第 3 章

深度神经网络

本章介绍了深度神经网络的几个相关概念，并给出了用 MindSpore 实现简单神经网络的样例。

3.1 前向网络

深度学习(Deep Learning)与传统机器学习最大的不同在于其利用神经网络对数据进行高级抽象。而最基础的神经网络结构为前向神经网络(Feed forward Neural Network，FNN)，又称多层感知机(Multi-Layer Perceptron，MLP)。

在介绍多层感知机之前，先来认识一下神经网络的基础单元——感知机。如图 3.1 所示，x_1，x_2，…，x_n 为输入，w_1，w_2，…，w_n 为与之对应的权重，w_0 为偏置。感知机对这些输入进行加权求和，再加上偏置值 w_0，最后通过激活函数 $f(\cdot)$得到神经元的输出。在分类问题中提到的逻辑函数 $\text{sigmoid}(x)=\frac{1}{1+e^{-x}}$为一种常用的激活函数(Activation Function)，目的是将一个在较大范围变化的值挤压到(0，1)的输出值范围

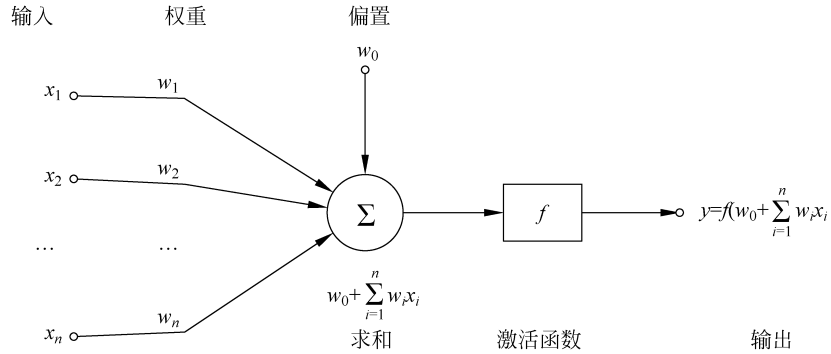

图 3.1 感知机

内，或者输出 0/1 对应的概率值。此外，双余弦函数 $\tanh(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ 以及修正线性单元(Rectified Linear Unit，ReLU)函数 $\mathrm{ReLU}(x)=\max(x,0)$ 也经常作为神经元的激活函数。这些激活函数的目的都是为神经元带来非线性运算。相比线性函数而言，非线性函数的表达能力更强，图 3.2 展示了这三种激活函数的形状。

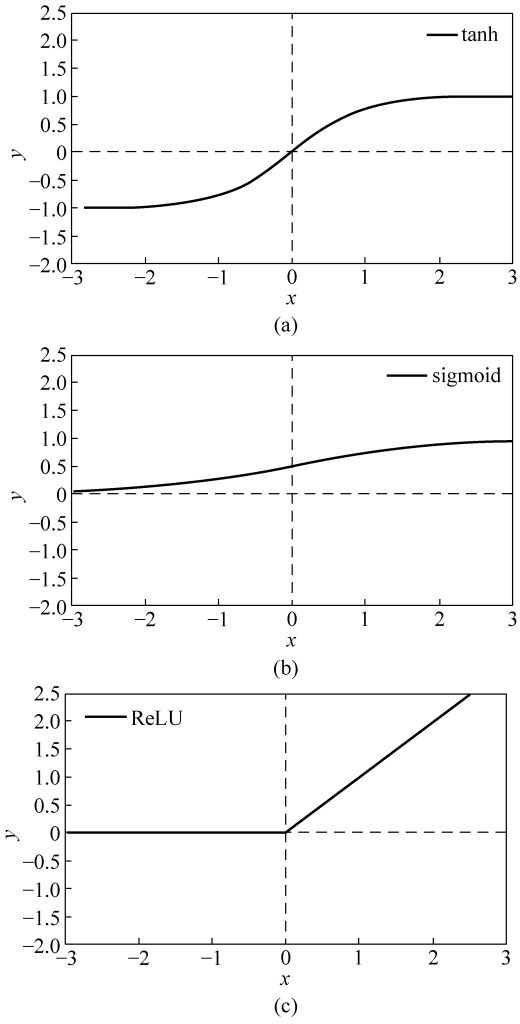

图 3.2　常见的三种激活函数的形状

尽管非线性激活函数的单个神经元带来了非线性特征，但它只拥有一层神经元，学习能力非常有限，仍然只能处理线性可分的问题。为了解决更复杂的非线性可分问题，多层感知机(MLP)被提出。

图 3.3 为一个简单的三层前向神经网络模型，包括输入层、隐藏层和输出层。数据 x 作为输入提供给输入层，经过线性映射和非线性激活函数，得到隐藏层。隐藏层再经过一层运算得到输出层。其中输入层的节点数由数据本身的属性数量决定，输出层的节点数可以是类别个数、抽象特征个数等。隐藏层的层数人为指定，并且层数可以是一层或多层，每个隐藏层上都可设置一类非线性激活函数。经过线性组合与非线性变换，这个由多层神经元组成的函数模型，具有更强大的学习能力。

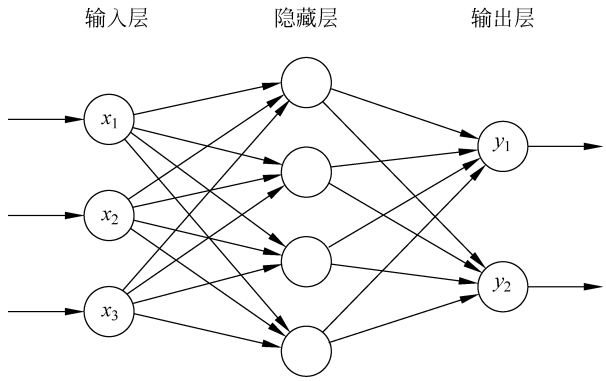

图 3.3　三层前向神经网络模型

3.2　反向传播

第 1 章中介绍了梯度下降算法训练回归模型，神经网络模型也一样需要使用梯度下降算法来更新参数。然而一个神经网络通常会有上百万的参数，那么如何高效地计算这百万级别的参数是需要重点考虑的问题。神经网络中使用反向传播(Backward Propagation)算法，使得计算梯度更加有效率。

在介绍反向传播之前，先来介绍一下链式法则。假设有两个函数 $y=g(x)$ 和 $z=h(y)$，那么 z 对 x 的求导过程如下：

$$\frac{\partial z}{\partial x}=\frac{\partial z}{\partial y}\frac{\partial y}{\partial x} \tag{3.1}$$

假设有三个函数 $x=g(s)$、$y=h(s)$ 和 $z=k(x,y)$，z 对 s 的求导过程如下：

$$\frac{\partial z}{\partial s}=\frac{\partial z}{\partial x}\frac{\partial x}{\partial s}+\frac{\partial z}{\partial y}\frac{\partial y}{\partial s} \tag{3.2}$$

神经网络的梯度计算，就是依赖链式法则一层层反向传播的。

如图 3.4 所示的前向神经网络，输入层有 n 个属性 $x_1,x_2,\cdots,x_n$，中间隐藏层有 p 个神经元，第 j 个神经元为 h_j，$j\in(0,p-1)$。

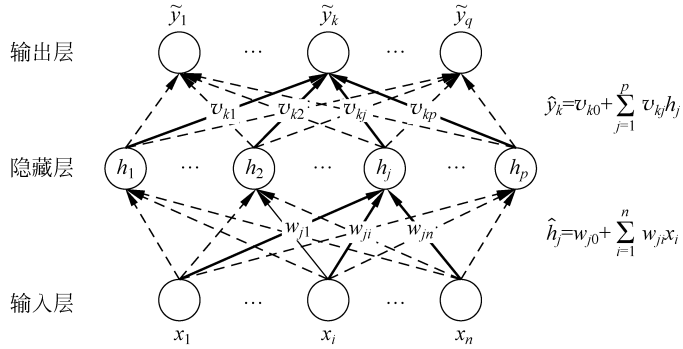

图 3.4　神经网络结构

输出层为 q 维。对隐藏层的每一个神经元 h_j，先经过一个线性变换，公式如下：

$$\hat{h}_j=w_{j0}+\sum_{i=1}^{n}w_{ji}x_i \tag{3.3}$$

式中，w_{j0}——偏置值；

$w_{j1},w_{j2},\cdots,w_{jn}$——作用在属性 $x_1,x_2,\cdots,x_n$ 上的权重。

$\hat{h}_j$ 输入给神经元后，经过激活函数的作用得到 $h_j=\alpha_1(\hat{h}_j)$。第二层同理有神经元输入：

$$\hat{y}_k=v_{k0}+\sum_{j=1}^{p}v_{kj}h_j \tag{3.4}$$

输出 $\tilde{y}_j=\alpha_2(\hat{y}_j)$。以上为前向神经网络的前向传播(Forward Propagation)过程。

对单个数据样本$(\boldsymbol{x},\boldsymbol{y})$，假设损失函数为均方差，则对第 k 个输出项的损失为：

$$J_k=\frac{1}{2}(\tilde{y}_k-y_k)^2 \tag{3.5}$$

通过链式法则，损失函数对权重 v_{kj} 的梯度为：

$$\frac{\partial J_k}{\partial v_{kj}}=\frac{\partial J_k}{\partial \tilde{y}_k}\cdot\frac{\partial \tilde{y}_k}{\partial \hat{y}_k}\cdot\frac{\partial \hat{y}_k}{\partial v_{kj}} \tag{3.6}$$

式中，第一项$\frac{\partial J_k}{\partial \tilde{y}_k}=(\tilde{y}_k-y_k)$；

第三项$\frac{\partial \hat{y}_k}{\partial v_{kj}}=h_j$。

而对于第二项，假设激活函数为 sigmoid 函数，则梯度有一个很好的性质，即：

$$\frac{\partial \tilde{y}_k}{\partial \hat{y}_k}=\tilde{y}_k(1-\tilde{y}_k) \tag{3.7}$$

三项相乘可以得到：

$$\frac{\partial J_k}{\partial v_{kj}}=\tilde{y}_k(\tilde{y}_k-y_k)(1-\tilde{y}_k)h_j \tag{3.8}$$

由于真实标签 y_k 由数据给定，而输出值 $\tilde{y}_k$ 与 h_j 均由前向传播算法计算得到，则可以轻易地计算得到每个中间层权重 v_{kj}，并且该计算过程可以并行进行。

类似地，可以得到损失值在隐藏单元 h_j 上的累积梯度为

$$\begin{aligned} e_{h_j}&=\frac{\partial J}{\partial h_j}\\ &=\sum_{k=1}^{q}\frac{\partial J_k}{\partial \tilde{y}_k}\cdot\frac{\partial \tilde{y}_k}{\partial \hat{y}_k}\cdot\frac{\partial \hat{y}_k}{\partial h_j}\\ &=\sum_{k=1}^{q}\tilde{y}_k(\tilde{y}_k-y_k)(1-\tilde{y}_k)v_{kj} \end{aligned} \tag{3.9}$$

同理，可以通过链式法则，得到损失函数对第一层权重 w_{ji} 的梯度为（假设隐藏层激活函数为 ReLU 函数）：

$$\begin{aligned} \frac{\partial J}{\partial w_{ji}}&=\frac{\partial J}{\partial h_j}\cdot\frac{\partial h_j}{\partial \hat{h}_j}\cdot\frac{\partial \hat{h}_j}{\partial w_{ji}}\\ &=e_{h_j}\cdot\frac{\partial h_j}{\partial \hat{h}_j}\cdot x_i \end{aligned} \tag{3.10}$$

式中，$\frac{\partial h_j}{\partial \hat{h}_j}=\begin{cases}0, & \hat{h}_j\leqslant 0\\ 1, & \hat{h}_j>0\end{cases}$。

在上一步计算得到 e_{h_j} 后，$\frac{\partial J}{\partial w_{ji}}$也可以高效并行地计算出。

在上述过程中，假设了损失函数是均方差，激活函数为 sigmoid 和 ReLU，其实这样的计算法则对任意可微的损失函数和激活函数都是有效的。从计算过程来看，在前向传

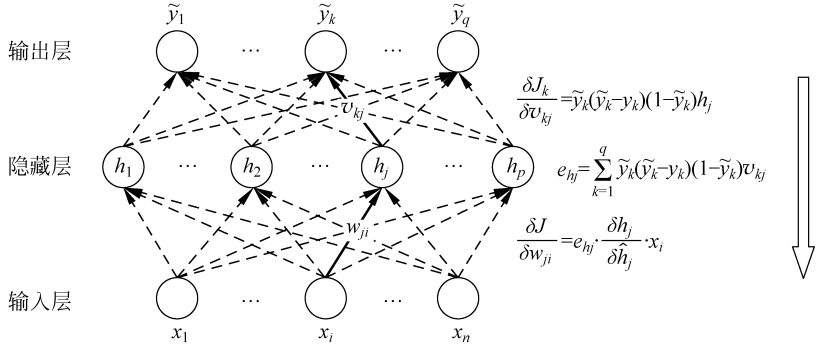

图 3.5　反向传播示意图

播得到隐藏层与输出层数值后，先从损失函数算起，接着从顶层逐渐计算梯度，将梯度逐层往输入层传播。这与前向传播的顺序是相反的，也是该算法为什么被称为反向传播的原因。反向传播（见图 3.5）得到所有参数的梯度之后，可以利用梯度下降算法对参数进行更新迭代，从而达到训练神经网络的目的。神经网络训练过程如算法 3.1 所示。

算法 3.1　神经网络训练过程

输入：数据集$\{(x^{(i)},y^{(i)})\}_{i=1}^{n}$，步长为 α，小批量训练样本的大小为 b，迭代次数为 T
输出：训练完成的神经网络
(1) 初始化网络参数 w_0
(2) for $t\in\{1,2,\cdots,T\}$
(3)　　从 m 个样本中均匀随机无放回选取 b 个样本 m_b
(4)　　前向传播逐层计算隐藏层的参数值，得到样本输出 $\tilde{\boldsymbol{y}}$
(5)　　根据损失函数计算误差，得到输出层梯度
(6)　　反向逐层计算隐藏层梯度
(7)　　计算连接参数梯度并更新参数

$$w_t \leftarrow w_{t-1}-\frac{\alpha}{b}\sum_{i\in m_b}\partial_{\boldsymbol{w}}J^{(i)}(\boldsymbol{w}) \tag{3.11}$$

3.3　泛化能力

在过拟合与欠拟合部分，介绍了泛化能力的概念，即机器学习算法对新样本的适应能力。神经网络具有大量的参数和很强的非线性变换能力，因而也很容易导致在训

练集上过拟合。训练集上准确率很高，损失很低，但在测试数据上效果很差，也就是缺乏泛化能力，不能适应新样本。从另一个角度来讲，模型在训练集上的准确度高，可能仅仅是记住了一些样本的实际标签，而没有学习到数据本身的特性，这种记忆学习在未见过的测试样本上是行不通的。

泛化能力不足的原因通常有以下几点：数据集有噪声、训练数据不足或训练模型过度导致模型非常复杂。为了提高模型的泛化能力，提出了很多解决方法，例如通过降低模型深度、宽度来减小模型复杂度；对数据集进行增强(Data Augmentation)，如将图像旋转、平移、缩放等；添加有规则的噪音，例如高斯噪声；加入正则化项(Regularizer)控制参数复杂度；训练过程中使用早停法(Early Stopping)等。

3.4节将介绍提高深度神经网络泛化能力的具体训练方法。

3.4　用 MindSpore 实现简单神经网络

说明：随着开发迭代 MindSpore 的接口及流程的不断演进，书中代码仅为示意代码，完整可运行代码请大家以线上代码仓中对应章节代码为准。

网址为：https://mindspore.cn/resource。读者可扫描右侧二维码获取相关资源。

LeNet 主要用来进行手写字符的识别与分类，并已在美国的银行中投入使用。LeNet 的实现确立了卷积神经网络(CNN)的结构，现在神经网络中的许多内容在 LeNet 的网络结构中都能看到，例如卷积层、池化(Pooling)层和 ReLU 层。虽然 LeNet 早在20世纪90年代就已经提出，但由于当时缺乏大规模的训练数据，计算机硬件的性能也较低，LeNet 神经网络在处理复杂问题时效果并不理想。LeNet 网络结构比较简单，刚好适合神经网络的入门学习。

3.4.1　各层参数说明

LeNet-5 是早期卷积神经网络中最有代表性的实验系统之一，它共有7层(不包含输入层)，每层都包含可训练参数和多个特征图(Feature Map)，每个特征图通过一种

卷积滤波器提取输入的一种特征，每个特征图有多个神经元。

1. Input 层——输入层

首先是数据 Input 层，输入图片的尺寸统一归一化为 32×32。

需要注意的是，本层不算 LeNet-5 的网络结构，传统上不将输入层视为网络层次结构之一。

2. C1 层——卷积层

C1 层详细信息如下：

(1) 输入图片大小：32×32。

(2) 卷积核大小：5×5。

(3) 卷积核种类：6。

(4) 输出特征图大小：28×28(28 由“32－5＋1”计算得出)。

(5) 神经元数量：28×28×6＝4704。

(6) 可训练参数：(5×5＋1)×6＝156。

(7) 连接数：(5×5＋1)×6×28×28＝122 304。

3. S2 层——池化层(降采样层)

S2 层详细信息如下：

(1) 输入大小：28×28。

(2) 采样区域：2×2。

(3) 采样方式：4 个输入相加，乘以可训练参数，再加上可训练偏置。

(4) 采样种类：6。

(5) 输出特征图大小：14×14(14 由 28/2 计算得出)。

(6) 神经元数量：14×14×6＝1176。

(7) 可训练参数：2×6＝12。

(8) 连接数：(2×2＋1)×6×14×14＝5880。

S2 中每个特征图的大小是 C1 中特征图大小的 1/4。

4. C3 层——卷积层

C3 层详细信息如下：

(1) 输入：S2 中所有 6 个或者几个特征图组合。

(2) 卷积核大小：5×5。

(3) 卷积核种类：16。

(4) 输出特征图大小：10×10 (C3 中的每个特征图是连接到 S2 中的所有 6 个特征,表示本层的特征图是上一层提取到的特征图的不同组合)。

(5) 可训练参数：6×(3×25＋1)＋6×(4×25＋1)＋3×(4×25＋1)＋(25×6＋1)＝1516。

(6) 连接数：10×10×1516＝151600。

5. S4 层——池化层(降采样层)

S4 层详细信息如下：

(1) 输入大小：10×10。

(2) 采样区域：2×2。

(3) 采样方式：4 个输入相加,乘以可训练参数,再加上可训练偏置。

(4) 采样种类：16。

(5) 输出特征图大小：5×5(5 由 10/2 计算得出)。

(6) 神经元数量：5×5×16＝400。

(7) 可训练参数：2×16＝32。

(8) 连接数：16×(2×2＋1)×5×5＝2000。

S4 中每个特征图的大小是 C3 中特征图大小的 1/4。

6. C5 层——卷积层

C5 层详细信息如下：

(1) 输入：S4 层的全部 16 个单元特征图(与 S4 全连接)。

(2) 卷积核大小：5×5。

(3) 卷积核种类：120。

(4) 输出特征图大小：1×1(1 由 5－5＋1 计算得出)。

(5) 可训练参数/连接：120×(16×5×5＋1)＝48120。

7. F6 层——全连接层

F6 层详细信息如下：

(1) 输入：C5 120 维向量。

(2) 计算方式：计算输入向量和权重向量之间的点积，再加上偏置。

(3) 可训练参数：84×(120+1)=10164。

8. Output 层——全连接层

Output 层也是全连接层，共有 10 个节点，分别用数字 0～9 表示。

3.4.2 详细步骤

下面描述使用 LeNet 网络训练和推理的详细步骤，并给出示例代码。

1. 加载 MindSpore 模块

使用 MindSpore API 前需要先导入 MindSpore API 和辅助模块，如代码 3.1 所示。

代码 3.1 导入 MindSpore API 和辅助模块

```
import mindspore.nn as nn
from mindspore.train import Model
from mindspore import context
```

2. 导入数据集

使用 MindSpore 数据格式 API 创建 Mnist 数据集，其中下面调用的 train_dataset()函数具体实现和 MindSpore 数据格式 API 介绍详见第 14 章。

3. 定义 LeNet 网络

定义 LeNet-5 网络结构，核心代码如代码 3.2 所示。

代码 3.2 定义 LeNet-5 网络结构

```
class LeNet5(nn.Cell):
  def __init__(self):
    super(LeNet5, self).__init__()
    self.conv1 = nn.Conv2d(1, 6, 5, pad_mode="valid")
    self.conv2 = nn.Conv2d(6, 16, 5, pad_mode="valid")
```

```
        self.fc1 = nn.Dense(16 * 5 * 5, 120)
        self.fc2 = nn.Dense(120, 84)
        self.fc3 = nn.Dense(84, 10)
        self.relu = nn.ReLU()
        self.max_pool2d = nn.MaxPool2d(kernel_size=2)
        self.flatten = nn.Flatten()

    def construct(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.max_pool2d(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.max_pool2d(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x
```

__init__()函数完成了卷积层和全连接层的初始化。初始化参数包括输入个数、输出个数、卷积层的参数以及卷积核大小。因为原始数据集的图片大小是 28×28，所以在导入数据集的过程中，需要将输入大小转变成 32×32。

construct()函数实现了前向传播。根据定义对输入依次进行卷积、激活、池化等操作，最后返回计算结果。在全连接层之前，先对数据进行展开操作，使用 Flatten()函数实现，这个函数可以在保留第 0 轴的情况下，对输入的张量进行扁平化(Flatten)处理。

4. 设置超参数并创建网络

定义损失函数和优化器。损失函数定义为 SoftmaxCrossEntropyWithLogits，采用 Softmax 进行交叉熵计算。选取 Momentum 优化器，学习率设置为 0.1，动量为 0.9，核心代码如代码 3.3 所示。

代码 3.3　设置超参数并创建网络

```
batch_size = 32
epoch_size = 2
lr = 0.1
```

```
momentum = 0.9

ds = train_dataset()
network = LeNet5()
network.set_train()
loss = nn.SoftmaxCrossEntropyWithLogits(is_grad = False, sparse = True)
opt = nn.Momentum(lr, momentum, network.trainable_params())
```

5. 训练网络模型

把网络、损失函数和优化器传入模型中，调用 train()方法即可开始训练，核心代码如代码 3.4 所示。

代码 3.4　训练网络模型

```
model = Model(network, loss, opt)
model.train(epoch_size, ds)
```