骤：

(1) 线性映射过程，即

$$z_j^{(l)}=\sum_{i=1}^{n_{l-1}}w_{ij}^{(l)}y_i^{(l-1)}+b_j^{(l)},\quad j=1,2,\cdots,n_l \tag{5-8}$$

其中，$\boldsymbol{B}^{(l)}=(b_1^{(l)},b_2^{(l)},\cdots,b_{n_l}^{(l)})$ 为第 l 隐含层偏置。

(2) 激活映射过程，即

$$y_j^{(l)}=\sigma(z_j^{(l)}),\quad j=1,2,\cdots,n_l \tag{5-9}$$

其中，σ 为激活函数。

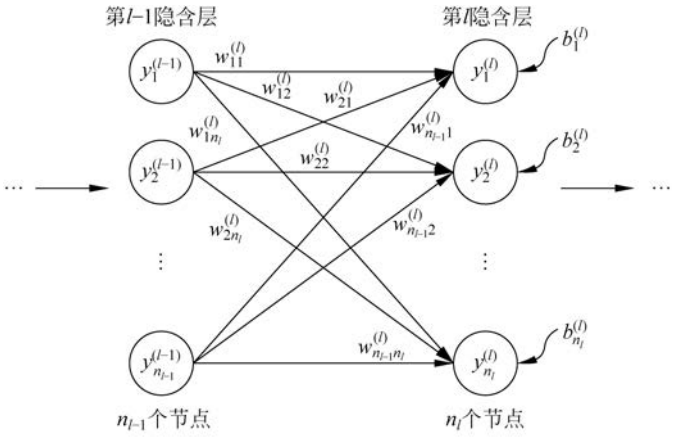

图 5-8 前向传播过程示意图

上述过程可以简写成矩阵形式，即

$$\boldsymbol{Y}^{(l)}=\sigma((\boldsymbol{W}^{(l)})^{\mathrm{T}}\boldsymbol{Y}^{(l-1)}+\boldsymbol{B}^{(l)}) \tag{5-10}$$

于是，整个前向传播可以写成以下复合函数：

$$\begin{aligned}\hat{\boldsymbol{Y}}&\triangleq\boldsymbol{Y}^{(s)}\\&=(\boldsymbol{W}^{(s)})^{\mathrm{T}}(\sigma(\cdots\sigma((\boldsymbol{W}^{(2)})^{\mathrm{T}}(\sigma((\boldsymbol{W}^{(1)})^{\mathrm{T}}\boldsymbol{Y}^{(1)}+\boldsymbol{B}^{(1)}))+\boldsymbol{B}^{(2)})\cdots))+\boldsymbol{B}^{(s)}\end{aligned} \tag{5-11}$$

整个前向传播过程的数据流向如式(5-12)所示。

$$\boldsymbol{X} \triangleq \boldsymbol{Y}^{(0)} \xrightarrow{\boldsymbol{Z}^{(1)}} \boldsymbol{Y}^{(1)} \to \cdots \to \boldsymbol{Y}^{(l-1)} \xrightarrow{\boldsymbol{Z}^{(l)}} \boldsymbol{Y}^{(l)} \to \cdots \to \boldsymbol{Y}^{(s)} \triangleq \hat{\boldsymbol{Y}} \tag{5-12}$$

从式(5-11)可以看出，深度神经网络其实是由线性函数和激活函数经过多层复合而成的一个高度非线性的映射，其非线性源自于激活函数。

5.2.3　训练模型

从前向传播过程可知，深度神经网络实际上就是一个高度非线性函数，这个函数可以完成对输入数据进行回归预测或分类的任务，但深度神经网络有大量的参数，确定这些参数是深度神经网络准确地完成回归预测和分类任务的前提，所谓的训练神经网络就是利用已知的先验知识(训练数据)来确定神经网络参数的过程。从数学上看，训练神经网络实际上是求解一个优化问题，该优化问题就是训练模型。

与5.2.1节中对训练数据的假设一样，设某一问题的带标签训练数据为$\{\boldsymbol{X}^i, \boldsymbol{Y}^i\}_{i=1}^N$，$\boldsymbol{X}^i$经过神经网络映射后得到预测输出$\hat{\boldsymbol{Y}}^i$，而根据训练数据，$\boldsymbol{X}^i$的目标输出应为$\boldsymbol{Y}^i$，当然，神经网络应该使$\hat{\boldsymbol{Y}}^i$和$\boldsymbol{Y}^i$越接近越好。当考虑所有训练数据时，应该让预测输出和目标输出的均方误差越小越好，即优化问题

$$\min_{\boldsymbol{W}} L \triangleq \frac{1}{N}\sum_{i=1}^{N} \| \hat{\boldsymbol{Y}}^i - \boldsymbol{Y}^i \|_2^2 \tag{5-13}$$

其中，$\| \cdot \|_2$为向量的2-范数，$\hat{\boldsymbol{Y}}^i$由$\boldsymbol{X}^i$经过深度神经网络映射得到。式(5-13)的目标函数在深度学习中一般称为损失函数，其决策变量就是深度神经网络的所有参数。从这一点来讲，深度神经网络的训练模型其实是一个参数优化问题。

均方误差损失函数只是常见损失函数的一种，主要用于训练回归预测的深度神经网络。根据神经网络解决的问题和拓扑结构的不同还有很多其他损失函数，具体将在5.3节详细介绍。

5.2.4　误差反向传播

优化问题式(5-13)并不好解。首先，对于参数量非常大或训练数据量非常大的问题，损失函数的计算开销是巨大的，常规优化方法是无能为力的；其次，由于式(5-11)的嵌套复合结构，若要像常规梯度下降算法一样一次性计算所有参数的梯度是不现实的，所以需要有特殊的梯度下降方案。

针对第一点困难的解决方案是小批量梯度下降(Mini-Batch Gradient Descent，MBGD)方法，每次只从所有训练数据中抽取一个小的批量组成一个优化问题。如随机抽取一个小批量数据$\{\boldsymbol{X}^i, \boldsymbol{Y}^i\}_{i=1}^B \subset \{\boldsymbol{X}^i, \boldsymbol{Y}^i\}_{i=1}^N$，求解优化问题

$$\min_{\boldsymbol{W}} L_B \triangleq \frac{1}{B}\sum_{i=1}^{B} \| \hat{\boldsymbol{Y}}^i - \boldsymbol{Y}^i \|_2^2 \tag{5-14}$$

通过小批量抽取并采用多次优化计算的方案解决数据量过大的问题。使用这种方案求出的参数$\boldsymbol{W}$并不一定总能使式(5-13)中的损失函数下降，但可以证明，它是以概率收敛到一个

局部最优解的，如图 5-9 所示。因为训练数据抽取的随机性和 $\boldsymbol{W}$ 向局部最优解收敛的过程的随机性，这种方法也叫小批量随机梯度下降（Mini-Batch Stochastic Gradient Descent，MBSGD）算法。

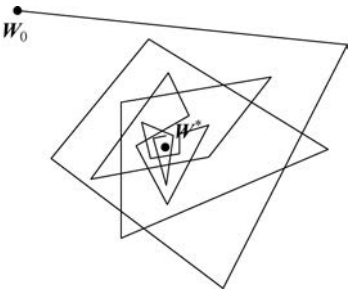

图 5-9　小批量随机梯度下降算法收敛过程示意图

针对第二点困难的解决方案是误差反向传播机制。与前向传播过程正好相反，误差反向传播是指将预测输出和目标输出的误差通过复合函数求导的链式法则逐层反向传播给各层参数，各层参数再利用反向传播的误差来调整自己，以达到训练网络的目的。

先计算损失函数关于最后一层参数和偏置的导数，考虑到最后一层的线性映射

$$\hat{\boldsymbol{Y}} \triangleq \boldsymbol{Y}^{(s)} = (\boldsymbol{W}^{(s)})^{\mathrm{T}}\boldsymbol{Y}^{(s-1)} + \boldsymbol{B}^{(s)} \tag{5-15}$$

和损失函数

$$\boldsymbol{L}_B \triangleq \frac{1}{B}\sum_{i=1}^{B}(\hat{\boldsymbol{Y}}^i - \boldsymbol{Y}^i)^2 = \frac{1}{B}\sum_{i=1}^{B}(\boldsymbol{Y}^{i(s)} - \boldsymbol{Y}^i)^{\mathrm{T}}(\boldsymbol{Y}^{i(s)} - \boldsymbol{Y}^i) \tag{5-16}$$

根据复合函数求导的链式法则，得

$$\begin{aligned}\frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{W}^{(s)}} &= \frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{Y}^{(s)}}\frac{\partial \boldsymbol{Y}^{(s)}}{\partial \boldsymbol{W}^{(s)}} \\ &= \frac{2}{B}\sum_{i=1}^{B}\mathrm{rep}(\mathrm{sum}(\boldsymbol{Y}^{i(s)} - \boldsymbol{Y}^i), n_{s-1}, n_s) \otimes \mathrm{rep}(\boldsymbol{Y}^{i(s-1)}, 1, n_s) \\ \frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{B}^{(s)}} &= \frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{Y}^{(s)}}\frac{\partial \boldsymbol{Y}^{(s)}}{\partial \boldsymbol{B}^{(s)}} = \frac{2}{B}\sum_{i=1}^{B}\mathrm{rep}(\mathrm{sum}(\boldsymbol{Y}^{i(s)} - \boldsymbol{Y}^i), n_s, 1)\end{aligned} \tag{5-17}$$

其中，$\mathrm{rep}(x, m, n)$ 是广播函数，其作用是将 x 复制成一个 $m \times n$ 维矩阵；$\mathrm{sum}(\boldsymbol{Y})$ 将 $\boldsymbol{Y}$ 的各分量相加；$\otimes$ 指矩阵的各分量分别相乘（Component-wise Multiplication）。

再计算从第 l 隐含层到第 $l-1$ 隐含层的误差传递，考虑到从第 $l-1$ 隐含层到第 l 隐含层的前向传播函数为

$$\begin{aligned}\boldsymbol{Z}^{(l)} &= (\boldsymbol{W}^{(l)})^{\mathrm{T}}\boldsymbol{Y}^{(l-1)} + \boldsymbol{B}^{(l)} \\ \boldsymbol{Y}^{(l)} &= \sigma(\boldsymbol{Z}^{(l)})\end{aligned} \tag{5-18}$$

于是

$$\frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{Y}^{(l)}}\frac{\partial \boldsymbol{Y}^{(l)}}{\partial \boldsymbol{Z}^{(l)}}\frac{\partial \boldsymbol{Z}^{(l)}}{\partial \boldsymbol{W}^{(l)}}$$

$$
\begin{aligned}
&=\mathrm{rep}\left(\left(\frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{Y}^{(l)}}\otimes\sigma'(\boldsymbol{Z}^{(l)})\right)^{\mathrm{T}},n_{l-1},1\right)\otimes\mathrm{rep}(\boldsymbol{Y}^{(l-1)},1,n_l)\\
\frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{B}^{(l)}}&=\frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{Y}^{(l)}}\frac{\partial \boldsymbol{Y}^{(l)}}{\partial \boldsymbol{Z}^{(l)}}\frac{\partial \boldsymbol{Z}^{(l)}}{\partial \boldsymbol{B}^{(l)}}=\frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{Y}^{(l)}}\otimes\sigma'(\boldsymbol{Z}^{(l)})
\end{aligned}
\tag{5-19}
$$

其中，$\partial \boldsymbol{L}_B/\partial \boldsymbol{Y}^{(l)}$ 可由第 $l+1$ 隐含层的 $\partial \boldsymbol{L}_B/\partial \boldsymbol{Y}^{(l+1)}$ 和从第 l 隐含层到第 $l+1$ 隐含层的前向传播函数计算得出，即

$$
\begin{aligned}
\frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{Y}^{(l)}}&=\frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{Y}^{(l+1)}}\frac{\partial \boldsymbol{Y}^{(l+1)}}{\partial \boldsymbol{Y}^{(l)}}\\
&=(\boldsymbol{W}^{(l+1)}\otimes\mathrm{rep}((\sigma'(\boldsymbol{Z}^{(l+1)}))^{\mathrm{T}},n_l,1))\frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{Y}^{(l+1)}}
\end{aligned}
\tag{5-20}
$$

这样，小批量训练数据的误差就可以自后向前一直传播到第 1 隐含层。图 5-10 描述了误差反向传播的过程。

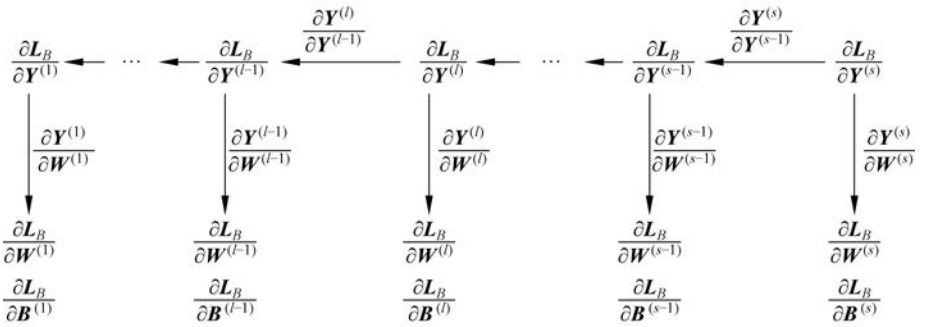

图 5-10　误差反向传播过程示意图

各层获得反向传播回的误差以后，根据梯度下降算法更新参数，即

$$
\begin{aligned}
\boldsymbol{W}^{(l)}&\leftarrow\boldsymbol{W}^{(l)}-\eta\frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{W}^{(l)}},\quad l=s,s-1,\cdots,1\\
\boldsymbol{B}^{(l)}&\leftarrow\boldsymbol{B}^{(l)}-\eta\frac{\partial \boldsymbol{L}_B}{\partial \boldsymbol{B}^{(l)}},\quad l=s,s-1,\cdots,1
\end{aligned}
\tag{5-21}
$$

这里 η 是更新步长(Step Size)，在深度学习中一般称为学习率(Learning Rate，LR)。学习率在训练过程中可以自适应地调整，反向传播的梯度也可以在传播过程中使用优化方法进行加速，这就可以得到不同的优化算法，如 SGD、Adam、Adagrad 等。关于优化方法的理论讨论已经超出了本书范围，感兴趣的读者可以查阅相关资料，5.4 节将介绍 PyTorch 中已经封装好的优化器的使用方法。

5.3　激活函数、损失函数和数据预处理

84min

本节介绍深度神经网络中常用的激活函数、损失函数及数据预处理方法。

5.3.1 激活函数

激活函数是深度神经网络的重要组成部分，也是其非线性的唯一来源。本节首先给出激活函数的一般性质，再介绍常见的激活函数。

激活函数的一般性质如下：

(1) 单调可微性，激活函数一般是单调可微的，单调性是为了保证在数据的前向传播过程中，输出随着输入的增加而增加，反之亦然，这样便于计算。可微性是为了保证在误差的反向传播过程中可计算，因为每层的误差传播都包含激活函数的导数项。

(2) 非线性，激活函数的非线性是深度神经网络非线性的唯一来源。

(3) 导数有界性，激活函数的导数必须是有界函数，这是为了保证在误差反向传播过程中不因为激活函数导数过大而出现梯度爆炸现象。

以下介绍常见的激活函数及其性质和优缺点。

1. Sigmoid 函数

Sigmoid 函数为

$$\sigma(x)=\frac{1}{1-\mathrm{e}^{-x}} \tag{5-22}$$

它的导函数为

$$\sigma'(x)=\sigma(x)(1-\sigma(x)) \tag{5-23}$$

它们的图像如图 5-11 所示。

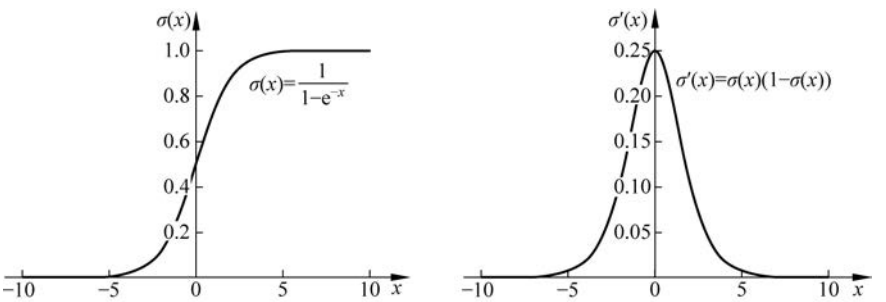

图 5-11 Sigmoid 激活函数及其导函数

以下是对 Sigmoid 函数的相关说明：

(1) 可以把 Sigmoid 函数想象成一个神经元的放电率，在中间斜率比较大的地方是神经元的敏感区，在两边斜率很平缓的地方是神经元的抑制区。

(2) 当输入稍微远离了坐标原点时，函数的导数就变得很小了，几乎为 0。在神经网络反向传播过程中，这会导致反向传播的梯度越来越小，以至于对权重的改变几乎没有影响，这不利于权重的优化，这种现象叫作梯度饱和或梯度弥散。

(3) Sigmoid 函数适用于二分类问题，但是它的输出不是以 0 为中心的，对于以 −1 和 1 为标签的二分类训练数据不适用。

(4) Sigmoid 函数及其导函数的计算都涉及指数函数，计算量比较大。

2. Tanh 函数

Tanh 函数为

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

它的导函数为

$$[\tanh(x)]'=1-\tanh^{2}(x) \tag{5-25}$$

它们的图像如图 5-12 所示。

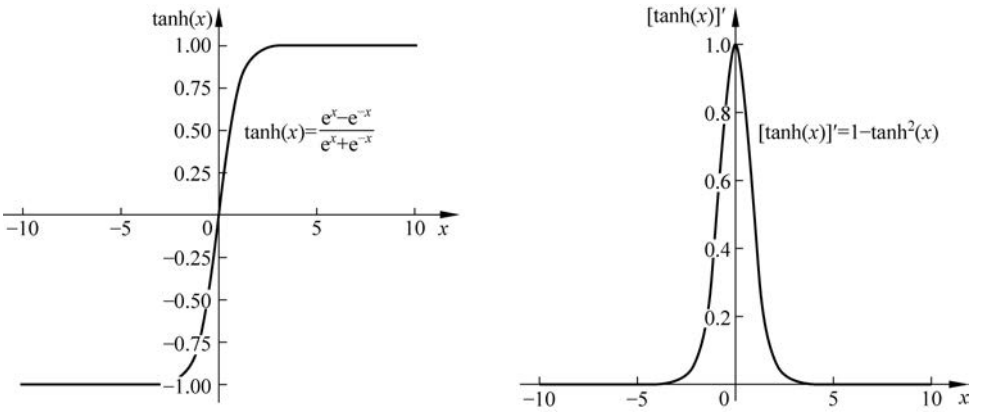

图 5-12　Tanh 激活函数及其导函数

以下是对 Tanh 函数的相关说明：

(1) Tanh 函数和 Sigmoid 函数的形状及导函数基本一样，不同的是 Tanh 函数的取值范围是[−1,1]，以 0 为中心，适用于以−1 和 1 为标签的二分类问题。

(2) 与 Sigmoid 函数一样，Tanh 函数当输入稍微远离了坐标原点时，函数的导数就变得很小了，几乎为 0，在神经网络反向传播过程中会造成梯度弥散问题。

(3) Tanh 函数及其导函数的计算也涉及指数函数，计算量比较大。

3. ReLU 函数

ReLU 函数为

$$f(x)=\max\{0,x\} \tag{5-26}$$

它的导函数为

$$f'(x)=\begin{cases}1, & x>0\\ 0, & x<0\end{cases} \tag{5-27}$$

它们的图像如图 5-13 所示。

以下是对 ReLU 函数的相关说明：

(1) ReLU 函数的最大优点是计算简单，一般在深度神经网络的中间隐含层使用。

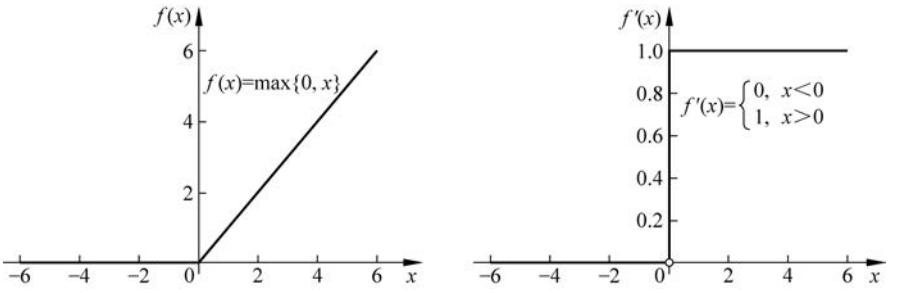

图 5-13 ReLU 激活函数及其导函数

(2) 当输入为正时,ReLU 函数的梯度恒为 1,不会出现梯度爆炸现象;但当输入为负时,ReLU 函数的梯度恒为 0,会出现梯度弥散现象。

(3) ReLU 函数的输出不对称。

4. ELU 函数

ELU 函数为

$$f(x)=\begin{cases}x, & x>0\\ \alpha(e^{x}-1), & x\leqslant 0\end{cases} \tag{5-28}$$

它的导函数为

$$f'(x)=\begin{cases}1, & x>0\\ \alpha e^{x}, & x\leqslant 0\end{cases} \tag{5-29}$$

它们的图像如图 5-14 所示。

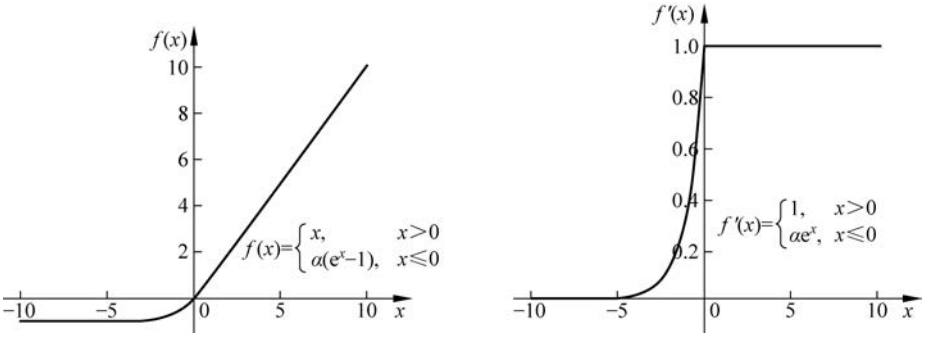

图 5-14 ELU 激活函数及其导函数

以下是对 ELU 函数的相关说明:

(1) ELU 函数是对 ReLU 函数的改进,主要改进点为当 $x=0$ 时,ELU 函数是光滑的,导数存在。

(2) 当 $x<0$ 且远离 0 时,ELU 函数的导数很接近 0,会出现梯度弥散问题。

(3) 相较于 ReLU 函数,ELU 函数的计算涉及指数函数,计算量更大一些,所以在实际使用中不及 ReLU 函数普遍。

5. PReLU 函数

PReLU 函数为

$$f(x)=\begin{cases}x, & x>0\\ \alpha x, & x\leqslant 0\end{cases} \tag{5-30}$$

它的导函数为

$$f'(x)=\begin{cases}1, & x>0\\ \alpha, & x<0\end{cases} \tag{5-31}$$

当 $\alpha=0.01$ 时，PReLU 函数退化为 Leaky ReLU 函数。

以下是对 PReLU 函数的相关说明：

(1) PReLU 函数是 ReLU 函数的另一个更简单的改进，主要修正了当 $x<0$ 时导数为 0 的问题，相较于 ELU 函数，其计算更为简单。

(2) 实验表明，Leaky ReLU 函数和 ReLU 函数的计算表现差别不大。

6. Softmax 函数

Softmax 函数是一种处理多分类问题的激活函数，一般用在神经网络输出层之后的多分类环节。

设神经网络的输出为 $\hat{\boldsymbol{Y}}=(y_1,y_2,\cdots,y_K)^{\mathrm{T}}$，其中 K 为总类别数，也是神经网络的输出层节点数，则 Softmax 函数为

$$p_i=\frac{\mathrm{e}^{y_i}}{\sum_{j=1}^{K}\mathrm{e}^{y_j}} \tag{5-32}$$

所以，Softmax 函数的输出 $\boldsymbol{P}=(p_1,p_2,\cdots,p_K)^{\mathrm{T}}$ 实际上是一个概率分布向量，利用该向量可以对神经网络的输入 $\boldsymbol{X}$ 进行分类。

例如，某输入 $\boldsymbol{X}$ 经过神经网络映射后预测输出为 $\hat{\boldsymbol{Y}}=(3,1,-3)^{\mathrm{T}}$，则

$$\sum=\sum_{i=1}^{3}\mathrm{e}^{y_i}=\mathrm{e}^3+\mathrm{e}^1+\mathrm{e}^{-3}=22.75$$

而

$$p_1=\frac{\mathrm{e}^{y_1}}{\sum}=\frac{\mathrm{e}^3}{\sum}=0.88$$

$$p_2=\frac{\mathrm{e}^{y_2}}{\sum}=\frac{\mathrm{e}^1}{\sum}=0.12$$

$$p_3=\frac{\mathrm{e}^{y_3}}{\sum}=\frac{\mathrm{e}^{-3}}{\sum}=0$$

故该输入 $\boldsymbol{X}$ 应归为第 1 类。

取定 $i=1,2,\cdots,K$，对 Softmax 函数的第 i 个分量 p_i 关于 y_k 求偏导，即当 $k=i$ 时，有

$$\frac{\partial p_i}{\partial y_k}=p_i(1-p_k) \tag{5-33}$$

当 $k\neq i$ 时，有

$$\frac{\partial p_i}{\partial y_k}=-p_i p_k \tag{5-34}$$

5.3.2 损失函数

损失函数是深度神经网络训练模型的重要组成部分，不同深度学习任务使用不同的损失函数，不同的损失函数也有各自的特点，本节介绍常见的损失函数及其特点。

1. L1 范数损失函数

L1 范数损失函数用于衡量小批量数据的预测输出和目标输出的绝对误差之和或均值。

设小批量训练数据为 $\{\boldsymbol{X}^i,\boldsymbol{Y}^i\}_{i=1}^B$，$\boldsymbol{X}^i$ 的预测输出为 $\hat{\boldsymbol{Y}}^i=\{\hat{y}_1^i,\hat{y}_2^i,\cdots,\hat{y}_m^i\}$，其中 m 为输出向量维度，则 L1 范数损失函数为

$$L_B\triangleq\sum_{i=1}^{B}\sum_{j=1}^{m}|\hat{y}_j^i-y_j^i| \quad 或 \quad L_B\triangleq\frac{1}{Bm}\sum_{i=1}^{B}\sum_{j=1}^{m}|\hat{y}_j^i-y_j^i| \tag{5-35}$$

L1 范数损失函数是不可微的，这会对训练模型的优化过程带来一些困难。

2. 均方误差损失函数

均方误差损失函数用于衡量小批量数据的预测输出和目标输出的误差平方之和或均值。

设小批量训练数据为 $\{\boldsymbol{X}^i,\boldsymbol{Y}^i\}_{i=1}^B$，$\boldsymbol{X}^i$ 的预测输出为 $\hat{\boldsymbol{Y}}^i=\{\hat{y}_1^i,\hat{y}_2^i,\cdots,\hat{y}_m^i\}$，其中 m 为输出向量维度，则均方误差损失函数为

$$L_B\triangleq\sum_{i=1}^{B}\sum_{j=1}^{m}(\hat{y}_j^i-y_j^i)^2 \quad 或 \quad L_B\triangleq\frac{1}{Bm}\sum_{i=1}^{B}\sum_{j=1}^{m}(\hat{y}_j^i-y_j^i)^2 \tag{5-36}$$

均方误差损失函数是二次凸函数，有比较成熟的优化算法，是使用比较多的损失函数，一般在回归预测问题中使用。

3. 负对数似然损失函数

负对数似然损失函数主要应用于 Logistic 回归（二分类问题）中，所以也称为 Logistic 回归损失函数。

假设小批量训练数据为 $\{\boldsymbol{X}^i,y_i\}_{i=1}^B$，其中 $y_i\in\{0,1\}$ 表示二分类问题的标签，设 $\boldsymbol{X}^i$ 经神经网络映射后的输出是 $\hat{y}_i=h_\theta(\boldsymbol{X}^i)$，这里 h 函数代表神经网络，θ 代表神经网络的参数，对于二分类问题有 $0\leqslant\hat{y}_i\leqslant1$，表示预测分类的概率，则经神经网络预测的样本 i 的条件概率分布为

$$p(y_i\mid\boldsymbol{X}^i;\theta)\triangleq(h_\theta(\boldsymbol{X}^i))^{y_i}(1-h_\theta(\boldsymbol{X}^i))^{1-y_i} \tag{5-37}$$

所有小批量样本的似然函数为

$$E \triangleq \prod_{i=1}^{B} p(y_i \mid \boldsymbol{X}^i;\theta) = \prod_{i=1}^{B} (h_\theta(\boldsymbol{X}^i))^{y_i}(1-h_\theta(\boldsymbol{X}^i))^{1-y_i} \tag{5-38}$$

为了计算方便，并考虑到最大化似然函数等价于最小化负对数似然函数，将式(5-38)写成负对数似然损失函数为

$$L_B \triangleq -\sum_{i=1}^{B} y_i \log(h_\theta(\boldsymbol{X}^i)) + (1-y_i)\log(1-h_\theta(\boldsymbol{X}^i)) \tag{5-39}$$

负对数似然损失函数也可以推广到多分类问题，它的推广形式和接下来要介绍的交叉熵损失函数等价。

4. 交叉熵损失函数

设 $x \in \boldsymbol{X}$ 是一个随机变量，$p(x)$是其概率分布，则一个事件 x 的信息量用

$$I(x) \triangleq -\log p(x) \tag{5-40}$$

来衡量。显然，如果一个事件发生的概率越确定(越大)，则其信息量就越小；若该事件以概率 1 发生，则信息量为 0。对随机变量的信息量关于分布 p 求期望即可得分布 p 的熵(Entropy)，即

$$H(p) \triangleq E_p(I(x)) = -\sum_{x \in \boldsymbol{X}} p(x)\log p(x) \tag{5-41}$$

交叉熵(Cross Entropy)是指随机分布 q 的信息量关于随机分布 p 的期望，即

$$\mathrm{CE}(p,q) \triangleq -\sum_{x \in \boldsymbol{X}} p(x)\log q(x) \tag{5-42}$$

交叉熵可以用于衡量分布 p 和 q 的近似程度，根据这一点可以得到交叉熵损失函数。

交叉熵损失函数主要应用于多分类问题的训练模型中，用于衡量小批量数据的预测输出和目标输出的交叉熵之和或均值。

设小批量训练数据为 $\{\boldsymbol{X}^i, \boldsymbol{Y}^i\}_{i=1}^B$，其中 $\boldsymbol{Y}^i$ 是 One-Hot 向量，代表输入 $\boldsymbol{X}^i$ 归属的类别，即 $\boldsymbol{Y}^i = (y_1^i, y_2^i, \cdots, y_m^i)$，并且 $y_j^i \in \{0,1\}$，$j=1,2,\cdots,m$。$\boldsymbol{X}^i$ 经过神经网络映射后的预测输出 $\hat{\boldsymbol{Y}}^i = \{\hat{y}_1^i, \hat{y}_2^i, \cdots, \hat{y}_m^i\}$ 再经过 Softmax 函数映射后得到概率分布向量 $\boldsymbol{P}^i = (p_1^i, p_2^i, \cdots, p_m^i)^{\mathrm{T}}$，则交叉熵损失函数为

$$L_B \triangleq -\frac{1}{B}\sum_{i=1}^{B}\sum_{j=1}^{m} y_j^i \ln p_j^i \quad \text{或} \quad L_B \triangleq -\frac{1}{Bm}\sum_{i=1}^{B}\sum_{j=1}^{m} y_j^i \ln p_j^i \tag{5-43}$$

式(5-43)中后一个求和就是预测输出 $\boldsymbol{P}^i$ 和目标输出 $\boldsymbol{Y}^i$ 的交叉熵。当 $m=2$ 时，多分类问题退化为二分类问题，交叉熵损失函数退化为 Logistic 回归损失函数。

5. KL 散度损失函数

KL 散度(Kullback-Leibler Divergence)又称为相对熵(Relative Entropy)或 KL 距离，是两个随机分布距离的一种度量，记为 $D_{\mathrm{KL}}(p \parallel q)$。它的意义是度量当真实分布为 p 时，假设分布 q 的无效性，即

$$D_{\mathrm{KL}}(p \parallel q) \triangleq E_p\left[\log \frac{p(x)}{q(x)}\right] = \sum_{x \in \boldsymbol{X}} p(x)\log \frac{p(x)}{q(x)}$$

$$\begin{aligned}&=\sum_{x\in\boldsymbol{X}}[p(x)\log p(x)-p(x)\log q(x)]\\&=-H(p)+\mathrm{CE}(p,q)\end{aligned}\tag{5-44}$$

显然，当 $p=q$ 时，$D_{\mathrm{KL}}(p\|q)=0$。

从式(5-44)可以看出交叉熵 $\mathrm{CE}(p,q)$ 等于 KL 散度 $D_{\mathrm{KL}}(p\|q)$ 和熵 $H(p)$ 之和，当分布 p 确定时，$H(p)$ 为常数，所以最小化 KL 散度等价于最小化交叉熵，从这一点上看，KL 散度损失函数和交叉熵损失函数是等价的，但由于交叉熵损失函数更易于计算，所以在实际使用中更为普遍。

6. Hinge 损失函数

Hinge 损失函数主要应用于 maximum-margin 分类任务中，典型的应用场景是支持向量机，所以也称为支持向量机损失函数。

假设小批量训练数据为 $\{\boldsymbol{X}^i,y_i\}_{i=1}^B$，其中 $y_i\in\{1,2,\cdots,K\}$ 为 $\boldsymbol{X}^i$ 的类别，K 为总类别数，$\boldsymbol{X}^i$ 经神经网络映射后的输出为 $\hat{\boldsymbol{Y}}^i=(\hat{y}_1^i,\hat{y}_2^i,\cdots,\hat{y}_K^i)$。这里，$\hat{y}_j^i,j=1,2,\cdots,K$ 可以看作对 $\boldsymbol{X}^i$ 归属于各类别的打分，其中只有一个打分是针对正确分类的，即 $\hat{\boldsymbol{Y}}^i$ 中下标为 y_i 的分量，其他均为针对错误分类的打分。Hinge 损失函数的思想是增加对正确分类的打分，降低对错误分类的打分，但又要让对正确类别的打分和对错误类别的打分的差值控制在一定的范围内，即 margin，所以对于训练样本 $(\boldsymbol{X}^i,y_i)$ 的 Hinge 损失为

$$l(\boldsymbol{X}^i,y_i)\triangleq\sum_{j=1}^{K}\max\{0,M-(\hat{y}_{y_i}^i-\hat{y}_j^i)\}\tag{5-45}$$

这里 M 表示正确和错误分类打分之差的控制范围，若超出该范围，则不会得到任何奖励，即损失为 0。在支持向量机中，M 相当于分离超平面的间隙。

对于所有小批量样本，Hinge 损失函数为

$$L_B\triangleq\frac{1}{B}\sum_{i=1}^{B}\sum_{j=1}^{K}\max\{0,M-(\hat{y}_{y_i}^i-\hat{y}_j^i)\}\tag{5-46}$$

5.3.3 数据预处理

数据预处理是指在将数据输入神经网络之前先进行一些预备处理过程，主要用在神经网络的输出层或隐含层的线性映射之前。数据预处理主要分为归一化处理和标准化处理。

1. 归一化处理

由于多维度数据的每维表达的实际意义不同或单位不同，会造成数据的尺度千差万别，这会造成在训练过程中出现“大数吃小数”的问题，所以需要在训练开始前先统一数据的尺度，归一化处理是指通过线性变换按维度将原始数据映射到[0,1]或[−1,1]上。

以[0,1]归一化为例，设输入数据第 j 维的所有批量数据为 $x_j=\{x_j^1,x_j^2,\cdots,x_j^N\}$，则

$$\bar{x}_j^i=\frac{x_j^i-x_{\min}}{x_{\max}-x_{\min}},\quad i=1,2,\cdots,N\tag{5-47}$$

显然，$\bar{x}_j^i \in [0,1]$，$i=1,2,\cdots,N$。

值得注意的是，使用归一化的数据训练好模型以后再做模型测试时要将测试输入做相同的归一化处理。

2. 标准化处理

标准化处理是将服从正态分布的输入数据按维度平移和伸缩为标准正态分布。设第 j 维的所有输入数据为 $x_j=\{x_j^1,x_j^2,\cdots,x_j^N\}$，则

$$\bar{x}_j^i=\frac{x_j^i-\mu}{\sigma},\quad i=1,2,\cdots,N \tag{5-48}$$

其中

$$\mu=\frac{1}{N}\sum_{i=1}^{N}x_j^i$$

和

$$\sigma=\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_j^i-\mu)^2} \tag{5-49}$$

分别为输入数据的均值和标准差。

5.4　PyTorch 深度学习软件包

103min

PyTorch 起源于 Facebook 公司的深度学习框架——Torch，在底层 Torch 框架的基础上，使用 Python 对其进行了重写，使 PyTorch 在支持 GPU 的基础上实现了与 NumPy 的无缝衔接。另外，PyTorch 还提供了 torchaudio(用于处理声频)、torchtext(用于处理文本)、torchvision(用于处理视频)等专门的库，内置大量已经预训练的深度神经网络和高质量的训练数据集，这些库为直接使用经典神经网络进行项目开发和迁移学习提供了便利。

PyTorch 的内容博大精深，本书不可能覆盖所有方面，本节仅针对在后文中要用到的内容及关键知识点进行简单介绍。本书所使用的 PyTorch 版本是 1.9.0。

5.4.1　数据类型及类型的转换

1. Tensor 数据类型

PyTorch 的基本数据结构是张量(Tensor)，所有的计算都是通过张量进行的。PyTorch 中的张量和 NumPy 中的数组(ndarray)具有极高的相似度，二者可以相互转换。唯一不同的是，Tensor 可以在 GPU 上运行，而 NumPy 中的 ndarray 只能在 CPU 上运行。PyTorch 的这种设计是为了充分利用 NumPy 丰富的数组处理函数，同时又兼顾了 GPU 的高计算性能。

PyTorch 支持的数据类型包括浮点型、复数型、整型和布尔型，它们的相关信息见表 5-3。

表 5-3 PyTorch 数据类型

数据类型	dtype	CPU Tensor	GPU Tensor
32-bit floating point	torch.float32 或 torch.float	torch.FloatTensor	torch.cuda.FloatTensor
64-bit floating point	torch.float64 或 torch.double	torch.DoubleTensor	torch.cuda.DoubleTensor
16-bit floating point	torch.float16 或 torch.half	torch.HalfTensor	torch.cuda.HalfTensor
32-bit complex	torch.complex32		
64-bit complex	torch.complex64		
128-bit complex	torch.complex128 或 torch.cdouble		
8-bit integer (unsigned)	torch.uint8	torch.ByteTensor	torch.cuda.ByteTensor
8-bit integer (signed)	torch.int8	torch.CharTensor	torch.cuda.CharTensor
16-bit integer (signed)	torch.int16 或 torch.short	torch.ShortTensor	torch.cuda.ShortTensor
32-bit integer (signed)	torch.int32 或 torch.int	torch.IntTensor	torch.cuda.IntTensor
64-bit integer (signed)	torch.int64 或 torch.long	torch.LongTensor	torch.cuda.LongTensor
Boolean	torch.bool	torch.BoolTensor	torch.cuda.BoolTensor

PyTorch 在构造张量时，浮点型默认使用 torch.float32，整型默认使用 torch.int64，也可以在定义时指定数据类型，可以通过 dtype 属性来查看张量的数据类型，代码如下：

```
import torch
import numpy as np

#In[Tensor 默认数据类型]
a = torch.rand((3,))
print(a.dtype)

b = torch.randint(1,10,(3,))
print(b.dtype)

#In[Tensor 指定数据类型]
a = torch.rand((3,),dtype = torch.float64)
print(a.dtype)

b = torch.randint(1,10,(3,),dtype = torch.int32)
print(b.dtype)
```

运行结果如下：

```
torch.float32
torch.int64
torch.float64
torch.int32
```

2. Tensor 数据类型转换

可以通过在 Tensor 后面加上数据类型的方式来改变 Tensor 的数据类型，代码如下：

```
#In[Tensor 数据类型转换]
a = torch.rand((3,))
print(a,a.dtype)

b = a.int()
print(b,b.dtype)

c = b.float()
print(c,c.dtype)
```

运行结果如下：

```
tensor([0.9472, 0.5125, 0.2198]) torch.float32
tensor([0, 0, 0], dtype=torch.int32) torch.int32
tensor([0., 0., 0.]) torch.float32
```

值得注意的是，当从 torch.float32 转换成 torch.int32 时，只保留了原来数据的整数部分，所以再转回 torch.float32 后就和原来的数据不相等了，从上面的运行结果也可以看出 $a \neq c$。

3. Tensor 和 ndarray 相互转换

PyTorch 提供了 NumPy 的 ndarray 数据和 Tensor 相互转换的工具，这在实际编程中非常适用和重要，因为大部分原始的和生成的数据最初是 ndarray 格式的，在将它们灌入深度神经网络进行训练之前需要先将其转换成 Tensor；另外，从深度神经网络输出的数据都是 Tensor，要对它们进行一般的计算，又要先将它们转换成 ndarray 格式。

使用 from_numpy 可以将由 NumPy 生成的 ndarray 数据转换成相应的 Tensor，也可以使用 torch.FloatTensor()函数来生成 ndarray 数组对应的 Tensor，代码如下：

```
#In[ndarray 转 Tensor]
a = np.array([1.,2.,3.])
tensor_a1 = torch.from_numpy(a)
tensor_a2 = torch.FloatTensor(a)
```

```
print(a,a.dtype)
print(tensor_a1,tensor_a1.dtype)
print(tensor_a2,tensor_a2.dtype)
```

运行结果如下：

```
[1. 2. 3.] float64
tensor([1., 2., 3.], dtype = torch.float64) torch.float64
tensor([1., 2., 3.]) torch.float32
```

使用 Tensor 的 NumPy 功能函数可以将 Tensor 转换成相应的 ndarray 数据格式，也可以直接使用 numpy.array()函数来生成与 Tensor 相应的数组，代码如下：

```
# In[Tensor 转 ndarray]
t = torch.rand((3,))
array_t1 = t.NumPy()
array_t2 = np.array(t)
print(t,t.dtype)
print(array_t1,array_t1.dtype)
print(array_t2,array_t2.dtype)
```

运行结果如下：

```
tensor([0.2360, 0.0514, 0.6417]) torch.float32
[0.23601848 0.05141479 0.64165056] float32
[0.23601848 0.05141479 0.64165056] float32
```

Tensor 和 ndarray 数据格式的相互转换在深度神经网络编程中经常会用到，但又是非常容易出错的部分，在编程过程中一定要清楚各个数据是 Tensor 还是 ndarray，以及它们的位数。一般只在神经网络内部使用 Tensor，在其他地方均使用 ndarray，浮点数类型一般设置为 float32。

5.4.2 张量的维度和重组操作

1. 维度的定义

在 PyTorch 中，张量的维度是一个重要概念，许多操作都和维度有关。从形式上看，张量和 NumPy 中的多维数组一样，其中 0 维张量表示标量，即一个数；一维张量表示向量，即一维数组；二维张量表示矩阵，即二维数组；多维张量相当于多维数组。张量的维度计数从 0 维开始，自外向里计数，在 Tensor 中的关键字是 axis 或 dim。

这里需要注意的是张量的维度和每维上的分量数是两个概念，但是人们一般将两者都称为维数。为了区别，本书将张量每维上的分量的个数称为分量数。

以一个三维张量为例，代码如下：

```
t = torch.Tensor(np.arange(24)).reshape((2,3,4))
```

运行结果如下：

```
tensor([[[ 0., 1., 2., 3.],
         [ 4., 5., 6., 7.],
         [ 8., 9., 10., 11.]],

         [[12., 13., 14., 15.],
         [16., 17., 18., 19.],
         [20., 21., 22., 23.]]])
```

这里 t 是一个三维张量，第 0 维、第 1 维和第 2 维上的分量数分别为 2、3 和 4。

可以采用剥掉中括号的方法来确认张量各维度的分量具体是什么。将 t 最外层的中括号剥掉，得到两个尺寸为 3×4 的二维张量，即

```
[[ 0., 1., 2., 3.],
 [ 4., 5., 6., 7.],
 [ 8., 9., 10., 11.]],

[[12., 13., 14., 15.],
 [16., 17., 18., 19.],
 [20., 21., 22., 23.]]
```

这就是 t 的第 0 维上的两个分量。若继续将第 1 个二维张量的最外层中括号剥掉，则可得到 3 个分量数均为 4 的一维张量，即

```
[ 0., 1., 2., 3.],
[ 4., 5., 6., 7.],
[ 8., 9., 10., 11.]
```

这就是 t 的第 1 维上的分量。若继续将第 1 个一维张量的最外层中括号剥掉，则可得到 4 个标量，即零维张量，即

```
0., 1., 2., 3.
```

这就是 t 的第 2 维上的分量。

弄清楚张量各维度具体如何得到以后，与张量维度有关的操作就容易理解了。例如对 t 的第 0 维求和，就是将第 0 维上的两个二维张量相加，即

```
print(t.sum(dim = 0))   #print(t.sum(axis = 0))
```

运行结果如下：

```
tensor([[12., 14., 16., 18.],
        [20., 22., 24., 26.],
        [28., 30., 32., 34.]])
```

可见得到的是一个尺寸为 3×4 的二维张量。若对 t 的第 1 维求和，就是将第 2 个二维张量上的一维张量分别相加，即

```
print(t.sum(dim = 1))
```

运行结果如下：

```
tensor([[12., 15., 18., 21.],
        [48., 51., 54., 57.]])
```

可见得到的是一个尺寸为 2×4 的二维张量。若对 t 的第 2 维求和，就是将第 2 维上的标量分别相加，即

```
print(t.sum(dim = 2))
```

运行结果如下：

```
tensor([[ 6., 22., 38.],
        [54., 70., 86.]])
```

可见得到的是一个尺寸为 2×3 的二维张量。

总而言之，对张量某一维的操作要先弄清这一维上的分量是什么才不至于出错。值得注意的是，Tensor 和 ndarray 都没有行向量和列向量的概念，这和 MATLAB 是很不一样的，熟悉 MATLAB 编程的读者要注意区分。

2. 张量的维度重组

张量的维度重组使用 view 或 reshape 函数，这两个函数的功能基本相同，代码如下：

```
#In[张量的维度重组]
t = torch.Tensor(np.arange(24)).reshape((2,3,4))
t1 = t.view(2,2,6)
t2 = t.reshape(8, -1)
print(t)
print(t1)
print(t2)
```

运行结果如下：

```
tensor([[[ 0., 1., 2.,  3.],
         [ 4., 5., 6.,  7.],
         [ 8., 9., 10., 11.]],

        [[12., 13., 14., 15.],
         [16., 17., 18., 19.],
         [20., 21., 22., 23.]]])
tensor([[[ 0., 1., 2., 3., 4.,  5.],
         [ 6., 7., 8., 9., 10., 11.]],

        [[12., 13., 14., 15., 16., 17.],
         [18., 19., 20., 21., 22., 23.]]])
tensor([[ 0., 1., 2.],
       [ 3., 4., 5.],
       [ 6., 7., 8.],
       [ 9., 10., 11.],
       [12., 13., 14.],
       [15., 16., 17.],
       [18., 19., 20.],
       [21., 22., 23.]])
```

可以看出，数据重组的方式是原始数据按照最后一维依次填入重组后的张量。t2 中的−1 会自动计算剩下维度的分量数，即(2×3×4)/8。

3. 张量的维度添加和压缩

在神经网络数据流动过程中经常需要对张量的维度进行对齐，这就需要对张量的维度进行添加或压缩。

维度添加使用 unsequeeze 函数，用于在张量的指定维度上添加一维；维度压缩使用 squeeze 函数，用于将分量数压缩为 1 的维度，代码如下：

```
#In[维的添加和压缩]
t = torch.Tensor(np.arange(6)).reshape((2,3))
t1 = t.unsqueeze(dim = 0)
t2 = t1.unsqueeze(dim = 3)
t3 = t1.squeeze()
t4 = t2.squeeze()
print(t,t.shape)
print(t1,t1.shape)
print(t2,t2.shape)
print(t3,t3.shape)
print(t4,t4.shape)
```

运行结果如下：

```
tensor([[0., 1., 2.],
        [3., 4., 5.]]) torch.Size([2, 3])
tensor([[[0., 1., 2.],
         [3., 4., 5.]]]) torch.Size([1, 2, 3])
tensor([[[[0.],
          [1.],
          [2.]],

         [[3.],
          [4.],
          [5.]]]]) torch.Size([1, 2, 3, 1])
tensor([[0., 1., 2.],
        [3., 4., 5.]]) torch.Size([2, 3])
tensor([[0., 1., 2.],
        [3., 4., 5.]]) torch.Size([2, 3])
```

上例中 t 是一个尺寸为 2×3 的二维张量；t1 在 t 的第 0 维上添加一维，故 t1 是一个尺寸为 1×2×3 的三维张量；t2 再在 t1 的第 3 维上增加一维，将 t1 的每个标量元素变成一维张量，故 t2 是一个尺寸为 1×2×3×1 的四维张量；最后 t3 和 t4 分别将 t1 和 t2 的所有分量数为 1 的维度压缩。使用 squeeze 压缩维度时也可以指定压缩某一维度，代码如下：

```
t5 = t2.squeeze(dim = 3)
print(t5,t5.shape)
```

运行结果如下：

```
tensor([[[0., 1., 2.],
         [3., 4., 5.]]]) torch.Size([1, 2, 3])
```

可以看出，只压缩了 t2 的第 3 维。

4. 张量的转置

用于张量转置的有 3 个函数，即 t、transpose 和 permute。t 只能用于二维张量，和矩阵转置一样，代码如下：

```
t = torch.Tensor(np.arange(6)).reshape((2,3))
t_t = t.t()
print(t_t,t_t.shape)
```

运行结果如下：

```
tensor([[0., 3.],
```

```
        [1., 4.],
        [2., 5.]]) torch.Size([3, 2])
```

transpose 函数用于张量的某两个维度的转置，代码如下：

```
t = torch.Tensor(np.arange(24)).reshape((2,3,4))
t_trans = t.transpose(0,1)      #第零维和一维转置
print(t,t.shape)
print(t_trans,t_trans.shape)
```

运行结果如下：

```
tensor([[[ 0., 1., 2., 3.],
         [ 4., 5., 6., 7.],
         [ 8., 9., 10., 11.]],

        [[12., 13., 14., 15.],
         [16., 17., 18., 19.],
         [20., 21., 22., 23.]]]) torch.Size([2, 3, 4])
tensor([[[ 0., 1., 2., 3.],
         [12., 13., 14., 15.]],

        [[ 4., 5., 6., 7.],
         [16., 17., 18., 19.]],

        [[ 8., 9., 10., 11.],
         [20., 21., 22., 23.]]]) torch.Size([3, 2, 4])
```

permute 给出维度转置的一个排列方式，代码如下：

```
t_perm = t.permute((1,0,2))        #将维度按照 1,0,2 方式转置,即(3,2,4)
print(t_perm,t_perm.shape)
```

运行结果如下：

```
tensor([[[ 0., 1., 2., 3.],
         [12., 13., 14., 15.]],

        [[ 4., 5., 6., 7.],
         [16., 17., 18., 19.]],

        [[ 8., 9., 10., 11.],
         [20., 21., 22., 23.]]]) torch.Size([3, 2, 4])
```

张量的转置比较容易造成数据混乱，在实际应用中应尽量减少使用。

5. 张量的广播

张量的常规加减乘除运算只有在相同尺寸的张量上才能进行，但在实际计算中经常会遇到一个二维张量加一个一维张量的问题，这就需要先对一维张量进行复制，得到一个和二维张量尺寸一样的张量以后，再进行加法运算，这就是张量的广播机制，代码如下：

```
t1 = torch.Tensor(np.arange(6)).reshape((2,3))
t2 = torch.ones((3,))
t3 = t1 + t2
print(t3)
```

运行结果如下：

```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

这里 t1 是一个尺寸为 2×3 的二维张量，但 t2 是一个分量数为 3 的一维张量，所以需要先将 t2 复制两次，成为一个尺寸也为 2×3 的二维张量，代码如下：

```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

这样才能和 t1 相加。值得注意的是，只能对分量数为 1 的维度进行广播，而且被广播的张量维度也要合适才行。例如，若上例中 t2 是一个分量数为 4 的一维张量，则会报错。

5.4.3 组装神经网络的模块

用 PyTorch 构建神经网络就像搭积木一样，只需使用已经封装好的模块，按照约定的结构搭建。这些封装好的模块都放在 torch.nn 模块库中，包括卷积层（Convolution Layer）、池化层（Pooling Layer）、边界填充层（Padding Layer）、激活函数（Activation Function）、线性层（Linear Layer）等。本节选择介绍一些后文中要用到的模块。

torch.nn 中的模块都是以类的形式给出的，在使用它们时要先创建一个类的实例，然后传入参数进行使用。

1. 线性层

线性层是神经网络最基本的映射层，用公式表示是

$$\boldsymbol{y} = \boldsymbol{x}\boldsymbol{A}^{\mathrm{T}} + \boldsymbol{b}$$

线性层使用的模块类是 Linear，调用语法为

```
torch.nn.Linear(in_features, out_features, bias = True, device = None, dtype = None)
```

其中

(1) in_features：输入 $\boldsymbol{x}$ 的维度。

(2) out_features：输出 $\boldsymbol{y}$ 的维度。

(3) bias：是否添加偏置，默认添加。

(4) device：计算设备为 CPU 还是 GPU，默认为 CPU。

(5) dtype：数据类型，默认为 torch.float32。

线性层支持批量数据运算，可以一次性输入一个批量的数据，代码如下：

```
#In[线性层]
B = 10                                  #batch-size
linear_layer = nn.Linear(20,30)         #创建线性层实例
x = torch.randn(B,20)                   #输入批量数据,单个输入维度为 20
y = linear_layer(x)                     #线性层映射,输出 y
print(y.size(),y.dtype)                 #查看输出的尺寸
```

运行结果如下：

```
torch.Size([10, 30]) torch.float32
```

线性层默认支持的数据类型是 torch.float32。可以在 dtype 关键字中修改数据类型，例如修改为 dtype=torch.float64，这时 $\boldsymbol{x}$ 在创建时也要使用相同的数据类型，否则会报错。使用默认数据类型对于实际问题来讲精度已经足够了。

上例中创建的 linear_layer 是 torch.nn.Linear 类的一个实体，有自己的属性和功能函数，可以通过 dir 函数查询这些属性和功能函数名，一般用得较多的是查询其权重和偏置，代码如下：

```
dir(linear_layer)                       #查询 linear_layer 的所有属性和功能函数名
print(linear_layer.weight)              #查询权重
print(linear_layer.bias)                #查询偏置
```

2. 激活函数

激活函数是神经网络非线性的唯一来源，不可或缺。关于常见激活函数的具体表达式和导数已经在 5.3.1 节中详细介绍过，不再赘述。此处以最常用的激活函数 ReLU 来介绍激活函数的使用方法。

ReLU 函数的调用语法为

```
torch.nn.ReLU(inplace=False)
```

其中，inplace 表示输出数据是否直接使用输入数据的内存，默认值为 False，即使用新的内存输出数据。inplace=True 可以节省内存空间，省去了反复申请和释放内存的时间，但会覆盖输入数据。

ReLU 函数是单变量函数，它会分别作用于输入张量的每个标量数据，所以输出数据和输入数据具有相同的尺寸，代码如下：

```
#In[ReLU 激活函数]
B = 10                                          #batch-size
relu = nn.ReLU()                                #创建 ReLU 函数实体
x = torch.randn(B,20)                           #输入批量数据,单个输入维度为 20
y = relu(x)                                     #ReLU 函数映射
print(x.shape,y.shape)                          #输入输出尺寸一样
```

运行结果如下：

```
torch.Size([10, 20]) torch.Size([10, 20])
```

可见输入数据和输出数据的尺寸一样。

3. 损失函数

损失函数是在训练神经网络时必需的模块，也放在 torch.nn 模块库中。常见的损失函数的具体原理和表达式已经在 5.3.2 节详细介绍过，不再赘述。此处以常用的均方误差损失函数为例介绍损失函数的使用方法。

均方误差损失函数的调用语法为

```
torch.nn.MSELoss(reduction = 'mean')
```

其中 reduction 取'none''mean'或'sum'，若 reduction＝'mean'，则输出批量运算结果之和按输入尺度平均后的标量；若 reduction＝'sum'，则输出批量运算结果之和按输入尺度相加后的标量和；若 reduction＝'none'，则只输出批量和，在输入尺度上不做处理，输入数据和输出数据有相同的尺度。默认为 reduction＝'mean'。

均方误差损失函数也支持批量运算，可以输入一个批量的数据，代码如下：

```
#In[MSELoss 函数]
B = 10                                          #batch-size
loss = nn.MSELoss()                             #创建 MSELoss 函数,reduction = 'mean'
loss_sum = nn.MSELoss(reduction = 'sum')        #创建 MSELoss 函数,reduction = 'sum'
loss_none = nn.MSELoss(reduction = 'none')      #创建 MSELoss 函数,reduction = 'none'
y_hat = torch.randn((3,5))                      #预测输出
y_tar = torch.randn((3,5))                      #目标输出
out = loss(y_hat,y_tar)                         #损失函数值
out_sum = loss_sum(y_hat,y_tar)
out_none = loss_none(y_hat,y_tar)
print(out,out.shape)
print(out_sum,out_sum.shape)
print(out_none,out_none.shape)
```

运行结果如下：

```
tensor(1.4108) torch.Size([])
tensor(21.1614) torch.Size([])
tensor([[1.1905e-02,  4.8586e-04,  1.0308e+00,  1.2013e+00,  9.7473e-02],
        [1.5531e+00,  6.4626e-01,  8.4639e-01,  3.3398e+00,  7.0021e+00],
        [1.6918e+00,  2.0903e+00,  2.6002e-02,  1.3863e+00,  2.3753e-01]])
torch.Size([3, 5])
```

上例中输入尺度是3×5，所以out_sum/15=out，而out_none的所有元素之和等于out_sum。

5.4.4　自动梯度计算

在神经网络的误差反向传播过程中，梯度计算是一个必不可少的步骤，PyTorch针对这一步骤专门开发了一个自动梯度计算引擎torch.autograd，它支持任何计算图的自动梯度计算。本节以一个简单的函数为例来介绍相关概念及操作。

1. 计算图

考虑二次函数

$$y = \boldsymbol{x}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{b}^{\mathrm{T}}\boldsymbol{b} \tag{5-50}$$

式(5-50)的计算过程可以用如图5-15所示的一个计算流程图来表示。

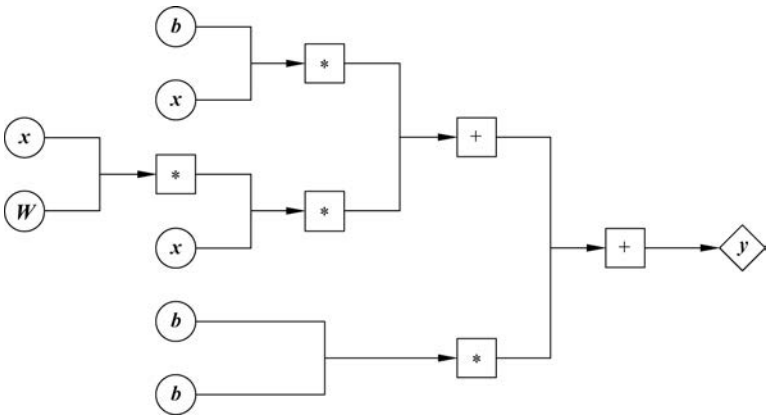

图5-15　计算图

在图5-15中，圆圈内的元素表示输入流程图的计算单元，称为叶节点(Leaf Node)；方框内的符号表示各计算单元的运算方式，称为计算节点(Computation Node)；菱形内的元素是计算流程的最后结果，称为根节点(Root Node)。

torch.autograd引擎计算梯度的过程是这样的：首先在前向传播过程中建立计算图，并保留一些计算梯度需要的中间结果，然后根据计算图自动计算用链式法则计算变量梯度所需要的中间函数，最后利用这些中间函数和前向传播中保留的中间结果根据链式法则计算各变量的梯度。整个过程的代码如下：

```
import torch

#In[搭建计算图]
x = torch.ones((5,),requires_grad = True)          #变量,需要计算梯度
W = torch.ones((5,5),requires_grad = False)        #参数,不需要计算梯度
b = torch.ones((5,))                               #参数,默认不需要计算梯度
Q = torch.matmul(torch.matmul(x,W),x)              #二次项,中间结果
L = torch.matmul(b,x)                              #一次项,中间结果
C = torch.matmul(b,b)                              #常数项
y = Q + L + C                                      #前向传播,建立计算图
                                                   #查看需要求梯度的量
print(x.requires_grad,Q.requires_grad,C.requires_grad,y.requires_grad)
print(y.grad_fn)                                   #对最终结果 y 的梯度函数
print(Q.grad_fn)                                   #对中间结果 Q 的梯度函数
print(C.grad_fn)                                   #对中间结果 C 的梯度函数
print(x.grad_fn)                                   #对叶节点 x 的梯度函数
```

运行结果如下：

```
True True False True
<AddBackward0 object at 0x0000023A5DCA4208>
<DotBackward object at 0x0000023A5DCA4208>
None
None
```

每个张量都有一个 requires_grad 属性，为 True 时表示该张量需要计算梯度，为 False 时表示该张量不需要计算梯度。输入张量 $\boldsymbol{x}$、$\boldsymbol{W}$、$\boldsymbol{b}$ 的 requires_grad 属性是在输入时给定的，默认值为 False；中间结果和输出结果是否需要求梯度要根据链式法则来定，例如在上例中 Q 和 y 需要求梯度，但作为常数项的 C 就不需要求梯度。

同样，根据求梯度的链式法则，要对 $\boldsymbol{x}$ 求梯度，首先要求输出结果 y 的梯度函数，然后要求中间结果 Q 和 L 的梯度函数，但 C 不需要求梯度函数，因为 C 是常数项，不参与梯度计算，$\boldsymbol{x}$ 也不需要求梯度函数，因为 $\boldsymbol{x}$ 已经是叶节点了。从上例中后 4 行打印出的结果也能清楚地看到这一点。

2. 自动梯度计算

计算图搭建好以后，就可以用 backward 函数进行梯度计算了，用 grad 属性可以查看计算好的梯度，代码如下：

```
#In[梯度计算和查看]
y.backward()                    #自动梯度计算
print(x.grad)                   #查看 y 对于 x 的梯度
print(b.grad)                   #查看 y 对于 b 的梯度
print(Q.grad)                   #查看 y 对于 Q 的梯度
```

运行结果如下：

```
tensor([11., 11., 11., 11., 11.])
None
None
__main__:4: UserWarning: The .grad attribute of a Tensor that is not a leaf Tensor is being accessed. Its .grad attribute won't be populated during autograd.backward(). If you indeed want the gradient for a non-leaf Tensor, use .retain_grad() on the non-leaf Tensor. If you access the non-leaf Tensor by mistake, make sure you access the leaf Tensor instead. See github.com/PyTorch/PyTorch/pull/30531 for more information.
```

可以看出，只有 y 关于 $\boldsymbol{x}$ 的梯度输出了有效值，y 关于 $\boldsymbol{b}$ 的梯度未输出是因为 $\boldsymbol{b}$ 的 requires_grad 属性为 False，不需要计算梯度，而 y 关于 Q 的梯度未输出是因为 Q 是中间变量，规定中间变量的梯度不能获取。

值得注意的是，PyTorch 中的计算图是动态图，在前向传播时构建，梯度计算完毕后释放，因此，若在以上代码中再次执行 y.backward()命令，就会出现以下错误提醒：

```
RuntimeError: Trying to backward through the graph a second time (or directly access saved variables after they have already been freed). Saved intermediate values of the graph are freed when you call .backward() or autograd.grad(). Specify retain_graph=True if you need to backward through the graph a second time or if you need to access saved variables after calling backward.
```

也就是说，在第 1 次执行 y.backward()命令后，计算图就被释放了，如果再次执行该命令，因为已经不存在计算图，所以就不能顺利计算梯度了。保存计算图的方法是在 backward 函数中传入 retain_graph=True 参数，即 y.backward(retain_graph=True)。也就是在执行 y.backward()命令后，计算图会被保留下来。

PyTorch 这种在前向传播时构建计算图，梯度计算完成后释放计算图的范式叫作动态计算图，相较于 TensorFlow 的静态计算图而言。动态计算图的优点是灵活，可以随时改变计算图的结构，这在训练一些动态神经网络或有分叉的神经网络中很有用处，缺点是构建和释放计算图需要一些时间和计算资源。

另外，y.backward()函数求出的梯度是在原来的梯度上的累加值，代码如下：

```
y.backward(retain_graph=True)
print(x.grad)
```

运行结果如下：

```
tensor([22., 22., 22., 22., 22.])
```

可见 y 关于 $\boldsymbol{x}$ 的梯度变成了原来的 2 倍，这是因为之前已经求过一次梯度，第 2 次求出

的梯度值要累加第 1 次求出的结果。为避免这种情况的出现，在每次求梯度之前需要先使用 zero_函数将梯度归零，代码如下：

```
x.grad.zero_()
print(x.grad)
y.backward(retain_graph = True)
print(x.grad)
```

运行结果如下：

```
tensor([0., 0., 0., 0.])
tensor([11., 11., 11., 11.])
```

可见在使用了 zero_函数以后，y 关于 $\boldsymbol{x}$ 的梯度就归零了，再次计算梯度后输出值恢复正常。

3. 关闭自动梯度计算

自动梯度计算通常用在模型训练中，但在前向传播过程中并不需要计算梯度，这时关闭自动梯度计算会让计算效率更高。有两种方式可以局部关闭自动梯度计算，一种是用 with torch.no_grad 块将本来需要计算梯度的代码包起来，代码如下：

```
y = torch.matmul(b,x)
print(y.requires_grad)

with torch.no_grad():
    y1 = torch.matmul(b,x)
print(y1.requires_grad)
```

运行结果如下：

```
True
False
```

可以看出在 with torch.no_grad 块中构建的计算图是不能计算梯度的。

另一种方法是用张量的 detach 函数，代码如下：

```
y2 = y.detach()
print(y2.requires_grad)
```

运行结果如下：

```
False
```

5.4.5　训练数据自由读取

为了更方便地管理和读取训练数据，PyTorch 提供了 torch. utils. data. Datasets 和 torch. utils. data. DataLoader 两个类。torch. utils. data. Datasets 是 torchvision 中所有预存训练数据的基类，保存着训练数据集的基本信息，例如数据和标签等。用户也可以用 Dataset 基类自定义训练数据。DataLoader 是一个用于加载训练数据的类，它可以对原始训练数据进行打乱顺序、划分小批量和循环加载等操作。以下以著名的图像处理数据集 FashionMNIST 为例介绍数据读取和加载。

1. 数据下载

首先从云端下载 FashionMNIST 数据集，代码如下：

```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt

#In[下载数据集]
training_data = datasets.FashionMNIST(
        root = 'data',
        train = True,
        download = True,
        transform = ToTensor()
        )

test_data = datasets.FashionMNIST(
        root = 'data',
        train = False,
        download = True,
        transform = ToTensor()
        )
```

关键字中 root 表示数据集存储的地址，程序运行后会在当前目标下生成一个名为 data 的文件，里面就是下载的 FashionMNIST 数据；train＝True 表示下载训练集，train＝False 表示下载测试集；download＝True 表示如果本地没有该数据集则从网上下载；transform 接收用于将原始数据转化成训练所需要的数据类型的函数，如 ToTensor 函数将 ndarray 数据转换成浮点型张量，并标准化。

2. 查看数据

可以用 dir 函数查看 training_data 和 test_data 的相关属性和功能函数，常用的是 data 和 targets 属性，分别代表数据本身和对应的标签，所有的标签名可以用 classes 属性读取，

代码如下：

```
#In[查看数据结构]
print(training_data.data.shape,training_data.targets.shape)
print(test_data.data.shape,test_data.data.shape)
print(training_data.data.dtype,training_data.targets.dtype)
print(training_data.classes)
```

运行结果如下：

```
torch.Size([60000, 28, 28]) torch.Size([60000])
torch.Size([10000, 28, 28]) torch.Size([10000, 28, 28])
torch.uint8 torch.int64
['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle
boot']
```

可见 FashionMNIST 数据集一共包括 60 000 条训练数据和 10 000 条测试数据，每条训练数据是一个 dtype 为 torch.unit8 的 28×28 维的矩阵，数据标签一共有 10 类，用 torch.int64 数据表示。

3. 训练数据加载

用 torch.utils.data.DataLoader 加载训练数据的语法如下：

```
dataloader = torch.utils.data.DataLoader(dataset, batch_size=1,shuffle=False)
```

其中，

(1) dataset：表示要加载的 torch.utils.data.Dataset 类的训练数据，例如上例中下载的 FashionMNIST 数据集。

(2) batch_size：小批量数据的批量尺度。

(3) shuffle：将数据划分成批量数据时，是否要先打乱数据顺序，默认为不打乱。

将上例中已经下载好的 FashionMNIST 训练数据用 torch.util.data.DataLoader 加载的代码如下：

```
#In[训练数据加载]
batch_size = 32
training_data_loader = DataLoader(training_data,batch_size,shuffle=True)
test_data_loader = DataLoader(test_data,batch_size,shuffle=True)

for X,y in training_data_loader:
    print('Shape of X is ',X.shape)
    print('Shape of y is ',y.shape)
    break
```

运行结果如下：

```
Shape of X is torch.Size([32, 1, 28, 28])
Shape of y is torch.Size([32])
```

可见所有训练数据已经被分成了 32 条一份的小批量数据集，数据标签也做了相应的划分。

5.4.6　模型的搭建、训练和测试

本节用一个完整的案例来介绍深度神经网络模型的搭建、训练和测试。以 torchvision 中已经预存的 FashionMNIST 训练数据为例。

1. 数据准备

数据准备包括下载和加载数据，代码如下：

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda, Compose
import matplotlib.pyplot as plt

#In[数据准备]
training_data = datasets.FashionMNIST(
        root = 'data',
        train = True,
        download = True,
        transform = ToTensor()
        )
test_data = datasets.FashionMNIST(
        root = 'data',
        train = False,
        download = True,
        transform = ToTensor()
        )
batch_size = 64
train_dataloader = DataLoader(training_data,batch_size = batch_size)
test_dataloader = DataLoader(test_data,batch_size = batch_size)
```

2. 构建模型

一个神经网络模型包括两个基本部分：拓扑结构和 forward 函数。拓扑结构是由线性层、激活函数、卷积层等基本模块搭建而成，封装在 nn.Sequential 类中，forward 函数用于前向传播计算过程。用户自定义神经网络模型一般继承 nn.Module 类，代码如下：

```
#In[构建模型]
device = 'CUDA' if torch.cuda.is_available() else 'cpu'

class NeuNet(nn.Module):
    def __init__(self):
        nn.Module.__init__(self)
        self.flatten = nn.Flatten()                    #将多维张量拉直成一维张量
                                                       #定义网络拓扑
        self.linear_ReLU_stack = nn.Sequential(
                nn.Linear(28*28,512),
                nn.relu(),
                nn.Linear(512,512),
                nn.relu(),
                nn.Linear(512,10),
                )
    def forward(self,x):                               #前向传播函数
        x=self.flatten(x)
        logits = self.linear_ReLU_stack(x)
        return logits

model =NeuNet().to(device)                             #创建一个神经网络实体
print(model)
```

运行结果如下：

```
NeuNet(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_ReLU_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): relu()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): relu()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

从打印的结果可以清楚地看到该神经网络的拓扑结构。

3. 损失函数和优化器

因为 FashionMNIST 数据集可在处理多分类问题时使用，所以使用交叉熵损失函数和 SGD 优化器，代码如下：

```
#In[损失函数和优化器]
loss = nn.CrossEntropyLoss(reduction='mean')
opt = torch.optim.SGD(model.parameters(),lr=1e-3)
```

优化器定义时的两个必需参数分别表示需要优化的模型参数和学习率。

4. 训练函数

训练函数的主要任务是误差反向传播和调整参数，代码如下：

```
#In[训练函数]
def train(dataloader,model,loss_fn,optimizer):
    size = len(dataloader.dataset)
    model.train()                                    #声明以下是训练环境
    for batch, (X,y) in enumerate(dataloader):
        X,y = X.to(device),y.to(device)
        pred = model(X)                              #计算预测值
        loss = loss_fn(pred,y)                       #计算损失函数
        opt.zero_grad()                              #梯度归零
        loss.backward()                              #误差反向传播
        opt.step()                                   #调整参数

        if batch%100 == 0:                           #每隔100批次打印训练进度
            loss,current = loss.item(),batch*len(X)
            print(f"loss: {loss:>7f} [{current:>5d}/{size:>5d}]")
```

代码中 model.train()用于声明下面的代码是训练环境，与此对应，测试环境的声明方式是 model.eval()。这个声明在本例中是没有任何意义的，可加也可不加，但若训练过程中使用了 DropOut 层就必须加了，因为在训练过程中 DropOut 层是要起作用的，但测试过程中却不能启动 DropOut 层。

5. 测试函数

测试函数的主要任务是用测试数据测试训练出的模型的泛化性能，可以根据测试结果来修改模型或者训练过程。测试函数的代码如下：

```
#In[测试函数]
def test(dataloader,model,loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()                                     #声明模型评估状态
    test_loss,correct = 0,0
    with torch.no_grad():
        for X,y in dataloader:
            X,y = X.to(device),y.to(device)
            pred= model(X)                           #计算预测值
            test_loss += loss_fn(pred,y).item()      #计算误差
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    correct /= size                                  #预测分类正确率
    test_loss /= num_batches                         #测试数据凭据误差
    print('Accuracy is {}, Average loss is {}'.format(correct,test_loss))
```

6. 模型训练和测试

训练和测试代码如下：

```
#In[训练和测试]
epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, opt)
    test(test_dataloader, model, loss_fn)
print("Done!")
```

超参数 epochs 是指训练和测试的回合数，每回合训练后都会进行测试。运行结果如下（为节约空间，仅输出第一回合训练过程结果和每回合测试结果）：

```
Epoch 1
-------------------------------
loss: 2.163912 [ 0/60000]
loss: 2.159833 [ 6400/60000]
loss: 2.095972 [12800/60000]
loss: 2.120173 [19200/60000]
loss: 2.076414 [25600/60000]
loss: 2.007832 [32000/60000]
loss: 2.031443 [38400/60000]
loss: 1.956872 [44800/60000]
loss: 1.955904 [51200/60000]
loss: 1.886647 [57600/60000]
Accuracy is 0.6081, Average loss is 1.891194589578422
Epoch 2
-------------------------------
Accuracy is 0.6186, Average loss is 1.5148332749202753
Epoch 3
-------------------------------
Accuracy is 0.6345, Average loss is 1.2477369035125538
Epoch 4
-------------------------------
Accuracy is 0.6474, Average loss is 1.0845678323393415
Epoch 5
-------------------------------
Accuracy is 0.6607, Average loss is 0.9791825618713524
Done!
```

可以看出正确率在增加，而平均误差在减小。

5.4.7　模型的保存和重载

模型的保存和重载是模型应用的重要工具。一个训练好的模型需要保存起来才能继续使用；如果想获得模型训练的一些中间结果，则需要进行模型保存；另外迁移学习中要继续训练已经有过预训练的模型，也需要保存和重载原有模型。

1. 保存模型或模型参数

接着 5.4.6 节中的代码，使用 torch.save 函数将训练好的模型和模型参数保存成.pth 文件，代码如下：

```
#In[保存模型或模型参数]
torch.save(model,'model')                                    #保存整个模型
torch.save(model.state_dict(),'model_parameter.pth')         #保存模型参数
```

这里 model.state_dict()是一个字典，保存着 model 模型的所有参数信息，可以通过键值访问神经网络每层参数的具体值，代码如下：

```
model.state_dict()['linear_ReLU_stack.4.bias']
```

这样就可以访问第 4 层的偏置参数值了，运行结果如下：

```
tensor([-0.0567,  0.0019,  -0.0235,  0.0068,  -0.0487,  0.1576,  -0.0035,
0.0590,  -0.0523,  -0.0582])
```

2. 重载模型或模型参数

重载模型或模型参数可用 torch.load 函数，代码如下：

```
#In[重载模型或模型参数]
model1 = torch.load('model')          #直接重载整个模型，包括网络拓扑和参数
model2 = NeuNet()                     #只重载参数，需要先创建一个相同网络拓扑的初始模型
                                      #重载模型参数
model2.load_state_dict(torch.load('model_parameter.pth'))

with torch.no_grad():
    for X,y in train_dataloader:
        print(model(X)[0])
        print(model1(X)[0])
        print(model2(X)[0])
        break
```

若重载整个模型，则包括模型的网络拓扑重载和参数重载，不需要另外创建模型；若只重载模型参数，则需要预先创建一个具有相同网络拓扑的模型作为模型参数的载体。程序

运行的结果如下：

```
tensor([ - 2.4643, - 4.8268, - 0.6808, - 3.1310, - 1.1310, 2.9940, - 1.2384, 2.7196,
3.1995, 4.8576])
tensor([ - 2.4643, - 4.8268, - 0.6808, - 3.1310, - 1.1310, 2.9940, - 1.2384, 2.7196,
3.1995, 4.8576])
tensor([ - 2.4643, - 4.8268, - 0.6808, - 3.1310, - 1.1310, 2.9940, - 1.2384, 2.7196,
3.1995, 4.8576])
```

可见，3 个模型是完全一样的。

3. torchvision 库中的模型

torchvision 库中已经预存了许多已经训练好的经典神经网络模型，例如 AlexNet、DenseNet、ResNet 等，用户可以在项目中直接使用这些模型，也可以基于这些模型进行迁移学习，代码如下：

```
#In[torchvision 预训练模型加载]
import torchvision.models as models
model_VGG - 16 = models.VGG - 16(pretrained = True)      #创建一个已经训练好的 VGG - 16 网络
                                                         #保存模型参数
torch.save(model_VGG - 16.state_dict(),'model_VGG - 16_parameter')
model_VGG - 16_1 = models.VGG - 16()                     #创建一个未训练的 VGG - 16 网络
                                                         #加载模型参数
model_VGG - 16_1.load_state_dict(torch.load('model_VGG - 16_parameter'))
model_VGG - 16.eval()                                    #评估模型
model_VGG - 16_1.eval()                                  #评估模型
```

5.5 深度学习案例

一个完整的深度学习过程主要包括构建网络、定义训练函数、定义测试函数、训练和测试、结果展示等几个过程。本节给出两个基于 PyTorch 的深度学习案例，以此来展示基于 PyTorch 的深度学习全过程。

5.5.1 函数近似

本例用一个深度神经网络来近似一元二次函数，待近似的一元二次函数为

$$y = x^2 + 3x + 4 \tag{5-51}$$

使用全连接前馈式深度神经网络，各层节点数为输入层：1；第 1 隐含层：20；第 2 隐含层：40；第 3 隐含层：20；输出层：1；损失函数使用均方误差损失(MSE)；优化器使用随机梯度下降(SGD)，全部代码如下：

```
#【代码 5-1】深度神经网络逼近一元二次函数代码
#In[导入包]
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt

#In[超参数]
LR = 1e-3
BATCH_SIZE = 32
EPOCHS = 40

#In[原函数]
def fun(x):
    return x*x+3*x+4

x = np.linspace(-np.pi,np.pi,100)
y = fun(x)

#In[创建神经网络]
class NeuNet(nn.Module):
    def __init__(self,in_size,out_size):
        nn.Module.__init__(self)
        self.flatten = nn.Flatten()
        self.layers = nn.Sequential(
                      nn.Linear(in_size,20),
                      nn.relu(),
                      nn.Linear(20,40),
                      nn.relu(),
                      nn.Linear(40,20),
                      nn.relu(),
                      nn.Linear(20,out_size),
                      )
    def forward(self,x):
        self.flatten(x)
        return self.layers(x)

model = NeuNet(1,1)

#In[损失函数和优化器]
loss = torch.nn.MSELoss()
opt = torch.optim.SGD(model.parameters(),lr=LR)

#In[训练函数]
def train(model,loss,opt):
    x_batch = -np.pi+2*np.pi*np.random.rand(BATCH_SIZE,1)      #训练输入
```

```
        y_tar_batch = fun(x_batch)                                    #目标输出

        x_batch = torch.from_numpy(x_batch).float()                   #数据格式转换
        y_tar_batch = torch.from_numpy(y_tar_batch).float()           #数据格式转换
        y_pre_batch = model(x_batch).float()                          #预测输入

        loss_fn = loss(y_tar_batch,y_pre_batch)                       #损失函数

        model.train()                                                 #声明训练
        opt.zero_grad()                                               #梯度归零
        loss_fn.backward()                                            #误差反向传播
        opt.step()                                                    #参数调整

#In[测试函数]
def test(model):
    model.eval()
    with torch.no_grad():
        y_pre_test = model(torch.from_numpy(x).float().unsqueeze(dim=1))
        loss_value = loss(torch.from_numpy(y).float(),y_pre_test.float())
    print('loss_fn = ',loss_value)

    return loss_value

#In[训练和测试]
Loss = []
for i in range(EPOCHS):
    print('EPOCH {} ---------- '.format(i))
    train(model,loss,opt)
    loss_value = test(model)
    Loss.append(loss_value)
print('DONE')

#In[作图比较]
with torch.no_grad():
    y_test = model(torch.from_numpy(x).float().unsqueeze(dim=1))
    y_test = y_test.squeeze().NumPy()

plt.figure(1)
plt.plot(Loss)
plt.xlabel('EPOCHS')
plt.ylabel('Loss')
plt.title('Loss via EPOCHS')
plt.savefig('loss.jpg')

plt.figure(2)
plt.plot(x,y,label='real')
```

```
plt.plot(x,y_test,label = 'approximated')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Real vs approximated graph')
plt.legend()
plt.savefig('graph.jpg')
plt.show()
```

使用 print(model)命令可以查看构建的神经网络结构，代码如下：

```
NeuNet(
  (flatten): Flatten(start_dim = 1, end_dim = -1)
  (layers): Sequential(
    (0): Linear(in_features = 1, out_features = 20, bias = True)
    (1): relu()
    (2): Linear(in_features = 20, out_features = 40, bias = True)
    (3): relu()
    (4): Linear(in_features = 40, out_features = 20, bias = True)
    (5): relu()
    (6): Linear(in_features = 20, out_features = 1, bias = True)
  )
)
```

程序运行的结果如图 5-16 所示。

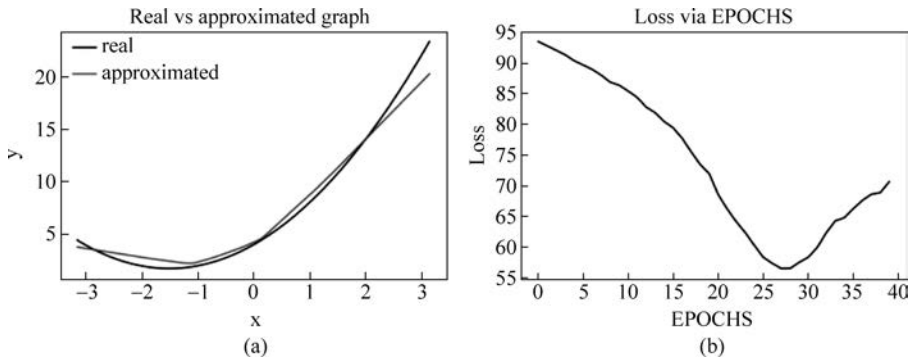

图 5-16　函数逼近运行结果

5.5.2　数字图片识别

本节给出一个用深度神经网络识别 FashionMNIST 图片库的案例。深度神经网络共有 4 层，节点数分别为 28×28(这也是 FashionMNIST 图片的像素尺寸)、512、512 和 10。因为是分类问题，损失函数使用交叉熵损失函数(CrossEntropyLoss)，优化器使用随机梯度

下降(SGD),全部代码如下:

```
#【代码 5-2】深度神经网络数字图片识别代码
#In[导入包]
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt

#In[超参数]
BATCH_SIZE = 64
LR = 1e-3
EPOCHS = 5

#In[数据下载]
training_data = datasets.FashionMNIST(
        root="data",
        train=True,
        download=True,
        transform=ToTensor(),
        )

test_data = datasets.FashionMNIST(
        root="data",
        train=False,
        download=True,
        transform=ToTensor(),
        )

#In[数据加载]
train_dataloader = DataLoader(training_data,batch_size=BATCH_SIZE)
test_dataloader = DataLoader(test_data,batch_size=BATCH_SIZE)

#In[创建网络]
device = 'CUDA' if torch.cuda.is_available() else 'cpu'

class NeuralNetwork(nn.Module):
    def __init__(self):
        nn.Module.__init__(self)
        self.flatten = nn.Flatten()
        self.linear_ReLU_stack=nn.Sequential(
                nn.Linear(28*28,512),
                nn.relu(),
                nn.Linear(512,512),
```

```
                nn.relu(),
                nn.Linear(512,10)
                )

    def forward(self,x):
        x = self.flatten(x)
        logits = self.linear_ReLU_stack(x)
        return logits

model = NeuralNetwork().to(device)

#In[损失函数和优化器]
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),lr=LR)

#In[训练函数]
def train(dataloader,model,loss_fn,optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X,y) in enumerate(dataloader):
        X,y = X.to(device),y.to(device)

        pred = model(X)
        loss = loss_fn(pred,y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch%100 == 0:
            loss, current = loss.item(),batch*len(X)
            print(f"loss: {loss:>7f} [{current:>5d}/{size:>5d}]")

#In[测试函数]
def test(dataloader,model,loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss,correct = 0,0
    with torch.no_grad():
        for X,y in dataloader:
            X,y = X.to(device),y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred,y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
```

```
    correct /= size
    print(f"Accuracy: {(100 * correct):> 0.1f} %, Avg loss: {test_loss:> 8f}")

#In[模型训练和测试]
for t in range(EPOCHS):
    print(f"Epoch {t + 1}\n-------------------- ")
    train(train_dataloader,model,loss_fn,optimizer)
    test(test_dataloader,model,loss_fn)
print("Done!")

#In[训练结果展示]
classes = training_data.classes
model.eval()
x,y = test_data[0][0],test_data[0][1]
with torch.no_grad():
    pred = model(x)
    predicted,actual = classes[pred[0].argmax(0)],classes[y]
print(f'Predicted: "{predicted}", Actual: "{actual}"')
```

使用 print(model)函数可以得到神经网络模型的拓扑结构,代码如下:

```
NeuralNetwork(
  (flatten): Flatten(start_dim = 1, end_dim = -1)
  (linear_ReLU_stack): Sequential(
    (0): Linear(in_features = 784, out_features = 512, bias = True)
    (1): relu()
    (2): Linear(in_features = 512, out_features = 512, bias = True)
    (3): relu()
    (4): Linear(in_features = 512, out_features = 10, bias = True)
  )
)
```

程序运行的结果如下:

```
------ Epoch 1 ------
loss: 2.308854 [ 0/60000]
loss: 2.285936 [ 6400/60000]
loss: 2.274783 [12800/60000]
loss: 2.276982 [19200/60000]
loss: 2.243695 [25600/60000]
loss: 2.230343 [32000/60000]
loss: 2.230508 [38400/60000]
loss: 2.203054 [44800/60000]
loss: 2.198021 [51200/60000]
loss: 2.176432 [57600/60000]
```

```
Accuracy: 46.0 %, Avg loss: 2.161742
------ Epoch 2 ------
Accuracy: 58.9 %, Avg loss: 1.907152
------ Epoch 3 ------
Accuracy: 61.6 %, Avg loss: 1.540300
------ Epoch 4 ------
Accuracy: 63.3 %, Avg loss: 1.265945
------ Epoch 5 ------
Accuracy: 64.5 %, Avg loss: 1.097099
Done!
Predicted: "Ankle boot", Actual: "Ankle boot"
```