2.5 神经网络

从海量数据中自动学习

从 1950 年代的感知机到如今的大模型（见 1.3 节的介绍），神经网络可以说是历史最悠久的机器学习模型之一。其发展过程几起几落，终有大成。那么读者一定想知道，神经网络究竟有什么独到之处呢？笔者认为主要可归结为两点：第一，神经网络可以从海量数据中自动学习特征；第二，神经网络可以进行端对端的学习，只需要告诉他输入是什么、输出是什么，他会自动的学习输入到输出的映射。关于这两点，1.2.1 节有详细讨论，读者可以回顾一下。

本节将着眼于一种最常见的神经网络——全连接多层神经网络（Fully Connected Multiple Layers Neural Network）[1]，对神经网络的基本原理和反向传播算法进行详细介绍。

2.5.1 全连接多层神经网络

如果将 2.3 节介绍的“对数几率回归模型”作为基本单元（称为神经元），就能够类似“搭积木”一样方便地构建出一种最常见的神经网络——全连接多层神经网络。如图 2.5.1 所示，将多个神经元按照从左到右的顺序一层层叠起来，层与层之间所有神经元均连接在一起，最右边用一个多类对数几率回归（Softmax）完成最终的多类分类，这就得到了一个很常见的全连接多层神经网络。由于层与层之间所有神经元均连接在一起，所以称这种神经网络为“全连接”。

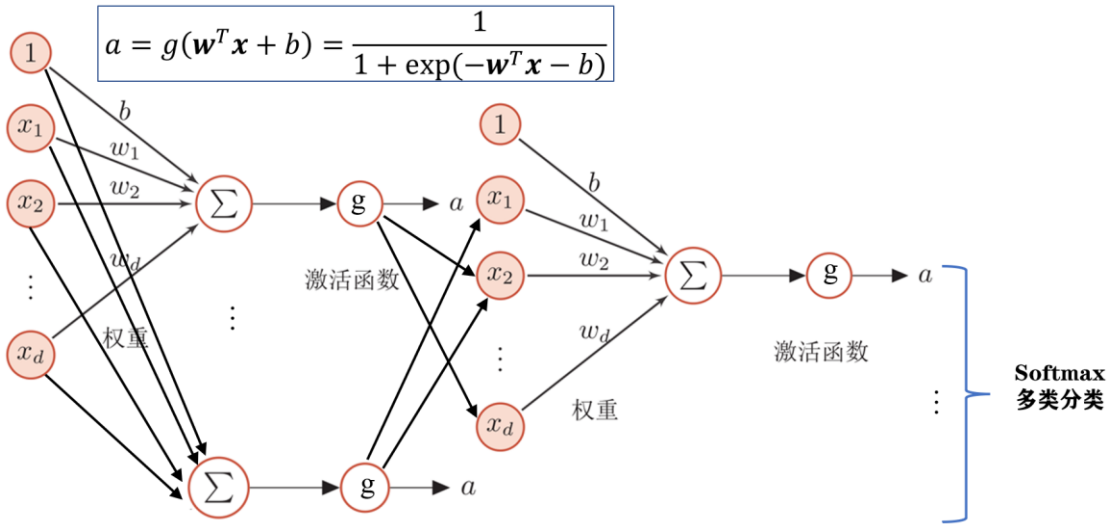

图 2.5.1 全连接多层神经网络

图 2.5.1 中，按照神经网络的习惯，将函数$g()$称为激活函数（Activation Functions），相应的将函数$g()$的输出值a称为激活值（Activation Value）。图 2.5.1 的最左边是整个神经网络的第 1 层——输入层（Input Layer），用输入向量$(1, x_1, x_2, \ldots, x_d)^T$表示。最右边是整个神经网络的最后一层——输出层（Output Layer），给出多类分类的类别值。其余中间的层次均接受前一层的输出作为输入，并将本层的输出作为后一层的输入。中间层一般也称为“隐藏层”

注 1：全连接多层神经网络也常被称为前馈网络（Feedforward）。

（Hidden Layer），简称“隐层”。图 2.5.2 给出了一个整体上更清楚的三层全连接神经网络的网络结构图，输入层有 784 个神经元（对应 784 维的输入向量），隐藏层有 15 个神经元，输出层有 10 个神经元（对应 10 个类别）。注意，为了整体结构的清楚，图 2.5.2 中忽略了神经元的细节（用圆圈表示），而且输入层的 784 个神经元只画了 8 个出来。这个三层网络由于只有一个隐层，因此也常被称为“单隐层神经网络”。如果有多个隐层，就可以称为“深度神经网络”了。实际上，后面将采用这个三层网络来解决一个经典问题：Mnist 手写数字识别。（隐藏层自动从数据中学习特征；端对端，只需给定输入和输出，而输入和输出由问题完全决定；隐藏层的黑盒特性）

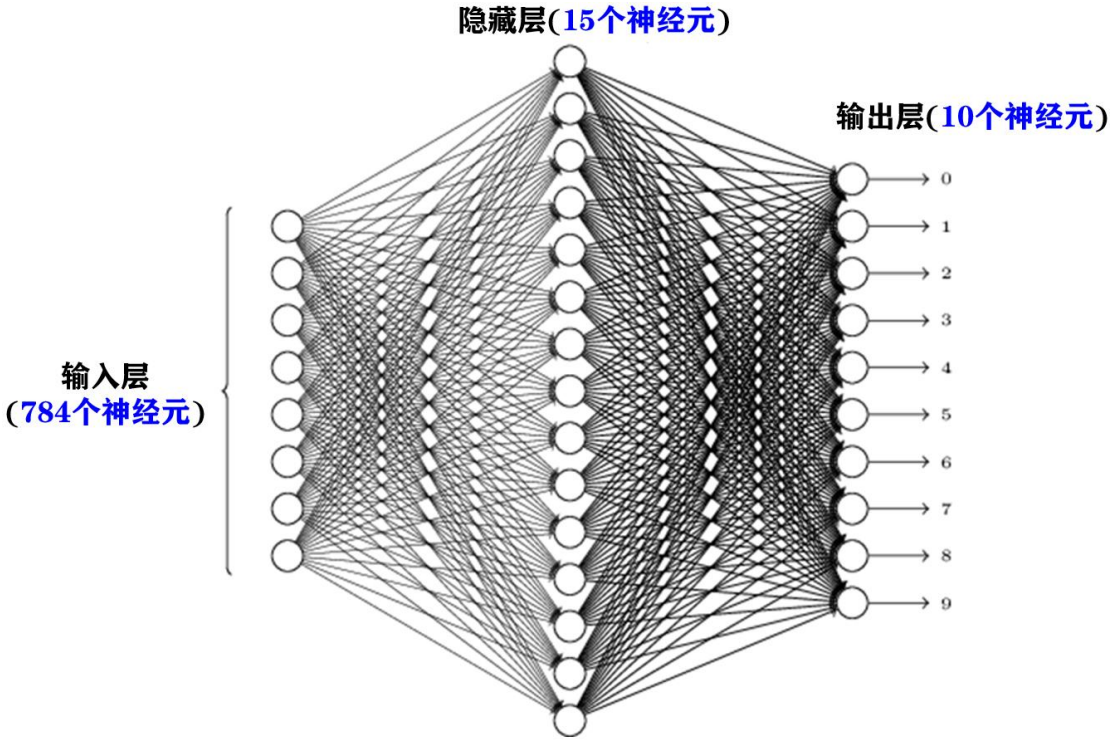

图 2.5.2 一个三层全连接神经网络

2.5.2 万能逼近定理

细心的读者一定会问一个问题：图 2.5.2 所示神经网络的隐藏层神经元个数是如何确定的呢？或者，引申一下这个问题：神经网络的学习能力有多强呢？神经网络的结构（有多少个隐层，每个隐层有多少个神经元）应该如何设计呢？可以说，神经网络的万能逼近定理试图从某种程度上回答这个问题。

这个定理的大意是：单隐层神经网络与任意连续 S 型激活函数能够以任意精度逼近任何目标连续函数，只要不对网络的结点数加以限制。说明一下，“S 型激活函数”就是 2.3.2 节谈到的“S型曲线函数”。

这就意味着，只要不对隐层的神经元个数加以限制，类似图 2.5.2 的单隐层神经网络能够完美解决任何机器学习问题，即神经网络具有“万能”的学习能力。这当然是个好消息。同时，这个定理建议了一种神经网络的结构：单个隐层，并且隐层的神经元个数可以任意多。以图 2.5.2 所示神经网络为例，图中相邻两层神经元之间的每个连接就是一个模型参数（回顾 1.2.2 节谈到的含参模型以及 2.3.1 节的线性模型），由此，在输入层和隐层之间就有 784x15 个参数，类似的，在隐层和输出层之间就有 15x10 个参数。参数的个数反映了模型的复杂程度，因此如果隐层的神经元个数可以任意多，则该单隐层神经网络可以任意复杂，从而可以

拟合任意复杂的输入数据。

谈到这里，聪明的读者一定会思考一个问题：在实践中，这样做是否经济划算？一方面，为了简化定理的证明，采用单隐层神经网络或许是很明智的。另一方面，在实际应用中也这样做，就不见得明智了，甚至会不可行。为什么这么说？以识别人脸为例，假如有 1 百万张不同的人脸图像数据，为了表示这些数据，对于单隐层网络我可能需要 1 百万个隐层神经元（因为每张人脸都不一样），存储开销将非常巨大。换一种做法，既然人脸都是眼睛、鼻子、嘴巴构成，那 100 种眼睛（对应第一个隐层）与 100 种鼻子（对应第二个隐层）再与 100 种嘴巴（对应第三个隐层）组合在一起不就表示出了这 1 百万张不同人脸了吗。这种多层组合式的表示方法总共只需要 300 个隐层神经元就解决了问题，经济划算得多。这就是为什么采用多个隐层，构建“深度神经网络”的根本原因所在。

当然，上面所说的“第一个隐层对应 100 种眼睛”这些只是一种示意的说法，是为了帮助读者理解“深度”的重要性。实际网络中一般很难说清楚某个隐层神经元所表示的具体内容（回顾 1.3 节谈到的神经网络的“不可解释性”）。因此，神经网络的结构设计（采用多少个隐层，每个隐层有多少个神经元）就成为了一个工程问题——“网络工程”，需要在实践中积累经验，不断优化。

2.5.3 学习算法

回顾一下 2.3 节，既然对率回归单元（神经元）可以通过负对数似然函数的梯度下降算法来进行学习，那么由对率回归单元堆叠而成的全连接神经网络也可以采用类似的方法吗？回答是肯定的，但是也有关键的不同：多层神经网络的损失函数是关于各层参数$\boldsymbol{w}$的复合函数。这就导致了需要面向复合函数的误差反向传播算法（Back Propagation, BP）。下面就来详谈。

假设目标变量$\boldsymbol{y}$有C个类别取值，为了后面公式推导的方便，将其取值定义为独热向量（One-hot Vector），即C维的标准单位向量$\boldsymbol{y}_i = (\dots,1,\dots)^T$：第$k$维$y_i^k$为 1 表示其类别为$k$。对应的，要求网络的输出$\boldsymbol{a}_i$也是一个$C$维向量。这样，就可以定义一个二次损失函数，用以衡量网络的输出与类别标签一致性如何。如式（2.5.1）所示，对于训练样本i，如果输出$\boldsymbol{a}_i$与类别标签$\boldsymbol{y}_i$比较一致，则损失就比较小；否则损失就比较大。其中N为总的训练样本数。注意，式（2.5.1）对求和的值除以N，以起到归一化的效果（对比 2.3 节的式(2.3.9)）。至于N前面的 2 倍则纯粹是为了后面求梯度得到一个更简洁的数学形式（2.4 节我们也曾采用了类似的小技巧）。

$$L(\boldsymbol{w},\boldsymbol{b}) = \frac{1}{2N}\sum_{i=1}^{N}\|\boldsymbol{y}_i - \boldsymbol{a}_i\|^2,$$

$$\text{其中}\boldsymbol{a_i} = \boldsymbol{g}(\boldsymbol{w^T x}_i + \boldsymbol{b}) = \frac{\mathbf{1}}{\mathbf{1}+\boldsymbol{exp}\,(-\boldsymbol{w^T x}_i-\boldsymbol{b})} \qquad (2.5.1)$$

要使损失$L(\boldsymbol{w},\boldsymbol{b})$最小，同样可以采用 2.3 节介绍的梯度下降方法。首先计算$L(\boldsymbol{w},\boldsymbol{b})$的梯度$\nabla_{\boldsymbol{w}}L(\boldsymbol{w},\boldsymbol{b})$和$\nabla_{\boldsymbol{b}}L(\boldsymbol{w},\boldsymbol{b})$。然后用公式$\boldsymbol{w} = \boldsymbol{w} - \alpha\nabla_{\boldsymbol{w}}L(\boldsymbol{w},\boldsymbol{b})$和$\boldsymbol{b} = \boldsymbol{b} - \alpha\nabla_{\boldsymbol{b}}L(\boldsymbol{w},\boldsymbol{b})$更新参数，直到$\boldsymbol{w}$和$\boldsymbol{b}$收敛或者达到最大迭代次数。关键的不同点在于，式（2.5.1）中的$\boldsymbol{a}_i$实际上本身也是一个关于$\boldsymbol{w}$和$\boldsymbol{b}$的函数；$\boldsymbol{x}_i$可能是输入层的输入，也可能是前一层的输出。因此需要用到多层复合函数的链式求导规则。这个求导的过程也被形象的称为误差反向传播算法——从输出层往输入层“反向”进行。下面就来详细介绍误差反向传播算法。

为了讨论的方便，先约定一些数学符号。用$i = 1,\dots,I$表示一个I层神经网络的各层，其中第1层是输入层，第I层是最后一层——输出层。用$z_j^i = \sum_k w_{jk}^i a_k^{i-1} + b_j^i$表示第$i$层第$j$个神

经元的权重输入，其值由第$i-1$层所有与第i层第j个神经元相连的神经元加权求和，再加上偏置b_j^i而得到。如果把第i层的所有神经元的权重输入记为向量$\boldsymbol{z}^i$，则有更简洁的向量形式的表示$\boldsymbol{z}^i=\boldsymbol{w}^i\boldsymbol{a}^{i-1}+\boldsymbol{b}^i$。用$\boldsymbol{a}^i=\sigma(\boldsymbol{z}^i)=\sigma(\boldsymbol{w}^i\boldsymbol{a}^{i-1}+\boldsymbol{b}^i)$表示第$i$层神经元的输出激活值，激活函数此处采用对数几率函数$\sigma(x)$。图 2.5.3 给出了一个简单全连接神经网络的示例，其中$I=3$。在图 2.5.3 中，比如，$z_2^2=w_{21}^2a_1^1+w_{22}^2a_2^1+b_2^2$，由于$a_1^1$和$a_2^1$分别就是$x_1$和$x_2$，因此$z_2^2=w_{21}^2x_1+w_{22}^2x_2+b_2^2$。注意，$b_2^2$在图中未画出。

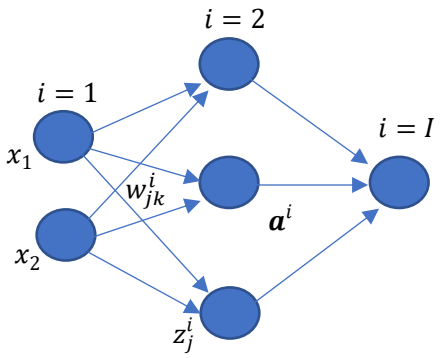

图 2.5.3 一个简单全连接神经网络的示例

由此，式（2.5.1）可重写为$L=\frac{1}{2N}\sum_{n=1}^{N}\|\boldsymbol{y}_n-\boldsymbol{a}^I\|^2$，将$L$分别对$\boldsymbol{w}$和$\boldsymbol{b}$求偏导，得到对应的梯度向量$\nabla_{\boldsymbol{w}}L(\boldsymbol{w},\boldsymbol{b})$和$\nabla_{\boldsymbol{b}}L(\boldsymbol{w},\boldsymbol{b})$：

$$\nabla_{\boldsymbol{w}}L(\boldsymbol{w},\boldsymbol{b})=\frac{1}{N}\sum_{n=1}^{N}(\boldsymbol{a}^I-\boldsymbol{y}_n)\nabla_{\boldsymbol{w}}\boldsymbol{a}^I \tag{2.5.2}$$

$$\nabla_{\boldsymbol{b}}L(\boldsymbol{w},\boldsymbol{b})=\frac{1}{N}\sum_{n=1}^{N}(\boldsymbol{a}^I-\boldsymbol{y}_n)\nabla_{\boldsymbol{b}}\boldsymbol{a}^I \tag{2.5.3}$$

可见，对每一个样本n，分别计算其梯度向量$(\boldsymbol{a}^I-\boldsymbol{y}_n)\nabla_{\boldsymbol{w}}\boldsymbol{a}^I$和$(\boldsymbol{a}^I-\boldsymbol{y}_n)\nabla_{\boldsymbol{b}}\boldsymbol{a}^I$，再累加起来即可。

对于一个特定的样本n，定义$\delta_j^i=\frac{\partial L}{\partial z_j^i}$表示第$i$层第$j$个神经元的误差，其中为了方便，用$L$表示$L_n=\frac{1}{2}\|\boldsymbol{y}_n-\boldsymbol{a}^I\|^2$。反向传播的过程就是为了计算这个误差$\delta_j^i$，然后基于误差就能得到梯度$\nabla_{\boldsymbol{w}}L$和$\nabla_{\boldsymbol{b}}L$。实际上，$\delta_j^i$就是损失函数$L$关于输入$z$的偏导，所以“误差反向传播”也称为“梯度反向传播”。

先来看第一个反向传播方程。由上面的定义有$\delta_j^I=\frac{\partial L}{\partial z_j^I}=\sum_k\frac{\partial L}{\partial a_k^I}\frac{\partial a_k^I}{\partial z_j^I}=\frac{\partial L}{\partial a_j^I}\frac{\partial a_j^I}{\partial z_j^I}=\frac{\partial L}{\partial a_j^I}\sigma'(z_j^I)$，这就得到了式（2.5.4）——输出层的误差。其中，第二步推导就是复合函数的链式求导。第三步推导是由于$\frac{\partial a_k^I}{\partial z_j^I}$当$k\neq j$时为 0，因为同层的不同神经元之间无连接（见图 2.5.3）。

$$\delta_j^I=\frac{\partial L}{\partial a_j^I}\sigma'(z_j^I)=(a_j^I-y_n^j)\sigma'(z_j^I) \tag{2.5.4}$$

式（2.5.4）的向量形式为：

$$\boldsymbol{\delta}^{\boldsymbol{I}}=(\boldsymbol{a}^{\boldsymbol{I}}-\boldsymbol{y}_{\boldsymbol{i}})\nabla_{\boldsymbol{z}}\sigma(\boldsymbol{z}^{\boldsymbol{I}}) \tag{2.5.5}$$

继续推导第二个反向传播方程。$\delta_j^i = \frac{\partial L}{\partial z_j^i} = \sum_k \frac{\partial L}{\partial z_k^{i+1}} \frac{\partial z_k^{i+1}}{\partial z_j^i} = \sum_k \delta_k^{i+1} \frac{\partial z_k^{i+1}}{\partial z_j^i}$，又$z_k^{i+1} = \sum_j w_{kj}^{i+1} a_j^i + b_k^{i+1} = \sum_j w_{kj}^{i+1} \sigma(z_j^i) + b_k^{i+1}$，$\frac{\partial z_k^{i+1}}{\partial z_j^i} = w_{kj}^{i+1} \sigma'(z_j^i)$，代入则得到：

$$\delta_j^i = \sum_k \delta_k^{i+1} w_{kj}^{i+1} \sigma'(z_j^i) \tag{2.5.6}$$

式（2.5.6）的向量形式为：

$$\boldsymbol{\delta}^i = [(\boldsymbol{w}^{i+1})^T \boldsymbol{\delta}^{i+1}] \nabla_{\boldsymbol{z}} \sigma(\boldsymbol{z}^i) \tag{2.5.7}$$

式（2.5.7）是第i层相对于第$i+1$层的误差，实际上就是反向传播第$i+1$层的误差，从而得到第i层的误差。由式（2.5.5）和（2.5.7）即可从最后一层I开始得到任一层i的误差：$\boldsymbol{\delta}^I \to \boldsymbol{\delta}^{I-1} \to \cdots \to \boldsymbol{\delta}^1$。类似，很容易推导出剩余的两个反向传播方程，即式（2.5.8）和（2.5.9）：

$$\frac{\partial L}{\partial b_j^i} = \delta_j^i \tag{2.5.8}$$

$$\frac{\partial L}{\partial w_{jk}^i} = a_k^{i-1} \delta_j^i \tag{2.5.9}$$

写成向量形式则分别为：

$$\nabla_{\boldsymbol{b}} L = \boldsymbol{\delta}^i \tag{2.5.10}$$

$$\nabla_{\boldsymbol{w}} L = (\boldsymbol{a}^{i-1})^T \boldsymbol{\delta}^i \tag{2.5.11}$$

这就得到了最终需要的梯度向量$\nabla_{\boldsymbol{w}} L$和$\nabla_{\boldsymbol{b}} L$。基于此，用梯度下降公式$\boldsymbol{w} = \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} L(\boldsymbol{w}, \boldsymbol{b})$和$\boldsymbol{b} = \boldsymbol{b} - \alpha \nabla_{\boldsymbol{b}} L(\boldsymbol{w}, \boldsymbol{b})$更新参数，就可完成神经网络的训练。

2.5.4 关于可解释性的讨论

1.3 节和 2.5.2 节都谈到了神经网络的“不可解释性”。这里，进一步以 2.4 节介绍的 SVM 来类比，因为 SVM 无论从理论支撑、模型设计思想还是模型求解来讲，都具有良好的“可解释性”。反观神经网络，则缺乏理论支撑（脑科学还处于发展的早期），端到端的学习范式导致模型内部是一个“黑盒子”，模型不能保证得到全局最优解等。

话说回来，SVM 真的就能彻底被解释吗？以其核函数的选取为例，线性核函数只能解决线性可分问题，这个能力感知机就已经具备，所以体现不出 SVM 有什么优势。具有非线性核函数的 SVM 才是其真正的优势所在，即只要正确选取核函数，SVM 就能以最优的方式解决任何非线性可分问题。然而，正如 2.4.7 节所谈到的，核函数如何选取是没有理论支撑的、是不可解释的。所以，必须认识到，理论支撑和可解释性是一个历史过程，不可能一蹴而就。对于“神经网络与深度学习”而言，我们理应同等对待，在接受其带来的好处的同时也必须同时接受其不可解释性。

计算机科学有一个经典命题：P 是否不等于 NP。大多数人相信“P 不等于 NP”，这意味着：不可解的问题都一样，永远也不可解。或许，“神经网络与深度学习”的不可解释，SVM 核函数的不可解释，都是一样的不可解问题？这个问题的解答只能留给将来。

2.5.5 全连接神经网络的核心代码实现

定义一个名为 Network 的类，该类包括构造函数、前向计算、反向传播、随机梯度下降、性能评估等方法。

（1）构造函数__init__()：

```
class Network:
    def __init__(self, sizes):
        self.sizes = sizes
        self.num_layers = len(sizes)
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
```

构造函数有一个额外参数：名为 sizes 的列表。该列表依次指定了每一层的神经元个数，比如[2, 3, 1]意味着一个三层网络，第一层有 2 个神经元，第二层有 3 个，而最后一层有一个。实例属性 sizes 和 num_layers 分别记录了神经网络的结构和层数。另外两个实例属性 weights 和 biases 分别对应权重$\boldsymbol{w}$和偏置$\boldsymbol{b}$。值得注意的是，weights 和 biases 都采用标准高斯分布进行了随机初始化，使用的是 Numpy 的 randn()函数。以 sizes=[2, 3, 1]为例，sizes[:-1]=[2, 3]，sizes[1:]=[3, 1]，则 weights 就是一个包含两个二维数组的列表：[3x2 数组，1x3 数组]，而 biases 则形为[3x1 数组，1x1 数组]。注意 zip()函数返回的是 zip 对象，此处的作用是组合两个参数对应位置的元素。为什么这样组合？如图 2.5.3 所示，第一层有 2 个神经元，第二层有 3 个，两层之间全连接就得到总共 3x2=6 个连接，对应 weights 的第一个元素：一个 3x2 的 Numpy 数组。类似，第二层和第三层之间的全连接对应 weights 的第二个元素：一个 1x3 的 Numpy 数组。至于 biases，第一层（输入层）并不需要；第二层有 3 个神经元，而 1 个神经元需要一个偏置，故共有 3 个偏置，对应 biases 的第一个元素：一个 3x1 的 Numpy 数组；第三层（输出层）类似，对应 biases 的第二个元素：一个 1x1 的 Numpy 数组。

（2）前向计算函数 feedforward()：

```
    def feedforward(self, a):
        """如果 a 是输入，则该函数返回网络的最终输出"""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a
```

这个函数有一个额外参数 a，如果 a 就是第一层（输入层）的输入，那么这个函数将返回最后一层（输出层）的输出，即网络最终的输出。还是以[2, 3, 1]这个网络为例，输入 a 为 2x1 的数组，w 和 b 分别取 weights 和 biases 的第一个元素，则 np.dot(w, a)完成 w 和 a 的矩阵乘，得到 3x1 的数组，再加上 b，结果还是 3x1 的数组。用这个结果更新 a 的值后，继续进行下一层的运算，就得到了最终的输出：1x1 的数组。注意，sigmoid()函数就是对数几率函数，定义在 2.3.6 节中，读者可以回顾下。

（3）随机梯度下降函数 SGD()：

```
    def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
        n = len(training_data)

        if test_data:
            n_test = len(test_data)

        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [training_data[k:k+mini_batch_size] for k in range(0, n,
mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
```

```
            if test_data:
                print("Epoch {} : {} / {}".format(j,self.evaluate(test_data),n_test));
            else:
                print("Epoch {} complete".format(j))
```

这个函数有五个额外参数，依次对应训练数据（元组(x, y)的列表）、训练轮数、小 batch 大小、学习率、测试数据。其中测试数据参数 test_data 指定了默认值 None。对于每一轮训练：首先用 shuffle()函数将训练数据 training_data 随机打乱；然后按照小 batch 大小 mini_batch_size 把 training_data 等分成 k 个小 batch；接着，就是对每一个小 batch 调用 update_mini_batch()函数完成梯度下降训练；最后，如果指定了测试数据，则调用 evaluate()函数完成模型的性能评估。

```
    def update_mini_batch(self, mini_batch, eta):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]
```

update_mini_batch()函数需要两个额外参数：小 batch 和学习率。该函数首先将梯度向量 nabla_w 和 nabla_b（分别对应$\nabla_{\boldsymbol{w}}L$和$\nabla_{\boldsymbol{b}}L$）初始化为 0。然后对小 batch 的每一个样本调用 backprop()函数计算其梯度，并将其累加到 nabla_w 和 nabla_b。最后，就是应用梯度下降公式$\boldsymbol{w}=\boldsymbol{w}-\frac{\alpha}{m}\nabla_{\boldsymbol{w}}L$和$\boldsymbol{b}=\boldsymbol{b}-\frac{\alpha}{m}\nabla_{\boldsymbol{b}}L$更新$\boldsymbol{w}$和$\boldsymbol{b}$。注意，这里的$m$表示小 batch 的大小，对应代码中的 len(mini_batch)。为什么要除以 m 呢？读者可以思考下（习题 2.5.6）。实际上，这与式（2.5.1）中的除以 N 是一样的作用。

（4）反向传播函数 backprop()：

```
    def backprop(self, x, y):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        #前馈计算
        activation = x
        activations = [x] #逐层存储所有层的输出
        zs = [] #逐层存储所有层的 z 向量
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        #反向传播
        delta = self.cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
```

```
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        for l in range(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)
```

这个函数以数据样本(x, y)为参数，返回该样本的梯度。同样是先将梯度向量 nabla_w 和 nabla_b（分别对应$\nabla_{\boldsymbol{w}}L$和$\nabla_{\boldsymbol{b}}L$）初始化为 0。然后开始前向计算过程，得到样本(x, y)的各层输出 activation，并将其保存到 activations 列表中。类似的，每一层的权重输入向量 z 保存到 zs 列表中。

接下来，开始反向传播过程。delta 是输出层的输出误差，通过式（2.5.3）进行计算。由式（2.5.8）和（2.5.9），就可分别得到最后一层关于$\boldsymbol{b}$和$\boldsymbol{w}$的梯度向量，并保存到 nabla_b[-1]和 nabla_w[-1]之中。得到了倒数第一层（输出层）的梯度向量，根据式（2.5.5）、式（2.5.8）和（2.5.9），接下来的 for 循环就计算出了倒数第二层（[-2]）一直到第二层（[2]）的梯度向量。

最后提个问题：计算输出层的输出误差 delta 时，用到了 cost_derivative()函数，请根据对应公式完成这个函数的实现。（习题 2.5.8）

（5）性能评估函数 evaluate()：

```
    def evaluate(self, test_data):
        test_results = [(np.argmax(self.feedforward(x)), y) for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)
```

该函数需要给定参数 test_data。所完成的功能就是统计 test_data 的所有数据样本中被网络正确分类的样本有多少个。注意一下 np.argmax(self.feedforward(x))这个语句，由前面的介绍，feedforward()函数计算出网络的最终输出，因此输出层有多少个神经元（对应类别数C）就有多少个输出，即构成一个C维的向量。函数 argmax 得到这个C维向量取最大值（对应最大的激活值）的那个神经元的索引值，这个索引值就是网络对样本 x 的类别预测值。

2.5.6 应用到 Mnist 手写数字识别

Mnist 手写数字识别数据集[1]是机器学习领域的经典数据集，本节将应用 2.5.5 节的代码来解决这个数据集所提出的手写数字识别问题。

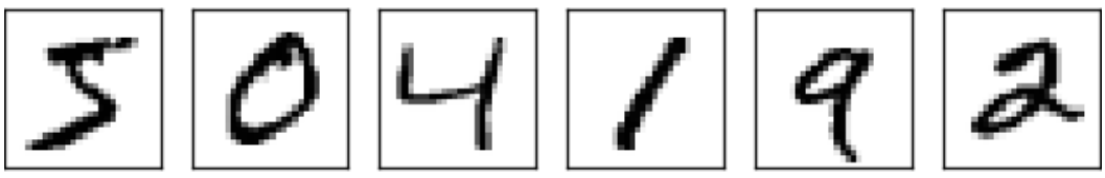

图 2.5.4 Mnist 数据集的 6 个数据样本

如图 2.5.4 所示，Mnist 数据集的每个数据样本（比如最左边的“5”）是一幅 28x28 像素的 8 位灰度图像。解释一下，所谓“28x28 像素”表示图像由 28 行 28 列总共 784 个像素构成，即图像的分辨率是 28x28。而“8 位灰度图像”则意味着图像的每个像素用一个字节来表示，其取值范围就为 0~255。假如值 0 表示黑色，而值 255 表示白色，那么介于其间的值就表示介于黑色和白色之间的“灰色”，所以称为“8 位灰度图像”。

该数据集包含 6 万个样本的训练集和 1 万个样本的测试集。其中，训练集来自于 250 个不同的人（一半是美国人口统计局员工，一般是美国中学生）；而测试集则来自于另外 250 个不同的人（仍然来自于美国人口统计局和中学生）。

（1）把名为 mnist.pkl.gz 的 Mnist 数据文件 [2] 下载下来后，就可以用 load_data()函数装入 Mnist：

```
def load_data():
    f = gzip.open('mnist.pkl.gz', 'rb')
    training_data, validation_data, test_data = pickle.load(f, encoding="latin1")
    f.close()
    return (training_data, validation_data, test_data)
```

这个函数无需参数，用到了 gzip 和 pickle 两个标准库。从文件读入的数据被分成 3 个子集，分别对应训练集、验证集和测试集。在解释器里执行这个函数如下：

```
>>> tr_d, va_d, te_d = load_data()
>>> tr_d
(array([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]], dtype=float32), array([5, 0, 4, ..., 8, 4, 8], dtype=int64))
>>> va_d
(array([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]], dtype=float32), array([3, 8, 6, ..., 5, 6, 8], dtype=int64))
>>> te_d
(array([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]], dtype=float32), array([7, 2, 1, ..., 4, 5, 6], dtype=int64))
```

可见，三个子集的格式是一样的：是一个两个元素的元组，其中第一个元素是表示图像数据本身的二维数组，第二个元素是对应的类别标签一维数组。用 np.shape()函数可以方便的查看每个元素的形状：

```
>>> np.shape(tr_d[0])
(50000, 784)
>>> np.shape(tr_d[1])
(50000,)
```

注 1：**http://yann.lecun.com/exdb/mnist/**

2：https://github.com/mnielsen/neural-networks-and-deep-learning/blob/master/data/mnist.pkl.gz

```
>>> np.shape(va_d[0])
(10000, 784)
>>> np.shape(va_d[1])
(10000,)
>>> np.shape(te_d[0])
(10000, 784)
>>> np.shape(te_d[1])
(10000,)
```

可见，训练集 tr_d 有 5 万个样本，每个样本是 784 维的向量（28x28=784）。验证集 va_d 和测试集 te_d 各有 1 万个样本。查看 tr_d 中第一个样本的数据：

```
>>> tr_d[0][0]
array([0.        , 0.        , 0.        ,
0.        , 0.        , ... ... ,
0.        , 0.        , 0.01171875, 0.0703125 , 0.0703125 ,
0.0703125 , 0.4921875 , 0.53125   , 0.68359375, 0.1015625 , 0.6484375 ,
0.99609375, 0.96484375, 0.49609375, 0.        , 0.        ,
0.        , 0.        , 0.        ,
0.        , ......, 0.        , 0.        ,
0.        , 0.        ], dtype=float32)
```

可见，每个像素的 8 位灰度(0~255)已经被归一化到[0,1]，比如 0.0703125 对应灰度级 18。

（2）为了方便数据的使用，写了一个包装函数 load_data_wrapper()：

```
def load_data_wrapper():
    tr_d, va_d, te_d = load_data()
    training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
    training_results = [vectorized_result(y) for y in tr_d[1]]
    training_data = zip(training_inputs, training_results)
    validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
    validation_data = zip(validation_inputs, va_d[1])
    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
    test_data = zip(test_inputs, te_d[1])
    return (training_data, validation_data, test_data)
```

这个函数无需参数，首先调用 load_data()函数装入数据。对于训练集 tr_d，语句[np.reshape(x, (784, 1)) for x in tr_d[0]]会把每个 784 维的数据向量变换为 784x1 的二维数组，这样 training_inputs 就成为了一个含有 5 万个元素的列表，其中每个元素是一个 784x1 的数据数组。语句[vectorized_result(y) for y in tr_d[1]]则会把每个样本的标签进行向量化，得到一个C维的标签向量（见 2.5.3 节）。然后，zip()函数将数据与其标签对应，返回对象给 training_data。对于验证集 va_d 和测试集 te_d，数据的处理是类似的，分别得到 validation_inputs 和 test_inputs。不同在于标签的处理，无需对标签进行向量化，直接用 zip()函数与数据对应即可。

（3）调用以上实现的代码，完成 Mnist 数据集的训练和测试：

```
training_data, validation_data, test_data = load_data_wrapper()
training_data = list(training_data) #convert zipped object to list
test_data = list(test_data)
net = Network([784, 30, 10])
```

```
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

以上代码的一次运行结果如下：

```
Epoch 0 : 9106 / 10000
Epoch 1 : 9285 / 10000
Epoch 2 : 9330 / 10000
Epoch 3 : 9378 / 10000
Epoch 4 : 9360 / 10000
Epoch 5 : 9411 / 10000
Epoch 6 : 9442 / 10000
Epoch 7 : 9395 / 10000
Epoch 8 : 9435 / 10000
Epoch 9 : 9451 / 10000
Epoch 10 : 9479 / 10000
Epoch 11 : 9474 / 10000
Epoch 12 : 9482 / 10000
Epoch 13 : 9459 / 10000
Epoch 14 : 9487 / 10000
Epoch 15 : 9506 / 10000
Epoch 16 : 9489 / 10000
Epoch 17 : 9513 / 10000
Epoch 18 : 9522 / 10000
Epoch 19 : 9512 / 10000
Epoch 20 : 9517 / 10000
Epoch 21 : 9510 / 10000
Epoch 22 : 9516 / 10000
Epoch 23 : 9514 / 10000
Epoch 24 : 9514 / 10000
Epoch 25 : 9533 / 10000
Epoch 26 : 9488 / 10000
Epoch 27 : 9519 / 10000
Epoch 28 : 9495 / 10000
Epoch 29 : 9533 / 10000
```

可以看到，随着训练的推进，网络的精度呈现局部随机波动、整体稳定提升的趋势，符合预期。实际上，正如 2.3.3 节里讨论的，神经网络的优化是一个典型的非凸优化问题，训练的轮数、小 batch 的大小、学习率这些超参数都需要根据经验来设定和调整。读者可以尝试去调整这些超参数，甚至修改网络结构，以获得更多感性的经验，这是有益的也是必须的。

本节虽然力求完整、系统地探讨全连接神经网络的原理、设计、实现及应用，但限于篇幅，神经网络很多重要的方面都尚未涉及，比如样本增扩、交叉熵损失函数、过拟合与正则化、梯度不稳定问题、ReLU 激活函数、面向图像数据的卷积神经网络、残差网络、Transformer 等。希望读者可以在实验部分以及今后的学习实践中进一步的探索。

小故事：信念的力量

他从未正式上过计算机课程，本科在剑桥大学读的是生理学和物理学，期间曾转向哲学，但最终拿到的却是心理学方向的学士学位；他曾因为一度厌学去做木匠，但遇挫后还是回到爱丁堡大学，并拿到「冷门专业」人工智能方向的博士学位；数学不好让他在做研究时倍感绝望，当了教授之后，对于不懂的神经科学和计算科学知识，他也总要请教自己手下的研究生。

学术道路看似踉踉跄跄，但 Geoffrey Hinton 却成了笑到最后的那个人，他被誉为「深度学习教父」，并且获得了计算机领域的最高荣誉「图灵奖」。

1973 年，在英国爱丁堡大学，他师从 Langer Higgins 攻读人工智能博士学位，但那时几乎没人相信神经网络，导师也劝他放弃研究这项技术。周遭的质疑并不足以动摇他对神经网络的坚定信念，在随后的十年，他接连提出了反向传播算法、玻尔兹曼机，不过他还要再等数十年才会等到深度学习迎来大爆发，到时他的这些研究将广为人知。

经过近半个世纪的技术坚守和生活磨砺，终于，2012 年曙光乍现，他与学生 Alex Krizhevsky、Ilya Sutskever 提出的 AlexNet 震动业界，就此重塑了计算机视觉领域，启动了新一轮深度学习（强大算力、海量数据以及随机梯度下降）的黄金时代。

也是在 2012 年底，他与这两位学生成立了三人组公司 DNN-research，并将其以 4400 万美元的「天价」卖给了 Google，他也从学者身份转变为 Google 副总裁、Engineering Fellow。2019 年，非计算机科班出身的 AI 教授 Hinton，与 Yoshua Bengio、Yann LeCun 共同获得了图灵奖。

饱经风霜之后，这位已经 74 岁的「深度学习教父」依然奋战在 AI 研究一线，他不惮于其他学者发出的质疑，也会坦然承认那些没有实现的判断和预言。不管怎样，他仍然相信，在深度学习崛起十年之后，这一技术会继续释放它的能量，而他也在思索和寻找下一个突破点。

Hinton 说：“经常有人说深度学习遇到了瓶颈，但事实上它一直在不断向前发展，我希望怀疑论者能将深度学习现在不能做的事写下来。五年后，我们会证明深度学习能做到这些事。”

启迪：

（1）人生的意义在于“坚持信念，追寻理想”

（2）逆境中要保持坚定，顺境中要保持清醒

（3）一项革命性科技的成功需要多方面的因素共同促成
（4）行胜于言

习题 2.5

2.5.1 在 2.5.1 节谈到了“全连接”这个概念，那么你认为“非全连接神经网络”应该是什么样的呢？这类网络可能有什么潜在优势呢？
2.5.2 试计算图 2.5.2 所示神经网络有多少个参数。
2.5.2’ 式（2.5.1）中的$\boldsymbol{a_i}$取值范围是多少呢？为什么？是否可以据此解释为何每个样本的分类损失一般不是 0 或 2 呢？
2.5.3 试证明式（2.5.8）和（2.5.9）。
2.5.4 标出图 2.5.3 中神经元及其连接的符号表示，并给出z_3^2和z_1^3的符号表示。
2.5.5 请解释式（2.5.6）中的$\frac{\partial z_k^{i+1}}{\partial z_j^i} = w_{kj}^{i+1}\sigma'(z_j^i)$这一步推导是如何得到的。
2.5.6 梯度下降公式$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{m}\nabla_{\boldsymbol{w}}L$和$\boldsymbol{b} = \boldsymbol{b} - \frac{\alpha}{m}\nabla_{\boldsymbol{b}}L$里，为什么要除以$m$呢？你认为这样做的目的是什么？
2.5.7 式（2.5.7）和（2.5.11）是对应分量形式的向量表达，那么为什么向量形式的表达需要转置操作呢？请从矩阵运算的角度进行分析。
2.5.8 backprop()函数里，计算输出层的输出误差 delta 时，用到了 cost_derivative()函数，请根据对应公式完成这个函数的实现。
2.5.9 根据 load_data_wrapper()函数的要求，实现 vectorized_result()函数。
2.5.10 load_data_wrapper()函数里，对于验证集 va_d 和测试集 te_d，请思考为何无需对标签进行向量化呢？
2.5.11 本节对 Mnist 进行多分类的 NN 模型里是采用的 Softmax 吗？请给出详细的原理和代码分析。