第 4 章 深度神经网络

前面章节介绍了线性代数、微积分和概率论的相关数学知识，以及 Anaconda、Jupyter Notebook 和 Python 语言的相关内容，并且深入学习了线性回归模型、logistic 模型和梯度下降法。从本章起，将正式开始介绍深度神经网络。本章将从以下 4 个方面来介绍。

- 为什么需要深度神经网络？
- 正向传播算法：使用深度神经网络得到预测值的方法。
- 反向传播算法：计算深度神经网络参数梯度的方法。
- 训练深度神经网络：结合正向传播算法和反向传播算法，使用梯度下降法训练深度神经网络。

4.1 为什么需要深度神经网络

4.1.1 简单神经网络

在第 3 章中我们学习了线性回归模型，

$$y = b + w_1 x_1 + w_2 x_2 + \cdots + w_p x_p + \epsilon$$

当$b, w_1, w_2, \cdots, w_p$已知时，自变量$x_1, x_2, \cdots, x_p$的加权和加上截距项b即为回归模型的预测值。可以用图 4-1 表示线性回归模型（假定只有两个自变量）。图 4-1 中左边圆圈表示输入值（截距项的输入值为1），在神经网络中称为输入层；右边圆圈表示输出值，在神经网络中称为输出层；椭圆形表示计算过程，在这里计算过程是输入值乘以对应权重后求和。

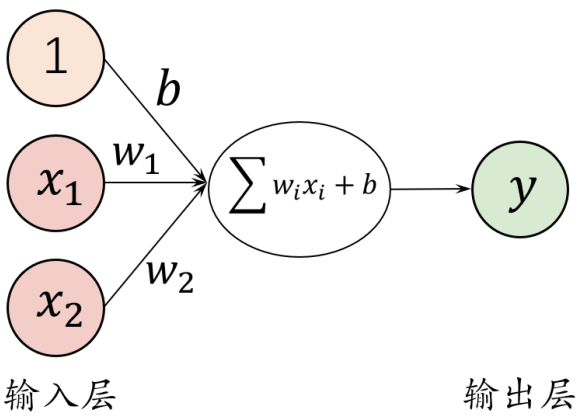


图 4-1 线性回归模型：一个简单神经网络

在神经网络中，通常把图 4-1 模型类比为一个神经元（神经细胞）。图 4-2 为生物神经元示意图，神经元有 3 个重要组成部分：细胞核、树突和轴突。在生物体内，神经元通过树突接收信息，然后把信息加工处理后，通过轴突传出信息。

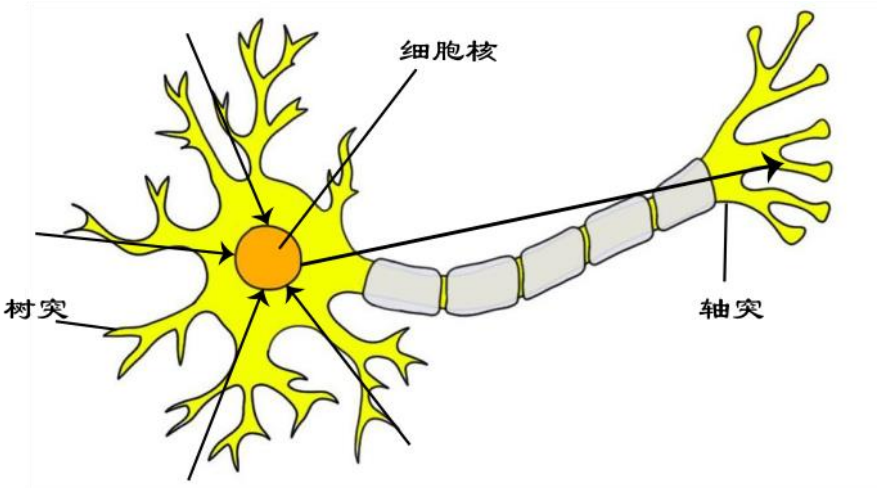


图 4-2 生物神经元

基于图 4-1 和图 4-2，可以做如下类比。

- 在线性回归模型中，左边输入层表示神经元树突接收到的信息。
- 在线性回归模型中，加权和可以看作神经元对接收到的信息进行加工处理。
- 在线性回归模型中，加权和为神经元轴突的传出信息。

通过这些简单类比可以看到，线性回归模型和生物神经元的工作机理有其相似之处。因此，我们称线性模型是一个神经网络（该神经网络只包含一个神经元）。

在 logistic 模型中，因变量Y服从概率为$P(Y=1)$的伯努利分布，其中，

$$P(Y=1)=\frac{1}{1+\mathrm{e}^{-(b+w_1x_1+\cdots+w_px_p)}}$$

当$b,w_1,w_2,\cdots,w_p$的估计值已知，首先可以计算自变量$x_1,x_2,\cdots,x_p$的加权和加上截距项b（即$\sum w_ix_i+b$），然后把所得结果代入 sigmoid 函数得到$Y=1$的概率。

可以用图 4-3 表示 logistic 模型（假定只有两个自变量）。图 4-3 中左边圆圈表示输入层；右边圆圈表示输出层，y表示输入信息加权和代入 sigmoid 后的值；椭圆形表示计算过程。同样也可以把 logistic 模型与神经元做类比：左边的输入层表示神经元树突接收到的信息；求加权和，然后代入 sigmoid 函数为神经元对接收到的信息的加工处理；$Y=1$的概率为传出信息。logistic 模型与线性回归模

型的不同点在于：logistic 模型对输入值加权和使用 sigmoid 函数做了非线性处理。除了这个不同点，logistic 模型和线性回归模型的原理是类似的。因此，logistic 模型也是一个神经网络。

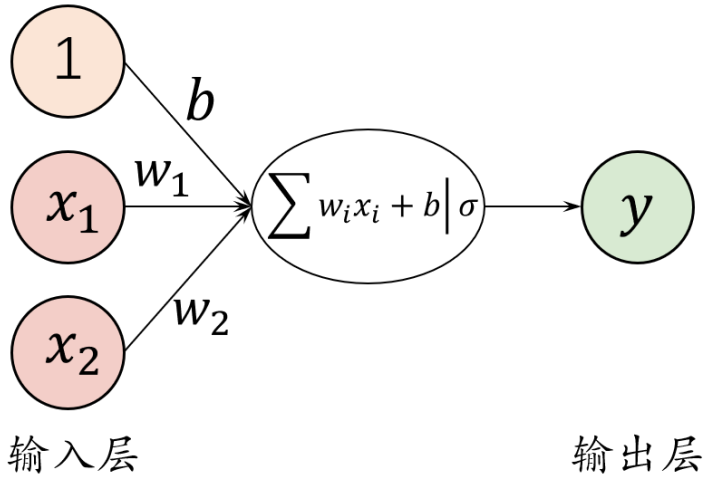


图 4-3 logistic 模型

基于第 3 章关于 logistic 模型的知识，我们把 logistic 模型的训练和预测过程写成 Python 函数。在 logistic 模型中，训练和预测过程经常会用到 sigmoid 函数，因此首先在代码中定义函数 `sigmoid()`。

```
"""
1. 载入需要用到的包
2. 定义函数 sigmoid()
"""
%config InlineBackend.figure_format = 'retina'
import numpy as np
import matplotlib.pyplot as plt

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

接着定义函数 `logit_model(x, y, w, b, lr=0.1)`，该函数有如下参数。

- `x`：自变量数组。
- `y`：因变量数组。
- `w`：权重初始值。
- `b`：截距项初始值。
- `lr`：学习步长，默认值设为 0.1。

函数 `logit_model()` 使用随机梯度下降法得到参数估计值，输出权重$\boldsymbol{w}$和截距项b的估计值。

```
"""
定义函数 logit_model(), 通过随机梯度下降法估计参数
"""
def logit_model(x, y, w, b, lr=0.1):
    for iter in range(60):
        loss = 0
        for i in range(len(x)):
            pred = sigmoid(np.dot(x[i:i+1], w) + b)  # 计算预测值
            loss += -(y[i:i+1] * np.log(pred) + \
                    (1-y[i:i+1]) * np.log(1-pred))   # 计算误差
            delta = pred - y[i:i+1]
            b -= lr * delta                          # 更新 b
            w -= lr * np.dot(x[i:i+1].T, delta)      # 更新 w

        if (iter%10==9 or iter==59):
            print("Loss:"+str(loss))
    return w, b
```

我们还定义如下 3 个辅助函数。

- 函数 `predict_logit_model(x, w, b)`：给定 `w`，`b` 时，计算自变量 `x` 的预测值。
- 函数 `scatter_simple_data(x, y)`：画 `x` 的散点图，可以根据 `y` 的不同，用不同颜色表示 `x`。这里，`x` 只能为二维向量。
- 函数 `plot_decision_bound(x, y, w, b)`：画 logistic 模型的决策边界。

```
"""
1. 定义函数 predict_logit_model()，给定 w,b 时，计算 logistic 模型的预测值
2. 定义函数 scatter_simple_data()，画 x 的散点图
3. 定义函数 plot_decision_bound()，画 logistic 模型的决策边界
"""
def predict_logit_model(x, w, b):
    pred = []
    for i in range(len(x)):
        tmp = sigmoid(np.dot(x[i:i+1], w) + b)     # 计算预测概率
        if tmp > 0.5:
            tmp = 1
        else:
            tmp = 0
        pred.append(tmp)
    return np.array(pred)

def scatter_simple_data(x, y):
    plt.scatter(x[y==0, 0], x[y==0, 1], label="0", marker="o")
    plt.scatter(x[y==1, 0], x[y==1, 1], s = 80, label="1", \
                marker="s")
    plt.legend()
    plt.xlabel("$x_1$", fontsize=16)
    plt.ylabel("$x_2$", fontsize=16)
    plt.show()
```

```
def plot_decision_bound(x, y, w, b):
    x1 = np.linspace(0, 1, 100)
    x2 = (-b - x1 * w[0])/w[1]
    plt.scatter(x[y==0, 0], x[y==0, 1], label="0", marker="o")
    plt.scatter(x[y==1, 0], x[y==1, 1], s = 80, label="1", \
                marker="s")
    plt.plot(x1, x2)
    plt.legend()
    plt.xlabel("$x_1$", fontsize=16)
    plt.ylabel("$x_2$", fontsize=16)
```

现在通过构造两个简单数据，观察在这些数据中 logistic 模型的表现。

- 数据 1：该数据有 4 个观测点。观测点(0,0)，(0,1)的因变量值为1；观测点(1,0)，(1,1)的因变量值为0；如图 4-4 所示，正方形表示因变量为 1 的观测点，圆点表示因变量为 0 的观测点。直观的看，圆点和正方形可以很容易被一条直线分割。

```
"""
定义函数 createDataSet_1()， 生成数据 1
"""
def createDataSet_1():
    x = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y = np.array([1, 1, 0, 0])
    return x, y

x, y = createDataSet_1()
scatter_simple_data(x, y)
```

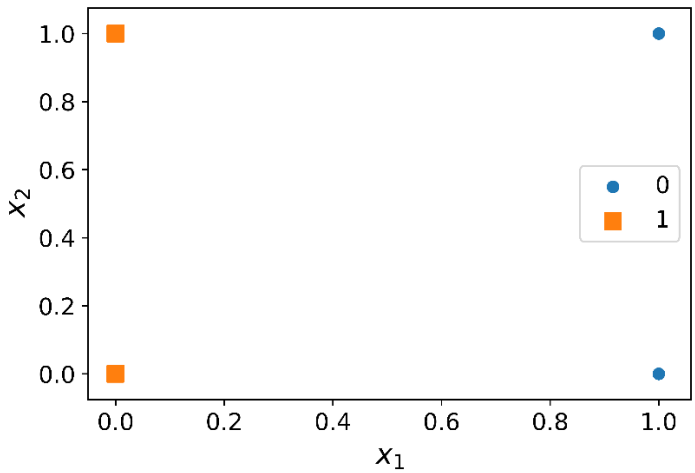


图 4-4 数据 1 散点图

我们把数据 1 的自变量 x，因变量 y 代入函数 logit_model()中训练 logistic 模型。这里从标准正态分布中随机生成初始权重w，截距项 b=0，学习步长 lr=0.1。从下面代码运行结果看，损失函数的值随着迭代不断变小，而且预测值全部等于真实因变量的值。

```
np.random.seed(4)
w = np.random.normal(size=2)
b, lr = 0, 0.1
w, b = logit_model(x, y, w, b, lr)

pred = predict_logit_model(x, w, b)
print("因变量的真实值为: " + str(y))
print("因变量的预测值为: " + str(pred))

Loss:[2.13470015]
Loss:[1.67673736]
Loss:[1.36190603]
Loss:[1.13485563]
Loss:[0.96600061]
Loss:[0.83689858]
因变量的真实值为: [1 1 0 0]
因变量的预测值为: [1 1 0 0]
```

从图 4-5 中可以看到，logistic 模型找到了一条斜向上的直线，该直线可以很好地分割因变量分别为 1 和 0 的观测点，即分割正方形和圆点。

```
plot_decision_bound(x, y, w, b)
```

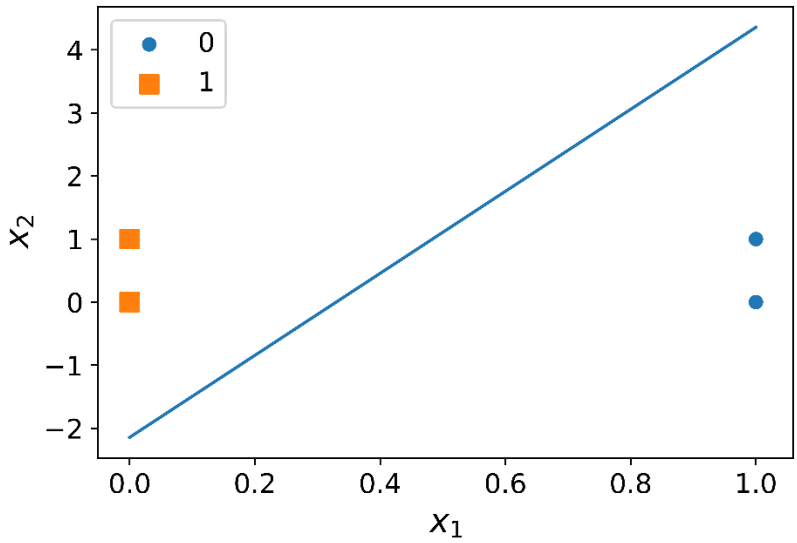


图 4-5 数据 1 决策边界

- 数据 2：该数据有 4 个观测点。观测点(0,0)，(1,1)的因变量值为1；观测点(0,1)，(1,0)的因变量值为0；如图 4-6 所示。直观的看，圆点和正方形不可能被一条直线分割。

```
"""
定义函数 createDataSet_2， 生成数据 2
"""
def createDataSet_2():
    x = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y = np.array([1, 0, 0, 1])
```

```
    return x, y

x, y = createDataSet_2()
scatter_simple_data(x, y)
```

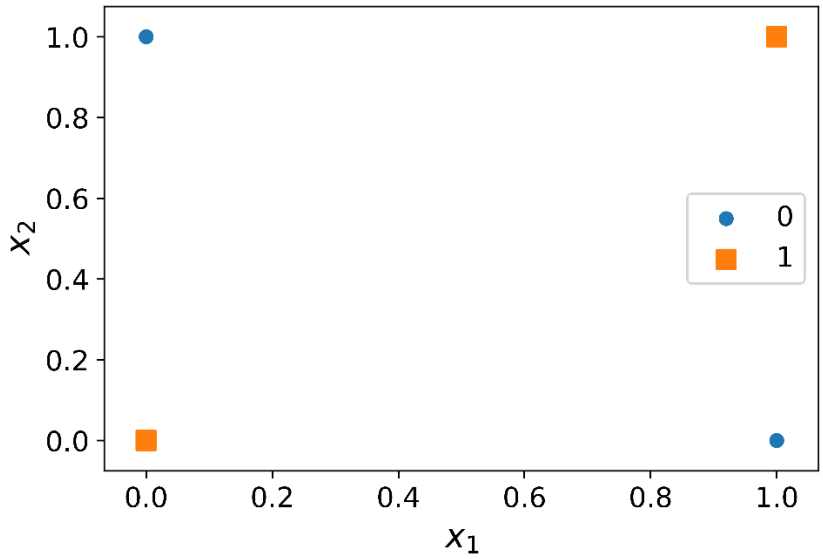


图 4-6 数据 2 散点图

我们把数据 2 的自变量 x，因变量 y 代入函数 `logit_model()`中，训练 logistic 模型。设置初始权重 `w=(0, -0.3)`，截距项 `b=0`，学习步长 `lr=0.1`。从下面代码运行结果看，损失函数的值稳定在 2.88，不再继续变小，而且有一个观测点的预测值是错的。观测点(1,0)的因变量真实值为0，预测值为1。

```
w = np.array([0, -0.3])
b, lr = 0, 0.1
w, b = logit_model(x, y, w, b, lr=0.1)

y_pred = predict_logit_model(x, w, b)
print("因变量的真实值为："+str(y))
print("因变量的预测值为："+str(y_pred))

Loss:[2.88349771]
Loss:[2.8812307]
Loss:[2.8797981]
Loss:[2.87874047]
Loss:[2.87800035]
Loss:[2.87749129]
因变量的真实值为：[1 0 0 1]
因变量的预测值为：[1 0 1 1]
```

从图 4-7 可以看出，logistic 模型找到了一条斜向上的直线，把直线右下方的点都预测为 1，直线左上方的点预测为 0；右下方的圆点预测错误。总的来说，logistic 模型在该简单数据中表现并不好，4 个观测点中有 1 个观测点预测错误。

从第 3 章关于 logistic 模型的介绍，我们知道 logistic 模型的决策边界是一条直线。而从直观上看，该数据不可能被一条直线很好地分割。因此，logistic 模型在该数据中表现不好其实是情理之中的。

```
plot_decision_bound(x, y, w, b)
```

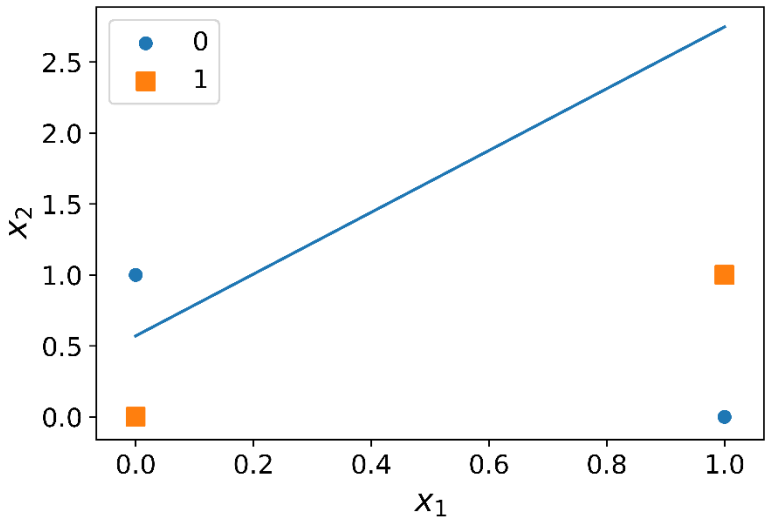


图 4-7 数据 2 决策边界

4.1.2 具有隐藏层的神经网络

从 4.1.1 节的实验可以看到，决策边界是一条直线的 logistic 模型可以很好地完成数据 1 的分类任务，但是不能很好地完成数据 2 的分类任务。原因是 logistic 模型的决策边界是线性的，无法很好完成决策边界是非线性的分类任务。如何改进呢？我们进一步观察数据 2 的散点图；从直观上看，用两根直线，可以更好地完成数据 2 的分类任务。用两根直线分割这 4 个点的方式有很多，图 4-8 展示了其中一种方式。从图 4-8 中可以看到，夹在两条直线中间的是因变量为 1 的正方形，在两条直线外侧的是因变量为 0 的圆点。也就是说，如果可以先得到两条如图 4-8 所示的直线，然后再判断观测点在两条直线中所处位置，那么就可以很好地完成该分类任务，即两条直线中间的应该预测为 1，两条直线外侧的应该预测为 0。

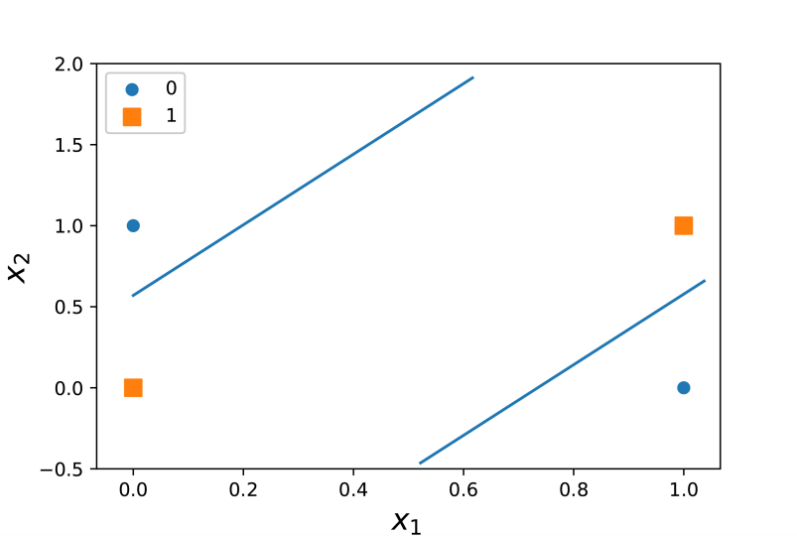


图 4-8 两条直线分割数据 2 的 4 个观测点

根据上面的分析，我们需要建立一个更加复杂的模型才能完成数据 2 的分类任务。

- 先建立两个 logistic 模型（这两个 logistic 模型的决策边界为图 4-8 所示的两条直线）。
- 然后把这两个 logistic 模型的结果作为输入，建立一个 logistic 回归模型（该 logistic 模型用于判断观测点在两条直线中所处位置）。

该模型的结构可以用图 4-9 表示。图 4-9 左边圆圈表示输入层（截距项和自变量：1，x_1，x_2），计算输入层的两个加权和，然后把加权和代入 sigmoid 函数，分别得到h_1和h_2。h_1和h_2是两个 logistic 模型的预测概率。中间圆圈表示隐藏层。然后计算隐藏层h_1和h_2的加权和，把该加权和代入 sigmoid 函数，得到输出值；这也是一个 logistic 模型；右边圆圈表示输出层。注意：在图 4-9 中，3 个椭圆形中的权重和截距项都写成统一的w_i和b，这里只是为了记号方便，实际中 3 个椭圆形中的权重和截距项的值是不同的。在神经网络中，除了输入层和输出层，其他层都称为隐藏层。这是我们建立的第一个带有隐藏层的神经网络。在神经网络的术语中，sigmoid 函数称为激活函数。

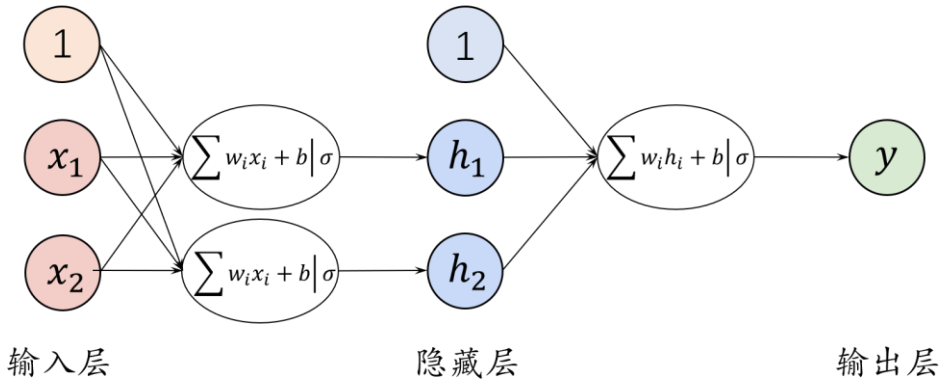


图 4-9 具有 1 个隐藏层的神经网络

我们可以通过下面代码实现图 4-9 所示的神经网络。这里可以暂时忽略代码的实现过程，在本章的后面部分会对这部分代码进行详细讲解。

```
"""
训练有隐藏层的神经网络
可以暂时忽略代码细节，只需要看代码运行结果
"""
np.random.seed(3)
def sigmoid(input):
    return 1.0/(1+np.exp(-input))

def sigmoid2deriv(output):
    return output * (1-output)

lr = 2
hidden_size = 2

b_0_1 = np.zeros((1, 2))
b_1_2 = 0
weights_0_1 = np.random.normal(size=(2, hidden_size))
weights_1_2 = np.random.normal(size=(hidden_size, 1))

for iter in range(600):
    total_loss = 0
    for i in range(len(x)):
        layer_0 = x[i:i+1]
        layer_1 = sigmoid(np.dot(layer_0, weights_0_1) + b_0_1)
        layer_2 = sigmoid(np.dot(layer_1, weights_1_2) + b_1_2)

        total_loss += -(y[i:i+1]*np.log(layer_2) + \
                        (1-y[i:i+1])*np.log(1-layer_2))

        delta_2 = (layer_2 - y[i:i+1])
        delta_1 = delta_2.dot(weights_1_2.T) * \
                                    sigmoid2deriv(layer_1)

        b_1_2 -= lr * delta_2
        b_0_1 -= lr * delta_1
```

```
        weights_1_2 -= lr * layer_1.T.dot(delta_2)
        weights_0_1 -= lr * layer_0.T.dot(delta_1)

    if (iter%100==9 or iter==599):
        print("Loss: "+str(total_loss))

Loss: [[4.05018207]]
Loss: [[2.93567526]]
Loss: [[2.82539712]]
Loss: [[0.07301895]]
Loss: [[0.02847156]]
Loss: [[0.01767868]]
Loss: [[0.01317388]]
"""
应用上面训练的神经网络，得到 4 个观测点的预测值以及隐藏层 h1 和 h2 的值
可以暂时忽略代码细节，只需要看代码运行结果
"""
y_pred = []
layer_1_record = np.zeros((4, 2))
for i in range(len(x)):
    layer_0 = x[i:i+1]
    layer_1 = sigmoid(np.dot(layer_0, weights_0_1) + b_0_1)
    layer_1_record[i,:] = layer_1
    layer_2 = sigmoid(np.dot(layer_1, weights_1_2) + b_1_2)
    y_pred.append(int(layer_2>0.5))

print("因变量的真实值为："+str(y))
print("因变量的预测值为："+str(np.array(y_pred)))
因变量的真实值为：[1 0 0 1]
因变量的预测值为：[1 0 0 1]
```

通过上面代码运行结果可以看到，不同于 logistic 模型，该神经网络的损失函数值随着迭代不断变小；而且，因变量的预测值全部等于其真实值；包含隐藏层的神经网络可以很好地完成数据 2 的分类任务。这样的结果符合预期。

现在，进一步观察上面神经网络的隐藏层和输出层的值。数据 2 的 4 个观测点，隐藏层和输出层的结果总结如表 4-1。

表 4-1 数据 2 中 4 个观测点的隐藏层和输出层结果

序号	自变量	因变量 y	隐藏层(h_1, h_2)	预测值$\hat{y}$
1	(0,0)	1	(0.01, 0.96)	1
2	(0,1)	0	(0.00, 0.03)	0
3	(1,0)	0	(0.98, 1.00)	0
4	(1,1)	1	(0.01, 0.98)	1

图 4-10 中画出了神经网络从输入层到隐藏层h_1和h_2的两个 logistic 模型对应的决策边界。结合表 4-1 可以看到，h_1 和h_2对应的 logistic 模型都把决策边界左上的点预测为 0，决策边界右下的点预测为 1。具体可以得到如下结论。

- 神经网络把第二个观测点的输入值(0,1)变换为隐藏层的(0.00,0.03)（该观测点在两个决策边界之上）；
- 把第三个观测点的输入值(1,0)变换为隐藏层的(0.98,1.00)（该观测点在两个决策边界之下）；
- 同时，神经网络分别把第一个和第四个观测点的输入值的(0,0)和(1,1)变换为隐藏层的(0.01,0.96)和(0.01,0.98)（这两个观测点在第一个决策边界之上，在第二个决策边界之下）。

换句话说，两个隐藏层的 logistic 模型把两条直线之外的观测点分别大致预测为(0,0),(1,1)，把两条直线之间的观测点大致预测为(0,1)。

```
"""
从输入层到隐藏层 h1 的 logistic 模型
可以暂时忽略代码细节，只需要看代码运行结果
"""
plt.subplot(1,2,1)
plot_decision_bound(x, y, weights_0_1[:,0], b_0_1[0,0])

# 从输入层到隐藏层 h2 的 logistic 模型
plt.subplot(1,2,2)
plot_decision_bound(x, y, weights_0_1[:,1], b_0_1[0,1])
plt.tight_layout()
```

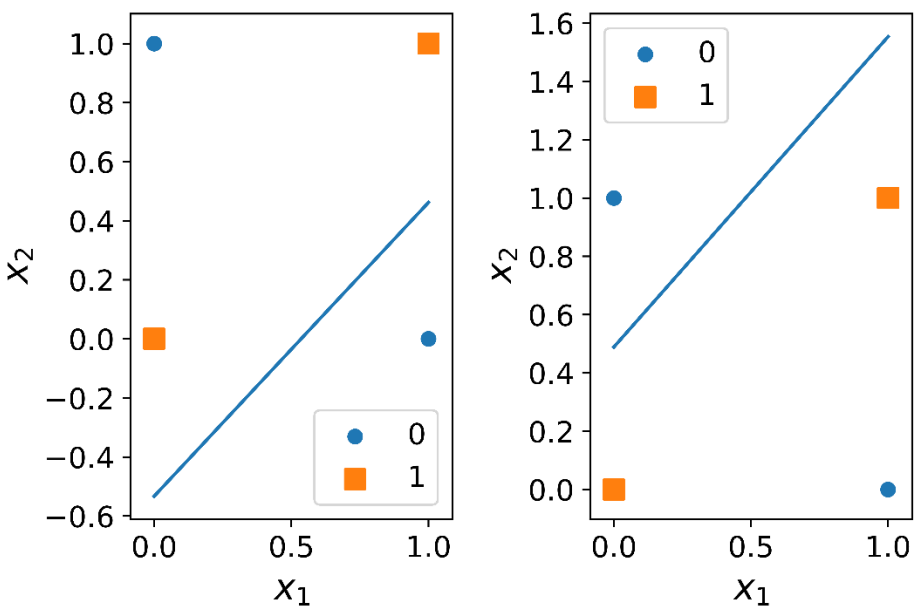


图 4-10 两个隐藏层的决策边界

此时画出隐藏层的散点图，如图 4-11 所示。因变量为 0 的两个点的隐藏层大致位于(0,0)，(1,1)处；而因变量为 1 的两个点的隐藏层大致位于(0,1)处（两个正方形位置接近，几乎重合在一起了）。这时，从隐藏层到输出层的 logistic 模型可以完美地把因变量为 0 的点和因变量为 1 的点分隔开了。

```
"""
从隐藏层到输出层的 logistic 模型
可以暂时忽略代码细节，只需要看代码运行结果
"""
w = weights_1_2[:,0]
b = b_1_2[0,0]
h1 = np.linspace(0, 1, 100)
h2 = (-b - h1 * w[0])/w[1]
plt.scatter(layer_1_record[y==0,0], layer_1_record[y==0,1],\
            label="0", marker="o")
plt.scatter(layer_1_record[y==1,0], layer_1_record[y==1,1], \
            s = 80, label="1", marker="s")
plt.plot(h1, h2)
plt.legend()
plt.xlabel("$h_1$", fontsize=16)
plt.ylabel("$h_2$", fontsize=16)
plt.show()
```

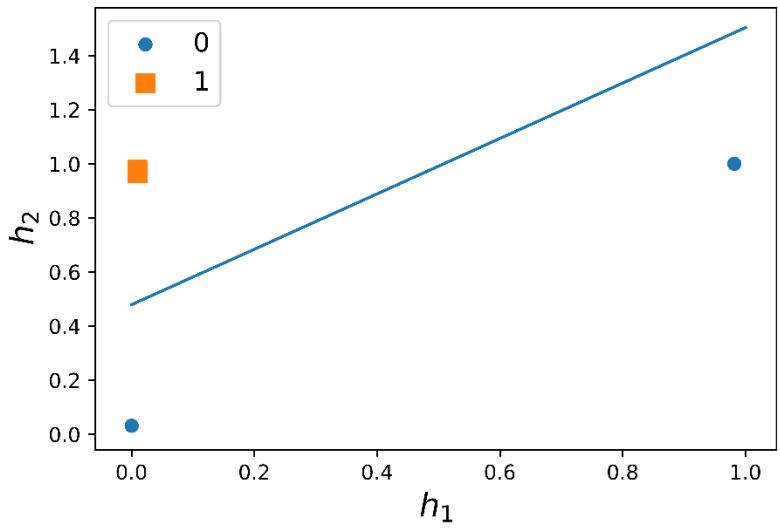


图 4-11 输出层的决策边界

从上面的例子可以看到，神经网络可以通过隐藏层学习数据的不同特征（例如，数据 2 的例子中观测点在两条直线中所处位置），然后再根据隐藏层得到的特征做出更好地预测。也就是说，通过增加隐藏层，神经网络可以找到输入层和因变量之间更复杂的关系。

为了方便，我们通常用图 4-12 表示图 4-9 所示的神经网络。左边圆圈表示输入层，中间圆圈表示隐藏层，右边圆圈表示输出层。在神经网络中，圆圈也称为节点。节点与节点之间的联系用箭头表示。例如，输入层的 3 个节点都有箭头指向h_1，表示由输入层的1，x_1，x_2的加权和代入激活函数得到h_1。

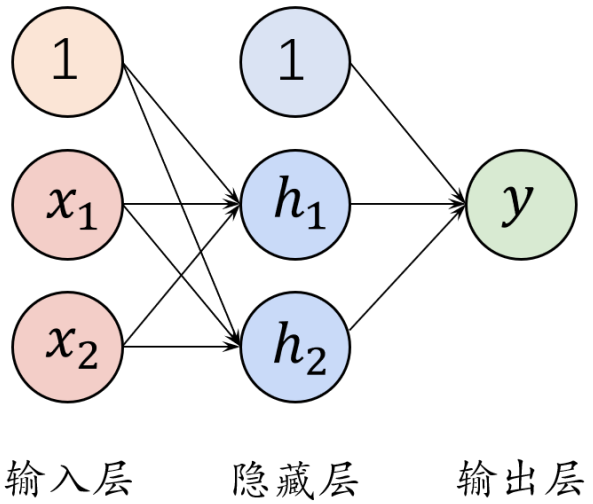


图 4-12 具有 1 个隐藏层的神经网络图

在数据 2 的例子中，我们用具有两个节点的隐藏层的神经网络解决了 logistic 模型无法处理的问题（计算节点数时，不包含截距项），那是不是可以通过增加隐藏层的数量和每个隐藏层的节点数处理更加复杂的问题呢？答案是可以的！拥有多个隐藏层的神经网络就是深度学习。但是，简单增加隐藏层的数量和每个隐藏层的节点数并不意味着一定可以提高模型的预测能力，其中需要更多的技巧来训练并发挥这些隐藏层的作用。我们将在接下来的几章中陆续介绍这些技巧。

图 4-13 展示了一个具有两个隐藏层的神经网络。输入层有 4 个节点，隐藏层 1 有 5 个节点，隐藏层 2 有 4 个节点，输出层有 1 个节点。神经网络模型一般是包含截距项的。但是，为了方便，有时候我们将不在图形里体现截距项。

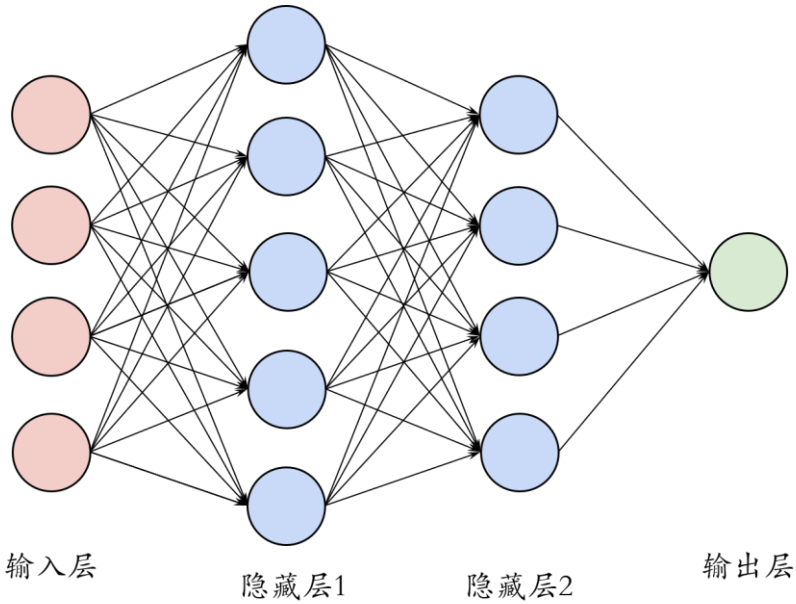


图 4-13 具有两个隐藏层的神经网络

4.2 正向传播算法

正向传播算法（Forward Propagation，FP）指输入值通过神经网络得到输出值的方法。我们以图 4-14 中具有 1 个隐藏层的神经网络为例说明正向传播算法的计算过程。从左到右依次把神经网络的各个层记为$\boldsymbol{l}_0$，$\boldsymbol{l}_1$，l_2（在这里，$\boldsymbol{l}_0$和$\boldsymbol{l}_1$为行向量，l_2为一个数；在代码中，$\boldsymbol{l}_0$，$\boldsymbol{l}_1$，l_2记为 layer_0，layer_1，layer_2），即，

$$\boldsymbol{l}_0 = (x_1 \quad x_2), \quad \boldsymbol{l}_1 = (h_1 \quad h_2), \quad l_2 = y$$

- 从$\boldsymbol{l}_0$到$\boldsymbol{l}_1$，虚线箭头表示从输入层x_1，x_2到h_1的信息传递（h_1由x_1，x_2计算所得，因此也可以理解为输入层x_1，x_2的信息传递给了隐藏层的h_1）；实线箭头表示从输入层x_1，x_2到h_2的信息传递。
  - 截距项记为$\boldsymbol{b}_{01}$（在代码中，$\boldsymbol{b}_{01}$记为 b_0_1），是一个1×2的行向量。$\boldsymbol{b}_{01}$的第一个元素表示从$\boldsymbol{l}_0$到h_1的截距项；$\boldsymbol{b}_{01}$的第二个元素表示从$\boldsymbol{l}_0$到h_2的截距项。
  - 权重记为$\boldsymbol{W}_{01}$（在代码中，$\boldsymbol{W}_{01}$记为 w_0_1），是一个2×2的矩阵。$\boldsymbol{W}_{01}$的第一列为$\boldsymbol{l}_0$到h_1的权重；$\boldsymbol{W}_{01}$的第二列为$\boldsymbol{l}_0$到h_2的权重。

$$\boldsymbol{W}_{01} = \begin{pmatrix} (\boldsymbol{W}_{01})_{11} & (\boldsymbol{W}_{01})_{12} \\ (\boldsymbol{W}_{01})_{21} & (\boldsymbol{W}_{01})_{22} \end{pmatrix}$$

- 从$\boldsymbol{l}_1$到$\boldsymbol{l}_2$，h_1，h_2到y的信息传递。
  - 截距项记为b_{12}（在代码中，b_{12}记为 b_1_2），是一个数。
  - 权重记为$\boldsymbol{W}_{12}$（在代码中，$\boldsymbol{W}_{12}$记为 w_1_2），是一个2×1的矩阵。

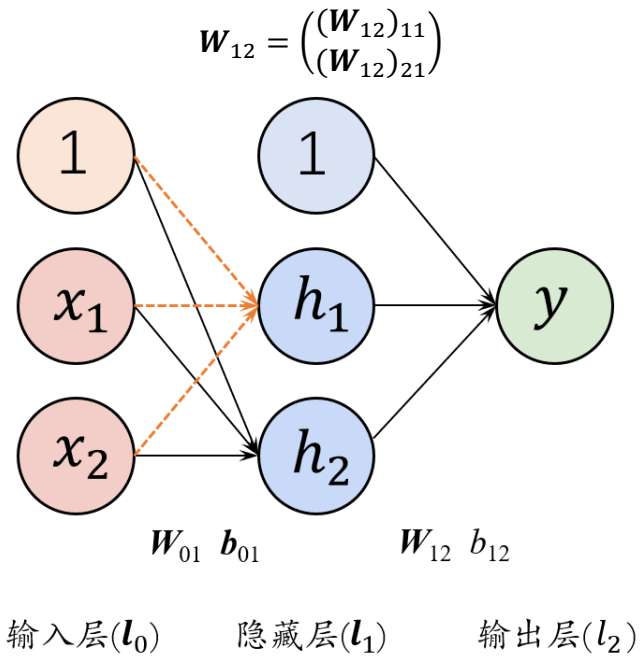


图 4-14 正向传播算法

如图 4-14 所示，从$\boldsymbol{l}_0$开始，当$\boldsymbol{b}_{01}$，$\boldsymbol{W}_{01}$已知，便可以计算$\boldsymbol{l}_1$；接着，当b_{12}，$\boldsymbol{W}_{12}$已知，便可以计算$\boldsymbol{l}_2$，得到神经网络模型的预测。从输入层到输出层的逐层计算过程即为正向传播算法。正向传播算法也可以用如图 4-15 所示的计算图详细表示。

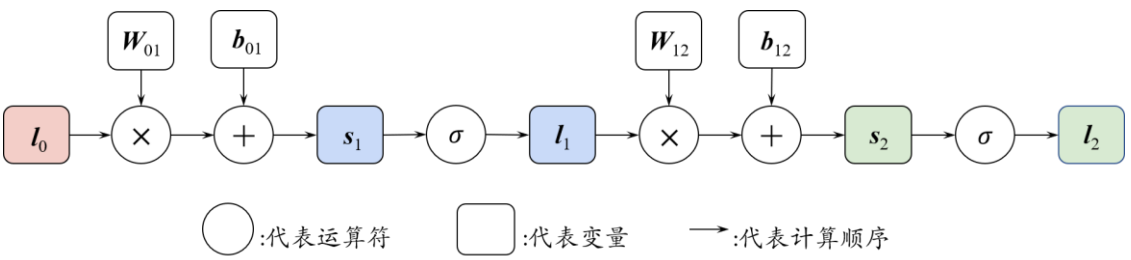


图 4-15 正向传播算法计算图

- 从输入层到隐藏层（从$\boldsymbol{l}_0$到$\boldsymbol{l}_1$）。公式下面的数字表示向量或者矩阵的维度。$\boldsymbol{l}_0$的加权和为$\boldsymbol{s}_1$，$\boldsymbol{s}_1$通过激活函数得到隐藏层$\boldsymbol{l}_1$。

$$\underset{1\times 2}{\boldsymbol{s}_1} = \underset{1\times 2}{\boldsymbol{l}_0}\ \underset{2\times 2}{\boldsymbol{W}_{01}} + \underset{1\times 2}{\boldsymbol{b}_{01}},\quad \underset{1\times 2}{\boldsymbol{l}_1} = \underset{1\times 2}{\sigma(\boldsymbol{s}_1)}$$

σ表示 sigmoid 函数，当 sigmoid 函数的输入值为向量或者矩阵时，函数将作用于向量或者矩阵的所有元素。

例如，当$\boldsymbol{l}_0 = (0 \quad 1)$，$\boldsymbol{b}_{01} = (0 \quad 0)$，

$$\boldsymbol{W}_{01} = \begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{pmatrix}$$

可以得到

$$\boldsymbol{s}_1 = (0 \quad 1)\begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{pmatrix} + (0 \quad 0) = (0.3 \quad 0.4)$$

$$\boldsymbol{l}_1 = \sigma\big((0.3 \quad 0.4)\big) = \left(\frac{1}{1+\mathrm{e}^{-0.3}} \ \frac{1}{1+\mathrm{e}^{-0.4}}\right) = (0.574 \quad 0.599)$$

在 Python 中，可以通过函数 `np.dot()`计算矩阵乘法，然后通过自定义函数 `sigmoid()`得到$\boldsymbol{l}_1$。这里以数据 2 为例，`layer_0` 为数据 2 的第二个观测点。

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

b_0_1 = np.zeros((1, 2))                          # 初始化截距项 b_0_1
w_0_1 = np.array([[0.1,0.2],[0.3,0.4]])           # 初始化权重 w_0_1

x, y = createDataSet_2()                          # 产生数据
layer_0 = x[1:2]                                  # 第二个观测点

print("b_0_1:\n"+str(b_0_1))
print("w_0_1:\n"+str(w_0_1))
print("layer_0:\n"+str(layer_0))

# ***** 计算 layer_1
layer_1 = sigmoid(np.dot(layer_0, w_0_1) + b_0_1)
print("----------------------\nlayer_1:\n"+str(layer_1))
```

```
b_0_1:
[[0. 0.]]
w_0_1:
[[0.1 0.2]
 [0.3 0.4]]
layer_0:
[[0 1]]
--------------------------
layer_1:
[[0.57444252 0.59868766]]
```

- 从隐藏层到输出层（从$\boldsymbol{l}_1$到l_2）。$\boldsymbol{l}_1$的加权和为s_2，s_2通过激活函数得到输入层l_2。

$$\underset{1\times 1}{s_2} = \underset{1\times 2}{\boldsymbol{l}_1}\ \underset{2\times 1}{\boldsymbol{W}_{12}} + \underset{1\times 1}{b_{12}}, \quad \underset{1\times 1}{l_2} = \underset{1\times 1}{\sigma(s_2)}$$

当$b_{12} = 0$，$\boldsymbol{W}_{12} = \begin{pmatrix} 0.1 \\ 0.2 \end{pmatrix}$，可以得到$\boldsymbol{s}_2 = (0.574 \quad 0.599)\begin{pmatrix} 0.1 \\ 0.2 \end{pmatrix} + 0 = 0.177$，$l_2 = \sigma(s_2) = 1/(1+\mathrm{e}^{-0.177}) = 0.544$。在 Python 中，可以通过如下方式计算$l_2$。

```
b_1_2 = 0
w_1_2 = np.array([0.1,0.2])
print("b_1_2:\n"+str(b_1_2))
```

```
print("w_1_2:\n"+str(w_1_2))
print("layer_1:\n"+str(layer_1))

# ***** 计算 layer_2
layer_2 = sigmoid(np.dot(layer_1, w_1_2) + b_1_2)
print("----------------------------\nlayer_2:\n"+str(layer_2))
```

```
b_1_2:

0
w_1_2:

[0.1 0.2]
layer_1:

[[0.57444252 0.59868766]]
----------------------------
layer_2:

[0.54417993]
```

到目前为止，我们从输入层开始，逐层计算加权和，然后代入激活函数，最终得到输出值。现在，可以根据因变量类型选择合适方式计算模型的损失函数，L。在第 3 章中，我们介绍过

- 如果因变量是定量数据，那么，损失函数可以是$L = \frac{1}{2}(y - \hat{y})^2$。这里$\hat{y}$为输出层的值。
- 如果因变量是二分类定性变量，那么，损失函数可以是$L = -y\log(p) - (1 - y)\log(1 - p)$。这里，$p$为输出层的值。

在图 4-14 的神经网络中，正向传播算法计算图可以拓展为图 4-16。即，得到l_2后，可以通过ℓ计算损失函数L，其中ℓ表示求解损失函数的运算。

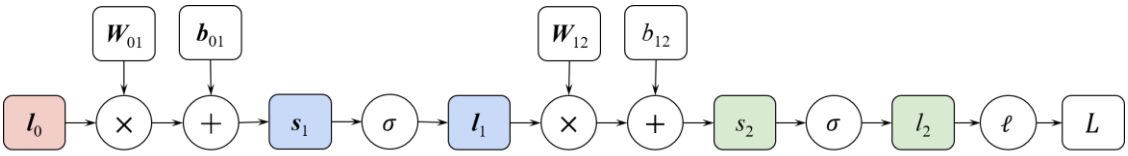


图 4-16 正向传播算法计算图（包含损失函数）

通过上面例子可以总结如下规律：除了输入层，神经网络每一层的值都由前一层的值乘以相应的权重矩阵，加上截距项，然后代入激活函数得到。

对于一般神经网络，我们可以用总结的规律正向逐层计算。例如，对于图 4-13 所示的神经网络模型，记$\boldsymbol{l}_0$为输入层，$\boldsymbol{l}_1$为隐藏层 1，$\boldsymbol{l}_2$为隐藏层 2，l_3为输出层。那么，

- $\boldsymbol{l}_0$为1×4的行向量；

- $\boldsymbol{l}_1 = \sigma(\boldsymbol{l}_0\boldsymbol{W}_{01} + \boldsymbol{b}_{01})$，$\boldsymbol{W}_{01}$为$4 \times 5$的矩阵，$\boldsymbol{b}_{01}$为$1 \times 5$的行向量，$\boldsymbol{l}_1$为$1 \times 5$的行向量；
- $\boldsymbol{l}_2 = \sigma(\boldsymbol{l}_1\boldsymbol{W}_{12} + \boldsymbol{b}_{12})$，$\boldsymbol{W}_{12}$为$5 \times 4$的矩阵，$\boldsymbol{b}_{12}$为$1 \times 4$的行向量，$\boldsymbol{l}_2$为$1 \times 4$的行向量；
- $l_3 = \sigma(\boldsymbol{l}_2\boldsymbol{W}_{23} + b_{23})$，$\boldsymbol{W}_{23}$为$4 \times 1$的矩阵，$b_{23}$为一个数，$l_3$为一个数；
- 最后，计算损失函数$L = \ell(l_3, y)$。

4.3 反向传播算法

为了训练神经网络，需要反复更新神经网络的参数，使神经网络的损失函数L变小。从第 3 章我们知道，梯度下降法可以实现该任务。参数梯度表示损失函数下降的方向和大小，是实现梯度下降法的重要部分。反向传播算法（Backward Propagation，BP）是神经网络中逐层计算参数梯度的方法。我们以图 4-14 所示的具有 1 个隐藏层的神经网络为例介绍反向传播算法。

首先，用正向传播算法得到输出值，

- $\boldsymbol{l}_1 = \sigma(\boldsymbol{l}_0\boldsymbol{W}_{01} + \boldsymbol{b}_{01})$
- $\boldsymbol{l}_2 = \sigma(\boldsymbol{l}_1\boldsymbol{W}_{12} + b_{12})$

这里σ为 sigmoid 函数，$\boldsymbol{W}_{01}$为一个2×2的矩阵，$\boldsymbol{b}_{01}$为一个1×2的向量，$\boldsymbol{W}_{12}$为一个2×1的矩阵，b_{12}为一个数。上面第一个式子代入第二个式子可以得到

$$l_2 = \sigma(\sigma(\boldsymbol{l}_0\boldsymbol{W}_{01} + \boldsymbol{b}_{01})\boldsymbol{W}_{12} + b_{12})$$

l_2是权重$\boldsymbol{W}_{01}$，$\boldsymbol{W}_{12}$和截距项$\boldsymbol{b}_{01}$，b_{12}的函数。以因变量是二分类定性变量为例，神经网络模型的损失函数为

$$L = -y\log(l_2) - (1-y)\log(1-l_2)$$

因此，损失函数L为权重$\boldsymbol{W}_{01}$，$\boldsymbol{W}_{12}$和截距项$\boldsymbol{b}_{01}$，b_{12}的函数。在梯度下降法中需要得到L关于$\boldsymbol{W}_{01}$，$\boldsymbol{W}_{12}$，$\boldsymbol{b}_{01}$，b_{12}的偏导数。为了更容易解释反向传播算法，我们把正向传播算法的过程写成如下更详细的形式，

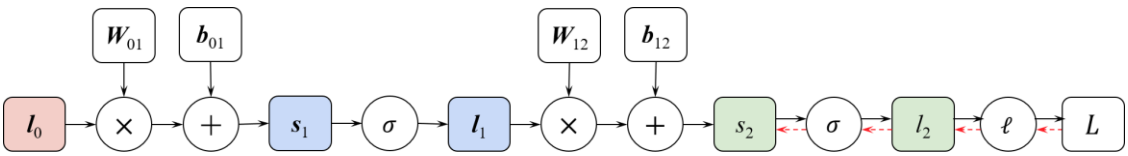
- $\boldsymbol{s}_1 = \boldsymbol{l}_0\boldsymbol{W}_{01} + \boldsymbol{b}_{01}$
- $\boldsymbol{l}_1 = \sigma(\boldsymbol{s}_1)$
- $s_2 = \boldsymbol{l}_1\boldsymbol{W}_{12} + b_{12}$
- $l_2 = \sigma(s_2)$
- $L = \ell(l_2, y)$

现在，我们用反向传播算法计算损失函数L关于$\boldsymbol{W}_{01}$，$\boldsymbol{W}_{12}$，$\boldsymbol{b}_{01}$，b_{12}的偏导数，整个过程分为 4 个步骤。

- 步骤 1：求损失函数L关于s_2的偏导数，记为 $\delta_2 = \frac{\partial L}{\partial s_2}$。
- 步骤 2：求损失函数L关于$\boldsymbol{W}_{12}$和b_{12}的偏导数，记为$\nabla_{\boldsymbol{W}_{12}} = \frac{\partial L}{\partial \boldsymbol{W}_{12}}$和$\nabla_{b_{12}} = \frac{\partial L}{\partial b_{12}}$。
- 步骤 3：求损失函数L关于$\boldsymbol{s}_1$的偏导数，记为$\boldsymbol{\delta}_1 = \frac{\partial L}{\partial \boldsymbol{s}_1}$。
- 步骤 4：求损失函数L关于$\boldsymbol{W}_{01}$和$\boldsymbol{b}_{01}$的偏导数，记为$\nabla_{\boldsymbol{W}_{01}} = \frac{\partial L}{\partial \boldsymbol{W}_{01}}$和$\nabla_{\boldsymbol{b}_{01}} = \frac{\partial L}{\partial \boldsymbol{b}_{01}}$。

步骤 1：计算δ_2

δ_2为损失函数L关于s_2的偏导数，$\delta_2 = \frac{\partial L}{\partial s_2}$。根据正向传播算法，我们知道$s_2$通过 sigmoid 函数变换成$l_2$，然后计算损失函数$L$。因此，$\delta_2$可以沿着图 4-17 虚线箭头所示方向，通过链式法则得到。



步骤1：计算δ_2

图 4-17 反向传播算法计算图（计算δ_2）

通过链式法则，我们知道$\delta_2 = \frac{\partial L}{\partial s_2} = \frac{\partial L}{\partial l_2}\frac{\partial l_2}{\partial s_2}$。容易求得

$$\frac{\partial L}{\partial l_2} = \frac{\partial\{-y\log(l_2)-(1-y)\log(1-l_2)\}}{\partial l_2} = -\frac{y}{l_2} + \frac{1-y}{1-l_2}, \quad \frac{\partial l_2}{\partial s_2} = l_2(1-l_2)$$

因此，

$$\begin{aligned} \underset{1\times 1}{\delta_2} &= (-\frac{y}{l_2} + \frac{1-y}{1-l_2})(l_2(1-l_2)) \\ &= \underset{1\times 1}{l_2} - \underset{1\times 1}{y} \end{aligned}$$

也就是说，δ_2等于预测值l_2与真实值y的差。在这个例子中，$L = -y\log(l_2) - (1-y)\log(1-l_2)$，我们得到$\delta_2$等于预测值$l_2$与真实值$y$的差。但是，当损失函数定义不同时，$\delta_2$有可能是不一样的。因此，需要注意$\delta_2$的具体形式取决于损失函数的定义。

在 4.2 节的例子中，`layer_2=0.544`，`y=0`，因此，

- $L = -0 \times \log(0.544) - (1-0) \times \log(1-0.544) = 0.786$

- $\delta_2 = l_2 - y = 0.544 - 0 = 0.544$

在 Python 中，该运算可以通过如下代码实现。

```
target = y[1:2]
loss = -(target * np.log(layer_2) + \
       (1-target) * np.log(1-layer_2))          #模型误差
print("layer_2:\n"+str(layer_2))
print("y:\n"+str(target))
print("loss:\n"+str(loss))

# ***** 计算 delta_2
delta_2 = layer_2 - target
print("----------------------------\ndetla_2:\n"+str(delta_2))

layer_2:
[0.54417993]

y:
[0]

loss:
[0.78565712]
----------------------------

detla_2:
[0.54417993]
```

步骤 2：计算$\nabla_{W_{12}}$和$\nabla_{b_{12}}$

现在计算L关于$\boldsymbol{W}_{12}$和b_{12}的偏导数，记$\nabla_{\boldsymbol{W}_{12}} = \frac{\partial L}{\partial \boldsymbol{W}_{12}}$和$\nabla_{b_{12}} = \frac{\partial L}{\partial b_{12}}$。根据正向传播算法，我们知道$s_2 = \boldsymbol{l}_1\boldsymbol{W}_{12} + b_{12}$。也就是说，$\boldsymbol{l}_1$通过与$\boldsymbol{W}_{12}$和$b_{12}$的运算得到$s_2$，进而经过一系列计算得到损失函数$L$。因此，$\nabla_{\boldsymbol{W}_{12}}$和$\nabla_{b_{12}}$可以沿着图 4-18 虚线箭头所示方向，通过链式法则得到。

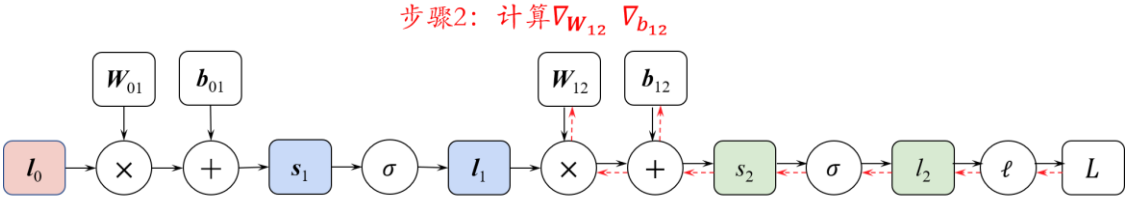


图 4-18 反向传播算法计算图（计算$\nabla_{W_{12}}$和$\nabla_{b_{12}}$）

根据链式法则可知$\nabla_{\boldsymbol{W}_{12}} = \frac{\partial L}{\partial s_2}\frac{\partial s_2}{\partial \boldsymbol{W}_{12}}$，$\frac{\partial L}{\partial b_{12}} = \frac{\partial L}{\partial s_2}\frac{\partial s_2}{\partial b_{12}}$。在步骤 1，已经得到$\delta_2 = \frac{\partial L}{\partial s_2}$。因此，只需要计算$\frac{\partial s_2}{\partial \boldsymbol{W}_{12}}$和$\frac{\partial s_2}{\partial b_{12}}$。 我们知道$s_2 = \boldsymbol{l}_1\boldsymbol{W}_{12} + b_{12}$，即

$$s_2 = ((\boldsymbol{l}_1)_1 \quad (\boldsymbol{l}_1)_2)\begin{pmatrix}(\boldsymbol{W}_{12})_{11}\\(\boldsymbol{W}_{12})_{21}\end{pmatrix} + b_{12}$$

简单计算可以得到，$s_2 = (\boldsymbol{l}_1)_1(\boldsymbol{W}_{12})_{11} + (\boldsymbol{l}_1)_2(\boldsymbol{W}_{12})_{21} + b_{12}$。因此，

$$\frac{\partial s_2}{\partial (\boldsymbol{W}_{12})_{11}} = (\boldsymbol{l}_1)_1，\ \frac{\partial s_2}{\partial (\boldsymbol{W}_{12})_{21}} = (\boldsymbol{l}_1)_2，\ \frac{\partial s_2}{\partial b_{12}} = 1$$

也就是说

$$\frac{\partial s_2}{\partial \boldsymbol{W}_{12}} = \begin{pmatrix} (\boldsymbol{l}_1)_1 \\ (\boldsymbol{l}_1)_2 \end{pmatrix} = \boldsymbol{l}_1^{\mathrm{T}}$$

把$\frac{\partial s_2}{\partial \boldsymbol{W}_{12}} = \boldsymbol{l}_1^{\mathrm{T}}$和$\frac{\partial s_2}{\partial b_{12}} = 1$分别代入$\nabla_{\boldsymbol{W}_{12}} = \frac{\partial L}{\partial s_2}\frac{\partial s_2}{\partial \boldsymbol{W}_{12}}$，$\frac{\partial L}{\partial b_{12}} = \frac{\partial L}{\partial s_2}\frac{\partial s_2}{\partial b_{12}}$，可以得到，

$$\underset{2\times 1}{\nabla_{\boldsymbol{W}_{12}}} = \underset{2\times 1}{\boldsymbol{l}_1^{\mathrm{T}}}\ \underset{1\times 1}{\delta_2}，\quad \underset{1\times 1}{\nabla_{b_{12}}} = \underset{1\times 1}{\delta_2}$$

接着上面的例子，我们可以如下计算$\nabla_{\boldsymbol{W}_{12}}$和$\nabla_{b_{12}}$。

- $\nabla_{\boldsymbol{W}_{12}} = \begin{pmatrix} 0.5744 \\ 0.5987 \end{pmatrix} \times 0.544 = \begin{pmatrix} 0.3126 \\ 0.3258 \end{pmatrix}$
- $\nabla_{b_{12}} = 0.544$

在 Python 中，可以用如下代码计算$\nabla_{\boldsymbol{W}_{12}}$和$\nabla_{b_{12}}$，

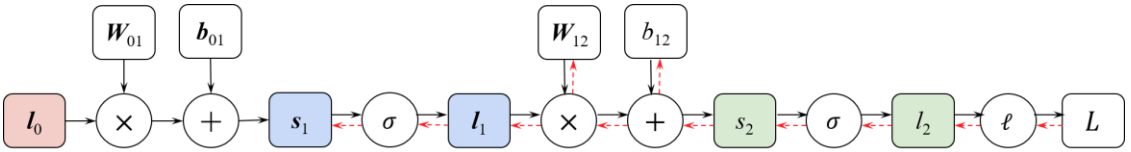
```
# ***** 计算损失函数关于 W_12 和 b_12 的偏导数
dw_1_2 = layer_1.T.dot(delta_2)
db_1_2 = delta_2

print("损失函数关于 W_12 的偏导数:")
print(dw_1_2)
print("--------------------")
print("损失函数关于 b_12 的偏导数:")
print(db_1_2)
```

```
损失函数关于 W_12 的偏导数:
[0.31260009 0.32579381]
--------------------
损失函数关于 b_12 的偏导数:
[0.54417993]
```

步骤 3：计算$\boldsymbol{\delta}_1$

$\boldsymbol{\delta}_1$为损失函数L关于$\boldsymbol{s}_1$的偏导数，记$\boldsymbol{\delta}_1 = \frac{\partial L}{\partial \boldsymbol{s}_1}$。根据正向传播算法，我们知道$\boldsymbol{s}_1$通过 sigmoid 函数变换成$\boldsymbol{l}_1$，然后经过一系列运算得到损失函数$L$。因此，$\boldsymbol{\delta}_1$可以沿着图 4-19 虚线箭头所示方向，通过链式法则得到。



步骤3：计算$\boldsymbol{\delta_1}$

图 4-19 反向传播算法计算图（计算δ_1）

根据链式法则可知$\boldsymbol{\delta_1} = \frac{\partial L}{\partial \boldsymbol{s_1}} = \frac{\partial L}{\partial s_2}\frac{\partial s_2}{\partial \boldsymbol{l_1}}\frac{\partial \boldsymbol{l_1}}{\partial \boldsymbol{s_1}}$。已知$\frac{\partial L}{\partial s_2}$，且容易得到$\frac{\partial \boldsymbol{l_1}}{\partial \boldsymbol{s_1}} = \boldsymbol{l_1} \circ (1 - \boldsymbol{l_1})$。因此，这里只需要计算$\frac{\partial s_2}{\partial \boldsymbol{l_1}}$。我们知道$s_2 = (\boldsymbol{l_1})_1(\boldsymbol{W_{12}})_{11} + (\boldsymbol{l_1})_2(\boldsymbol{W_{12}})_{21} + b_{12}$，简单计算可以得到$\frac{\partial s_2}{\partial (\boldsymbol{l_1})_1} = (\boldsymbol{W_{12}})_{11}$和$\frac{\partial s_2}{\partial (\boldsymbol{l_1})_2} = (\boldsymbol{W_{12}})_{21}$，即$\frac{\partial s_2}{\partial \boldsymbol{l_1}} = ((\boldsymbol{W_{12}})_{11} \quad (\boldsymbol{W_{12}})_{21}) = \boldsymbol{W_{12}^{\mathsf{T}}}$。把$\frac{\partial L}{\partial s_2} = \delta_2$，$\frac{\partial s_2}{\partial \boldsymbol{l_1}} = \boldsymbol{W_{12}^{\mathsf{T}}}$，$\frac{\partial \boldsymbol{l_1}}{\partial \boldsymbol{s_1}} = \boldsymbol{l_1} \circ (1 - \boldsymbol{l_1})$代入$\boldsymbol{\delta_1} = \frac{\partial L}{\partial \boldsymbol{s_1}} = \frac{\partial L}{\partial s_2}\frac{\partial s_2}{\partial \boldsymbol{l_1}}\frac{\partial \boldsymbol{l_1}}{\partial \boldsymbol{s_1}}$，可以得到

$$\underset{1\times 2}{\boldsymbol{\delta_1}} = (\underset{1\times 1}{\delta_2} \quad \underset{1\times 2}{\boldsymbol{W_{12}^{\mathsf{T}}}}) \circ (\underset{1\times 2}{\boldsymbol{l_1} \circ (1 - \boldsymbol{l_1})})$$

也就是说，$\boldsymbol{\delta_1}$为δ_2乘以$\boldsymbol{l_1}$到$\boldsymbol{l_2}$的权重的转置再逐点乘以$\boldsymbol{l_1}$关于$\boldsymbol{s_1}$的偏导数。

接着上面例子，我们可以计算$\boldsymbol{\delta_1}$如下。

$$\begin{aligned}\boldsymbol{\delta_1} &= (\delta_2 \boldsymbol{W_{12}^{\mathsf{T}}}) \circ \left(\boldsymbol{l_1} \circ (1 - \boldsymbol{l_1})\right) \\ &= \left(0.544 \times \begin{pmatrix} -12.41 \\ 12.11 \end{pmatrix}^{\mathsf{T}}\right) \circ \left((0.5744 \quad 0.5987) \circ \left(1 - (0.5744 \quad 0.5987)\right)\right) \\ &= (-1.651 \quad 1.584)\end{aligned}$$

下面定义函数 `sigmoid2deriv()`，该函数可以计算 sigmoid 函数的导数。计算$\boldsymbol{\delta_1}$的具体代码如下。

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

# ***** 计算 delta_1
delta_1 = delta_2.dot(weights_1_2.T) * sigmoid2deriv(layer_1)
print("delta_1:\n"+str(delta_1))

delta_1:
[[-1.65108245  1.58364831]]
```

步骤 4：计算$\nabla_{\boldsymbol{W_{01}}}$和$\nabla_{\boldsymbol{b_{01}}}$

现在计算L关于$\boldsymbol{W_{01}}$和$\boldsymbol{b_{01}}$的偏导数，记$\nabla_{\boldsymbol{W_{01}}} = \frac{\partial L}{\partial \boldsymbol{W_{01}}}$和$\nabla_{\boldsymbol{b_{01}}} = \frac{\partial L}{\partial \boldsymbol{b_{01}}}$。根据正向传播算法可知$\boldsymbol{s_1} = \boldsymbol{l_0}\boldsymbol{W_{01}} + \boldsymbol{b_{01}}$。也就是说，$\boldsymbol{l_0}$通过与$\boldsymbol{W_{01}}$和$\boldsymbol{b_{01}}$运算得到$\boldsymbol{s_1}$，进

而经过一系列计算得到损失函数L。因此，$\nabla_{\boldsymbol{W}_{01}}$和$\nabla_{\boldsymbol{b}_{01}}$可以沿着图 4-20 虚线箭头所示方向，通过链式法则得到。

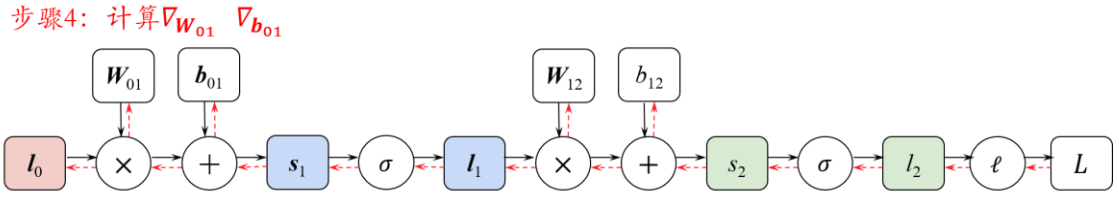


图 4-20 反向传播算法计算图（计算$\nabla_{\boldsymbol{W}_{01}}$和$\nabla_{\boldsymbol{b}_{01}}$）

根据链式法则可知$\nabla_{\boldsymbol{W}_{01}} = \frac{\partial L}{\partial \boldsymbol{s}_1}\frac{\partial \boldsymbol{s}_1}{\partial \boldsymbol{W}_{01}}$，$\nabla_{\boldsymbol{b}_{01}} = \frac{\partial L}{\partial \boldsymbol{s}_1}\frac{\partial \boldsymbol{s}_1}{\partial \boldsymbol{b}_{01}}$。在步骤 3 中已经得到$\boldsymbol{\delta}_1 = \frac{\partial L}{\partial \boldsymbol{s}_1}$。在这里，需要计算$\frac{\partial \boldsymbol{s}_1}{\partial \boldsymbol{W}_{01}}$。因为$\boldsymbol{s}_1$是一个$1 \times 2$的向量，$\boldsymbol{W}_{01}$是一个$2 \times 2$的矩阵，所以，理论上$\frac{\partial \boldsymbol{s}_1}{\partial \boldsymbol{W}_{01}}$应该是一个$2 \times 2 \times 2$的多维数组（因为$\boldsymbol{s}_1$的每个元素关于$\boldsymbol{W}_{01}$的偏导数都是一个$2 \times 2$的矩阵）。

但是，实际运算$\frac{\partial \boldsymbol{s}_1}{\partial \boldsymbol{W}_{01}}$的过程中，可能不需要涉及复杂的多维数组。接下来，我们将逐步介绍计算$\frac{\partial \boldsymbol{s}_1}{\partial \boldsymbol{W}_{01}}$的详细过程。已知$\boldsymbol{s}_1 = \boldsymbol{l}_0\boldsymbol{W}_{01} + \boldsymbol{b}_{01}$，即

$$((\boldsymbol{s}_1)_1 \quad (\boldsymbol{s}_1)_2) = ((\boldsymbol{l}_0)_1 \quad (\boldsymbol{l}_0)_2)\begin{pmatrix}(\boldsymbol{W}_{01})_{11} & (\boldsymbol{W}_{01})_{12}\\ (\boldsymbol{W}_{01})_{21} & (\boldsymbol{W}_{01})_{22}\end{pmatrix} + ((\boldsymbol{b}_{01})_1 \quad (\boldsymbol{b}_{01})_2)$$

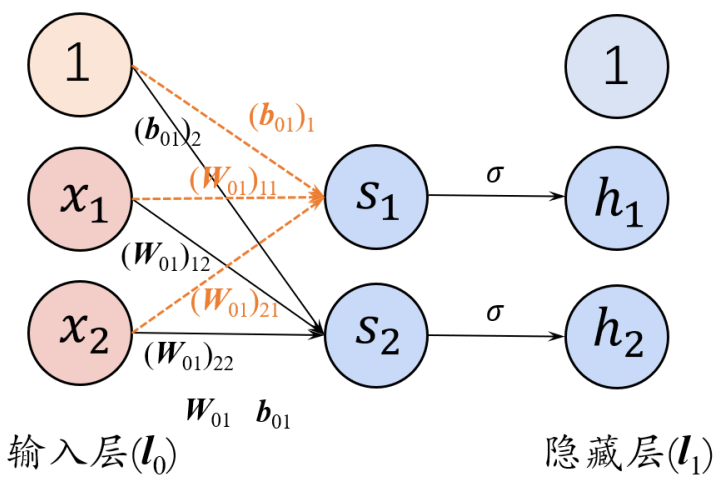


图 4-21 从输入层（$\boldsymbol{l}_0$）到隐藏层（$\boldsymbol{l}_1$）详细计算过程

如图 4-21 所示，$(\boldsymbol{s}_1)_1 = (\boldsymbol{l}_0)_1(\boldsymbol{W}_{01})_{11} + (\boldsymbol{l}_0)_2(\boldsymbol{W}_{01})_{21} + (\boldsymbol{b}_{01})_1$，$(\boldsymbol{s}_1)_2 = (\boldsymbol{l}_0)_1(\boldsymbol{W}_{01})_{12} + (\boldsymbol{l}_0)_2(\boldsymbol{W}_{01})_{22} + (\boldsymbol{b}_{01})_2$。可以得到

- $\frac{\partial(\boldsymbol{s_1})_1}{\partial(\boldsymbol{W_{01}})_{11}}=(\boldsymbol{l_0})_1$，$\frac{\partial(\boldsymbol{s_1})_1}{\partial(\boldsymbol{W_{01}})_{21}}=(\boldsymbol{l_0})_2$，$\frac{\partial(\boldsymbol{s_1})_2}{\partial(\boldsymbol{W_{01}})_{12}}=(l_0)_1$，$\frac{\partial(\boldsymbol{s_1})_2}{\partial(\boldsymbol{W_{01}})_{22}}=(\boldsymbol{l_0})_2$，$\frac{\partial(\boldsymbol{s_1})_1}{\partial(\boldsymbol{b_{01}})_1}=1$，$\frac{\partial(\boldsymbol{s_1})_2}{\partial(\boldsymbol{b_{01}})_2}=1$
- $\frac{\partial(\boldsymbol{s_1})_2}{\partial(\boldsymbol{W_{01}})_{11}}=0$，$\frac{\partial(\boldsymbol{s_1})_2}{\partial(\boldsymbol{W_{01}})_{21}}=0$，$\frac{\partial(\boldsymbol{s_1})_1}{\partial(\boldsymbol{W_{01}})_{12}}=0$，$\frac{\partial(\boldsymbol{s_1})_1}{\partial(\boldsymbol{W_{01}})_{22}}=0$，$\frac{\partial(\boldsymbol{s_1})_2}{\partial(\boldsymbol{b_{01}})_1}=0$，$\frac{\partial(\boldsymbol{s_1})_1}{\partial(\boldsymbol{b_{01}})_2}=0$

虽然$\frac{\partial \boldsymbol{s_1}}{\partial \boldsymbol{W_{01}}}$应该是一个$2\times2\times2$的多维数组，但是$\frac{\partial \boldsymbol{s_1}}{\partial \boldsymbol{W_{01}}}$中有一半的元素为 0。从图 4-21 也可以看到，$(\boldsymbol{W_{01}})_{11}$和$(\boldsymbol{W_{01}})_{21}$与$(\boldsymbol{s_1})_2$没有关系，$(\boldsymbol{W_{01}})_{12}$和$(\boldsymbol{W_{01}})_{22}$与$(\boldsymbol{s_1})_1$没有关系，因此，$\frac{\partial(\boldsymbol{s_1})_2}{\partial(\boldsymbol{W_{01}})_{11}}$，$\frac{\partial(\boldsymbol{s_1})_2}{\partial(\boldsymbol{W_{01}})_{21}}$，$\frac{\partial(\boldsymbol{s_1})_1}{\partial(\boldsymbol{W_{01}})_{12}}$，$\frac{\partial(\boldsymbol{s_1})_1}{\partial(\boldsymbol{W_{01}})_{22}}$都等于 0。同样的，虽然理论上$\frac{\partial \boldsymbol{s_1}}{\partial \boldsymbol{b_{01}}}$应该是一个$2\times2$的矩阵，但是$\frac{\partial \boldsymbol{s_1}}{\partial \boldsymbol{b_{01}}}$中有一半的元素为 0，即$\frac{\partial(\boldsymbol{s_1})_2}{\partial(\boldsymbol{b_{01}})_1}$，$\frac{\partial(\boldsymbol{s_1})_1}{\partial(\boldsymbol{b_{01}})_2}$都等于 0。

基于这些观察，我们不直接使用$\nabla_{\boldsymbol{W_{01}}}=\frac{\partial L}{\partial \boldsymbol{s_1}}\frac{\partial \boldsymbol{s_1}}{\partial \boldsymbol{W_{01}}}$，$\nabla_{\boldsymbol{b_{01}}}=\frac{\partial L}{\partial \boldsymbol{s_1}}\frac{\partial \boldsymbol{s_1}}{\partial \boldsymbol{b_{01}}}$计算$\nabla_{\boldsymbol{W_{01}}}$和$\nabla_{\boldsymbol{b_{01}}}$。因为把$\boldsymbol{s_1}$的元素关于$\boldsymbol{W_{01}}$的所有元素的偏导数排成$2\times2\times2$的多维数会很麻烦，而且我们也没有定义矩阵和多维数组的乘法。在这里，损失函数L是一个数，而$\boldsymbol{W_{01}}$是一个2×2的矩阵，$\boldsymbol{b_{01}}$是一个1×2的矩阵；可以知道$\nabla_{\boldsymbol{W_{01}}}$是一个$2\times2$的矩阵，$\nabla_{\boldsymbol{b_{01}}}$是一个$1\times2$的矩阵。我们可以先求出$\nabla_{\boldsymbol{W_{01}}}$和$\nabla_{\boldsymbol{b_{01}}}$的所有元素。

根据链式法则可以得到

- $\frac{\partial L}{\partial(\boldsymbol{W_{01}})_{11}}=\frac{\partial L}{\partial(\boldsymbol{s_1})_1}\frac{\partial(\boldsymbol{s_1})_1}{\partial(\boldsymbol{W_{01}})_{11}}=(\boldsymbol{\delta_1})_1(\boldsymbol{l_0})_1$
- $\frac{\partial L}{\partial(\boldsymbol{W_{01}})_{21}}=\frac{\partial L}{\partial(\boldsymbol{s_1})_1}\frac{\partial(\boldsymbol{s_1})_1}{\partial(\boldsymbol{W_{01}})_{21}}=(\boldsymbol{\delta_1})_1(\boldsymbol{l_0})_2$
- $\frac{\partial L}{\partial(\boldsymbol{W_{01}})_{12}}=\frac{\partial L}{\partial(\boldsymbol{s_1})_2}\frac{\partial(\boldsymbol{s_1})_2}{\partial(\boldsymbol{W_{01}})_{12}}=(\boldsymbol{\delta_1})_2(\boldsymbol{l_0})_1$
- $\frac{\partial L}{\partial(\boldsymbol{W_{01}})_{22}}=\frac{\partial L}{\partial(\boldsymbol{s_1})_2}\frac{\partial(\boldsymbol{s_1})_2}{\partial(\boldsymbol{W_{01}})_{22}}=(\boldsymbol{\delta_1})_2(\boldsymbol{l_0})_2$
- $\frac{\partial L}{\partial(\boldsymbol{b_{01}})_1}=\frac{\partial L}{\partial(\boldsymbol{s_1})_1}\frac{\partial(\boldsymbol{s_1})_1}{\partial(\boldsymbol{b_{01}})_1}=(\boldsymbol{\delta_1})_1$
- $\frac{\partial L}{\partial(\boldsymbol{b_{01}})_2}=\frac{\partial L}{\partial(\boldsymbol{s_1})_2}\frac{\partial(\boldsymbol{s_1})_2}{\partial(\boldsymbol{b_{01}})_2}=(\boldsymbol{\delta_1})_2$

因此，可以得到L关于$\boldsymbol{W_{01}}$的偏导数

$$\frac{\partial L}{\partial \boldsymbol{W}_{01}} = \begin{pmatrix} \frac{\partial L}{\partial (\boldsymbol{W}_{01})_{11}} & \frac{\partial L}{\partial (\boldsymbol{W}_{01})_{12}} \\ \frac{\partial L}{\partial (\boldsymbol{W}_{01})_{21}} & \frac{\partial L}{\partial (\boldsymbol{W}_{01})_{22}} \end{pmatrix} = \begin{pmatrix} (\boldsymbol{\delta}_1)_1(\boldsymbol{l}_0)_1 & (\boldsymbol{\delta}_1)_2(\boldsymbol{l}_0)_1 \\ (\boldsymbol{\delta}_1)_1(\boldsymbol{l}_0)_2 & (\boldsymbol{\delta}_1)_2(\boldsymbol{l}_0)_2 \end{pmatrix} = \begin{pmatrix} (\boldsymbol{l}_0)_1 \\ (\boldsymbol{l}_0)_2 \end{pmatrix} ((\boldsymbol{\delta}_1)_1 \quad (\boldsymbol{\delta}_1)_2) = \boldsymbol{l}_0^{\mathrm{T}} \boldsymbol{\delta}_1$$

以及L关于$\boldsymbol{b}_{01}$的偏导数

$$\frac{\partial L}{\partial \boldsymbol{b}_{01}} = \begin{pmatrix} \frac{\partial L}{\partial (\boldsymbol{b}_{01})_1} & \frac{\partial L}{\partial (\boldsymbol{b}_{01})_2} \end{pmatrix} = ((\boldsymbol{\delta}_1)_1 \quad (\boldsymbol{\delta}_1)_2) = \boldsymbol{\delta}_1$$

因此，可以总结如下，

$$\underset{2\times 2}{\nabla_{\boldsymbol{W}_{01}}} = \underset{2\times 1}{\boldsymbol{l}_0^{\mathrm{T}}} \underset{1\times 2}{\boldsymbol{\delta}_1}, \quad \underset{1\times 2}{\nabla_{\boldsymbol{b}_{01}}} = \underset{1\times 2}{\boldsymbol{\delta}_1}$$

接着上面的例子，我们可以如下计算$\nabla_{\boldsymbol{W}_{01}}$和$\nabla_{\boldsymbol{b}_{01}}$。

- $\nabla_{\boldsymbol{W}_{01}} = (0 \quad 1)^{\mathrm{T}}(-1.65 \quad 1.58) = \begin{pmatrix} 0 & 0 \\ -1.65 & 1.58 \end{pmatrix}$
- $\nabla_{\boldsymbol{b}_{01}} = (-1.65 \quad 1.58)$

这个运算在 Python 中可以用如下代码实现。

```
# ***** 计算损失函数关于 W_01 和 b_01 的偏导数
derivate_w_01 = layer_0.T.dot(delta_1)
derivate_b_01 = delta_1

print("损失函数关于 weights_0_1 的偏导数:\n"+str(derivate_w_01))
print("损失函数关于 b_0_1 的偏导数:\n"+str(derivate_b_01))

损失函数关于 weights_0_1 的偏导数:
[[ 0.          0.        ]
 [-1.65108245  1.58364831]]
损失函数关于 b_0_1 的偏导数:
[[-1.65108245  1.58364831]]
```

综合正向传播算法和反向传播算法，对本节考虑的具有 1 个隐藏层的神经网络（图 4-14），可以通过正向传播算法得到输出层的值，进而计算损失函数$\boldsymbol{L}$。

- $\boldsymbol{s}_1 = \boldsymbol{l}_0\boldsymbol{W}_{01} + \boldsymbol{b}_{01}$
- $\boldsymbol{l}_1 = \sigma(\boldsymbol{s}_1)$
- $s_2 = \boldsymbol{l}_1\boldsymbol{W}_{12} + b_{12}$

- $l_2 = \sigma(s_2)$
- $L = \ell(l_2, y)$

然后通过反向传播算法计算所有参数的梯度。

- $\delta_2 = l_2 - y$，δ_2的维度为1×1。在这里，损失函数定义为$L = -y\log(p) - (1-y)\log(1-p)$；
- $\nabla_{\boldsymbol{W}_{12}} = \boldsymbol{l}_1^{\mathrm{T}}\delta_2$，$\nabla_{b_{12}} = \delta_2$，$\nabla_{\boldsymbol{W}_{12}}$的维度为$2 \times 1$，$\nabla_{b_{12}}$的维度为$1 \times 1$；
- $\boldsymbol{\delta}_1 = (\delta_2 \boldsymbol{W}_{12}^{\mathrm{T}}) \circ (\boldsymbol{l}_1 \circ (1 - \boldsymbol{l}_1))$，$\boldsymbol{\delta}_1$的维度为$1 \times 2$；
- $\nabla_{\boldsymbol{W}_{01}} = \boldsymbol{l}_0^{\mathrm{T}}\boldsymbol{\delta}_1$，$\nabla_{\boldsymbol{b}_{01}} = \boldsymbol{\delta}_1$，$\nabla_{\boldsymbol{W}_{01}}$的维度为$2 \times 2$，$\nabla_{\boldsymbol{b}_{01}}$的维度为$1 \times 2$。

在 4.4.2 节中，我们将进一步总结在一般的神经网络模型中反向传播算法的主要步骤。

4.4 神经网络训练完整流程

4.4.1 随机梯度下降法

基于正向传播算法、反向传播算法，我们可以利用神经网络计算预测值，并且计算所有参数的梯度。如图 4-14 所示的神经网络，随机梯度下降法可以通过如下步骤实现。

循环直至收敛

对于训练数据的每个观测点：

1. 用正向传播算法得到输出值l_2:
   - $\boldsymbol{s}_1 = \boldsymbol{l}_0\boldsymbol{W}_{01} + \boldsymbol{b}_{01}$
   - $\boldsymbol{l}_1 = \sigma(\boldsymbol{s}_1)$
   - $s_2 = \boldsymbol{l}_1\boldsymbol{W}_{12} + b_{12}$
   - $l_2 = \sigma(s_2)$
2. 计算损失函数，L；当因变量是二分类定性数据时，损失函数可以是$L = -y\log(l_2) - (1-y)\log(1-l_2)$
3. 计算$\delta_2 = l_2 - y$和$\boldsymbol{\delta}_1 = (\delta_2 \boldsymbol{W}_{12}^{\mathrm{T}}) \circ (\boldsymbol{l}_1 \circ (1 - \boldsymbol{l}_1))$
4. 计算参数梯度$\nabla_{\boldsymbol{W}_{12}} = \boldsymbol{l}_1^{\mathrm{T}}\delta_2$，$\nabla_{b_{12}} = \delta_2$，$\nabla_{\boldsymbol{W}_{01}} = \boldsymbol{l}_0^{\mathrm{T}}\boldsymbol{\delta}_1$，$\nabla_{\boldsymbol{b}_{01}} = \boldsymbol{\delta}_1$
5. 给定学习步长α，更新权重和截距项：
   - $\boldsymbol{W}_{12} = \boldsymbol{W}_{12} - \alpha\nabla_{\boldsymbol{W}_{12}}$
   - $\boldsymbol{W}_{01} = \boldsymbol{W}_{01} - \alpha\nabla_{\boldsymbol{W}_{01}}$
   - $b_{12} = b_{12} - \alpha\nabla b_{12}$
   - $\boldsymbol{b}_{01} = \boldsymbol{b}_{01} - \alpha\nabla\boldsymbol{b}_{01}$

对于数据 2，在 Python 中可以使用如下代码实现完整的神经网络训练流程。

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

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

# 定义函数 sigmoid2deriv(): 求 sigmoid 函数的导数
def sigmoid2deriv(output):
    return output * (1-output)

lr, hidden_size = 2, 2           # 给定学习步长和隐藏层的节点个数

b_0_1 = np.zeros((1, 2))          # 初始化 b_0_1
b_1_2 = 0                         # 初始化 b_1_2
# 初始化 weights_0_1, 服从标准正态分布
weights_0_1 = np.random.normal(size=(2, hidden_size))
# 初始化 weights_1_2, 服从标准正态分布
weights_1_2 = np.random.normal(size=(hidden_size, 1))
for iter in range(600):
    total_loss = 0
    for i in range(len(x)):
        layer_0 = x[i:i+1]                                  #  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: 计算 layer_2
        layer_2 = sigmoid(np.dot(layer_1, weights_1_2)+b_1_2)
        total_loss += -(y[i:i+1]*np.log(layer_2)+\
                (1-y[i:i+1])*np.log(1-layer_2))  #  计算模型误差

        # ***** 反向传播算法: 计算 delta_2
        delta_2 = (layer_2 - y[i:i+1])
        # ***** 反向传播算法: 计算 delta_1
        delta_1 = delta_2.dot(weights_1_2.T) * \
                  sigmoid2deriv(layer_1)
        # ***** 更新 b_0_1 和 b_1_2
        b_1_2 -= lr * delta_2
        b_0_1 -= lr * delta_1
        # ***** 更新 weights_1_2 和 weights_0_1
        weights_1_2 -= lr * layer_1.T.dot(delta_2)
        weights_0_1 -= lr * layer_0.T.dot(delta_1)

    if (iter % 100 == 9 or iter==599):
        print("iter: %3d; Loss: %0.5f" % (iter, total_loss))

iter:   9; Loss: 4.05018
iter: 109; Loss: 2.93568
iter: 209; Loss: 2.82540
iter: 309; Loss: 0.07302
iter: 409; Loss: 0.02847
iter: 509; Loss: 0.01768
iter: 599; Loss: 0.01317
```

4.4.2 批量随机梯度下降法

图 4-22 表示一个具有两个隐藏层的神经网络模型。我们将应用正向传播算法、反向传播算法以及批量随机梯度下降法训练该模型。图 4-22 的神经网络可以表示如下（随机梯度下降法的 batch size 设为 10）。

$$\underset{10\times2}{\boldsymbol{l}_0}\left(\underset{2\times5}{\boldsymbol{W}_{01}},\underset{1\times5}{\boldsymbol{b}_{01}}\right)\rightarrow\underset{10\times5}{\boldsymbol{l}_1}\left(\underset{5\times5}{\boldsymbol{W}_{12}},\underset{1\times5}{\boldsymbol{b}_{12}}\right)\rightarrow\underset{10\times5}{\boldsymbol{l}_2}\left(\underset{5\times1}{\boldsymbol{W}_{23}},\underset{1\times1}{b_{23}}\right)\rightarrow\underset{10\times1}{l_3}$$

- $\boldsymbol{l}_0$为输入层，$\boldsymbol{l}_1$为隐藏层 1，$\boldsymbol{l}_2$为隐藏层 2，l_3为输出层；
- $\boldsymbol{W}_{01}$，$\boldsymbol{W}_{12}$，$\boldsymbol{W}_{23}$分别为$\boldsymbol{l}_0$到$\boldsymbol{l}_1$，$\boldsymbol{l}_1$到$\boldsymbol{l}_2$，$\boldsymbol{l}_2$到l_3的权重；
- $\boldsymbol{b}_{01}$，$\boldsymbol{b}_{12}$，b_{23}分别为$\boldsymbol{l}_0$到$\boldsymbol{l}_1$，$\boldsymbol{l}_1$到$\boldsymbol{l}_2$，$\boldsymbol{l}_2$到l_3的截距项。

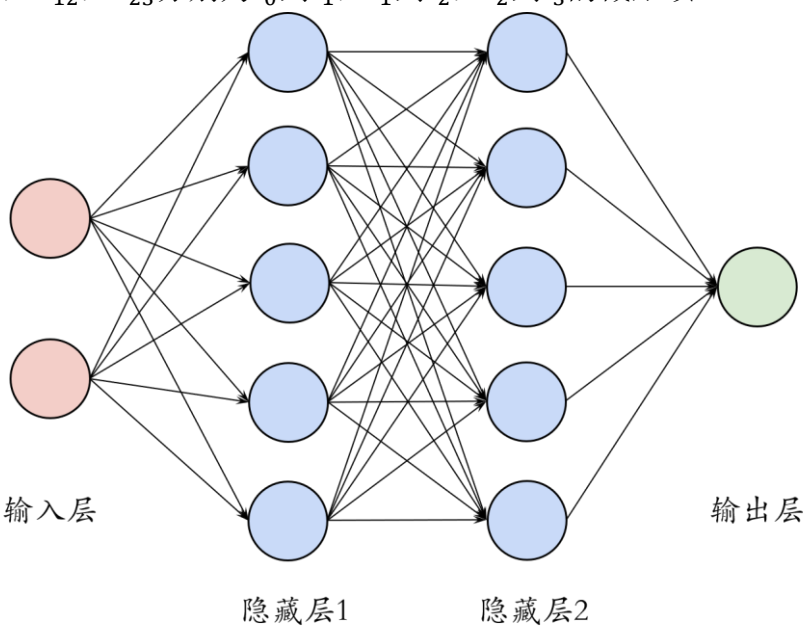


图 4-22 具有两个隐藏层的神经网络，每个隐藏层的节点数都为 5

批量随机梯度下降法的学习步长为α,batch size 为q。对于图 4-22 所示神经网络，批量随机梯度下降法可以通过如下步骤实现。

循环直至收敛

对于训练数据的每批数据：

1. 用正向传播算法得到输出值$\boldsymbol{l_3}$:
   - $\boldsymbol{s_1}=\boldsymbol{l}_0\boldsymbol{W}_{01}+\boldsymbol{b}_{01}$
   - $\boldsymbol{l_1}=\sigma(\boldsymbol{s_1})$
   - $\boldsymbol{s_2}=\boldsymbol{l_1}\boldsymbol{W}_{12}+\boldsymbol{b}_{12}$
   - $\boldsymbol{l_2}=\sigma(\boldsymbol{s_2})$
   - $\boldsymbol{s_3}=\boldsymbol{l_2}\boldsymbol{W}_{23}+b_{23}$
   - $\boldsymbol{l_3}=\sigma(\boldsymbol{s_3})$

2. 计算损失函数L；并且计算$\boldsymbol{\delta_3} = \frac{\partial L}{\partial \boldsymbol{l_3}} \frac{\partial \boldsymbol{l_3}}{\partial \boldsymbol{s_3}} = \frac{\partial L}{\partial \boldsymbol{l_3}} \circ (\boldsymbol{l_3} \circ (1 - \boldsymbol{l_3}))\ /q$
3. 逐层计算$\boldsymbol{\delta_2}$，$\boldsymbol{\delta_1}$
   - $\boldsymbol{\delta_2} = (\delta_3 \boldsymbol{W_{23}^{\mathrm{T}}}) \circ (\boldsymbol{l_2} \circ (1 - \boldsymbol{l_2}))$
   - $\boldsymbol{\delta_1} = (\boldsymbol{\delta_2} \boldsymbol{W_{12}^{\mathrm{T}}}) \circ (\boldsymbol{l_1} \circ (1 - \boldsymbol{l_1}))$
4. 逐层计算梯度，$\nabla_{\boldsymbol{W_{23}}}$，$\nabla_{\boldsymbol{W_{12}}}$，$\nabla_{\boldsymbol{W_{01}}}$，$\nabla_{\boldsymbol{b_{23}}}$，$\nabla_{\boldsymbol{b_{12}}}$，$\nabla_{\boldsymbol{b_{01}}}$。$\mathbf{1}$表示维度为$10 \times 1$，全部元素都为 1 的列向量。
   - $\nabla_{\boldsymbol{W_{23}}} = \boldsymbol{l_2^{\mathrm{T}}} \boldsymbol{\delta_3}$
   - $\nabla_{\boldsymbol{W_{12}}} = \boldsymbol{l_1^{\mathrm{T}}} \boldsymbol{\delta_2}$
   - $\nabla_{\boldsymbol{W_{01}}} = \boldsymbol{l_0^{\mathrm{T}}} \boldsymbol{\delta_1}$
   - $\nabla_{\boldsymbol{b_{23}}} = \mathbf{1}^{\mathrm{T}} \boldsymbol{\delta_3}$
   - $\nabla_{\boldsymbol{b_{12}}} = \mathbf{1}^{\mathrm{T}} \boldsymbol{\delta_2}$
   - $\nabla_{\boldsymbol{b_{01}}} = \mathbf{1}^{\mathrm{T}} \boldsymbol{\delta_1}$
5. 更新权重和截距项：
   - $\boldsymbol{W_{23}} = \boldsymbol{W_{23}} - \alpha \nabla_{\boldsymbol{W_{23}}}$
   - $\boldsymbol{W_{12}} = \boldsymbol{W_{12}} - \alpha \nabla_{\boldsymbol{W_{12}}}$
   - $\boldsymbol{W_{01}} = \boldsymbol{W_{01}} - \alpha \nabla_{\boldsymbol{W_{01}}}$
   - $b_{23} = b_{23} - \alpha \nabla_{b_{23}}$
   - $\boldsymbol{b_{12}} = \boldsymbol{b_{12}} - \alpha \nabla_{\boldsymbol{b_{12}}}$
   - $\boldsymbol{b_{01}} = \boldsymbol{b_{01}} - \alpha \nabla_{\boldsymbol{b_{01}}}$

因此，在算法中无论神经网络有多少层，每一层有多少个节点，都可以通过正向传播算法计算预测值和损失函数，并且可以自动化地按照反向传播算法计算损失函数关于所有参数的梯度。下面是计算$\boldsymbol{\delta_k}$、$\nabla_{\boldsymbol{W_{i,i+1}}}$以及$\nabla_{\boldsymbol{b_{i,i+1}}}$的通用方法。

1. 计算$\boldsymbol{\delta_k}$，$\boldsymbol{\delta_k}$表示最小化损失函数L需要改变的$\boldsymbol{s_k}$的方向和大小。在本例中，$k = 1,2,3$。
   - $\boldsymbol{\delta_3}$的维度为10×1，与$\boldsymbol{l_3}$的维度相同。
   - $\boldsymbol{\delta_2}$的维度为10×5，与$\boldsymbol{l_2}$的维度相同。$\boldsymbol{\delta_2} = (\delta_3 \boldsymbol{W_{23}^{\mathrm{T}}}) \circ \sigma'(\boldsymbol{s_2})$，即$\boldsymbol{\delta_2}$为$\delta_3$乘以$\boldsymbol{l_2}$到$l_3$的权重的转置再逐点乘以$\boldsymbol{l_2}$关于$\boldsymbol{s_2}$的偏导数。
   - $\boldsymbol{\delta_1}$的维度为10×5，与$\boldsymbol{l_1}$的维度相同；$\boldsymbol{\delta_1} = (\boldsymbol{\delta_2} \boldsymbol{W_{12}^{\mathrm{T}}}) \circ \sigma'(\boldsymbol{s_1})$，即$\boldsymbol{\delta_1}$为$\boldsymbol{\delta_2}$乘以$\boldsymbol{l_1}$到$\boldsymbol{l_2}$的权重的转置再逐点乘以$\boldsymbol{l_1}$关于$\boldsymbol{s_1}$的导数。

   因此，除了输出层的$\boldsymbol{\delta}$，其余层的$\boldsymbol{\delta}$都可以通过如下公式逐层计算得到：

$$\boldsymbol{\delta_i} = (\boldsymbol{\delta_{i+1}} \boldsymbol{W_{i,i+1}^{\mathrm{T}}}) \circ \sigma'(\boldsymbol{s_i})$$

2. 计算$\nabla_{\boldsymbol{W_{i,i+1}}}$，$\nabla_{\boldsymbol{W_{i,i+1}}}$表示最小化损失函数$L$需要改变的$\boldsymbol{W_{i,i+1}}$方向和大小。在本例中，$i = 0,1,2$。
   - $\nabla_{\boldsymbol{W_{23}}}$的维度为$5 \times 1$，与$\boldsymbol{W_{23}}$相同；$\nabla_{\boldsymbol{W_{23}}} = \boldsymbol{l_2^{\mathrm{T}}} \boldsymbol{\delta_3}$，$\boldsymbol{l_2}$的维度为$10 \times 5$，$\boldsymbol{\delta_3}$的维度为$10 \times 1$。

- $\nabla_{\boldsymbol{W}_{12}}$的维度为$5\times5$，与$\boldsymbol{W}_{12}$相同；$\nabla_{\boldsymbol{W}_{12}}=\boldsymbol{l}_1^{\mathrm{T}}\boldsymbol{\delta}_2$，$\boldsymbol{l}_1$的维度为$10\times5$，$\boldsymbol{\delta}_2$的维度为$10\times5$。
- $\nabla_{\boldsymbol{W}_{01}}$的维度为$2\times5$，与$\boldsymbol{W}_{01}$相同；$\nabla_{\boldsymbol{W}_{01}}=\boldsymbol{l}_0^{\mathrm{T}}\boldsymbol{\delta}_1$，$\boldsymbol{l}_0$的维度为$10\times2$，$\boldsymbol{\delta}_1$的维度为$10\times5$。

因此，权重$\boldsymbol{W}_{i,i+1}$的梯度都可以通过如下公式逐层计算得到：

$$\nabla_{\boldsymbol{W}_{i,i+1}}=\boldsymbol{l}_i^{\mathrm{T}}\boldsymbol{\delta}_{i+1}$$

3. 计算$\nabla_{\boldsymbol{b}_{i,i+1}}$，$\nabla_{\boldsymbol{b}_{i,i+1}}$表示最小化损失函数$L$需要改变的$\boldsymbol{b}_{i,i+1}$方向和大小。在本例中$i=0,1,2$。
   - $\nabla_{b_{23}}$的维度为1×1，与b_{23}相同；$\nabla_{b_{23}}=\mathbf{1}^{\mathrm{T}}\boldsymbol{\delta}_3$，$\boldsymbol{\delta}_3$的维度为$10\times1$。
   - $\nabla_{\boldsymbol{b}_{12}}$的维度为$1\times5$，与$\boldsymbol{b}_{12}$相同；$\nabla_{\boldsymbol{b}_{12}}=\mathbf{1}^{\mathrm{T}}\boldsymbol{\delta}_2$，$\boldsymbol{\delta}_2$的维度为$10\times5$。
   - $\nabla_{\boldsymbol{b}_{01}}$的维度为$1\times5$，与$\boldsymbol{b}_{01}$相同；$\nabla_{\boldsymbol{b}_{01}}=\mathbf{1}^{\mathrm{T}}\boldsymbol{\delta}_1$，$\boldsymbol{\delta}_1$的维度为$10\times5$。

   因此，截距项$\boldsymbol{b}_{i,i+1}$的梯度可以通过如下公式逐层计算得到：

$$\nabla_{\boldsymbol{b}_{i,i+1}}=\mathbf{1}^{\mathrm{T}}\boldsymbol{\delta}_{i+1}$$

现在以 Default 数据为例，在 Python 中建立如图 4.22 所示的神经网络，并使用批量随机梯度下降法更新参数。批量随机梯度下降法的 batch size 设为 10，学习步长设为 0.5。

```
"""
定义函数 loadDataSet()
载入数据 Default.txt
"""
def loadDataSet():
    x = []; y = []
    # 打开 data 文件中的文件 Default.txt
    f = open("./data/Default.txt")
    for line in f.readlines(): # 函数 readlines()读入文件 f 的所有行
        lineArr = line.strip().split()
        x.append([float(lineArr[0]), float(lineArr[1])])
        y.append(lineArr[2])
    return np.array(x), np.array(y)
x, y = loadDataSet()

y01 = np.zeros(len(y))
for i in range(len(y)):
    if y[i] == "Yes":
        y01[i] = 1
y = y01
"""
训练有两个隐藏层的神经网络
批量随机梯度下降法
"""
np.random.seed(3)
```

```
def sigmoid(x):                                    # 定义函数 sigmoid
    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)

lr, hidden_size = 0.5, 5          # 给定学习步长和隐藏层的节点个数
batch_size = 10                   # 给定每一批数据的观测点个数
num_batch = int(np.floor(len(x)/batch_size))

scaled_x = (x - np.mean(x, axis=0, keepdims=True))/ \
            np.std(x, axis=0, keepdims=True)       #  自变量标准化
y = y.reshape((-1, 1))

b_0_1 = np.zeros((1, 5))                                    # 初始化 b_0_1
b_1_2 = np.zeros((1, 5))                                    # 初始化 b_1_2
b_2_3 = np.zeros((1, 1))                                    # 初始化 b_1_2

# 初始化 weights_0_1, weights_1_2, weights_2_3
# 服从(-0.5,0.5)的均匀分布
weights_0_1 = np.random.random(size=(2, 5))-0.5
weights_1_2 = np.random.random(size=(5, 5))-0.5
weights_2_3 = np.random.random(size=(5, 1))-0.5

for iter in range(200):
    total_loss = 0
    train_acc = 0
    for i in range(num_batch):

        batch_start, batch_end = i * batch_size, \
                                 (i+1) * batch_size
         # ***** 正向传播算法: 计算 layer_0, layer_1, layer_2
        layer_0 = scaled_x[batch_start:batch_end]
        layer_1 = sigmoid(np.dot(layer_0, weights_0_1) + b_0_1)
        layer_2 = sigmoid(np.dot(layer_1, weights_1_2) + b_1_2)
        layer_3 = sigmoid(np.dot(layer_2, weights_2_3) + b_2_3)

        labels_batch = y[batch_start:batch_end]
        total_loss -= np.sum(labels_batch  * np.log(layer_3) +\
                    (1 - labels_batch ) * np.log(1 - layer_3))
        for k in range(batch_size):
            if layer_3[k] >= 0.5:
                pred = 1
            else:
                pred = 0
            train_acc += np.sum(pred == labels_batch[k])
        # ***** 反向传播算法: 计算 delta_3, delta_2, delta_1
        delta_3 = (layer_3 - labels_batch)/batch_size
        delta_2 = delta_3.dot(weights_2_3.T)* \
                  sigmoid2deriv(layer_2)
        delta_1 = delta_2.dot(weights_1_2.T)* \
                  sigmoid2deriv(layer_1)
```

```
        # ***** 更新b_2_3，b_1_2，b_0_1
        b_2_3 -= lr * np.sum(delta_3, axis = 0, keepdims=True)
        b_1_2 -= lr * np.sum(delta_2, axis = 0, keepdims=True)
        b_0_1 -= lr * np.sum(delta_1, axis = 0, keepdims=True)

        # ***** 更新weights_1_2，weights_1_2，weights_0_1
        weights_2_3 -= lr * layer_2.T.dot(delta_3)
        weights_1_2 -= lr * layer_1.T.dot(delta_2)
        weights_0_1 -= lr * layer_0.T.dot(delta_1)

    if (iter%50 == 0 or iter==199):
        print("iter: %3d; Loss: %0.3f; Train Acc: %0.3f" %\
              (iter, total_loss, train_acc/len(x)))

iter:   0; Loss: 66.246; Train Acc: 0.700
iter:  50; Loss: 31.835; Train Acc: 0.920
iter: 100; Loss: 31.192; Train Acc: 0.890
iter: 150; Loss: 30.969; Train Acc: 0.890
iter: 199; Loss: 30.801; Train Acc: 0.890
```

4.5 本章小结

神经网络的目的是建立输入层和输出层之间的相关关系，进而利用建立的相关关系得到预测值。通过增加隐藏层，神经网络可以找到输入层和输出层之间较为复杂的相关关系。深度学习是拥有多个隐藏层的神经网络。在神经网络中，我们通过正向传播算法得到预测，并通过反向传播算法得到参数梯度，然后再用梯度下降法更新参数使得模型误差变小，最终得到一个训练好的神经网络模型。

在神经网络中，只要知道神经网络的结构，便可以自动地计算参数梯度（逐层计算$\boldsymbol{\delta}$，然后再逐层计算参数梯度），进而训练神经网络。因此，无论神经网络模型的结构有多复杂，我们都可以使用一套既定的算法训练神经网络模型。

正向传播算法、反向传播算法和梯度下降法是训练神经网络模型的关键步骤，也是理解神经网络原理的关键点。希望你可以逐行敲入代码，观察代码运行结果，最终能够独自编程实现神经网络模型。

习题

1. 当$\boldsymbol{l}_2 = \sigma(\boldsymbol{s}_2)$时，证明$\frac{\partial \boldsymbol{l}_2}{\partial \boldsymbol{s}_2} = \boldsymbol{l}_2 \circ (1 - \boldsymbol{l}_2)$。
2. 在数据 2 中，建立如图 4-12 所示的神经网络，使用随机梯度下降法求参数，尝试不同的学习步长。
3. 在数据 2 中，建立如图 4-12 所示的神经网络，尝试不同的隐藏层节点数。
4. 在 Default 数据中，建立如图 4-12 所示的神经网络，使用随机梯度下降法求参数。
5. 在 Default 数据中，建立如图 4-12 所示的神经网络，使用批量随机梯度下降法求参数，尝试不同的 batch size。

6. 在 Default 数据中，建立如图 4-22 所示的神经网络，使用全数据梯度下降法求参数。
7. 在 Advertising 数据中，建立如图 4-12 所示的神经网络，使用全数据梯度下降法求参数。