第 3 章

CHAPTER 3

分 类 问 题

人工智能可能会是“人类文明面临的最大风险”——伊隆·马斯克

前面已经介绍了用于连续值预测的线性回归模型，现在来挑战分类问题。分类问题的一个典型应用就是教会机器如何自动识别图片中物体的种类。考虑图片分类中最简单的任务之一：0～9 数字图片识别，它相对简单，而且也具有非常广泛的应用价值，例如邮政编码、快递单号、手机号码等都属于数字图片识别范畴。下面将以数字图片识别为例，探索如何用机器学习的方法去解决这个问题。

3.1 手写数字图片数据集

机器学习需要从数据中间学习，因此首先需要采集大量的真实样本数据。以手写的数字图片识别为例，如图 3.1 所示，需要收集大量的由真人书写的 0～9 的数字图片，为了便于存储和计算，一般把收集的原始图片缩放到某个固定的大小(Size 或 Shape)，例如 224 个像素的行和 224 个像素的列(224×224)，或者 96 个像素的行和 96 个像素的列(96×96)，这张图片将作为输入数据 $\boldsymbol{x}$。同时，需要给每一张图片标注一个标签(Label)，它将作为图片的真实值 y，这个标签表明这张图片属于哪一个具体的类别，一般通过映射方式将类别名一一对应到从 0 开始编号的数字，例如硬币的正反面，可以用 0 来表示硬币的反面，用 1 来表示硬币的正面，当然也可以反过来 1 表示硬币的反面，这种编码方式称为数字编码(Number Encoding)。对于手写数字图片识别问题，编码更为直观，用数字的 0～9 来表示类别名字为 0～9 的图片。

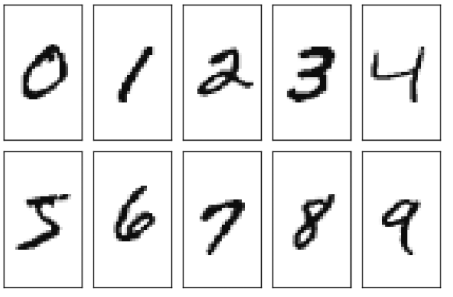


图 3.1 手写的数字图片样例

如果希望模型能够在新样本上也能具有良好的表现，即模型泛化能力(Generalization

Ability)较好,那么应该尽可能多地增加数据集的规模和多样性(Variance),使得用于学习的训练数据集与真实的手写数字图片的分布(Ground-truth Distribution)尽可能的逼近,这样在训练数据集上面学到了模型能够很好地用于未见过的手写数字图片的预测。

为了方便业界统一测试和评估算法,文献发布了手写数字图片数据集[1],命名为MNIST,它包含了0～9共10种数字的手写图片,每种数字一共有7000张图片,采集自不同书写风格的真实手写图片,一共70000张图片。其中60000张图片作为训练集D^{train}(Training Set),用来训练模型,剩下10000张图片作为测试集D^{test}(Test Set),用来预测或者测试,训练集和测试集共同组成了整个MNIST数据集。

考虑到手写数字图片包含的信息比较简单,每张图片均被缩放到28×28的大小,同时只保留了灰度信息,如图3.2所示。这些图片由真人书写,包含了如字体大小、书写风格、粗细等丰富的样式,确保这些图片的分布与真实的手写数字图片的分布尽可能地接近,从而保证了模型的泛化能力。

图3.2 MNIST数据集样例图片

现在来看图片的表示方法。一张图片包含了h行(Height/Row),w列(Width/Column),每个位置保存了像素(Pixel)值,像素值一般使用0～255的整形数值来表达颜色强度信息,例如0表示强度最低,255表示强度最高。如果是彩色图片,则每个像素点包含了R、G、B三个通道的强度信息,分别代表红色通道、绿色通道、蓝色通道的颜色强度,所以与灰度图片不同,它的每个像素点使用一个1维、长度为3的向量(Vector)来表示,向量的3个元素依次代表了当前像素点上面的R、G、B颜色强值,因此彩色图片需要保存为形状是$[h,w,3]$的张量(Tensor,可以通俗地理解为3维数组)。如果是灰度图片,则使用一个数值来表示灰度强度,例如0表示纯黑,255表示纯白,因此它只需要一个形状为$[h,w]$的二维矩阵(Matrix)来表示一张图片信息(也可以保存为$[h,w,1]$形状的张量)。图3.3演示了内容为8的数字图片的矩阵内容,可以看到,图片中黑色的像素用0表示,灰度信息用0～255表示,图片中越白的像素点,对应矩阵位置中数值也就越大。

目前常用的深度学习框架,如TensorFlow、PyTorch等,都可以非常方便地通过数行代

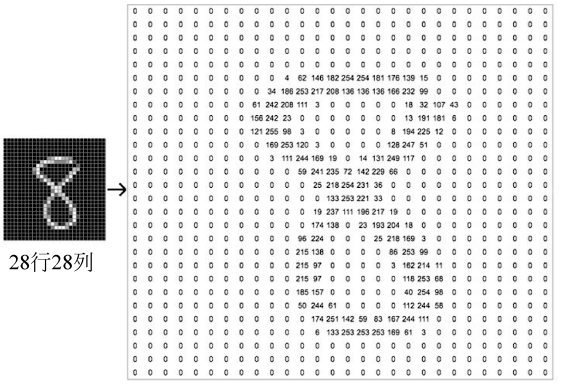


图 3.3 图片的表示①

码自动下载、管理和加载 MNIST 数据集，不需要额外编写代码，使用起来非常方便。这里利用 TensorFlow 自动在线下载 MNIST 数据集，并转换为 Numpy 数组格式。

```
import os
import tensorflow as tf                                          # 导入 TF 库
from tensorflow import keras                                     # 导入 TF 子库 keras
from tensorflow.keras import layers, optimizers, datasets        # 导入 TF 子库等
(x, y), (x_val, y_val) = datasets.mnist.load_data()              # 加载 MNIST 数据集
x = 2 * tf.convert_to_tensor(x, dtype = tf.float32)/255. - 1     # 转换为浮点张量,并缩放到 - 1～1
y = tf.convert_to_tensor(y, dtype = tf.int32)                    # 转换为整型张量
y = tf.one_hot(y, depth = 10)                                    # one - hot 编码
print(x.shape, y.shape)
train_dataset = tf.data.Dataset.from_tensor_slices((x, y))       # 构建数据集对象
train_dataset = train_dataset.batch(512)                         # 批量训练
```

load_data()函数返回两个元组(tuple)对象，第一个是训练集，第二个是测试集，每个 tuple 的第一个元素是多个训练图片数据 $\boldsymbol{X}$，第二个元素是训练图片对应的类别数字 $\boldsymbol{Y}$。其中训练集 $\boldsymbol{X}$ 的大小为(60000,28,28)，代表了 60000 个样本，每个样本由 28 行、28 列构成，由于是灰度图片，故没有 RGB 通道；训练集 $\boldsymbol{Y}$ 的大小为(60000)，代表了这 60000 个样本的标签数字，每个样本标签用一个范围为 0～9 的数字表示。测试集 $\boldsymbol{X}$ 的大小为(10000,28,28)，代表了 10000 张测试图片，$\boldsymbol{Y}$ 的大小为(10000)。

从 TensorFlow 中加载的 MNIST 数据图片，数值的范围为[0,255]。在机器学习中间，一般希望数据在 0 周围的小范围内分布。通过预处理步骤，把[0,255]像素范围归一化(Normalize)到[0,1]区间，再缩放到[－1,1]区间，从而有利于模型的训练。

每一张图片的计算流程是通用的，在计算的过程中可以一次进行多张图片的计算，充分

① 素材来自 https://towardsdatascience.com/how-to-teach-a-computer-to-see-with-convolutional-neural-networks-96c120827cd1。

利用 CPU 或 GPU 的并行计算能力。用形状为$[h,w]$的矩阵来表示一张图片，对于多张图片来说，在前面添加一个数量维度(Dimension)，使用形状为$[b,h,w]$的张量来表示，其中 b 代表了批量(Batch Size)；多张彩色图片可以使用形状为$[b,h,w,c]$的张量来表示，其中 c 表示通道数量(Channel)，彩色图片 $c=3$。通过 TensorFlow 的 Dataset 对象可以方便地完成模型的批量训练，只需要调用 batch()函数即可构建带 batch 功能的数据集对象。

3.2　模型构建

回顾在回归问题中讨论的生物神经元结构。把一组长度为 d_{in} 的输入向量 $\boldsymbol{x}=[x_1,x_2,\cdots,x_{d_{\text{in}}}]^{\mathrm{T}}$ 简化为单输入标量 x，模型可以表达成 $y=xw+b$。如果是多输入、单输出的模型结构，需要借助于向量形式：

$$\boldsymbol{y}=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b=[w_1,w_2,w_3,\cdots,w_{d_{\text{in}}}]\cdot\begin{bmatrix}x_1\\x_2\\x_3\\\vdots\\x_{d_{\text{in}}}\end{bmatrix}+b$$

更一般地，通过组合多个多输入、单输出的神经元模型，可以拼成一个多输入、多输出的模型：

$$\boldsymbol{y}=\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}$$

其中，$\boldsymbol{x}\in\mathbf{R}^{d_{\text{in}}}$，$\boldsymbol{b}\in\mathbf{R}^{d_{\text{out}}}$，$\boldsymbol{y}\in\mathbf{R}^{d_{\text{out}}}$，$\boldsymbol{W}\in\mathbf{R}^{d_{\text{out}}\times d_{\text{in}}}$。

对于多输出节点、批量训练方式，将模型写成批量形式：

$$\boldsymbol{Y}=\boldsymbol{X}@\boldsymbol{W}+\boldsymbol{b}\tag{3.1}$$

其中 $\boldsymbol{X}\in\mathbf{R}^{b\times d_{\text{in}}}$，$\boldsymbol{b}\in\mathbf{R}^{d_{\text{out}}}$，$\boldsymbol{Y}\in\mathbf{R}^{b\times d_{\text{out}}}$，$\boldsymbol{W}\in\mathbf{R}^{d_{\text{in}}\times d_{\text{out}}}$，$d_{\text{in}}$ 表示输入节点数，d_{out} 表示输出节点数；$\boldsymbol{X}$ 形状为$[b,d_{\text{in}}]$，表示 b 个样本的输入数据，每个样本的特征长度为 d_{in}；$\boldsymbol{W}$ 的形状为$[d_{\text{in}},d_{\text{out}}]$，共包含了 $d_{\text{in}}*d_{\text{out}}$ 个网络参数；偏置向量 $\boldsymbol{b}$ 形状为 d_{out}，每个输出节点上均添加一个偏置值；@符号表示矩阵相乘(Matrix Multiplication，简称 matmul)。由于 $\boldsymbol{X}@\boldsymbol{W}$ 的运算结果是形状为$[b,d_{\text{out}}]$的矩阵，与向量 $\boldsymbol{b}$ 并不能直接相加，因此批量形式的＋号需要支持自动扩展功能(Broadcasting)，将向量 $\boldsymbol{b}$ 扩展为形状为$[b,d_{\text{out}}]$的矩阵后，再与 $\boldsymbol{X}@\boldsymbol{W}$ 相加。

考虑两个样本，输入特征长度 $d_{\text{in}}=3$，输出特征长度 $d_{\text{out}}=2$ 的模型，式(3.1)展开为：

$$\begin{bmatrix}o_1^{(1)} & o_2^{(1)}\\o_1^{(2)} & o_2^{(2)}\end{bmatrix}=\begin{bmatrix}x_1^{(1)} & x_2^{(1)} & x_3^{(1)}\\x_1^{(2)} & x_2^{(2)} & x_3^{(2)}\end{bmatrix}\begin{bmatrix}w_{11} & w_{12}\\w_{21} & w_{22}\\w_{31} & w_{32}\end{bmatrix}+\begin{bmatrix}b_1\\b_2\end{bmatrix}$$

其中 $x_1^{(1)}$、$o_1^{(1)}$ 等符号的上标表示样本索引号(样本编号)，下标表示某个样本向量的元素。

对应模型结构如图 3.4 所示。

可以看到，通过矩阵形式表达网络结构，更加简洁清晰，同时也可充分利用矩阵计算的并行加速能力。如何将图片识别任务的输入和输出转变为满足格式要求的张量形式？

考虑输入格式，一张灰度图片 $\boldsymbol{x}$ 使用矩阵方式存储，形状为$[h,w]$，b 张图片使用形状为$[b,h,w]$的张量 $\boldsymbol{X}$ 存储。而模型只能接受向量形式的输入特征向量，因此需要将$[h,w]$的矩阵形式图片特征打平成$[h\cdot w]$长度的向量，如图 3.5 所示，其中输入特征的长度 $d_{\text{in}}=h\cdot w$。

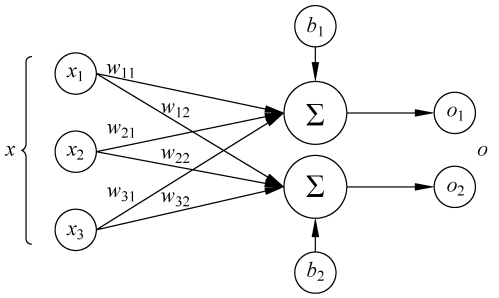


图 3.4　3 输入 2 输出模型

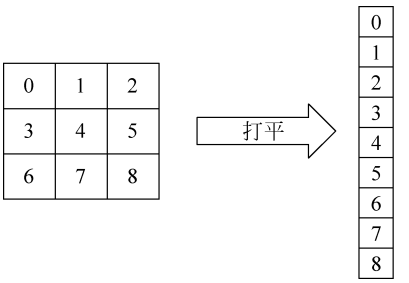


图 3.5　矩阵打平操作

对于输出标签 y，前面已经介绍了数字编码，它可以用一个数字来表示标签信息，此时输出只需要一个节点即可表示网络的预测类别值，例如数字 1 表示猫，数字 3 表示鱼等（编程实现时一般从 0 开始编号）。但是数字编码一个最大的问题是，数字之间存在天然的大小关系，例如 1＜2＜3，如果 1、2、3 分别对应的标签是猫、狗、鱼，它们之间并没有大小关系，所以采用数字编码时会迫使模型去学习这种不必要的约束。

如何解决这个问题？可以将输出设置为 d_{out} 个输出节点的向量，d_{out} 与类别数相同，让第 $i\in[1,d_{\text{out}}]$个输出节点的值表示当前样本属于类别 i 的概率 $P(\boldsymbol{x}$ 属于类别 $i\mid\boldsymbol{x})$。只考虑输入图片只输入一个类别的情况，此时输入图片的真实标签已经唯一确定：如果物体属于第 i 类，那么索引为 i 的位置上设置为 1，其他位置设置为 0，把这种编码方式称为 One-hot 编码（独热编码）。以图 3.6 中的猫、狗、鱼、鸟识别系统为例，所有的样本只属于猫、狗、鱼、鸟 4 个类别中其一，将第 1～4 号索引位置分别表示猫、狗、鱼、鸟的类别，对于所有猫的图片，它的数字编码为 0，One-hot 编码为[1,0,0,0]；对于所有狗的图片，它的数字编码为 1，One-hot 编码为[0,1,0,0]，以此类推。One-hot 编码方式在分类问题中应用非常广

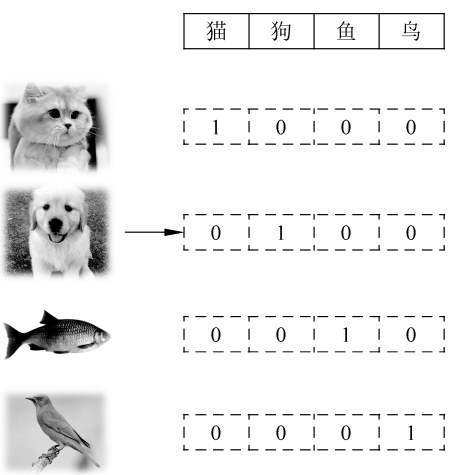


图 3.6　猫、狗、鱼、鸟系统 One-hot 编码

泛，需要理解并掌握。

手写数字图片的总类别数有 10 种，即输出节点数 $d_{out}=10$，那么对于某个样本，假设它属于类别 i，即图片中的数字为 i，只需要一个长度为 10 的向量 $\boldsymbol{y}$，向量 $\boldsymbol{y}$ 的索引号为 i 的元素设置为 1，其他位为 0。例如图片 0 的 One-hot 编码为[1,0,0,…,0]，图片 2 的 One-hot 编码为[0,0,1,…,0]，图片 9 的 One-hot 编码为[0,0,0,…,1]。One-hot 编码是非常稀疏(Sparse)的，相对于数字编码来说，占用较多的存储空间，所以一般在存储时还是采用数字编码，在计算时，根据需要把数字编码转换成 One-hot 编码，通过 tf.one_hot 函数即可实现。

```
In [1]:
y = tf.constant([0,1,2,3])                    # 数字编码的 4 个样本标签
y = tf.one_hot(y, depth = 10)                 # One - hot 编码,指定类别总数为 10
print(y)
Out[1]:
tf.Tensor(
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]              # 数字 0 的 One - hot 编码向量
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]              # 数字 1 的 One - hot 编码向量
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]              # 数字 2 的 One - hot 编码向量
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]], shape = (4, 10), dtype = float32)
```

现在回到手写数字图片识别任务，输入是一张打平后的图片向量 $\boldsymbol{x}\in\mathbf{R}^{784}$，输出是一个长度为 10 的向量 $\boldsymbol{o}\in\mathbf{R}^{10}$，图片的真实标签 $\boldsymbol{y}$ 经过 One-hot 编码后变成长度为 10 的非 0 即 1 的稀疏向量 $\boldsymbol{y}\in\{0,1\}^{10}$。预测模型采用多输入、多输出的线性模型 $\boldsymbol{o}=\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}$，其中模型的输出记为输入的预测值 $\boldsymbol{o}$，希望 $\boldsymbol{o}$ 越接近真实标签 $\boldsymbol{y}$ 越好。一般把输入经过一次(线性)变换称为一层网络。

3.3　误差计算

对于分类问题来说，我们的目标是最大化某个性能指标，例如准确度 acc，但是把准确度当作损失函数去优化时，会发现$\frac{\partial \text{acc}}{\partial \theta}$其实是不可导的，无法利用梯度下降算法优化网络参数 θ。一般的做法是，设立一个平滑可导的代理目标函数，例如优化模型的输出 $\boldsymbol{o}$ 与 One-hot 编码后的真实标签 $\boldsymbol{y}$ 之间的距离(Distance)，通过优化代理目标函数得到的模型，一般在测试性能上也能有良好的表现。因此，相对回归问题而言，分类问题的优化目标函数和评价目标函数是不一致的。模型的训练目标是通过优化损失函数$\mathcal{L}$来找到最优数值解 $\boldsymbol{W}^*,\boldsymbol{b}^*$：

$$\boldsymbol{W}^*,\boldsymbol{b}^*=\underbrace{\text{argmin}}_{\boldsymbol{W},\boldsymbol{b}}\mathcal{L}(\boldsymbol{o},\boldsymbol{y})$$

对于分类问题的误差计算来说，更常见的是采用交叉熵(Cross Entropy)损失函数，较少采用回归问题中介绍的均方差损失函数。将在后续章节介绍交叉熵损失函数，这里仍然使用均方差损失函数来求解手写数字识别问题。对于 n 个样本的均方差损失函数可以表达为：

$$\mathcal{L}(\boldsymbol{o},\boldsymbol{y})=\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{10}(o_j^{(i)}-y_j^{(i)})^2$$

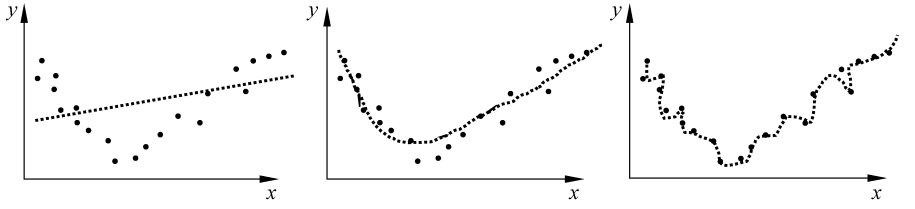
现在只需要采用梯度下降算法来优化损失函数得到$\boldsymbol{W},\boldsymbol{b}$的最优解，然后再利用求得的模型去预测未知的手写数字图片$\boldsymbol{x}\in D^{\text{test}}$。

3.4 真的解决了吗

按照上面的方案，手写数字图片识别问题真的得到了完美的解决吗？目前来看，至少存在两大问题：

- **线性模型**：线性模型是机器学习中间最简单的数学模型之一，参数量少，计算简单，但是只能表达线性关系。即使是简单如数字图片识别任务，它也属于图片识别的范畴，人类目前对于复杂大脑的感知和决策的研究尚处于初步探索阶段，如果只使用一个简单的线性模型去逼近复杂的人脑图片识别模型，很显然不能胜任。
- **表达能力**：表达能力体现为逼近复杂分布的能力。上面的解决方案只使用了少量神经元组成的一层网络模型，相对于人脑中千亿级别的神经元互联结构，它的表达能力明显偏弱。

模型的表达能力与数据模态之间的关系如图 3.7 所示，图中绘制了带观测误差的采样点的分布，人为推测数据的真实分布可能是某二次抛物线模型。如图 3.7(a)所示，如果使用表达能力偏弱的线性模型去学习，很难学习到比较好的模型；如果使用合适的多项式函数模型去学习，例如二次多项式，则能学到比较合适的模型，如图 3.7(b)所示；但模型过于复杂，表达能力过强时，例如 10 次多项式，则很有可能会过拟合，伤害模型的泛化能力，如图 3.7(c)所示。



(a) 表达能力偏弱　(b) 表达能力与数据模态匹配　(c) 表达能力过强

图 3.7　模型表达能力与数据模态

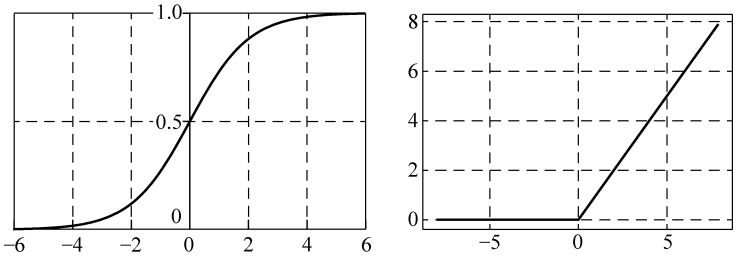
目前所采用的多神经元模型仍是线性模型，表达能力偏弱，接下来尝试解决这两个问题。

3.5 非线性模型

既然线性模型不可行，可以给线性模型嵌套一个非线性函数，即可将其转换为非线性模型。把这个非线性函数称为激活函数(Activation Function)，用σ表示：

$$\boldsymbol{o}=\sigma(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b})$$

这里的 σ 代表了某个具体的非线性激活函数，如 Sigmoid 函数（见图 3.8(a)）、ReLU 函数（见图 3.8(b)）。



(a) Sigmoid函数　　(b) ReLU函数

图 3.8　常见激活函数

ReLU 函数非常简单，在 $y=x$ 的基础上面截去了 $x<0$ 的部分，可以直观地理解为 ReLU 函数仅保留正的输入部分，清零负的输入，具有单边抑制特性。虽然简单，ReLU 函数却有优良的非线性特性，而且梯度计算简单，训练稳定，是深度学习模型使用最广泛的激活函数之一。这里通过嵌套 ReLU 函数将模型转换为非线性模型：

$$\boldsymbol{o}=\mathrm{ReLU}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b})$$

3.6　表达能力

针对模型的表达能力偏弱的问题，可以通过重复堆叠多次变换来增加其表达能力：

$$\boldsymbol{h}_1=\mathrm{ReLU}(\boldsymbol{W}_1\boldsymbol{x}+\boldsymbol{b}_1)$$
$$\boldsymbol{h}_2=\mathrm{ReLU}(\boldsymbol{W}_2\boldsymbol{h}_1+\boldsymbol{b}_2)$$
$$\boldsymbol{o}=\boldsymbol{W}_3\boldsymbol{h}_2+\boldsymbol{b}_3$$

把第 1 层神经元的输出值 $\boldsymbol{h}_1$ 作为第 2 层神经元模型的输入，把第 2 层神经元的输出 $\boldsymbol{h}_2$ 作为第 3 层神经元的输入，最后一层神经元的输出作为模型的输出 $\boldsymbol{o}$。

从网络结构上看，如图 3.9 所示，函数的嵌套表现为网络层的前后相连，每堆叠一个（非）线性环节，网络层数增加一层。把输入节点 $\boldsymbol{x}$ 所在的层称为输入层，每一个非线性模块的输出 $\boldsymbol{h}_i$ 连同它的网络层参数 $\boldsymbol{W}_i$ 和 $\boldsymbol{b}_i$ 称为一层网络层，特别地，对于网络中间的层，称为隐藏层，最后一层称为输出层。这种由大量神经元模型连接形成的网络结构称为神经网络（Neural Network）。可以看到，神经网络并不难理解，神经网络的每层的节点数和神经网络的层数决定了神经网络的复杂度。

现在网络模型已经升级为 3 层的神经网络，具有较好的非线性表达能力，接下来讨论如何优化网络参数。

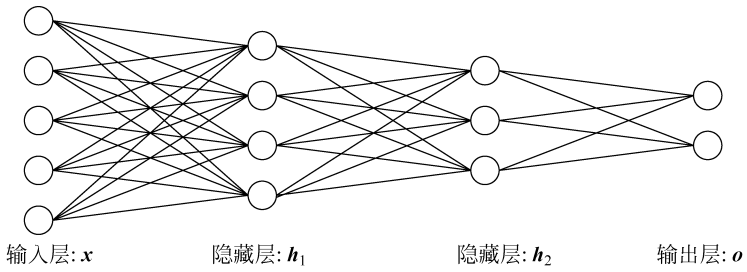


图 3.9　3 层神经网络结构

3.7　优化方法

对于仅一层的网络模型，如线性回归的模型，可以直接推导出$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}}$和$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}}$的偏导数表达式，然后直接计算每一步的梯度，根据梯度更新法则循环更新 $\boldsymbol{w}$ 和 $\boldsymbol{b}$ 参数即可。但是，当网络层数增加、数据特征长度增大以及添加复杂的非线性函数之后，模型的表达式将变得非常复杂，很难手动推导出模型和梯度的计算公式。而且一旦网络结构发生变动，网络的模型函数也随之发生改变，依赖手动计算梯度的方式显然不可行。

这时就是深度学习框架发明的意义所在，借助于自动求导(Autograd)技术，深度学习框架在计算神经网络每层的输出以及损失函数的过程中，会构建神经网络的计算图模型，并自动完成任意参数 θ 的偏导数$\frac{\partial \mathcal{L}}{\partial \theta}$的计算，用户只需要搭建出网络结构，梯度将自动完成计算和更新，使用起来非常便捷高效。

3.8　手写数字图片识别体验

本节将在未介绍 TensorFlow 的情况下，先带用户体验神经网络的乐趣。本节的主要目的并不是教会每个细节，而是让用户对神经网络算法有全面、直观的感受，为接下来介绍 TensorFlow 基础和深度学习理论打下基础。

3.8.1　网络搭建

对于第 1 层模型来说，它接受的输入 $\boldsymbol{x} \in \mathbf{R}^{784}$，输出 $\boldsymbol{h}_1 \in \mathbf{R}^{256}$ 设计长度为 256 的向量，不需要显式地编写 $\boldsymbol{h}_1 = \mathrm{ReLU}(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1)$ 的计算逻辑，在 TensorFlow 中通过一行代码即可实现：

```
# 创建一层网络，设置输出节点数为 256，激活函数类型为 ReLU
```

```
layers.Dense(256, activation = 'relu')
```

使用 TensorFlow 的 Sequential 容器可以非常方便地搭建多层的网络。对于 3 层网络，可以快速完成 3 层网络的搭建。

```
# 利用 Sequential 容器封装 3 个网络层，前网络层的输出默认作为下一层的输入
model = keras.Sequential([                    # 3 个非线性层的嵌套模型
    layers.Dense(256, activation = 'relu'),   # 隐藏层 1
    layers.Dense(128, activation = 'relu'),   # 隐藏层 2
    layers.Dense(10)])                        # 输出层，输出节点数为 10
```

第 1 层的输出节点数设计为 256，第 2 层设计为 128，输出层节点数设计为 10。直接调用这个模型对象 model(x)就可以返回模型最后一层的输出 $\boldsymbol{o}$。

3.8.2　模型训练

搭建完成 3 层神经网络的对象后，给定输入 $\boldsymbol{x}$，调用 model($\boldsymbol{x}$)得到模型输出 $\boldsymbol{o}$ 后，通过 MSE 损失函数计算当前的误差$\mathcal{L}$：

```
with tf.GradientTape() as tape:                # 构建梯度记录环境
    # 打平操作，[b, 28, 28] => [b, 784]
    x = tf.reshape(x, (-1, 28*28))
    # Step1. 得到模型输出 output [b, 784] => [b, 10]
    out = model(x)
    # [b] => [b, 10]
    y_onehot = tf.one_hot(y, depth = 10)
    # 计算差的平方和，[b, 10]
    loss = tf.square(out - y_onehot)
    # 计算每个样本的平均误差，[b]
    loss = tf.reduce_sum(loss) / x.shape[0]
```

再利用 TensorFlow 提供的自动求导函数 tape.gradient(loss, model.trainable_variables)求出模型中所有参数的梯度信息$\frac{\partial \mathcal{L}}{\partial \theta}$，$\theta \in \{\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2, \boldsymbol{W}_3, \boldsymbol{b}_3\}$。

```
# Step3. 计算参数的梯度 w1, w2, w3, b1, b2, b3
grads = tape.gradient(loss, model.trainable_variables)
```

计算获得的梯度结果使用 grads 列表变量保存。再使用 optimizers 对象自动按照梯度更新法则去更新模型的参数 θ。

$$\theta' = \theta - \eta \cdot \frac{\partial \mathcal{L}}{\partial \theta}$$

实现如下：

```
# 自动计算梯度
grads = tape.gradient(loss, model.trainable_variables)
```

```
    # w' = w - lr * grad,更新网络参数
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

循环迭代多次后，就可以利用学好的模型 f_θ 去预测未知的图片的类别概率分布。模型的测试部分暂不讨论。

手写数字图片 MNIST 数据集的训练误差曲线如图 3.10 所示，由于 3 层的神经网络表达能力较强，手写数字图片识别任务相对简单，误差值可以较快速、稳定地下降，其中，把对数据集的所有样本迭代一遍称为一个 Epoch，可以在间隔数个 Epoch 后测试模型的准确率等指标，方便监控模型的训练效果。

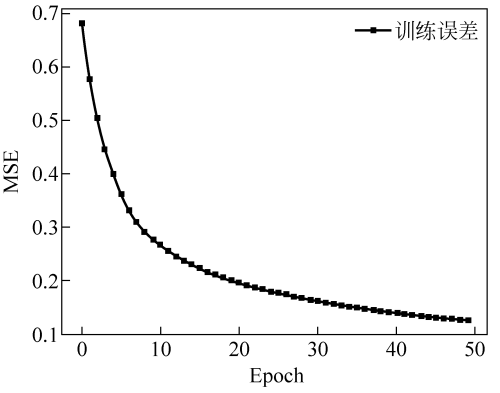


图 3.10　MNIST 数据集的训练误差曲线

本章我们通过将一层的线性回归模型类推到分类问题，提出了表达能力更强的三层非线性神经网络，去解决手写数字图片识别的问题。本章的内容以感受为主，学习完大家其实已经了解了（浅层的）神经网络算法，接下来我们将学习 TensorFlow 的一些基础知识，为后续正式学习、实现深度学习算法打下夯实的基石。

参考文献

[1] Lecun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 1998, 86: 2278-2324.