第 5 章 激活函数

在神经网络中，隐藏层和输出层的节点都是其上一层节点的加权和代入激活函数得到的函数值。例如，图 5-1 的神经网络中，s_1是1，x_1，x_2的加权和，s_1代入 sigmoid 函数得到h_1；同样的，s_2是1，x_1，x_2的加权和，s_2代入 sigmoid 函数得到h_2。得到隐藏层h_1，h_2之后，s是1，h_1，h_2 的加权和，s代入 sigmoid 函数得到输出值。在神经网络中，sigmoid 函数被称为激活函数。

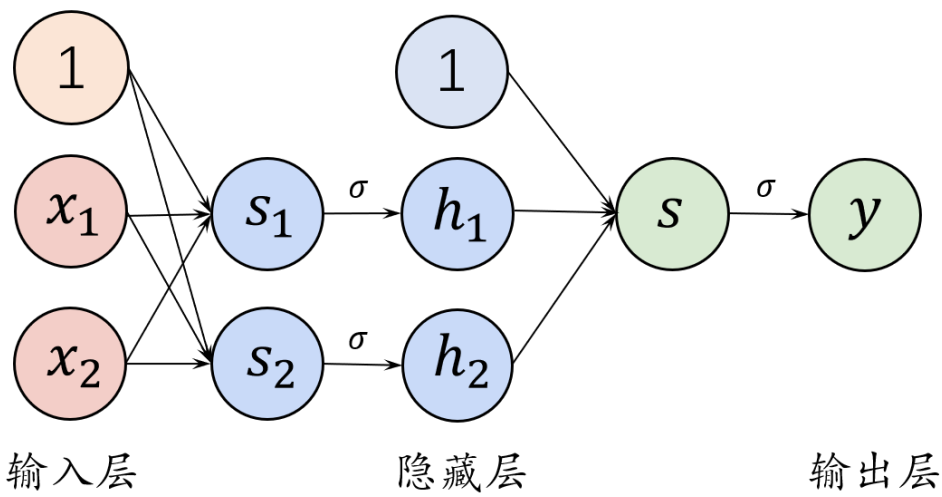

图 5-1 激活函数为 sigmoid 函数的神经网络

在实际应用中，激活函数不仅仅有 sigmoid 函数，还有很多其他函数可以选择。本章将介绍两个应用于输出层的激活函数：

- sigmoid 函数
- softmax 函数

以及 4 个常见的应用于隐藏层的激活函数：

- sigmoid 函数
- tanh 函数
- ReLU 函数
- Leaky ReLU 函数

5.1 激活函数的基本要求

如果没有激活函数，神经网络会变成什么呢？在图 5-1 的神经网络中，去除激活函数得到如图 5-2 所示的神经网络。

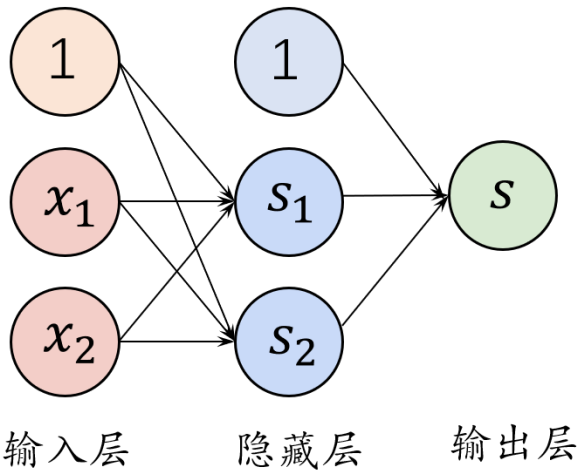

图 5-2 没有激活函数的神经网络

假设在图 5-2 所示的神经网络中，

（1）$s_1 = 0.5 + x_1 + x_2$
（2）$s_2 = 1 + 2x_1 + 3x_2$
（3）$s = 1.5 + 0.5s_1 + 0.6s_2$

将式子（1）和式子（2）代入到式子（3）中得到，

$$\begin{aligned} s &= 1.5 + 0.5 \times (0.5 + x_1 + x_2) + 0.6 \times (1 + 2x_1 + 3x_2) \\ &= 2.35 + 1.7x_1 + 2.3x_2 \end{aligned}$$

可以看到s依然是x_1和x_2的一个线性函数。因此，如果没有激活函数，无论神经网络的结构多复杂，它都将退化为一个线性模型；在这种情况下，面对非线性的回归或者分类问题（例如，第 4 章的数据 2），神经网络模型将不会比线性模型表现得更好。在第 4 章中，当使用 sigmoid 函数作为激活函数时，建立的 3 层神经网络可以很好地完成数据 2 的分类任务。数据 2 是一个相对简单的任务，只有 4 个观测点。现实的回归或者分类问题的决策边界通常都是复杂且非线性的，因此，要求模型具有产生复杂的非线性决策边界的能力。在这点上，激活函数在神经网络中扮演了非常重要的角色。通常，我们让隐藏层的每一个节点值都通过激活函数的变换，使得输出层是输入层的一个非线性函数。当神经网络有很多隐藏层，且每个隐藏层有很多节点时，加入了激活函数的神经网络可以得到非常复杂的非线性函数，从而提高神经网络解决实际问题的能力。

直观理解，可以认为激活函数在神经网络中的作用是对加权和进行变换。因此，理论上只要一个函数可以把一个数字变换成另一个数字，该函数便可以成为一个激活函数。我们可以找到成千上万的函数充当激活函数。而在实际中，并不是任意一个函数充当激活函数后神经网络都可以表现得很好。一个函数成为激活函数需要满足以下 4 个基本条件：

- 连续函数，且定义域是$(-\infty,\infty)$
- 单调函数
- 非线性函数
- 可导函数，且函数和它的导数都容易计算

条件 1：激活函数是连续函数，且定义域是$(-\infty,\infty)$

激活函数的输入值为上一层节点的加权和。而加权和没有数值上的界限，因此，为了使每一个可能的加权和都能经过激活函数的变换，要求激活函数是连续函数，且定义域为$(-\infty,\infty)$。例如，在图 5-3 左图中，函数只在 4 个区间中有定义，而在其他区间中没有定义。当上一层节点的加权和为0时，把0代入图 5-3 左图所示的函数中，将得不到任何结果。因此，如果在神经网络中使用图 5-3 左图的函数为激活函数，将不能顺利地利用正向传播算法得到最终的预测值。对于图 5-3 右图的函数，神经网络中每一层节点的加权和都可以找到对应的函数值。

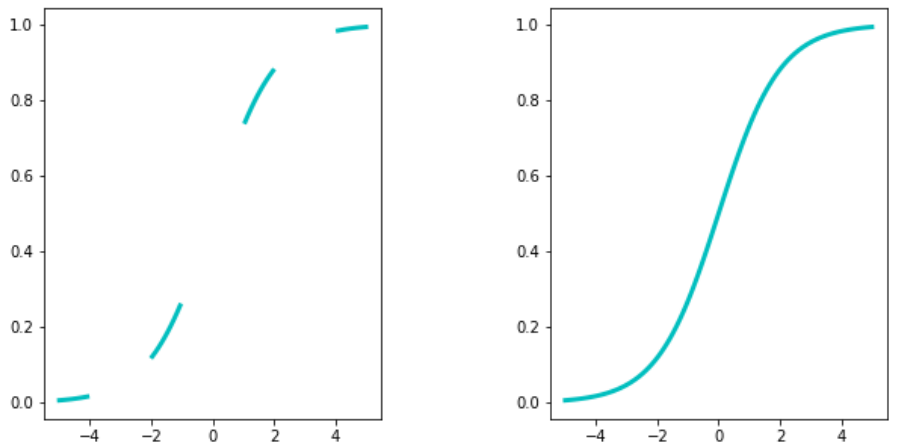

图 5-3 激活函数的连续性。左图：非连续函数；右图：连续函数

条件 2：激活函数是单调函数

激活函数的第二个条件是单调性。假设激活函数是图 5-4 左图的函数$y=x^2$，两个不同的x可以得到相同的y。在第 4 章中，我们使用反向传播算法计算梯度，然后用梯度下降法更新权重，使得损失函数逐渐变小。梯度下降法通过更新权重，

进而更新所有节点（除了输入层节点）的值，使得损失函数最小化。如果两个不同的值可以得到同一个结果，那么梯度下降法更新梯度时可以有多个不同更新参数的方向。乐观的说，这可能是好事，因为可以有多种方式使得损失函数下降；但是，悲观的说，可能更难找到一个正确方向更新参数，或者有可能在不同参数更新方向上摇摆，使得梯度下降法更难找到使得损失函数更小的路径。当然，该条件不是一个完全硬性的条件（条件 1 的函数连续性是一个硬性条件，每个激活函数都必须满足）。但是，我们还是应该尽量避免使用非单调的激活函数。

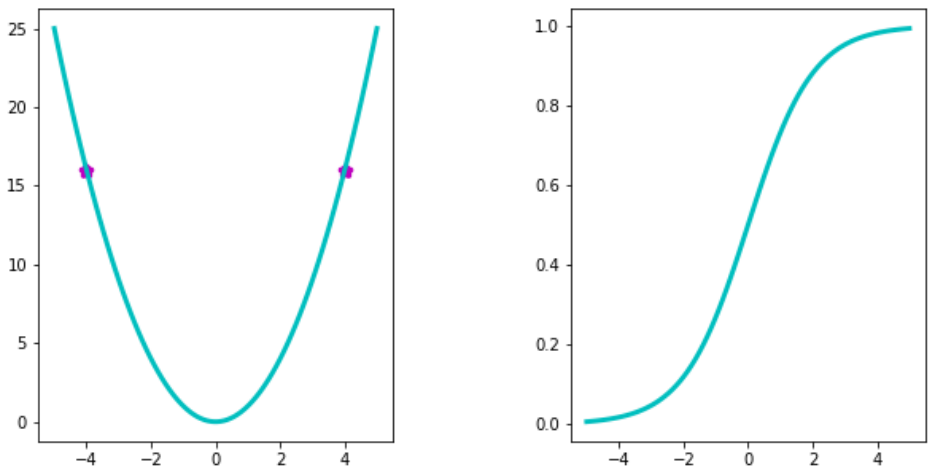

图 5-4 激活函数的单调性。左图：非单调函数；右图：单调函数

条件 3：激活函数是非线性函数

激活函数不能是线性函数。假设激活函数为图 5-5 左图的函数$h = 2s + 3$，加权和$s = 0.5 + x_1 + 2x_2$，把s代入激活函数中得到$h = 2 \times (0.5 + x_1 + 2x_2) + 3 = 4 + 2x_1 + 4x_2$。该线性激活函数的作用仅仅是把加权和从$0.5 + x_1 + 2x_2$变为$4 + 2x_1 + 4x_2$。因此，激活函数是线性函数的神经网络模型还是线性模型，不能让模型变得更加复杂。这要求激活函数必须是一个非线性函数。

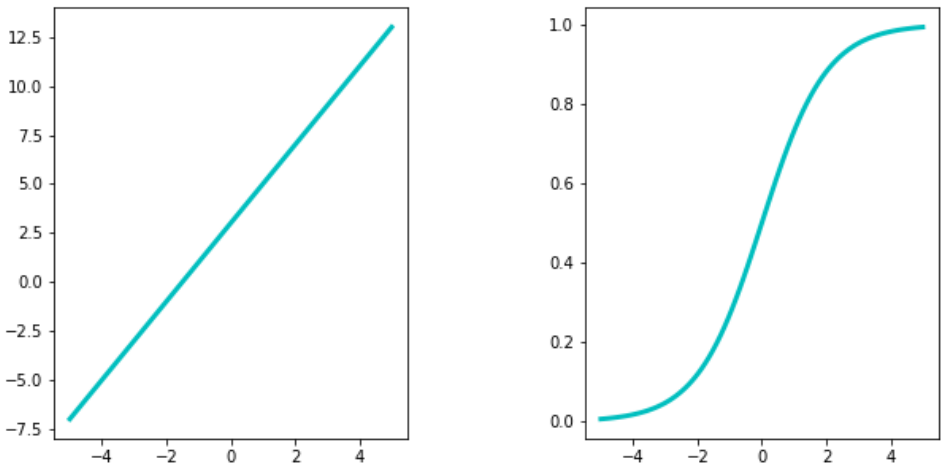

图 5-5 激活函数是非线性函数。左图：线性函数；右图：非线性函数

条件 4：激活函数是可导函数，且激活函数和它的导数都容易计算

在神经网络中需要使用反向传播算法计算参数梯度，然后用梯度下降法更新参数。反向传播算法的主要思想是通过链式法则逐步计算参数梯度，其过程需要计算激活函数的导数。而且在实际应用中，激活函数会应用于所有隐藏层和输出层节点，当神经网络的隐藏层很多，或者隐藏层的节点个数很多时，激活函数及其导数被计算的次数将会很多。因此，为了降低神经网络训练过程的计算复杂度，要求激活函数及其导数的计算复杂度较低。

5.2 输出层激活函数

总的来说，因变量的常见数据类型有 3 种：定量数据、二分类定性数据和多分类定性数据。输出层激活函数的选择主要取决于因变量数据类型。

5.2.1 因变量为定量数据

当因变量为定量数据时，它的取值范围通常为$(-\infty,\infty)$。因变量的预测值也需要在区间$(-\infty,\infty)$内。这种情况可以不使用激活函数，即因变量的预测值为最后一层隐藏层的加权和。例如，在图 5-6 中，因变量的预测值s等于1，h_1，h_2的加权和。记第i个观测点输出层的加权和为s_i，那么第i个观测点的预测值为$\hat{y}_i = s_i$。损失函数可以定义为均方误差，

$$L = \text{ave}_{i\in\mathcal{D}} \frac{1}{2}(y_i - \hat{y}_i)^2$$

其中，ave 表示求平均值，$\mathcal{D}$为观测点集合（如果使用随机梯度下降法，那么$\mathcal{D}$只包含一个观测点；如果使用全数据梯度下降法，那么$\mathcal{D}$包含训练数据的全部观测点；如果使用批量随机梯度下降法，那么$\mathcal{D}$包含训练数据中部分观测点）。

从第 4 章我们知道，在反向传播算法中计算权重梯度时，需要计算输出层的δ（即损失函数关于输出层加权和的偏导数）。在这个情况下，可以通过如下方式计算损失函数关于第i个观测点的输出层加权和s_i的偏导数，

$$\begin{aligned}\delta_i = \frac{\partial L}{\partial s_i} &= \frac{\partial L}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial s_i} \\ &= -(y_i - \hat{y}_i) \times 1 \\ &= \hat{y}_i - y_i\end{aligned}$$

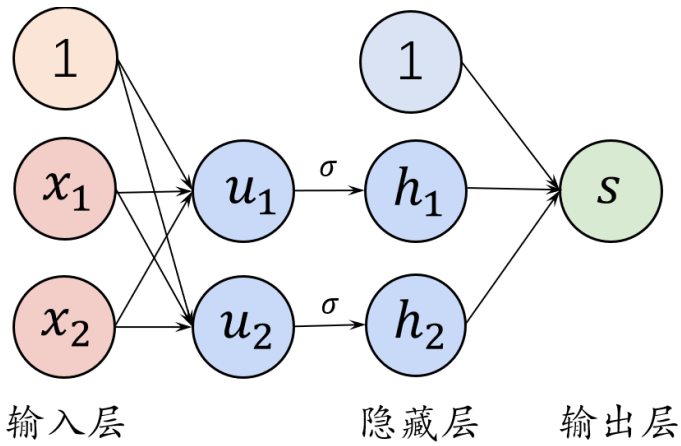

图 5-6 因变量为定量数据的神经网络

5.2.2 因变量为二分类定性数据

当因变量为二分类定性数据时，它可以用0和1表示。这时神经网络的输出层只需要一个节点。通过正向传播算法，第i个观测点的输出值为最后一层隐藏层的加权和代入 sigmoid 函数得到的函数值，表示神经网络预测第i个观测点为1的概率。如图 5-7 所示，s为1，h_1，h_2的加权和，p为s代入 sigmoid 函数的函数值。记第i个观测点输出层的加权和为s_i，那么第i个观测点的预测概率定义为$p_i = \frac{1}{1+\mathrm{e}^{-s_i}}$。这时可以定义损失函数$L$为

$$L = \mathrm{ave}_{i\in\mathcal{D}} - \{1_{y_i=1}\log(p_i) + 1_{y_i=0}\log(1-p_i)\}$$

这里，$1_{y_i=1}$是示性函数，当y_i等于1时，$1_{y_i=1}=1$；当y_i不等于1时，$1_{y_i=1}=0$。同样的，$1_{y_i=0}$也是示性函数，当y_i等于0时，$1_{y_i=0}=1$；当y_i不等于0时，$1_{y_i=0}=0$。

在这种情况下，可以通过如下方式计算损失函数关于第i个观测点的输出层加权和s_i的偏导数，

$$\begin{aligned}\delta_i = \frac{\partial L}{\partial s_i} &= \frac{\partial L}{\partial p_i}\frac{\partial p_i}{\partial s_i} \\ &= \left[-1_{y_i=1}\frac{1}{p_i} + 1_{y_i=0}\frac{1}{1-p_i}\right]\frac{\partial p_i}{\partial s_i} \\ &= \left[-1_{y_i=1}(1+\mathrm{e}^{-s_i}) + 1_{y_i=0}\frac{1+\mathrm{e}^{-s_i}}{\mathrm{e}^{-s_i}}\right]\frac{\mathrm{e}^{-s_i}}{(1+\mathrm{e}^{-s_i})^2} \\ &= p_i - y_i\end{aligned}$$

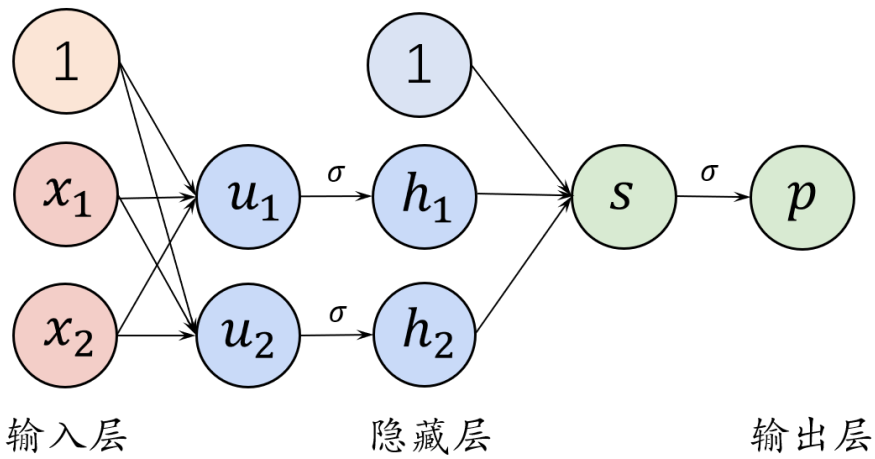

图 5-7 因变量为二分类定性数据的神经网络

5.2.3 因变量为多分类定性数据

当因变量为多分类定性数据时，它可以用 one-hot 编码表示。假设因变量的类的数量为K，第i个观测点的因变量属于第k类，那么第i个观测点的因变量y_i的 one-hot 编码为$\boldsymbol{y}_i = (0 \cdots 0\ 1\ 0 \cdots 0)$，即$\boldsymbol{y}_i$的第$k$个元素为1，其他元素都为0。假设$K = 3$且因变量取值为1，2或者3，当第$i$个观测点因变量为 1 时，$\boldsymbol{y}_i$的 one-hot 编码为$\boldsymbol{y}_i = (1\ 0\ 0)$；当第$i$个观测点因变量为 2 时，$\boldsymbol{y}_i$的 one-hot 编码为$\boldsymbol{y}_i = (0\ 1\ 0)$；当第$i$个观测点因变量为 3 时，$\boldsymbol{y}_i$的 one-hot 编码为$\boldsymbol{y}_i = (0\ 0\ 1)$。

当因变量为多分类定性数据时，神经网络模型输出层的节点个数将设为因变量的类的数量。在图 5-8 中，输出层的s_1，s_2，s_3分别为 1，h_1，h_2的加权和。p_1定义为$\frac{\mathrm{e}^{s_1}}{\sum_{i=1}^{3}\mathrm{e}^{s_i}}$；$p_2$定义为$\frac{\mathrm{e}^{s_2}}{\sum_{i=1}^{3}\mathrm{e}^{s_i}}$；$p_3$定义为$\frac{\mathrm{e}^{s_3}}{\sum_{i=1}^{3}\mathrm{e}^{s_i}}$。$p_1$，$p_2$，$p_3$分别表示预测为第 1 类，第 2 类，第 3 类的概率，而且$p_1 + p_2 + p_3 = 1$。

对于一般情况，当因变量的类的数量为K时，模型预测为第k类的概率定义为

$$p_k = \frac{e^{s_k}}{\sum_{j=1}^{K} e^{s_j}}$$

该函数称为 softmax 函数。

假设在图 5-8 的神经网络中输出层$(s_1 \quad s_2 \quad s_3) = (2 \quad 1 \quad 0)$，那么可以通过如下方式计算 softmax 函数值，

$$(2 \quad 1 \quad 0) \rightarrow (e^2 \quad e^1 \quad e^0) = (7.39 \quad 2.72 \quad 1.00) \rightarrow (0.665 \quad 0.245 \quad 0.090)$$

也就是说，神经网络预测第 1 类发生的概率为66.5%，第 2 类发生的概率为24.5%，第 3 类发生的概率为9%。在这里，第 1 类发生的概率最大，因此最终预测结果为第 1 类。

对于一般的K类的情况，我们通过如下方式得到神经网络的最终预测值，

$$\hat{y} = \arg\max_k p_k$$

即找出概率值最大节点对应的类。可以看到，softmax 函数是 sigmoid 函数的推广，但是 softmax 函数可以更自然地处理多分类问题。softmax 函数在神经网络中作为输出层的激活函数有如下优点。

- 加权和的结果在$(-\infty, \infty)$中，代入 softmax 函数之后，最终输出层的值全部变为正，且在 0 和 1 之间。
- 通过 softmax 函数的运算，输出层的所有结果的和为 1，因此可以认为输出层节点的值为概率值。

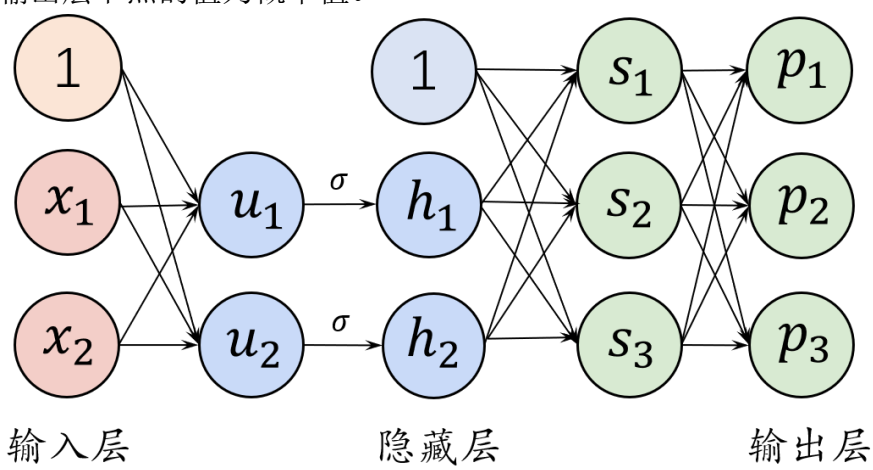

图 5-8 因变量为多分类定性数据的神经网络

在这个情况下，可以定义损失函数L为

$$L = \text{ave}_{i\in\mathcal{D}} - \sum_{j=1}^{K} y_{ij}\log(p_{ij}) = \text{ave}_{i\in\mathcal{D}} - \log(p_{ik})$$

这里，y_{ij}为第i个观测点的因变量$\boldsymbol{y}_i$的第j个元素（其中，$y_{ik}=1$），p_{ij}为第i个观测点预测为第j类的概率。该损失函数称为交叉熵（Cross Entropy）。直观上，当y_{ik}等于1时，我们希望第i个观测值最终被预测为第k类，因此希望概率p_{ik}可以很大；也就是说，希望$-\log(p_{ik})$很小。对于 softmax 函数，因为所有类的概率值加起来等于 1，如果p_{ik}很大，也就意味着其他概率很小。因此，最小化$\text{ave}_{i\in\mathcal{D}} - \log(p_{ik})$也意味着最大化$p_{ik}$，最小化其他类的概率值。

假设第i个观测点的因变量y_i（y_i为 one-hot 编码，且第k个元素为 1），该观测点的损失函数为

$$\begin{aligned} L_i &= -\log(p_{ik}) \\ &= -\log\frac{\mathrm{e}^{s_{ik}}}{\sum_{j=1}^{K}\mathrm{e}^{s_{ij}}} \\ &= -s_{ik} + \log(\sum_{j=1}^{K}\mathrm{e}^{s_{ij}}) \end{aligned}$$

记L_i关于$\boldsymbol{s}_i = (s_{i1} \quad s_{i2} \quad \cdots \quad s_{iK})$的导数为$\frac{\partial L_i}{\partial \boldsymbol{s}_i}$，$\frac{\partial L_i}{\partial \boldsymbol{s}_i}$的维度为$1\times K$。可以把$\frac{\partial L_i}{\partial \boldsymbol{s}_i}$的元素分成两种情况。

- $\frac{\partial L_i}{\partial \boldsymbol{s}_i}$的第$k$个元素（假设第$i$个观测点$y_{ik}=1$），$\frac{\partial L_i}{\partial s_{ik}}$，

$$\begin{aligned} \frac{\partial L_i}{\partial s_{ik}} &= \frac{\partial(-s_{ik} + \log(\sum_{j=1}^{K}\mathrm{e}^{s_{ij}}))}{\partial s_{ik}} \\ &= -1 + \frac{\mathrm{e}^{s_{ik}}}{\sum_{j=1}^{K}\mathrm{e}^{s_{ij}}} \\ &= p_{ik} - 1 \end{aligned}$$

- $\frac{\partial L_i}{\partial \boldsymbol{s}_i}$中第$k$个之外的元素，$\frac{\partial L_i}{\partial s_{ij}}, j\neq k$，

$$\begin{aligned} \frac{\partial L_i}{\partial s_{ij}} &= \frac{\partial(-s_{ik} + \log(\sum_{j=1}^{K}\mathrm{e}^{s_{ij}}))}{\partial s_{ij}} \\ &= -0 + \frac{\mathrm{e}^{s_{ij}}}{\sum_{j=1}^{K}\mathrm{e}^{s_{ij}}} \\ &= p_{ij} - 0 \end{aligned}$$

综合上面两种情况可以得到，$\frac{\partial L_i}{\partial \boldsymbol{s}_i} = \boldsymbol{p}_i - \boldsymbol{y}_i$。因此，损失函数关于第$i$个观测点的输出层加权和$\mathrm{s}_i$的偏导数可以写成如下形式，

$$\begin{aligned} \boldsymbol{\delta}_i &= \frac{\partial L_i}{\partial \boldsymbol{s}_i} \\ &= \boldsymbol{p}_i - \boldsymbol{y}_i \end{aligned}$$

在 Python 中，可以如下定义函数 `softmax()`。

```
# ***** 定义函数 softmax()
def softmax(x):
    temp = np.exp(x)
    return temp / np.sum(temp, axis = 1, keepdims=True)
```

5.2.4 mnist 手写数字

mnist 手写识别数据是机器学习文献中常用的数据。mnist 数据把手写数字图片，包含0，1，2，3，4，5，6，7，8，9，用28×28的像素点表示，每个像素点的灰度值是一个八位字节数字（$0 \sim 255$）。例如，图 5-9 是数字 8 的手写图片。观测点的自变量为手写数字图片的每一行灰度值链接在一起的向量，长度为$784 = 28 \times 28$。因变量为人工标注的该手写数字图像代表的数字，即 0 到 9 的整数。因变量用 one-hot 编码表示。例如，数字 8 的 one-hot 编码为(0 0 0 0 0 0 0 0 1 0)；数字 2 的 one-hot 编码为(0 0 1 0 0 0 0 0 0 0)。

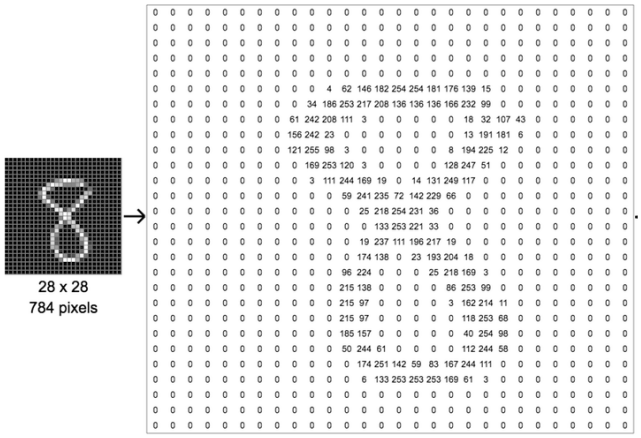

图 5-9 mnist 数据中的一幅图（数字 8）

包 idx2numpy 的函数 `convert_from_file()` 可用于载入 mnist 数据。训练数据由 6 万幅手写数字图片以及对应的数字组成，保存在 `x_train`，`y_train` 中。测试数据由 1 万幅手写数字图片及其对应的数字组成，保存在 `x_test`，`y_test` 中。

```
"""
```

```
载入需要用到的包和 mnist 数据
"""
%config InlineBackend.figure_format = 'retina'
import idx2numpy
import matplotlib.pyplot as plt
import numpy as np

x_train = idx2numpy.convert_from_file(   \
          './data/mnist/train-images.idx3-ubyte')
y_train = idx2numpy.convert_from_file(   \
          './data/mnist/train-labels.idx1-ubyte')
x_test = idx2numpy.convert_from_file(  \
          './data/mnist/t10k-images.idx3-ubyte')
y_test = idx2numpy.convert_from_file(   \
          './data/mnist/t10k-labels.idx1-ubyte')

print("训练数据的自变量维度:"+str(x_train.shape))
print("训练数据的因变量维度:"+str(y_train.shape))
print("测试数据的自变量维度:"+str(x_test.shape))
print("测试数据的因变量维度:"+str(y_test.shape))
print("训练数据因变量前 10 个数字："+str(y_train[0:10]))

训练数据的自变量维度:(60000, 28, 28)
训练数据的因变量维度:(60000,)
测试数据的自变量维度:(10000, 28, 28)
测试数据的因变量维度:(10000,)
训练数据因变量前 10 个数字：[5 0 4 1 9 2 1 3 1 4]
```

图 5-10 画出了训练数据的前 20 幅图片，图片对应的因变量写在图片的上方。

```
fig=plt.figure(figsize=(6, 4))  #设置每个数字图片的大小
columns = 5
rows = 4
for i in range(1, columns*rows+1):
    fig.add_subplot(rows, columns, i)
    plt.imshow(x_train[i-1]) # 函数 plt.imshow 可以根据像素点画出图片
    plt.title(y_train[i-1])
    plt.axis('off')
plt.tight_layout(True)
plt.show()
```

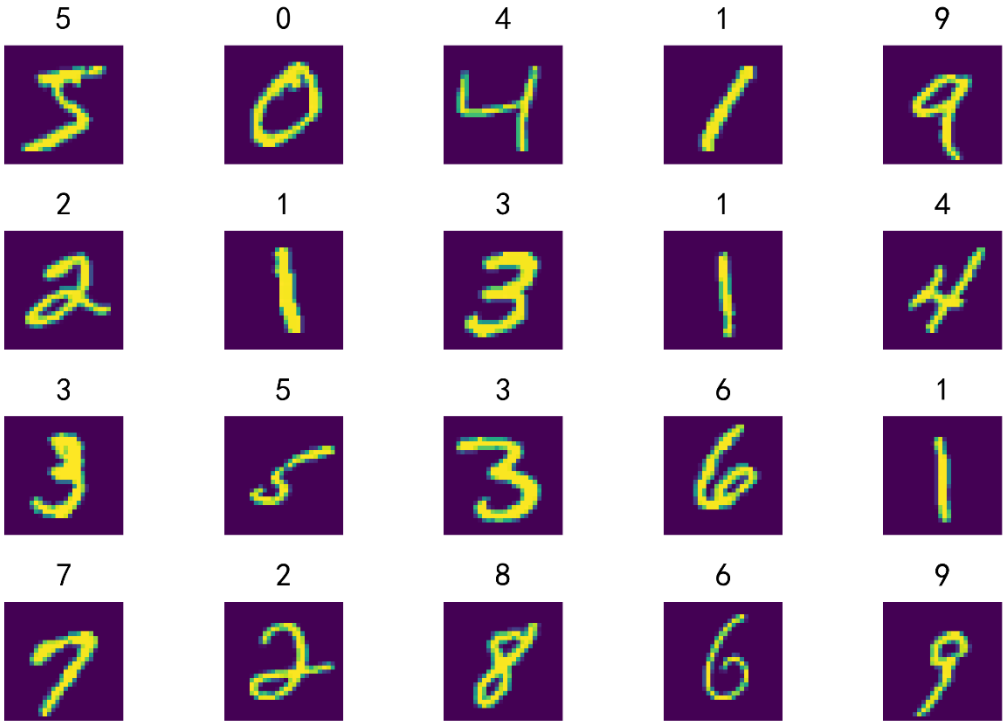

图 5-10 mnist 数据的前 20 幅图

我们为 mnist 数据建立如图 5-11 所示的神经网络。在该神经网络中，输出层节点数为 10，第一个节点表示输入图片为数字 0 的概率，第二个节点表示输入图片为数字 1 的概率，等等。该神经网络结构总结如下。

- 输入层有 784 个节点（注：计算节点数时，通常不包含截距项）。
- 隐藏层有 200 个节点，隐藏层的激活函数为 sigmoid 函数。
- 输出层有 10 个节点，输出层的激活函数为 softmax 函数。

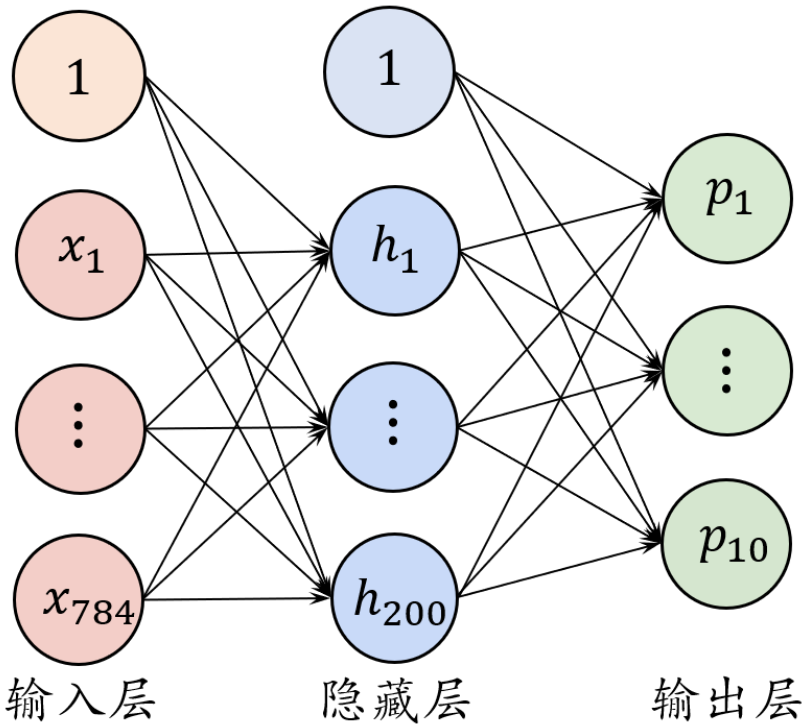

图 5-11 神经网络模型，该模型具有一个隐藏层，
隐藏层节点数为 200，输出层节点数为 10

为了减少计算时间，模型只用 x_train 的前 1000 幅手写数字图片作为训练数据。像素点的灰度值为 0~255 的数字，为了数值计算的稳定性，把所有灰度值除以 255，变换之后自变量的值在 0 到 1 之间。权重初始值随机从$(-0.5,0.5)$的均匀分布中产生，初始截距项系数都设为 0。设置 batch size 为 100，训练数据刚好有 10 批。下面代码和第 4 章处理 Default 数据的代码是类似的，不同之处在于输出层有 10 个节点，且使用 softmax 作为激活函数。从下面代码运行结果可以看出，预测误差迅速从 3000 多下降到 7 左右，训练准确率达到 1，测试准确率逐步稳定在 0.87。

```
"""
mnist 数据
训练具有 1 个隐藏层的神经网络
批量随机梯度下降法
"""
np.random.seed(3)

# 定义函数 sigmoid
def sigmoid(x):
    return 1.0/(1+np.exp(-x))

# 定义函数 sigmoid2deriv()求 sigmoid 函数的导数
def sigmoid2deriv(x):
```

```
    return x * (1-x)

# 定义 softmax 函数
def softmax(x):
    temp = np.exp(x)
    return temp / np.sum(temp, axis = 1, keepdims=True)

#  因变量为 10 类定性数据；自变量维度为 784
num_classes, pixels_per_image = 10, 784
n_train_images = 1000
images = x_train[0:n_train_images]
# 训练数据自变量：把 28*28 的像素点转换成向量形式
images = images.reshape(n_train_images, 28*28)/255
labels = y_train[0:n_train_images]                  # 训练数据因变量

# 训练数据因变量：one-hot 编码
one_hot_labels = np.zeros((len(labels), num_classes))
for i, j in enumerate(labels):
    one_hot_labels[i][j] = 1

# 测试数据自变量
test_images = x_test.reshape(len(x_test), 28*28)/255
# 测试数据因变量：one-hot 编码
one_hot_test_labels = np.zeros((len(y_test), num_classes))
for i, j in enumerate(y_test):
    one_hot_test_labels[i][j] = 1

lr, hidden_size = 0.05, 200    # 给定学习步长和隐藏层的节点个数
batch_size, epochs = 100, 2000 # 给定每一批数据的观测点个数和循环次数
num_batch = int(np.floor(n_train_images/batch_size))

b_0_1 = np.zeros((1, hidden_size))          # 初始化 b_0_1
b_1_2 = np.zeros((1, num_classes))          # 初始化 b_1_2
# 初始化 weights_0_1，服从均匀分布
weights_0_1 = np.random.random(size=(pixels_per_image, \
                                hidden_size))-0.5

# 初始化 weights_1_2，服从均匀分布
weights_1_2 = np.random.random(size=(hidden_size, \
                                     num_classes))-0.5

for e in range(epochs):
    total_loss = 0
    train_acc = 0
    for i in range(num_batch):

        batch_start = i * batch_size
        batch_end = (i+1) *batch_size
        layer_0 = images[batch_start:batch_end]    # layer_0
        # 正向传播算法：计算 layer_1
        layer_1 = sigmoid(np.dot(layer_0, weights_0_1) + b_0_1)
        # 正向传播算法：使用 softmax 函数，计算 layer_2
        layer_2 = softmax(np.dot(layer_1, weights_1_2) + b_1_2)
        labels_batch = one_hot_labels[batch_start:batch_end]
        for j in range(len(layer_0)):
```

```
            # 计算损失函数
            total_loss += - np.log(layer_2[j, \
                          np.argmax(labels_batch[j])])
            # 计算预测正确的观测点数量
            train_acc += int(np.argmax(labels_batch[j]) == \
                         np.argmax(layer_2[j]))
        # 反向传播算法：计算 delta_2
        layer_2_delta = (layer_2 - labels_batch)/batch_size
        # 反向传播算法：计算 delta_1
        layer_1_delta = layer_2_delta.dot(weights_1_2.T)* \
                        sigmoid2deriv(layer_1)
        # 更新 b_0_1 和 b_1_2
        b_1_2 -= lr * np.sum(layer_2_delta, axis = 0, \
                             keepdims=True)
        b_0_1 -= lr * np.sum(layer_1_delta, axis = 0, \
                             keepdims=True)
        # 更新 weights_1_2 和 weights_0_1
        weights_1_2 -= lr * layer_1.T.dot(layer_2_delta)
        weights_0_1 -= lr * layer_0.T.dot(layer_1_delta)

    if(e % 200 == 0 or e == epochs - 1):
        layer_0 = test_images
        # 计算测试准确率
        layer_1 = sigmoid(np.dot(layer_0, weights_0_1) + b_0_1)
        layer_2 = softmax(np.dot(layer_1, weights_1_2) + b_1_2)
        test_acc = 0
        for i in range(len(test_images)):
            test_acc += int(np.argmax(one_hot_test_labels[i])\
                            == np.argmax(layer_2[i]))

        print("Loss: %10.3f;   Train Acc: %0.3f;  Test  Acc:\
              %0.3f"%(total_loss, train_acc/n_train_images,
                      test_acc/len(test_images)))
```

```
Loss:   3043.921;   Train Acc: 0.133;  Test Acc:0.197
Loss:    150.026;   Train Acc: 0.979;  Test Acc:0.851
Loss:     65.080;   Train Acc: 0.999;  Test Acc:0.860
Loss:     37.565;   Train Acc: 1.000;  Test Acc:0.863
Loss:     25.463;   Train Acc: 1.000;  Test Acc:0.864
Loss:     18.916;   Train Acc: 1.000;  Test Acc:0.866
Loss:     14.890;   Train Acc: 1.000;  Test Acc:0.867
Loss:     12.193;   Train Acc: 1.000;  Test Acc:0.867
Loss:     10.275;   Train Acc: 1.000;  Test Acc:0.868
Loss:      8.848;   Train Acc: 1.000;  Test Acc:0.869
Loss:      7.753;   Train Acc: 1.000;  Test Acc:0.870
```

5.2.5 小结

输出层激活函数的选择取决于因变量的数据类型。选定激活函数之后，需要根据建模目标选择相应的损失函数。三个常用的激活函数以及对应的损失函数总结如表 5-1。从表 5-1 可以看到，softmax 函数是 sigmoid 函数的推广，多分类定

性数据的损失函数也是二分类定性数据的推广；虽然对于 3 种因变量的数据类型，激活函数的定义不同，损失函数的定义也不同，但是输出层$\boldsymbol{\delta}$的形式却是相似的。

表 5-1 输出层激活函数的选择

因变量的数据类型	输出层激活函数	损失函数	输出层$\boldsymbol{\delta_i}$
定量数据	不需要激活函数	$\text{ave}_{i\in\mathcal{D}}(y_i-\hat{y}_i)^2$	$\hat{y}_i-y_i$
二分类定性数据	sigmoid 函数	$\text{ave}_{i\in\mathcal{D}}-\{1_{y_i=1}\log(p_i)+1_{y_i=0}\log(1-p_i)\}$	p_i-y_i
多分类定性数据	softmax 函数	$\text{ave}_{i\in\mathcal{D}}-1_{\{y_{ik}=1\}}\log(p_{ik})$	$\boldsymbol{p}_i-\boldsymbol{y}_i$

5.3 隐藏层激活函数

根据激活函数需要满足的基本条件，其实可以有成千上万的函数供我们选择。但是，在实际应用中，有些激活函数在神经网络的发展过程中脱颖而出，获得神经网络科学家和应用者的青睐。本节将介绍其中 4 个常用且效果很好的隐藏层激活函数：

- sigmoid 函数
- tanh 函数
- ReLU 函数
- Leaky ReLU 函数

5.3.1 sigmoid 函数

我们已经对 sigmoid 函数比较熟悉了。这里将和其他激活函数一起系统地介绍 sigmoid 函数。sigmoid 函数定义如下：

$$\text{sigmoid}(x)=\frac{1}{1+\text{e}^{-x}}$$

sigmoid 函数的定义域为$(-\infty,\infty)$，可以将输入值变换到 0 和 1 之间，如图 5-12。该特征使得 sigmoid 函数可以既用于输出层也可以用于隐藏层。在 Python 中，可以如下定义函数 `sigmoid()`。

```
def sigmoid(x):
    return 1/(1 + np.exp(-x))

x = np.linspace(-10, 10, 1000)
plt.plot(x, sigmoid(x), linewidth=3)
```

```
plt.show()
```

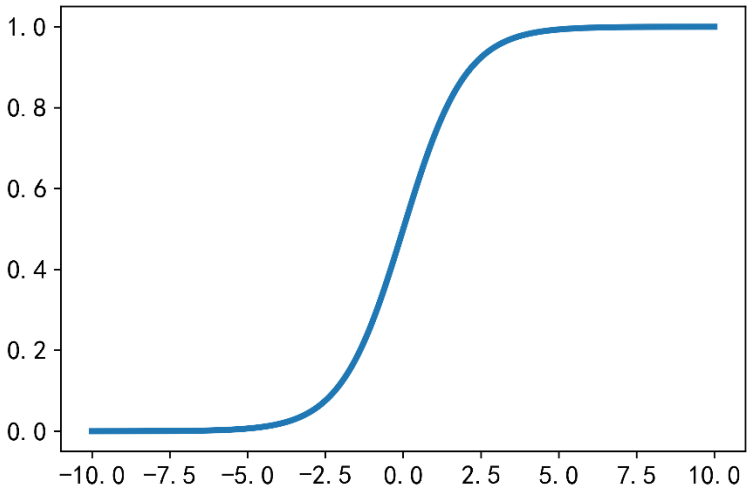

图 5-12 sigmoid 函数

sigmoid 函数的导数为

$$\frac{\partial \text{sigmoid}(x)}{\partial x} = \text{sigmoid}(x)(1 - \text{sigmoid}(x))$$

可以看到，sigmoid 函数的导数非常容易计算。在 Python 中，可以如下计算 sigmoid 函数的导数。

```
# ***** 计算 sigmoid 函数的导数
def sigmoid2deriv(output):
    return output*(1-output)

plt.rcParams['font.sans-serif']=['SimHei'] #用来正常显示中文标签
plt.rcParams['axes.unicode_minus']=False    #用来正常显示负号

plt.plot(x, sigmoid2deriv(sigmoid(x)), linewidth=3, \
         label="sigmoid 函数")
plt.plot(x, sigmoid(x), linewidth=3, \
         label="sigmoid 函数的导数")
plt.ylim(0, 1)
plt.legend()
plt.show()
```

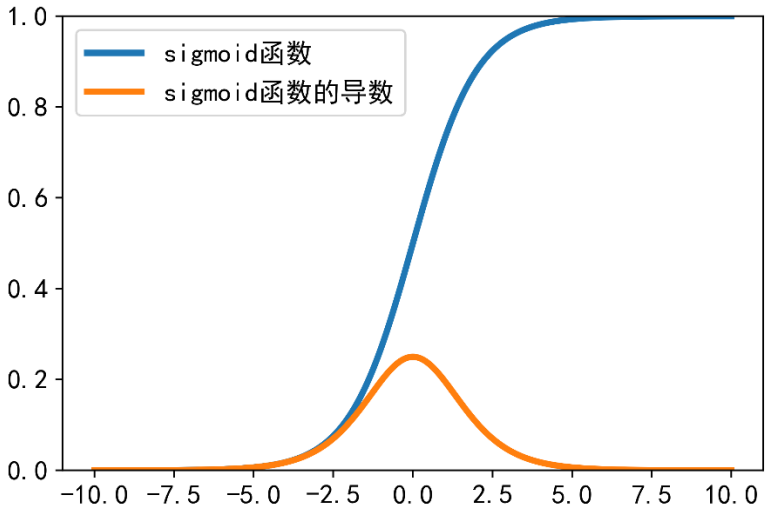

图 5-13 sigmoid 函数及其导数

从图 5-13 可以看出，sigmoid 函数的导数在 0 处达到最大值0.25。随着输入值远离 0，导数值迅速变小。当输入值的绝对值大于 5 时，sigmoid 函数的导数几乎等于 0。这是 sigmoid 函数作为隐藏层激活函数的最大缺点。我们知道在反向传播算法中，应用链式法则计算参数导数时需要乘以激活函数的导数。当多个隐藏层都使用 sigmoid 函数作为激活函数时，参数导数包含多个激活函数的导数乘积。但是，sigmoid 函数的导数都是较小的小数，这可能使得参数的梯度非常小，甚至接近 0，导致神经网络模型无法通过梯度下降法更新参数。这就是 sigmoid 函数的梯度消失问题（Vanishing Gradient）。

5.3.2 tanh 函数

tanh 函数定义为

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

可以看到，tanh 函数可以把输入值变换到−1到1之间，如图 5-14。在 Python 中可以使用 Numpy 函数 `np.tanh()`计算 tanh 函数。

```
# ***** 计算 tanh 函数
def tanh(x):
    return np.tanh(x)

x = np.linspace(-10, 10, 1000)
plt.plot(x, tanh(x), linewidth=3)
plt.show()
```

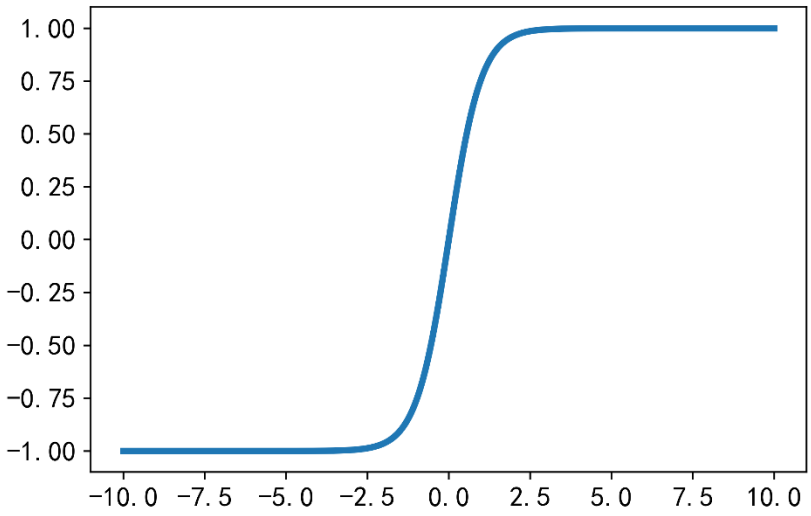

图 5-14 tanh 函数

tanh 函数的导数为

$$\frac{\partial \tanh(x)}{\partial x} = 1 - (\tanh(x))^2$$

即 tanh 函数的导数为 1 减 tanh 函数的函数值的平方，也非常容易计算。在 Python 中，可以如下计算 tanh 函数的导数。

```
# ***** 计算 tanh 函数的导数
def tanh2deriv(output):
    return 1 - output**2

plt.plot(x, tanh(x), linewidth=3, label="tanh 函数")
plt.plot(x, tanh2deriv(tanh(x)), linewidth=3, \
         label = "tanh 函数的导数")
plt.legend()
plt.show()
```

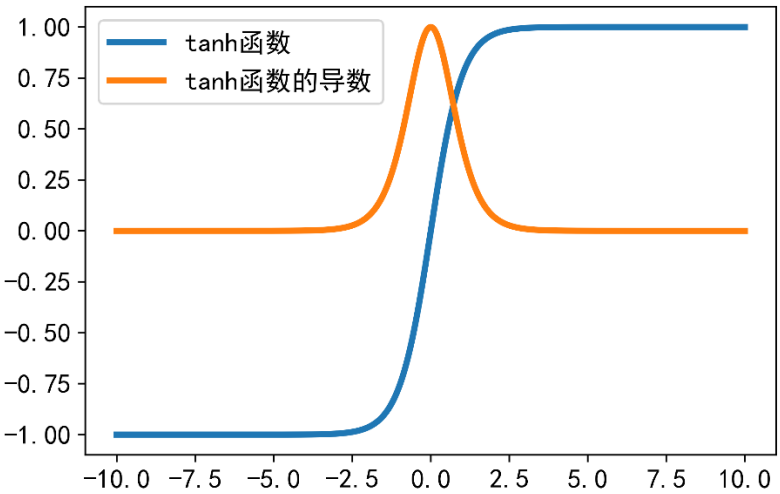

图 5-15 tanh 函数及其导数

从图 5-15 可以看出，tanh 函数的导数在 0 处达到最大值1。随着输入值远离 0 点，导数值迅速变小。当输入值的绝对值大于 2.5 时，tanh 函数的导数非常接近于 0。虽然 tanh 函数的导数最大值达到 1，但是其导数也在 0 和 1 之间，而且当输入值稍大时，tanh 函数的导数快速变小。因此，tanh 函数也有 sigmoid 函数的缺点，即梯度消失问题。比较 sigmoid 函数和 tanh 函数可以看到，sigmoid 函数的值域在 0 和 1 之间，而 tanh 函数以原点为中心，函数值在−1到1之间。tanh 函数的作用相当于把整个负无穷大到正无穷大的输入值变换到一个有界区间，而且不改变符号，这样可以增加数值的稳定性且实现模型非线性。相比于 sigmoid 函数，tanh 函数可以更好地传递信息。

5.3.3 ReLU 函数

ReLU（Rectified Linear Unit）函数是激活函数的新贵，发明和流行的时间都比较晚，却是现在深度学习中最为常用的激活函数。ReLU 函数定义为

$$\mathrm{ReLU}(x) = \max(x, 0)$$

ReLU 函数是一个非常简单的函数，它是一个折线函数，所有负的输入值都变换成 0，所有正的输入值的函数值都等于输入值本身，如图 5-16。在 Python 中，可以如下定义函数 `relu()`。

```
# ***** 计算 ReLU 函数
def relu(x):
    return (x>=0) * x

x = np.linspace(-10, 10, 1000)
plt.plot(x, relu(x), linewidth=3)
plt.show()
```

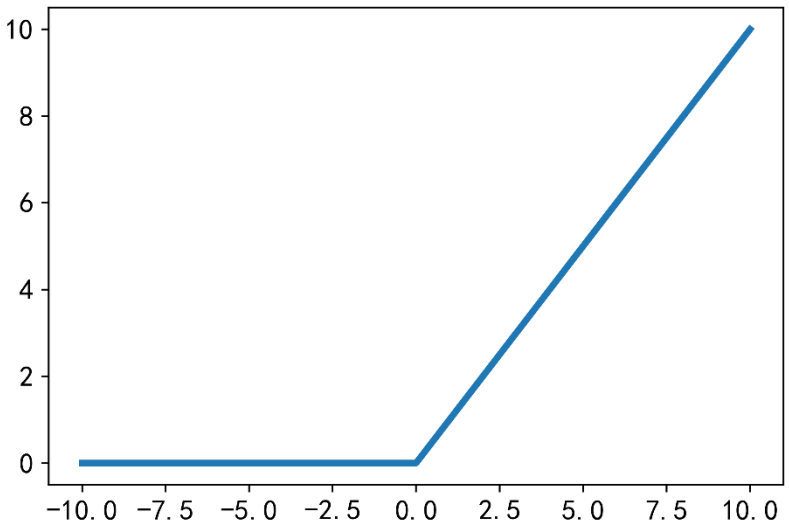

图 5-16 ReLU 函数

ReLU 函数的导数为

$$\frac{\partial \mathrm{ReLU}(x)}{\partial x} = \begin{cases} 1 & \mathrm{x} > 0 \\ 0 & \mathrm{x} < 0 \end{cases}$$

可以看到，ReLU 函数的导数非常容易计算，当x大于 0 时，其导数值为 1，当x小于 0 时，其导数值为 0。在 Python 中，可以如下计算 ReLU 函数的导数。

```
# ***** 计算 ReLU 函数的导数
# 当 output 等于 0 时，ReLU 函数的导数设为 0
def relu2deriv(output):
    return output > 0

plt.plot(x, relu(x), linewidth=2, label="ReLU 函数")
plt.plot(x, relu2deriv(relu(x)), linewidth=4, \
         label = " ReLU 函数的导数")
plt.legend()
plt.show()
```

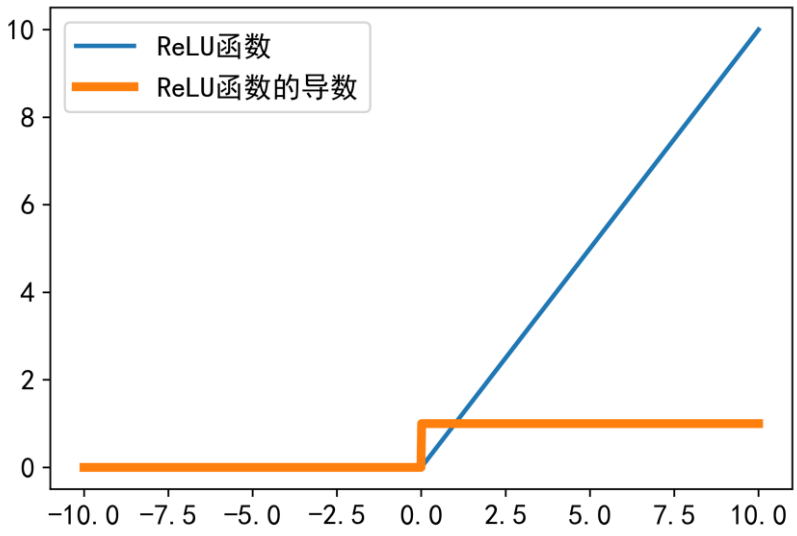

图 5-17 ReLU 函数及其导数

ReLU 函数形式非常简单，计算快，在正值区域没有梯度消失问题（ReLU 函数在所有正输入值的导数都是 1）。因为 ReLU 函数的这些优点，ReLU 函数是深度学习模型最为常用的激活函数之一。但是，ReLU 函数也有缺点，当输入值是负值时，ReLU 函数的梯度等于 0，因此使得相关节点的导数为 0，这类节点相关的权重梯度都是 0。这导致部分权重无法在梯度下降法中优化更新。

5.3.4 Leaky ReLU 函数

Leaky ReLU 函数定义为：

$$\text{Leaky ReLU}(x) = \max(x, \gamma x)$$

其中，γ是一个很小的正数。例如，可以设$\gamma = 0.01$。当输入值x为正，Leaky ReLU 函数的函数值为x本身；当输入值x为负，Leaky ReLU 函数的函数值为x乘以一个很小的正数，如图 5-18。在 Python 中，可以如下定义函数 `Leaky_ReLU()`。

```
# ***** 计算 Leaky ReLU 函数
def leaky_relu(x, gamma=0.01):
    return np.maximum(gamma*x, x)

x = np.linspace(-5, 5, 1000)
plt.plot(x, leaky_relu(x, 0.05), linewidth=3)
plt.plot([-5, 5], [0, 0], 'k-')
plt.plot([0, 0], [-0.5, 4.2], 'k-')

props = dict(facecolor='black', shrink=0.1)
plt.annotate('Leak', xytext=(-3.5, 0.5), xy=(-5, -0.2), \
             arrowprops=props, fontsize=14, ha="center")
plt.axis([-5, 5, -0.5, 4.2])
```

```
plt.show()
```

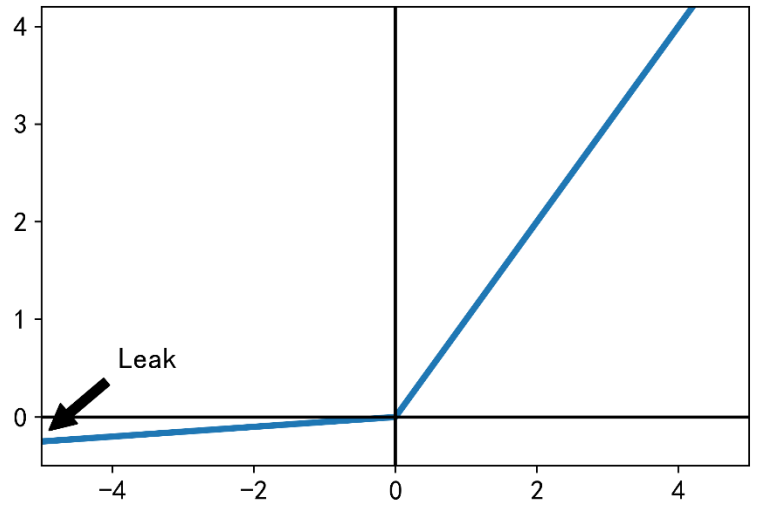

图 5-18 Leaky ReLU 函数

Leaky ReLU 函数的导数为：

$$\frac{\partial \text{Leaky ReLU}(x)}{\partial x} = \begin{cases} 1 & x > 0 \\ \gamma & x < 0 \end{cases}$$

可以看到，Leaky ReLU 函数的导数也非常容易计算，当x大于 0 时，其导数值为 1，当x小于 0 时，其导数值为γ。因此，Leaky ReLU 函数在一定程度上避免了 ReLU 函数的缺点（输入值为负数时，导数为 0）在 Python 中，可以如下计算 Leaky ReLU 函数的导数。

```
# ***** 计算 Leaky ReLU 函数的导数
# 当 output 等于 0 时，Leaky ReLU 函数的导数设为 gamma
def leakyrelu2deriv(output, gamma=0.01):
    deriv = np.ones_like(output)
    deriv[output <= 0] = gamma
    return deriv

x = np.linspace(-5, 5, 1000)
plt.plot(x, leaky_relu(x, 0.05), linewidth=3, \
         label="leaky relu 函数")
plt.plot([-5, 5], [0, 0], 'k-')
plt.plot(x, leakyrelu2deriv(leaky_relu(x, 0.05), 0.05), \
         label="leaky relu 函数的导数")
plt.axis([-5, 5, -0.5, 4.2])
plt.legend()
plt.show()
```

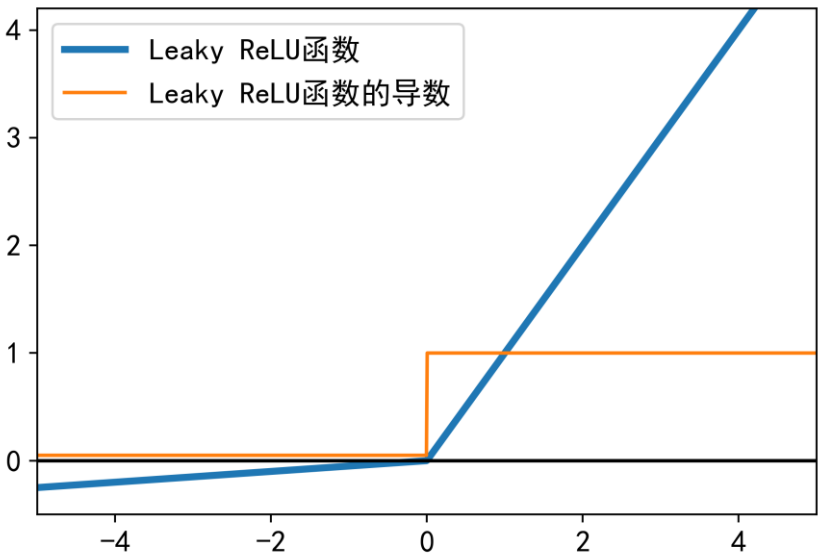

图 5-19 Leaky ReLU 函数及其导数

Leaky ReLU 函数拥有 ReLU 函数的优点，但是减轻了 ReLU 的缺点（输入值为负值时，Leaky ReLU 函数的导数为γ，而不是 0，使得输入值为负值的节点也可以在训练的过程中迭代）。只是建模时需要额外选择一个合适的超参数γ。根据文献记载和我们的实际经验，当我们能够选择合适的γ时，使用 Leaky ReLU 为激活函数的神经网络模型可以表现得更好。

对 mnist 数据的分类问题，现在用 Leaky ReLU 函数替换 5.2.4 节中神经网络模型使用的 sigmoid 函数。从代码运行结果可以看出，在该例子中，使用 Leaky ReLU 函数的神经网络模型的最终测试准确率大约为 0.868，没有比使用 sigmoid 函数的神经网络模型表现得更好。一个可能的原因是该神经网络模型只有一个隐藏层，sigmoid 函数作为隐藏层的激活函数不会造成严重的梯度消失问题。

```
"""
mnist 数据
训练具有 1 个隐藏层的神经网络
批量随机梯度下降法
"""
np.random.seed(1)

def leaky_relu(x, gamma=0.01):
    return np.maximum(gamma*x, x)

def leakyrelu2deriv(x, gamma=0.01):
    deriv = np.ones_like(x)
    deriv[x <= 0] = gamma
    return deriv

def softmax(x):
```

```
    temp = np.exp(x)
    return temp / np.sum(temp, axis = 1, keepdims=True)

#  因变量为 10 类的性定数据；自变量维度为 84
num_classes, pixels_per_image = 10, 784
n_train_images = 1000
# 训练数据自变量：把 28*28 的像素点转换成向量的形式
images = x_train[0:n_train_images]
images = images.reshape(n_train_images, 28*28)/255
labels = y_train[0:n_train_images]          # 训练数据因变量
# 训练数据因变量：one-hot 编码
one_hot_labels = np.zeros((len(labels), num_classes))
for i, j in enumerate(labels):
    one_hot_labels[i][j] = 1

# 测试数据自变量
test_images = x_test.reshape(len(x_test), 28*28)/255
# 测试数据因变量：one-hot 编码
one_hot_test_labels = np.zeros((len(y_test), num_classes))
for i, j in enumerate(y_test):
    one_hot_test_labels[i][j] = 1

lr, hidden_size = 0.05, 200    # 给定学习步长和隐藏层的节点个数
batch_size, epochs = 100, 2000 # 给定每一批数据的观测点个数和循环次数
num_batch = int(np.floor(n_train_images/batch_size))

# 初始化 b_0_1 和 b_1_2
b_0_1 = np.zeros((1, hidden_size))
b_1_2 = np.zeros((1, num_classes))

# 初始化 weights_0_1 和 weights_1_2, 服从均匀分布
weights_0_1 = 0.02*np.random.random(size=(pixels_per_image,\
                                    hidden_size))-0.01
weights_1_2 = 0.2*np.random.random(size=(hidden_size, \
                                   num_classes))-0.1

for e in range(epochs):
    total_loss = 0
    train_acc = 0
    for i in range(num_batch):

        batch_start = i * batch_size
        batch_end = (i+1) * batch_size
        layer_0 = images[batch_start:batch_end]      # layer_0
        # 正向传播算法：计算 layer_1
        layer_1 = leaky_relu(np.dot(layer_0, weights_0_1)+b_0_1)
        # 正向传播算法：计算 layer_2
        layer_2 = softmax(np.dot(layer_1, weights_1_2)+b_1_2)

        labels_batch = one_hot_labels[batch_start:batch_end]
        for j in range(len(layer_0)):
            # 计算损失函数
            total_loss += - np.log(layer_2[j, \
                      np.argmax(labels_batch[j])])
            # 计算预测正确的观测点数量
```

```
            train_acc += int(np.argmax(labels_batch[j]) == \
                             np.argmax(layer_2[j]))
        # 反向传播算法：计算 delta_2
        layer_2_delta = (layer_2 - labels_batch)/batch_size
        # 反向传播算法：计算 delta_1
        layer_1_delta = layer_2_delta.dot(weights_1_2.T)* \
                        leakyrelu2deriv(layer_1)
        # 更新 b_1_2 和 b_0_1
        b_1_2 -= lr * np.sum(layer_2_delta, axis = 0, \
                                            keepdims=True)
        b_0_1 -= lr * np.sum(layer_1_delta, axis = 0, \
                                             keepdims=True)
        # 更新 weights_1_2 和 weights_0_1
        weights_1_2 -= lr * layer_1.T.dot(layer_2_delta)
        weights_0_1 -= lr * layer_0.T.dot(layer_1_delta)

    if (e % 200 == 0 or e == (epochs - 1)):
        layer_0 = test_images
        layer_1 = leaky_relu(np.dot(layer_0, weights_0_1) + \
                                                    b_0_1)
        layer_2 = softmax(np.dot(layer_1, weights_1_2) + b_1_2)
        # 计算测试准确率
        test_acc = 0
        for i in range(len(test_images)):
            test_acc += int(np.argmax(one_hot_test_labels[i])
                            == np.argmax(layer_2[i]))

        print("Loss: %10.3f;   Train Acc: %0.3f;  Test Acc: \
              %0.3f"%(total_loss, train_acc/n_train_images,\
              test_acc/len(test_images)))
```

```
Loss:   2238.407;   Train Acc: 0.434;  Test Acc:0.682
Loss:     38.150;   Train Acc: 1.000;  Test Acc:0.868
Loss:     12.073;   Train Acc: 1.000;  Test Acc:0.868
Loss:      6.533;   Train Acc: 1.000;  Test Acc:0.868
Loss:      4.327;   Train Acc: 1.000;  Test Acc:0.868
Loss:      3.179;   Train Acc: 1.000;  Test Acc:0.868
Loss:      2.485;   Train Acc: 1.000;  Test Acc:0.868
Loss:      2.026;   Train Acc: 1.000;  Test Acc:0.868
Loss:      1.702;   Train Acc: 1.000;  Test Acc:0.868
Loss:      1.462;   Train Acc: 1.000;  Test Acc:0.868
Loss:      1.278;   Train Acc: 1.000;  Test Acc:0.868
```

5.4 本章小结

在本章中，我们介绍了激活函数在神经网络中的作用以及激活函数需要满足的基本要求。特别的，我们介绍了两个应用于输出层的激活函数：

- sigmoid 函数
- softmax 函数

以及 4 个应用于隐藏层的激活函数：

- sigmoid 函数
- tanh 函数
- ReLU 函数
- Leaky ReLU 函数

表 5-2 总结了 4 个隐藏层激活函数的定义、导数、优点以及缺点。实际应用中，有时候很难根据函数的性质和数据的特征确定适合的激活函数。我们可能需要在模型中尝试不同的激活函数，根据模型的数值结果选择合适的激活函数。

表 5-2 4 个隐藏层激活函数总结

激活函数	定义	导数	优点	缺点
sigmoid 函数	$\frac{1}{1+e^{-x}}$	$\text{sigmoid}(x)(1-\text{sigmoid}(x))$	1.连续导数	梯度消失
tanh 函数	$\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$1-(\tanh(x))^{2}$	1.连续导数 2.函数值原点对称	梯度消失
ReLU 函数	$\max(x,0)$	$\{0\}\text{when } x<0;\{1\} \text{ when } x>0$	1.不会梯度消失	输入值为负时，导数为 0
Leaky ReLU 函数	$\max(x,\gamma x)$	$\{\gamma\} \text{ when } x<0;\{1\} \text{ when } x>0$	1.输入值为负时，导数不为 0 2.不会梯度消失	需要确定γ

习题

1. 分析 mnist 数据，建立具有 1 个隐藏层的神经网络，尝试两种情况：a）隐藏层不使用激活函数；b）隐藏层使用 sigmoid 函数作为激活函数，比较两种情况下神经网络的表现。
2. 分析 mnist 数据，建立具有 1 个隐藏层的神经网络，隐藏层的激活函数为 tanh 函数或者 ReLU 函数，比较各种不同激活函数情况下，神经网络的表现。
3. 分析 mnist 数据，建立具有两个隐藏层的神经网络，隐藏层 1 和隐藏层 2 使用不同的激活函数。
4. 分析 Fashion-mnist 数据，建立具有一个隐藏层的神经网络，尝试不同的激活函数，比较各种不同激活函数情况下，神经网络的表现。可以使用

如下方式读入 Fashion-mnist 数据（注：在我们的电脑中，数据保存在工作目录路径的文件夹 data/fashion 中。在该文件夹中，还有文件 mnist_reader.py，我们需要该文件帮助读入 Fashion-mnist 数据）。

```
from data.fashion import mnist_reader

x_fashion_train, y_fashion_train = \
        mnist_reader.load_mnist('./data/fashion',  kind='train')
x_fashion_test, y_fashion_test =  \
        mnist_reader.load_mnist('./data/fashion', kind='t10k')

text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', \
      'coat','sandal', 'skirt', 'sneaker', 'bag', 'ankle boot']

fig=plt.figure(figsize=(6, 4))
columns = 5
rows = 2
for i in range(1, columns*rows+1):
    fig.add_subplot(rows, columns, i)
    plt.imshow(x_fashion_train[i-1].reshape((28, 28)))
    plt.title(text_labels[y_train[i-1]])
    plt.axis('off')
plt.tight_layout(True)
plt.show()
```

图 5-20 Fashion-mnist 数据的前 10 幅图

5. 现在，我们建立 1 个没有隐藏层的神经网络（如图 5-21），并使用该神经网络分析 mnist 数据。在神经网络中，权重矩阵的维度为784×10。在下面的代码中，计算权重时，如果权重小于 0，则令其等于 0。最后把权重矩阵的每一列都变成28×28的矩阵，然后画出图形。观察图 5-22，思考权重在神经网络中的意义。（我们知道权重矩阵的每一列表示

所有输入层节点到对应输出层节点的权重。例如，权重矩阵的第一列表示所有输入层节点计算数字0的概率的权重。）

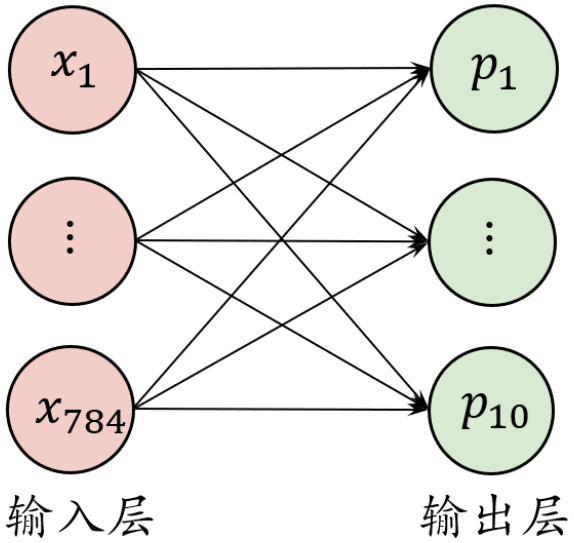

图 5-21 没有隐藏层的神经网络

```
"""
mnist 数据
训练没有隐藏层的神经网络
批量随机梯度下降法
"""
np.random.seed(3)

def softmax(x):
    temp = np.exp(x)
    return temp / np.sum(temp, axis = 1, keepdims=True)

num_classes, pixels_per_image = 10, 784
n_train_images = 1000
images = x_train[0:n_train_images]
images = images.reshape(n_train_images, 28*28)/255
labels = y_train[0:n_train_images]
one_hot_labels = np.zeros((len(labels), num_classes))
for i, j in enumerate(labels):
    one_hot_labels[i][j] = 1

test_images = x_test.reshape(len(x_test), 28*28)/255
one_hot_test_labels = np.zeros((len(y_test), num_classes))
for i, j in enumerate(y_test):
    one_hot_test_labels[i][j] = 1

lr = 0.05
batch_size, epochs = 100, 2000
num_batch = int(np.floor(n_train_images/batch_size))

b_0_1 = np.zeros((1, 10))
```

```
weights_0_1 = np.random.random(size=(pixels_per_image, 10))-0.5

for e in range(epochs):
    total_loss = 0
    train_acc = 0
    for i in range(num_batch):

        batch_start, batch_end = i * batch_size, \
                                 (i+1) * batch_size
        layer_0 = images[batch_start:batch_end]
        layer_1 = softmax(np.dot(layer_0, weights_0_1)+b_0_1)

        labels_batch = one_hot_labels[batch_start:batch_end]
        for j in range(len(layer_0)):
            total_loss += - np.log(layer_1[j, \
                                   np.argmax(labels_batch[j])])
            train_acc += int(np.argmax(labels_batch[j]) == \
                                   np.argmax(layer_1[j]))

        layer_1_delta = (layer_1 - labels_batch)/batch_size
        b_0_1 -= lr * np.sum(layer_1_delta, axis = 0, \
                             keepdims=True)
        weights_0_1 -= lr * layer_0.T.dot(layer_1_delta)
        weights_0_1[weights_0_1<=0] =0     # 令小于 0 的权重等于 0

    if(e % 200 == 0 or e == epochs - 1):
        layer_0 = test_images
        layer_1 = softmax(np.dot(layer_0, weights_0_1) + b_0_1)
        test_acc = 0
        for i in range(len(test_images)):
             test_acc += int(np.argmax(one_hot_test_labels[i])\
                             == np.argmax(layer_1[i]))

        print("Loss: %10.3f;   Train Acc: %0.3f;  Test Acc: \
               %0.3f"%(total_loss, train_acc/n_train_images, \
                                test_acc/len(test_images)))

Loss:   3071.014;   Train Acc: 0.125;  Test Acc:0.138
Loss:    234.250;   Train Acc: 0.953;  Test Acc:0.848
Loss:    152.757;   Train Acc: 0.977;  Test Acc:0.850
Loss:    112.962;   Train Acc: 0.989;  Test Acc:0.848
Loss:     89.277;   Train Acc: 0.997;  Test Acc:0.847
Loss:     73.852;   Train Acc: 0.998;  Test Acc:0.846
Loss:     63.029;   Train Acc: 1.000;  Test Acc:0.846
Loss:     54.979;   Train Acc: 1.000;  Test Acc:0.847
Loss:     48.776;   Train Acc: 1.000;  Test Acc:0.846
Loss:     43.844;   Train Acc: 1.000;  Test Acc:0.846
Loss:     39.847;   Train Acc: 1.000;  Test Acc:0.845

fig=plt.figure(figsize=(6, 4))
columns = 5
rows = 2
for i in range(1, columns*rows+1):
    fig.add_subplot(rows, columns, i)
    plt.imshow(weights_0_1[:,i-1].reshape((28, 28)))
```

```
    plt.title(str(i-1))
    plt.axis('off')
plt.tight_layout(True)
plt.show()
```

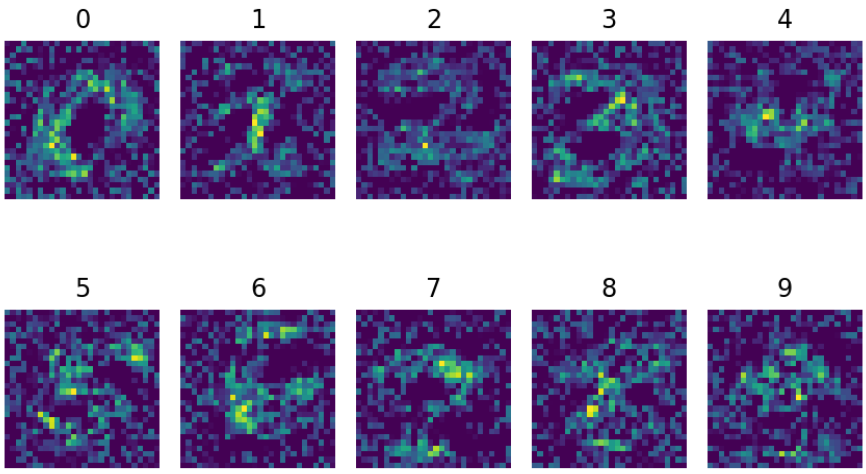

图 5-22 没有隐藏层的神经网络的权重