第 6 章 模型评估和正则化

现在我们已经知道如何建立一个基本的神经网络模型。在本章中，我们将学习评估神经网络模型的方法，讨论神经网络模型经常会出现的两个问题：欠拟合和过拟合。在神经网络建模的过程中，模型评估有两个作用：1. 了解神经网络的表现，例如预测精度和波动性等；2. 找出神经网络模型表现不好的原因，明确改进神经网络模型的方向。本章将介绍 4 种应对过拟合的方法：早停法、L_2惩罚和 Dropout 方法，增加观测点。

6.1 模型评估

我们用模型误差衡量神经网络模型的表现。直觉上，模型误差应该计算模型预测与真实因变量之间的差距。神经网络的因变量通常有两种数据类型：定量数据和定性数据。接下来，我们分别讨论不同因变量数据类型下模型误差的定义。

- 当因变量的数据类型是定量数据，模型最终得到的预测值也是连续型的数据。对于一个观测点$(\boldsymbol{x}, y)$，把自变量$\boldsymbol{x}$代入模型中得到该观测点的预测值$\hat{y}$，模型关于该观测点的误差可以定义为$(\hat{y}-y)^2$。$\hat{y}-y$表示预测值和真实值的差距。$\hat{y}-y$为正值表示预测值过大；$\hat{y}-y$为负值则表示预测值过小。模型预测的最终目标是$\hat{y}$尽可能接近y。因此可以定义模型误差为$\hat{y}-y$的平方。$(\hat{y}-y)^2$越小说明$\hat{y}$离y越近。如果有多个观测点，记包含这些观测点的集合为$\mathcal{D}$，模型误差定义为集合$\mathcal{D}$中观测点误差的平均值，i.e. $\text{ave}_{i\in\mathcal{D}}(\hat{y}_i-y_i)^2$。

- 当因变量的数据类型是定性数据，模型最终得到的预测值也是离散的。例如，对于一个观测点$(\boldsymbol{x}, y)$, $y\in\{1,2,\cdots,K\}$，因变量分成K类；把自变量$\boldsymbol{x}$代入模型中得到该观测点的预测值$\hat{y}$，这里，$\hat{y}=\arg\max p_k$, $k=1,\cdots,K$。该观测点的误差定义为$I_{y\neq\hat{y}}$，即如果预测值与真实值相同，则模型关于该观测点的误差为 0；如果预测值与真实值不同，则模型关于该观测点的误差为 1。如果有多个观测点，记包含这些观测点的集合为$\mathcal{D}$，模型误差定义为所有观测点误差的平均值，i.e. $\text{ave}_{i\in\mathcal{D}}I_{y_i\neq\hat{y}_i}$。该模型误差表示数据集$\mathcal{D}$中预测值与真实值不相同的比例。

当因变量为定性数据时，模型误差可以进一步分为两个类型，假阳性率和假阴性率。例如，在第 3 章信用卡违约的例子中，模型有可能会错误预测真实违约的人为不违约，或者预测真实不违约的人为违约。这两种错误对决策的影响可能是不一样的。在信用卡违约的例子中，如果银行希望尽量减少风险，那么银行便希望真实违约但被预测为不违约的人尽可能少；如果银行

希望扩大信用卡业务而适当放宽风险控制，那么银行可以让真实违约但被预测为不违约的人稍微多些。误差矩阵（confusion matrix）可以方便表示两类误差。表 6-1 是误差矩阵的一个例子。总的观测点个数为 100，模型误差总体误差为$(2+14)/100=16\%$。真实不违约的人(80+2)中，有 80 人正确预测为不违约，有两人错误预测为违约，错误率为$2/82=2.4\%$，该错误率称为假阳性率（False Positive Rate, FPR）；真实违约的人(14+4)中，有 14 人错误预测为不违约，4 人正确预测为违约，错误率为$14/18=77.8\%$，该错误率称为假阴性率（False Negative Rate, FNR）。从表 6-1 可以看到，信用卡违约模型可以很好地控制假阳性率，但是假阴性率很大。

表 6-1 误差矩阵

	真实不违约	真实违约
预测不违约	80	14
预测违约	2	4

模型误差的定义方式取决于因变量的数据类型是定量数据还是定性数据。无论是哪种定义方式，估计模型误差还需要选择合适的用于计算模型误差的观测点集合。估计模型误差的作用之一是估计应用于实际情况时模型的误差大小。而在实际应用中，输入值是模型没有使用过的。因此，为了更准确地得到模型误差估计，模拟模型实际应用的情形，我们用训练数据训练模型，而用测试数据计算模型误差。训练数据和测试数据没有任何交集。通常，用训练数据计算得到的模型误差称为训练误差；用测试数据计算得到的模型误差称为测试误差。

在实际中，训练误差常常偏小，不是模型真实误差的好的估计。这是因为模型是用训练数据训练出来的，而训练模型的目标是让模型损失函数最小（损失函数表示真实因变量与预测值之间的差距，与模型误差通常是相同或者相似的）。因此，用训练数据训练出来的模型会使得损失函数变小，也就会使得训练误差较小。做一个类比，假如我们通过做作业来学习函数相关知识。如果考试题目是做过的作业题，那么我们更容易得高分（因为学习的过程便是做作业题）；如果考试题目不是作业题，相对来说更不容易得到高分，但是这样的考试结果才能更好的评估我们对相关知识点的掌握程度。

如果没有测试数据，可以把数据集分成两个部分：训练数据（training data）和测试数据（test data）。例如，在图 6-1 中，把原始数据的n个观测点随机分成左边的训练数据和右边的测试数据。然后用训练数据训练模型，再把测试数据代入训练好的模型中计算测试误差。

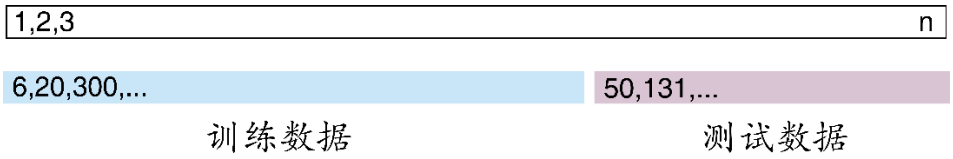

图 6-1 训练数据和测试数据

下面代码演示了将数据分成训练数据和测试数据的常用方法。在本例中，共有 4 个观测点，随机把两个观测点分为训练数据，另外两个观测点分为测试数据。

```
import numpy as np

"""
产生一个简单的数据
"""
x = np.arange(20).reshape((4,5))
y = np.arange(4).reshape((4,1))
print("x:")
print(x)
print("y:")
print(y)

x:
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
y:
[[0]
 [1]
 [2]
 [3]]

"""
把数据分成训练数据和测试数据
"""

# ***** 产生一个长度为观测点个数的向量，0,1,2...

index = np.arange(len(x))
# ***** 打乱 index 的顺序
np.random.shuffle(index)
# ***** 训练数据和测试数据各一半
half = int(len(x)/2)
x_train, y_train = x[index[0:half]], y[index[0:half]]
x_test, y_test = x[index[-half:]], y[index[-half:]]

print("training data X:")
print(x_train)
print("training data y:")
print(y_train)
print("test data X:")
print(x_test)
print("test data y:")
```

```
print(y_test)

training data X:
[[ 5  6  7  8  9]
 [10 11 12 13 14]]
training data y:
[[1]
 [2]]
test data X:
[[15 16 17 18 19]
 [ 0  1  2  3  4]]
test data y:
[[3]
 [0]]
```

在实践中，有时候需要进一步把训练数据分成两个部分：训练数据和验证数据（如图 6-2 所示）。训练数据用于正向传播算法、反向传播算法和梯度下降法等具体模型训练过程。验证数据主要用于选择超参数（hyperparameter）。超参数指在模型中用到，但是又无法直接通过迭代计算优化的参数。在神经网络模型中，超参数包含隐藏层的层数、隐藏层的节点数、学习步长、batch size 等。超参数的选择过程也是模型训练的一部分。因此，使用验证数据计算得到的验证误差也不能用于估计模型真实的预测误差。测试数据将一直被搁置一旁，直到模型完全建立好，用于计算模型的测试误差。

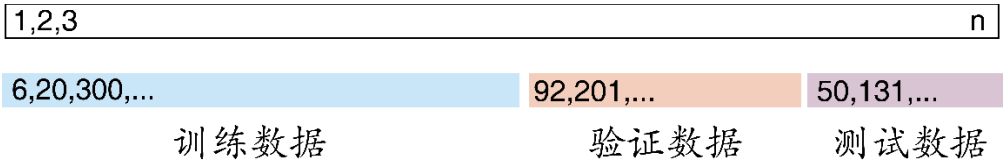

图 6-2 训练数据、验证数据和测试数据

6.2 欠拟合和过拟合

本节将讨论神经网络模型训练过程中经常出现的两类问题：欠拟合（underfitting）和过拟合（overfitting）。在实践中，无论是欠拟合还是过拟合，都会造成神经网络模型的实际预测效果较差。

首先通过一个类比例子理解什么是欠拟合和过拟合。想象学生通过做 100 道习题来学习函数相关知识，学习结束后将进行一次考试，测试学习效果（考试涉及知识点与 100 道习题一样，但是是题目完全不一样的）。分析考试成绩，可以把考试不好的原因分成两种情况。

- 第一种情况，没有很好地掌握 100 道习题，也意味着没能很好地理解习题涉及的知识点，当然考试题也做不好。
- 第二种情况，完全掌握了 100 道习题，但是过多地学习了习题的细节，考试题也有可能做不好。例如，习题里的函数都写成$y = f(x)$，但是有

的考试题把函数写成$b = g(a)$，学习过度的学生有可能不明白$g(a)$也是一个函数。

第一种情况是欠拟合，没能很好地学习习题中的知识；第二种情况是过拟合，过度学习，不仅学习了习题的知识，还学习了习题里过多细节，掌握这些细节反而让学生做不好没做过的题目。

在神经网络中，模型的拟合程度由模型复杂度决定。模型复杂度主要由神经网络模型隐藏层的层数和隐藏层的节点数决定，隐藏层的层数和隐藏层的节点数越多，模型复杂度越高。因为复杂度高的神经网络可以更灵活地调整参数使得损失函数变小，所以模型复杂度越高的神经网络模型学习训练数据的能力也越强。给定训练数据，模型复杂度和误差通常有图 6-3 所示的关系。

- 训练误差随着模型复杂度增大而变小。
- 测试误差随着模型复杂度增大先变小再变大。

测试误差最小的位置表示模型复杂度是适中的。适中模型的左边是欠拟合模型，训练误差和测试误差比较接近，两者都比较大；适中模型的右边是过拟合模型，训练误差和测试误差差距比较大，训练误差很小而测试误差较大。

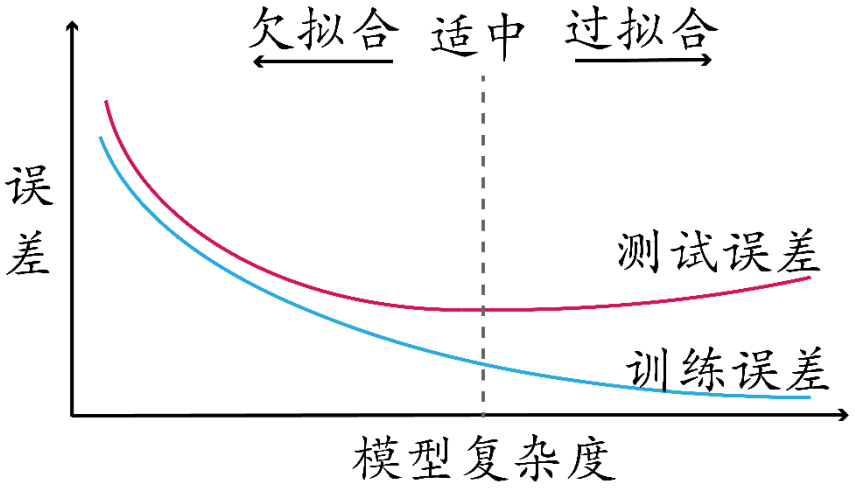

图 6-3 模型误差与模型复杂度的关系

接下来用 mnist 数据说明欠拟合和过拟合。为了节省计算时间，我们只用前 1000 幅数字图片作为训练数据。隐藏层的激活函数为 tanh 函数，输出层的激活函数为 softmax 函数。截距项的初始值都设为 0，输入层到隐藏层的权重从$(-0.01,0.01)$的均匀分布中产生，隐藏层到输出层的权重从$(-0.1,0.1)$的均匀分布中产生。

```
"""
载入需要用到的包和数据
对因变量进行 one-hot 编码
```

```
"""
%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')

images = x_train[0:1000].reshape(1000,28*28)/255
labels = y_train[0:1000]

one_hot_labels = np.zeros((len(labels), 10))
for i,l in enumerate(labels):
    one_hot_labels[i][l] = 1
labels = one_hot_labels

test_images = x_test.reshape(len(x_test), 28*28)/255
test_labels = np.zeros((len(y_test), 10))
for i,l in enumerate(y_test):
    test_labels[i][l] = 1
"""
定义激活函数 tanh, tanh2deriv, softmax
"""
def tanh(x):                    # 定义函数 tanh
    return np.tanh(x)

def tanh2deriv(x):              # 定义函数 tanh2deriv
    return 1 - (x**2)

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

下面代码实现的神经网络模型的隐藏层只有两个节点，迭代更新 500 次。从代码运行结果看，训练误差为 0.271，测试误差为 0.520。

```
"""
欠拟合
神经网络模型有一个隐藏层，隐藏层节点数为 2
"""

np.random.seed(1)
# 给定超参数
lr, epochs, hidden_size = (0.05,500, 2)
pixels_per_image, num_labels = (784,10)
batch_size = 100                              # 每一批数据的观测点个数
num_batch = int(len(images) / batch_size)# 数据的批数
```

```
# b_0_1 和 b_1_2 初始值,全部为 0
b_0_1 = np.zeros((1, hidden_size))
b_1_2 = np.zeros((1, num_labels))
# w_0_1 和 w_1_2 初始值
w_0_1 = 0.02*np.random.random((pixels_per_image,hidden_size))-\
                                                              0.01
w_1_2 = 0.2*np.random.random((hidden_size,num_labels))-0.1

train_err, test_err = [], []
for epoch in range(epochs):
    correct_cnt, test_correct_cnt = 0.0, 0.0
    for i in range(num_batch):
        batch_start, batch_end = ((i*batch_size), \
                                          ((i+1)*batch_size))
        # layer_0: 输入值
        layer_0 = images[batch_start:batch_end]
        # layer_1: 隐藏层
        layer_1 = tanh(np.dot(layer_0, w_0_1)+b_0_1)
        # layer_2: 输出值
        layer_2 = softmax(np.dot(layer_1,w_1_2)+b_1_2)
        labels_batch = labels[batch_start:batch_end]
        for k in range(batch_size):
            # 计算训练正确率
            correct_cnt += int(np.argmax(layer_2[k:k+1])== \
                    np.argmax(labels_batch[k:k+1]))
        # 输出层 delta
        layer_2_delta=(layer_2-labels[batch_start:batch_end])/\
                      batch_size
        # 隐藏层 delta
        layer_1_delta=layer_2_delta.dot(w_1_2.T) * \
                                          tanh2deriv(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 w_1_2 和 w_0_1
        w_1_2 -= lr * layer_1.T.dot(layer_2_delta)
        w_0_1 -= lr * layer_0.T.dot(layer_1_delta)


    layer_0 = test_images
    layer_1 = tanh(np.dot(layer_0, w_0_1)+b_0_1)
    layer_2 = softmax(np.dot(layer_1,w_1_2)+b_1_2)

    # 计算测试误差
    for i in range(len(test_images)):
        test_correct_cnt += int(np.argmax(layer_2[i:i+1])== \
                                np.argmax(test_labels[i:i+1]))

    # 记录训练误差
    train_err.append(1-correct_cnt/float(len(images)))
    # 记录测试误差
    test_err.append(1-test_correct_cnt/float(len(test_images)))
```

```
    # 每 100 epochs 输出训练误差，测试误差等信息
    if (epoch % 100==0 or epoch==epochs-1):
        print("e: %3d; Train_Err: %0.3f; Test_Err: %0.3f" % \
              (epoch,       train_err[-1],       test_err[-1]))
```

```
e:   0; Train_Err: 0.911; Test_Err: 0.860
e: 100; Train_Err: 0.505; Test_Err: 0.554
e: 200; Train_Err: 0.415; Test_Err: 0.532
e: 300; Train_Err: 0.350; Test_Err: 0.519
e: 400; Train_Err: 0.301; Test_Err: 0.518
e: 499; Train_Err: 0.271; Test_Err: 0.520
```

图 6-4 画出了训练误差和测试误差在训练过程中的变化情况。可以看到，刚开始训练误差和测试误差都较大；随后训练误差和测试误差一起快速变小；接着，训练误差依然下降较快，测试误差下降较慢；直到模型训练停止，训练误差依然较大且测试误差还没有上升。说明该神经网络没能很好地学习 1000 幅数字图片包含的信息，使得训练误差和测试误差都较大。这是一个典型的欠拟合情况。

```
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(train_err, label="训练误差")
plt.plot(test_err, label="测试误差 ")

plt.xlabel("迭代次数", fontsize=16)
plt.ylabel("模型误差", fontsize=16 )
plt.legend(fontsize=16)
plt.show()
```

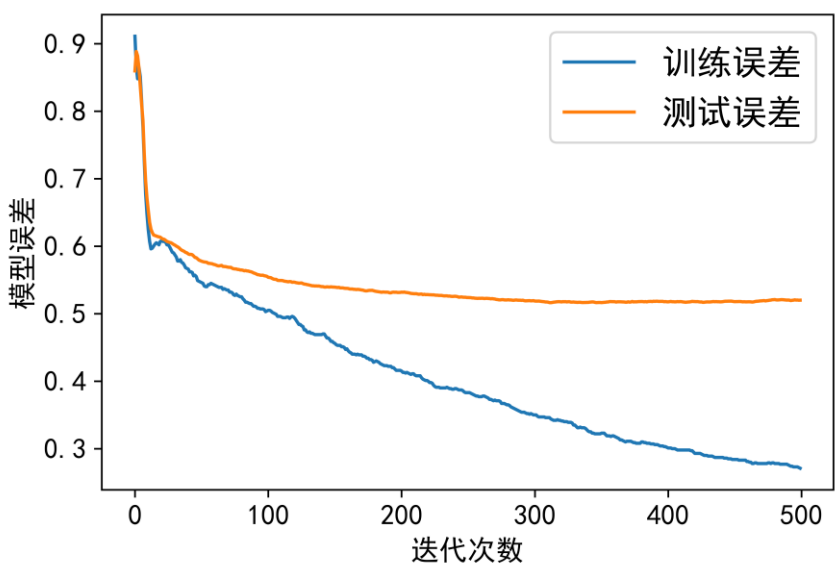

图 6-4 训练误差和测试误差（隐藏层节点数为 2）

解决欠拟合问题的方法比较简单，增加模型复杂度就可以了。在神经网络中，常见的方法是增加隐藏层的数量，或者增加隐藏层的节点数，或者同时增加隐藏层数量和隐藏层的节点数。这些方法都可以提高模型复杂度，改善模型的欠拟合情况。

下面的神经网络模型把隐藏层的节点数增加到 300，并且更新迭代 3000 次。可以看到现在训练误差接近于 0，测试误差大约为 0.14。

```
"""
过拟合
神经网络模型有 1 个隐藏层，隐藏层节点数为 300
"""
np.random.seed(1)
# 给定超参数
lr, epochs, hidden_size = (0.05,3000, 300)
pixels_per_image, num_labels = (784,10)
batch_size = 100                                  # 每一批数据的观测点个数
num_batch = int(len(images) / batch_size)              # 数据的批数

# b_0_1 和 b_1_2 初始值,全部为 0
b_0_1 = np.zeros((1, hidden_size))
b_1_2 = np.zeros((1, num_labels))
# w_0_1 和 w_1_2 初始值
w_0_1 = 0.02*np.random.random((pixels_per_image,hidden_size))-\
                                                                  0.01
w_1_2 = 0.2*np.random.random((hidden_size,num_labels))-0.1

train_err, test_err = [], []
for epoch in range(epochs):
    correct_cnt, test_correct_cnt = 0.0, 0.0
    for i in range(num_batch):
        batch_start = i*batch_size
        batch_end = (i+1)*batch_size
        # layer_0: 输入值
        layer_0 = images[batch_start:batch_end]
        # layer_1: 隐藏层
        layer_1 = tanh(np.dot(layer_0, w_0_1)+b_0_1)
        # layer_2: 输出值
        layer_2 = softmax(np.dot(layer_1,w_1_2)+b_1_2)
        labels_batch = labels[batch_start:batch_end]
        for k in range(batch_size):
            # 计算训练正确率
            correct_cnt += int(np.argmax(layer_2[k:k+1])==\
                               np.argmax(labels_batch[k:k+1]))

        layer_2_delta=(layer_2-labels[batch_start:batch_end])/\
                                    batch_size      # 输出层 delta
        layer_1_delta=layer_2_delta.dot(w_1_2.T)*   \
                            tanh2deriv(layer_1)     # 隐藏层 delta

        # 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 w_1_2 和 w_0_1
        w_1_2 -= lr * layer_1.T.dot(layer_2_delta)
        w_0_1 -= lr * layer_0.T.dot(layer_1_delta)

    layer_0 = test_images
    layer_1 = tanh(np.dot(layer_0, w_0_1)+b_0_1)
    layer_2 = softmax(np.dot(layer_1,w_1_2)+b_1_2)
    # 计算测试误差
    for i in range(len(test_images)):
        test_correct_cnt += int(np.argmax(layer_2[i:i+1])== \
                                np.argmax(test_labels[i:i+1]))

    # 记录训练误差
    train_err.append(1-correct_cnt/float(len(images)))
    # 记录测试误差
    test_err.append(1-test_correct_cnt/float(len(test_images)))

    if (epoch % 300==0 or epoch==epochs-1):
        # 每 300 epochs 输出训练误差，测试误差
        print("e: %4d; Train_Err:%0.3f; Test_Err:%0.3f" % \
              (epoch, train_err[-1], test_err[-1]))
```

```
e:    0; Train_Err:0.453; Test_Err:0.336
e:  300; Train_Err:0.000; Test_Err:0.138
e:  600; Train_Err:0.000; Test_Err:0.139
e:  900; Train_Err:0.000; Test_Err:0.140
e: 1200; Train_Err:0.000; Test_Err:0.140
e: 1500; Train_Err:0.000; Test_Err:0.140
e: 1800; Train_Err:0.000; Test_Err:0.140
e: 2100; Train_Err:0.000; Test_Err:0.140
e: 2400; Train_Err:0.000; Test_Err:0.140
e: 2700; Train_Err:0.000; Test_Err:0.140
e: 2999; Train_Err:0.000; Test_Err:0.140
```

图 6-5 画出了训练误差和测试误差在训练过程中的变化情况。可以看到，刚开始训练误差和测试误差都较大，随后训练误差和测试误差都快速变小；接着，训练误差依然持续下降，测试误差变得平稳甚至略有上升；当模型训练停止后，训练误差和测试误差的差距较大，训练误差接近于 0。说明该神经网络过度学习 1000 幅数字图片包含的信息，使得训练误差不断下降，而测试误差上升，训练误差和测试误差的差距较大。这是典型的过拟合情况。

```
plt.plot(train_err, label="训练误差")
plt.plot(test_err, label="测试误差")

plt.xlabel("迭代次数", fontsize=16)
plt.ylabel("模型误差", fontsize=16 )
plt.legend(fontsize=16)
plt.show()
```

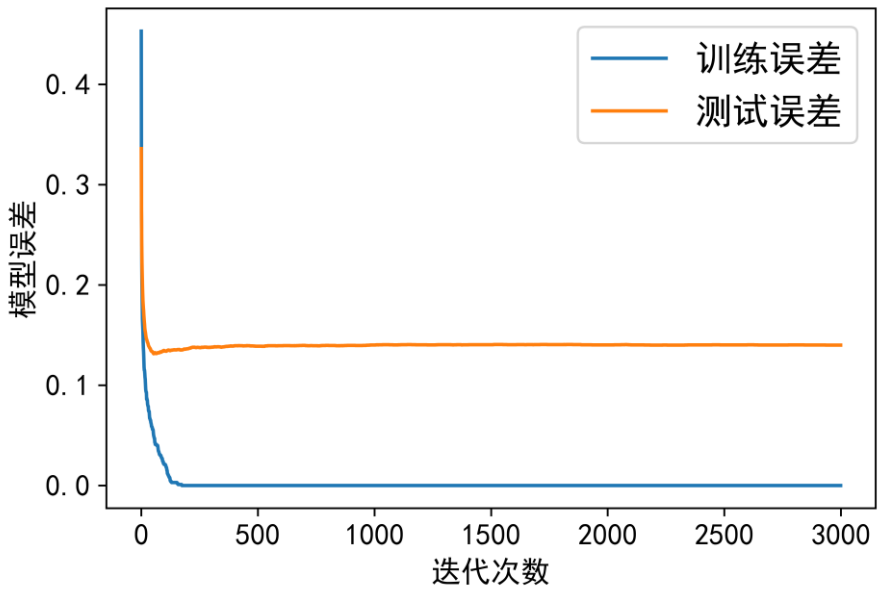

图 6-5 训练误差和测试误差（隐藏层节点数为 300）

在建立神经网络模型的初始阶段，通常很难确定合适的神经网络复杂度（没有过拟合和欠拟合）。解决欠拟合的方法比较简单，只需要增加模型复杂度。在实践中，通常先构造一个复杂的神经网络模型（该模型通常过拟合），然后应用一些方法控制复杂神经网络模型的过拟合现象。科学家提出了很多控制神经网络模型过拟合的方法，这些方法统称为正则化（regularization）方法。

6.3 正则化

6.3.1 早停法

从 6.2 节过拟合的例子可以看出，当过拟合发生时，随着神经网络模型的训练，测试误差不再减小，甚至增大，而训练误差还在逐步减小。所以，一个很自然的想法是：早点停止训练神经网络即可以防止神经网络模型过拟合。该策略称为早停法（early stopping）。

什么时候停止训练神经网络模型呢？回答该问题需要考虑两个因素。第一，需要一个合适的模型评价指标。该指标不能是训练误差，因为训练误差总是逐步减小，而且也不能反映实际模型误差。该指标也不能是测试误差，因为测试数据需要一直保留着，直到完全建立好神经网络模型，再用测试数据计算测试误差（但是，选择停止训练的时机也是训练的一部分）。这时，通常可以把数据分成 3 份，训练数据、验证数据和测试数据。在建模过程中，用训练数据训练模型，同时，在模型训练过程中，每次迭代之后（或者若干次迭代之后）用验证数据计

算验证误差。第二，需要一个停止模型训练的策略。一个简单的策略是，当最新的验证误差大于上次的验证误差时，停止训练模型。

理想情况下，早停法是一个既简单又有效的防止过拟合的方法。在图 6-6 中，早停法将在 A 点处停止模型训练，得到一个拟合程度适中的神经网络模型。

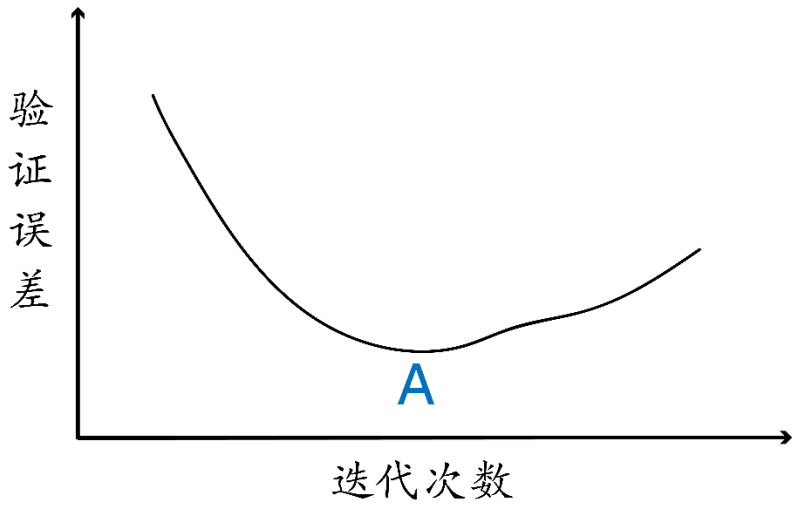

图 6-6 理想情况的早停法

但是，现实中，验证误差有可能不是单调减小的，而是如图 6-7 所示，先减小，增大，然后再减小，再增大。如果我们使用上面说的早停法准则，只要最新的验证误差大于上次的验证误差就停止训练， 早停法可能会使模型训练停在 B 点处。这时，早停法防止了过拟合，但是没能得到更好的神经网络模型（C 点处）。

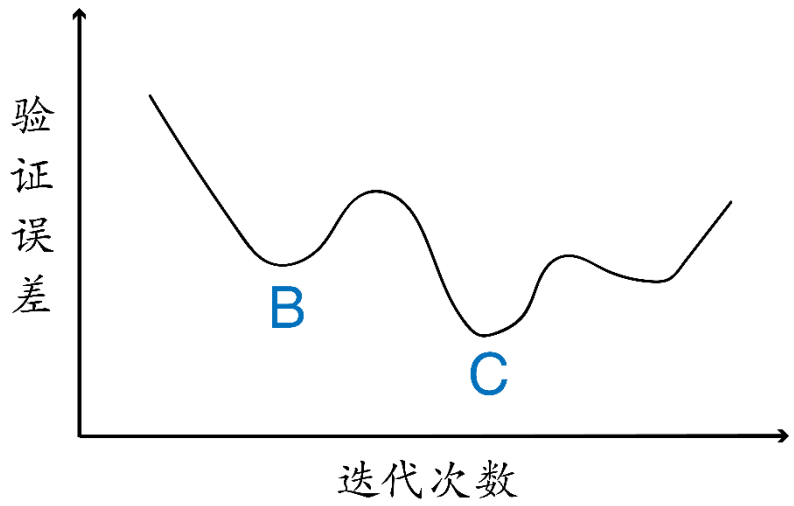

图 6-7 现实中的早停法

6.3.2 L_2惩罚

L_2惩罚是一个经典的正则化方法。该方法在原有损失函数基础上加上所有或者部分权重的L_2范数构造了一个新的损失函数。例如，在 6.2 节的例子中，神经网络模型有两个权重矩阵W_{01}和W_{12}，新的损失函数可以写成如下形式。

$$\text{ave}_{i\in\mathcal{D}}\{-(1_{\{y_{ik}=1\}}\log(p_{ik})\}+\lambda(\|\boldsymbol{W}_{01}\|_2^2+\|\boldsymbol{W}_{12}\|_2^2)$$

这里，λ为一个正的常数，$\mathcal{D}$为观测点集合，$\|\boldsymbol{W}\|_2$表示 Frobenius 范数，即$\|\boldsymbol{W}\|_2^2$表示矩阵$\boldsymbol{W}$所有元素的平方和。注意，L_2惩罚只惩罚权重，而不惩罚截距项系数。

对于一般情况，记模型误差为L，所有权重矩阵为$\boldsymbol{W}_1,\cdots,\boldsymbol{W}_m$，截距项系数为$\boldsymbol{b}_1,\cdots,\boldsymbol{b}_m$，那么带有惩罚项的损失函数可以写成

$$\underset{\boldsymbol{b}_1,\cdots,\boldsymbol{b}_m,\boldsymbol{W}_1,\cdots,\boldsymbol{W}_m}{\arg\min}\ L+\lambda\sum_{i=1}^{m}\|\boldsymbol{W}_i\|_2^2$$

从上式可以看出，最小化带有惩罚项的损失函数时，增大$\boldsymbol{W}_i$的某个元素，上式右边惩罚项的值会上升λ乘以该元素增加值，这时需要该元素值增加后可以使得L下降得更多，这样才可以使得带惩罚项的损失函数变小。因此，惩罚项可以限制权重大小。也就是说，带有惩罚项的损失函数可以让神经网络权重的绝对值变小。当λ值很大时，对权重的限制更大，会使得权重较小；当λ值很小时，对权重的限制较小，会使得权重较大。而限制权重大小实质等价于减少模型的复杂度（可以想象一个较为极端的情况，当λ非常大，有可能使得某些权重值压缩到接近于 0，等于减少了某些节点的作用），减少模型复杂度可以减少过拟合的程度。因此，L_2惩罚可以控制过拟合，且λ越大，控制过拟合的程度越大。

加入惩罚项后，如何最小化带有惩罚项的损失函数呢？我们知道，当损失函数没有惩罚项时，梯度下降法可以通过如下方式更新权重

$$\boldsymbol{W}_i=\boldsymbol{W}_i-\alpha\nabla_{\boldsymbol{W}_i},i=1,\cdots,m$$

其中，$\nabla_{\boldsymbol{W}_i}=\frac{\partial L}{\partial \boldsymbol{W}_i}$。现在，当损失函数带有惩罚项时，带有惩罚项的损失函数关于$\boldsymbol{W}_i$的偏导数为：

$$\frac{\partial L}{\partial \boldsymbol{W}_i}+2\lambda\boldsymbol{W}_i$$

其中，$2\lambda\boldsymbol{W}_i$为$\|\boldsymbol{W}\|_2^2$的偏导数。梯度下降法将通过如下方式更新权重

$$\boldsymbol{W}_i=\boldsymbol{W}_i-\alpha(\frac{\partial L}{\partial \boldsymbol{W}_i}+2\lambda\boldsymbol{W}_i),\quad i=1,\cdots,m$$

在下面代码中，神经网络模型中增加了L_2惩罚，并设$\lambda = 0.005$。从代码运行结果可以看出，模型过拟合现象略有减少，训练误差以更慢的速度趋近于 0，测试误差也略微降低到 0.128。在实践中，λ是一个超参数，我们需要尝试多个不同的λ，对每个λ利用验证数据计算验证误差，然后选择最小验证误差对应的λ。

```
"""
L2 惩罚
神经网络模型有 1 个隐藏层，隐藏层节点数为 300
"""
np.random.seed(1)

lr, epochs, hidden_size = (0.05,3000, 300)
pixels_per_image, num_labels = (784,10)
batch_size = 100
num_batch = int(len(images) / batch_size)
lam = 0.005                                    # 惩罚项参数设为 0.005

b_0_1 = np.zeros((1, hidden_size))
b_1_2 = np.zeros((1, num_labels))
w_0_1 = 0.02*np.random.random((pixels_per_image,hidden_size))-\
                                                          0.01
w_1_2 = 0.2*np.random.random((hidden_size,num_labels))-0.1

L2_train_err, L2_test_err = [], []
for epoch in range(epochs):
    correct_cnt, test_correct_cnt = 0.0, 0.0
    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_1 = tanh(np.dot(layer_0, w_0_1)+b_0_1)
        layer_2 = softmax(np.dot(layer_1,w_1_2)+b_1_2)

        labels_batch = labels[batch_start:batch_end]
        for k in range(batch_size):
            correct_cnt += int(np.argmax(layer_2[k:k+1])== \
                           np.argmax(labels_batch[k:k+1]))


        layer_2_delta=(layer_2-labels[batch_start:batch_end])/\
                      batch_size
        layer_1_delta=layer_2_delta.dot(w_1_2.T)*  \
                                          tanh2deriv(layer_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 w_1_2 和 w_0_1， 增加了惩罚项
        w_1_2 -= lr * (layer_1.T.dot(layer_2_delta) + \
                                          2*lam*w_1_2)
        w_0_1 -= lr * (layer_0.T.dot(layer_1_delta) + \
                                          2*lam*w_0_1)
```

```
    layer_0 = test_images
    layer_1 = tanh(np.dot(layer_0, w_0_1)+b_0_1)
    layer_2 = softmax(np.dot(layer_1,w_1_2)+b_1_2)
    for i in range(len(test_images)):
        test_correct_cnt += int(np.argmax(layer_2[i:i+1])==\
                                np.argmax(test_labels[i:i+1]))

    L2_train_err.append(1-correct_cnt/float(len(images)))
    L2_test_err.append(1-test_correct_cnt/ \
                       float(len(test_images)))

    # 每 300 epochs 输出训练误差，测试误差
    if (epoch % 300==0 or epoch==epochs-1):
        print("e: %4d; Train_Err: %0.3f; Test_Err: %0.3f" % \
              (epoch, L2_train_err[-1], L2_test_err[-1]))
```

```
e:    0; Train_Err: 0.453; Test_Err: 0.336
e:  300; Train_Err: 0.012; Test_Err: 0.129
e:  600; Train_Err: 0.009; Test_Err: 0.128
e:  900; Train_Err: 0.008; Test_Err: 0.128
e: 1200; Train_Err: 0.007; Test_Err: 0.128
e: 1500; Train_Err: 0.007; Test_Err: 0.129
e: 1800; Train_Err: 0.007; Test_Err: 0.128
e: 2100; Train_Err: 0.007; Test_Err: 0.128
e: 2400; Train_Err: 0.007; Test_Err: 0.128
e: 2700; Train_Err: 0.006; Test_Err: 0.128
e: 2999; Train_Err: 0.006; Test_Err: 0.128
```

图 6-8 进一步画出有L_2惩罚项和无L_2惩罚项的训练误差和测试误差随着迭代次数的增加而变化的情况。可以看到，和标准神经网络相比，应用了L_2惩罚的神经网络训练误差下降得更慢。最终，应用了L_2惩罚的神经网络模型虽然有更大的训练误差，但是测试误差更小了。

```
plt.plot(np.arange(len(L2_train_err)), L2_train_err, \
         label="训练误差（L2 惩罚）")
plt.plot(np.arange(len(train_err)), train_err, linestyle=":", \
         label="训练误差（无惩罚）")
plt.plot(np.arange(len(L2_test_err)), L2_test_err, \
         label="测试误差（L2 惩罚）")
plt.plot(np.arange(len(test_err)), test_err, linestyle=":", \
         label="测试误差（无惩罚）")
plt.ylim((-0.01, 0.3))
plt.legend()
plt.xlabel("迭代次数", fontsize=16)
plt.ylabel("模型误差", fontsize=16)
plt.show()
```

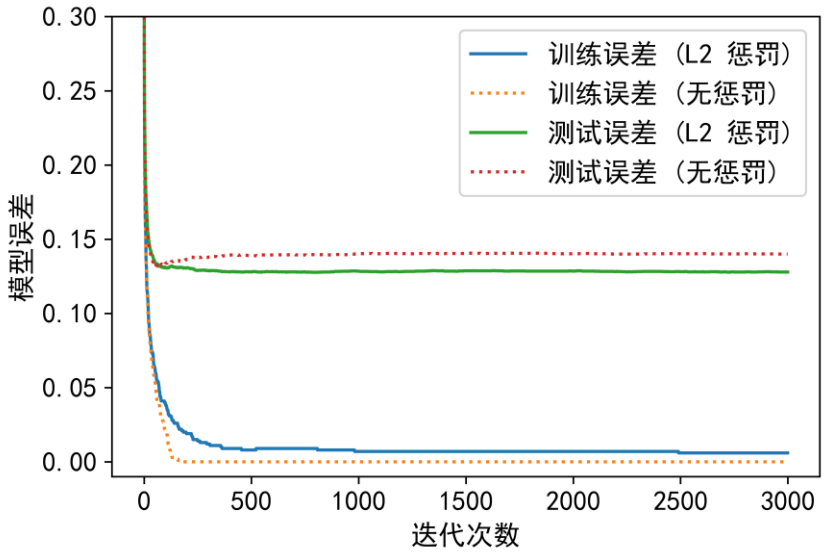

图 6-8 训练误差和测试误差（考虑带有惩罚项和没有惩罚项两种情况）

6.3.3 Dropout

在统计学习或者机器学习领域，模型集成（model ensemble）经常可以提高模型的预测准确度（主要原因是可以减少过拟合）。模型集成方法主要思想很简单，首先训练大量结构不同的模型，然后再通过平均或者投票方式综合所有模型的结果，获得最终预测。通常情况下，如果因变量是定量数据，模型集成的最终预测是所有模型预测结果的平均值；如果因变量是定性数据，模型集成的最终预测是所有模型预测最多的类别，这时，该过程也称为投票。

沿用 6.2 节的类比例子，在该例子中，学生通过 100 道习题学习函数相关知识，然后参加考试测试知识的掌握程度。如果学生学习过于勤奋，该学生的思维可能会限制在 100 道题提供的知识中，造成过拟合。例如，有可能认为$y = f(x)$是函数，而$b = g(a)$不是函数；或者因为 100 道题中出现的函数的定义域都是$(-\infty, \infty)$，就认为所有函数的定义域都是$(-\infty, \infty)$，这样，碰到函数$y = \log(x)$时就容易做错题了。如果让 1000 个学生，而不是 1 个学生学习这 100 道题。考试时，每一道题的最终答案都参考所有学生的意见，考试分数将可能更高。因为，一般情况下，不会所有学生都在同一个知识点中过度学习。例如，有的学生可能误认为$b = g(a)$不是函数，但是其他学生不会这样想，参考所有学生意见的答案将很可能不会犯该错误；有的学生可能误认为$y = \log(x)$的定义域是$(-\infty, \infty)$，但是，不会所有学生都会有这个误解，综合所有学生的意见可能不会犯这个错误。

在统计学习或者机器学习中，模型集成的原理和类比例子一样。虽然复杂度很高的神经网络或多或少都会过拟合，但是这些神经网络过拟合的地方将会不一样。因为不同的神经网络模型初始值不同，模型结构也可能不同，最终这些神经

网络模型得到的输入值与输出值的关系也不同。这就像学生一样，所有学生都会犯错，但是犯错的地方会不同。训练大量结构不同的模型（相当于大量的学生），然后再综合所有模型的结果，最终将会减少总的错误。因为对于某个错误，只有少量模型会出错，综合之后，这些错误将会减少。因此，理论上说，模型集成可以降低模型过拟合程度，得到更好的预测。

理论上，模型集成方法可以提高神经网络模型的预测准确度，如图 6-9 所示。但是，实践中，模型集成方法应用于神经网络模型却有较大限制，其原因主要有两个。

1. 实际应用中的神经网络可能有几十上百层隐藏层，每个隐藏层有大量的节点，调试和训练一个大型的神经网络模型已经需要大量时间、资金、人力和计算机资源。而模型集成需要大量的神经网络模型。因此，有限的时间和成本限制了模型集成方法在神经网络模型中的应用。

2. 实际应用的神经网络模型在预测时有时需要很快的反应速度。例如，自动驾驶车辆需要很快预测出车辆前方是否有人。而把输入值代入到大量的神经网络，然后再综合所有神经网络的结果需要耗费大量的时间和计算机资源。

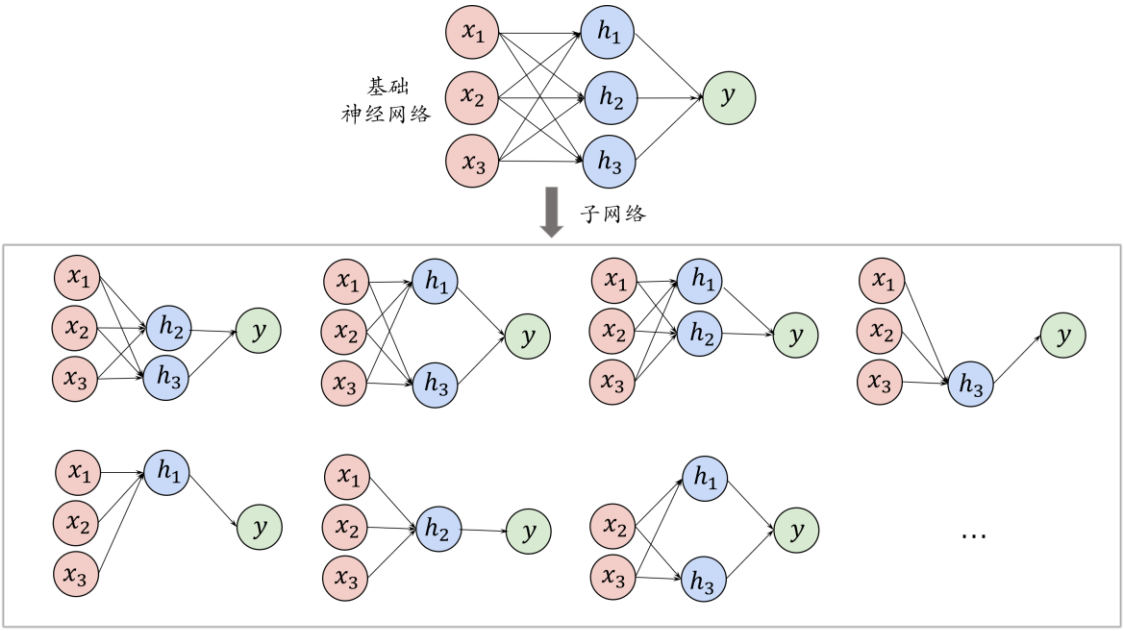

图 6-9 神经网络的模型集成方法

Geoffrey Hinton 及其团队在 2014 年发明了 Dropout 方法。Dropout 方法是一个非常有创意的方法，Dropout 可以在增加少量计算机资源的情况下近似地在神经网络训练过程中实现模型集成，从而控制过拟合，提高神经网络模型的预测准

确度。而且使用 Dropout 的神经网络与没有使用 Dropout 的神经网络预测时占用的计算资源是一样的。从 2014 年 Dropout 提出之后，Dropout 方法就被广泛地应用于学术界和工业界。

Dropout 的实现很简单。如图 6-10 和图 6-11，在标准神经网络情况下，训练神经网络时，每输入一个（或者一批）输入值，Dropout 方法以概率$1-p$随机失活某些节点（可以是输入层和隐藏层的节点）。这些失活的节点在当次正向传播算法和反向传播算法中将不起任何作用。也就是说，当输入一个（或者一批）观测点训练神经网络模型时，因为失活了部分节点，其实训练了一个更“瘦”的模型。这样，每次输入观测点训练神经网络模型都在一定程度上训练了一个“瘦”的不一样的神经网络模型。预测时，我们将用上所有的节点，这相当于综合这些不同的“瘦”模型的预测结果。

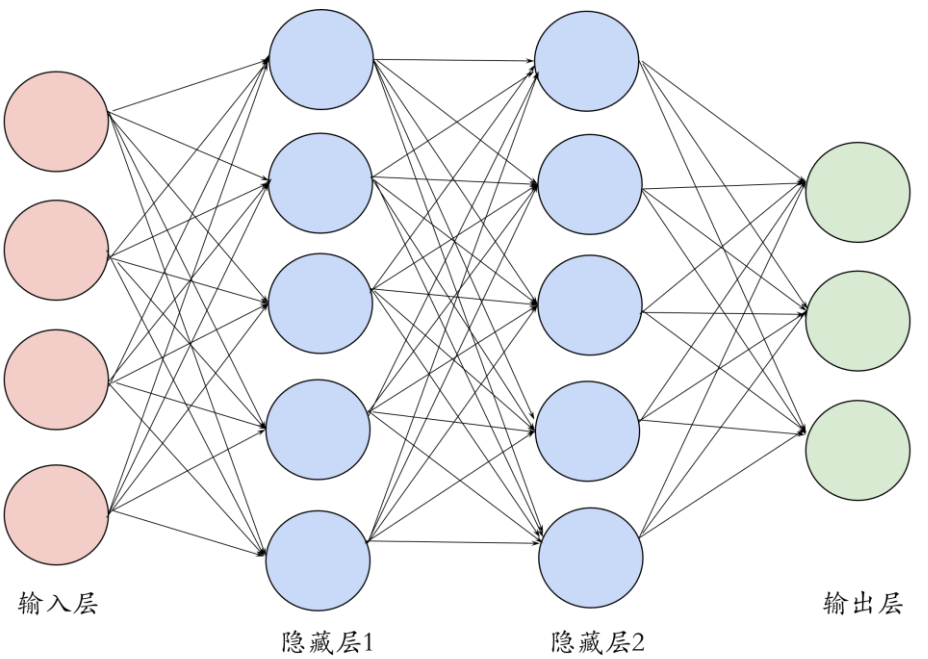

图 6-10 标准神经网络

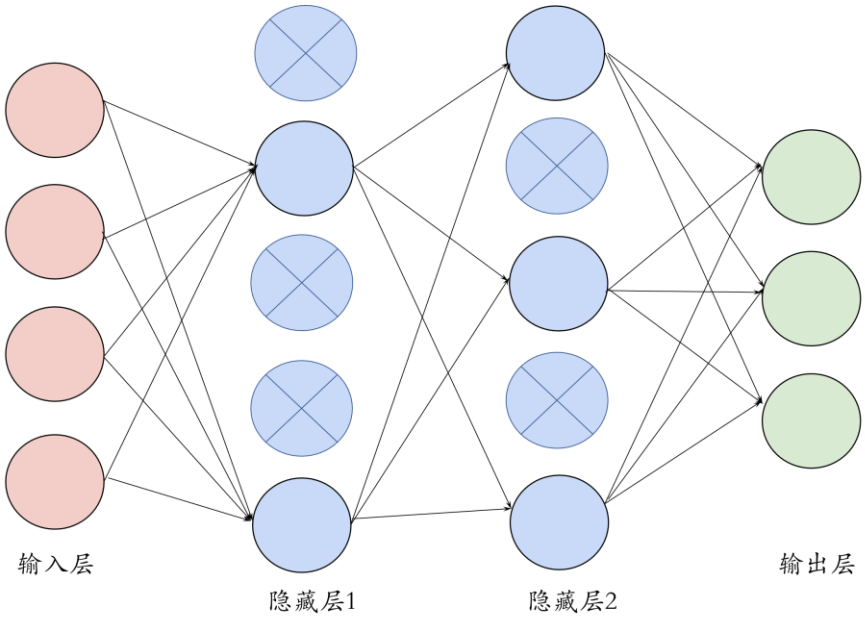

图 6-11 应用了 Dropout 的神经网络

下面以 mnist 数据为例，建立神经网络，说明 Dropout 的具体实现方法。Dropout 方法主要是在训练过程中让一部分输入层或者隐藏层的节点失活，也就是说随机让某些节点变为 0。在实际中，可以从二项分布Binomial$(1,p)$中产生随机数。Binomial$(1,p)$的参数中，1表示二项分布实验次数为 1，p表示成功的概率。因此，二项分布Binomial$(1,p)$产生的随机数要么是1要么是0，且随机数为1的比例大约等于p。在下面代码中，我们让 `layer_1` 乘以 `dropout`，而 `dropout` 是Binomial$(1,p)$产生的随机数的矩阵，所以，`layer_1 * dropout` 便失活了部分 `layer_1` 的节点。Dropout 保留的节点比例一般在 0.3 到 1 之间。在 mnist 数据中，我们设$p = 0.5$，即每次随机让大约一半的隐藏层节点失活。

实现 Dropout 方法时，还需要注意以下两点。

- `layer_1` 不变成 0 的部分需要除以p。`layer_2` 是 `layer_1` 的加权平均值再通过激活函数处理的结果。训练神经网络模型时，`layer_1` 在 Dropout 过程中只有$(100 \times p)\%$的节点会被保留，因此 `layer_1` 的加权平均值会相应的减小，进而使得 `layer_2` 的节点的值变小。而在使用神经网络做预测时，`layer_1` 的节点没有被失活。为了在训练模型和使用模型两种情况下，输入一个观测点，`layer_2` 节点的值大致相等，我们需要在训练神经网络时，让 `layer_1` 除以p。在 mnist 例子中，因为$p = 0.5$，所以 layer_1 需要除以 0.5。
- 另一个需要注意的地方是，在使用 Dropout 方法训练神经网络过程中，因为隐藏层的有些节点被设为 0，这些被设为 0 的节点的梯度也应该被设为 0。因此，在下面代码中，我们让 `layer_1_delta` 乘以 `dropout`。

```
# 下面两行代码得到 layer_0 和 layer_1
```

```
layer_0 = images[1:2]                              # 输入层
layer_1 = tanh(np.dot(layer_0, w_0_1)+b_0_1)   # 隐藏层

"""
从二项分布中产生随机数，失活 layer_1 的部分节点
"""
# 从二项分布中产生随机数
dropout = np.random.binomial(1, 0.5, size=layer_1.shape)

"""
Layer_1 乘以 dropout 失活部分节点
layer_1 除以 0.5，即 layer_1 未失活节点乘以 2
"""
layer_1 *= dropout
layer_1 *= 2

"""
计算输出层
并计算输出层和隐藏层 delta
"""
layer_2 = softmax(np.dot(layer_1,w_1_2) + b_1_2)
layer_2_delta=(layer_2 - labels[1:2])
layer_1_delta=layer_2_delta.dot(w_1_2.T) * tanh2deriv(layer_1)
"""
注意：隐藏层的 delta 乘以 dropout
"""
layer_1_delta *= dropout
```

接下来，我们把 Dropout 方法融入训练 mnist 数据的神经网络模型中。

```
"""
使用 Dropout
神经网络模型有 1 个隐藏层，隐藏层节点数为 300
"""
np.random.seed(1)
lr, epochs, hidden_size = (0.05,3000, 300)
pixels_per_image, num_labels = (784,10)
batch_size = 100
num_batch = int(len(images) / batch_size)
retain_prob = 0.5             # ***** Dropout 的保留节点的比例为 0.5

b_0_1 = np.zeros((1, hidden_size))
b_1_2 = np.zeros((1, num_labels))
w_0_1 = 0.02*np.random.random((pixels_per_image,hidden_size))-\
                                                              0.01
w_1_2 = 0.2*np.random.random((hidden_size,num_labels))-0.1

dropout_train_err, dropout_test_err = [], []
for epoch in range(epochs):
    correct_cnt, test_correct_cnt = 0.0, 0.0
    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_1 = tanh(np.dot(layer_0, w_0_1)+b_0_1)
```

```
        # ***** 从二项分布中产生随机数
        dropout = np.random.binomial(1, retain_prob, \
                    size=layer_1.shape)
        # ***** 失活 dropout 为 0 对应的节点，之后除以保留比例
        layer_1 *= dropout/retain_prob

        layer_2 = softmax(np.dot(layer_1,w_1_2)+b_1_2)

        labels_batch = labels[batch_start:batch_end]
        for k in range(batch_size):
            correct_cnt += int(np.argmax(layer_2[k:k+1])== \
                               np.argmax(labels_batch[k:k+1]))

        layer_2_delta=(layer_2-labels[batch_start:batch_end])/\
                                                    batch_size
        # 隐藏层 delta
        layer_1_delta=layer_2_delta.dot(w_1_2.T)* \
                                            tanh2deriv(layer_1)
        # ***** dropout 为 0 的节点对应的 delta 设为 0
        layer_1_delta *= dropout
        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)
        w_1_2 -= lr * layer_1.T.dot(layer_2_delta)
        w_0_1 -= lr * layer_0.T.dot(layer_1_delta)

    # 计算测试误差，正常使用神经网络
    layer_0 = test_images
    layer_1 = tanh(np.dot(layer_0, w_0_1)+b_0_1)
    layer_2 = softmax(np.dot(layer_1,w_1_2)+b_1_2)
    for i in range(len(test_images)):
        test_correct_cnt += int(np.argmax(layer_2[i:i+1])==\
                            np.argmax(test_labels[i:i+1]))

    dropout_train_err.append(1-correct_cnt/float(len(images)))
    dropout_test_err.append(1-test_correct_cnt/ \
                                        float(len(test_images)))

    if (epoch % 300==0 or epoch==epochs-1):
        # 每 300 epochs 输出训练误差，测试误差
        print("epoch: %4d; Train_Err: %0.3f; Test_Err: %0.3f"%\
              (epoch, dropout_train_err[-1], \
                      dropout_test_err[-1]))
```

```
epoch:    0; Train_Err: 0.586; Test_Err: 0.368
epoch:  300; Train_Err: 0.003; Test_Err: 0.131
epoch:  600; Train_Err: 0.003; Test_Err: 0.125
epoch:  900; Train_Err: 0.001; Test_Err: 0.124
epoch: 1200; Train_Err: 0.000; Test_Err: 0.123
epoch: 1500; Train_Err: 0.001; Test_Err: 0.121
epoch: 1800; Train_Err: 0.000; Test_Err: 0.120
epoch: 2100; Train_Err: 0.000; Test_Err: 0.119
epoch: 2400; Train_Err: 0.000; Test_Err: 0.120
epoch: 2700; Train_Err: 0.001; Test_Err: 0.119
```

```
epoch: 2999; Train_Err: 0.000; Test_Err: 0.119
```

从代码运行结果可以看出，Dropout 方法在 mnist 数据中可以较好地控制过拟合，训练误差下降得更慢，同时测试误差下降到了 0.119。图 6-12 画出有 Dropout 方法和无 Dropout 方法两种情况下训练误差和测试误差随着迭代次数的增加而变化的曲线。从图 6-12 可以看出，和标准神经网络相比，应用了 Dropout 的神经网络训练误差下降得更慢，测试误差也下降得更慢。但是，应用了 Dropout 的神经网络模型最终取得了更小的测试误差。

```
plt.plot(np.arange(len(dropout_train_err)), dropout_train_err,\
         label="训练误差（dropout）")
plt.plot(np.arange(len(train_err)), train_err, linestyle=":", \
         label="训练误差（no dropout）")
plt.plot(np.arange(len(dropout_test_err)), dropout_test_err, \
         label="测试误差（dropout）")
plt.plot(np.arange(len(test_err)), test_err, linestyle=":", \
         label="测试误差（no dropout）")
plt.ylim((-0.01, 0.3))
plt.legend()
plt.xlabel("迭代次数", fontsize=16)
plt.ylabel("模型误差", fontsize=16)
plt.show()
```

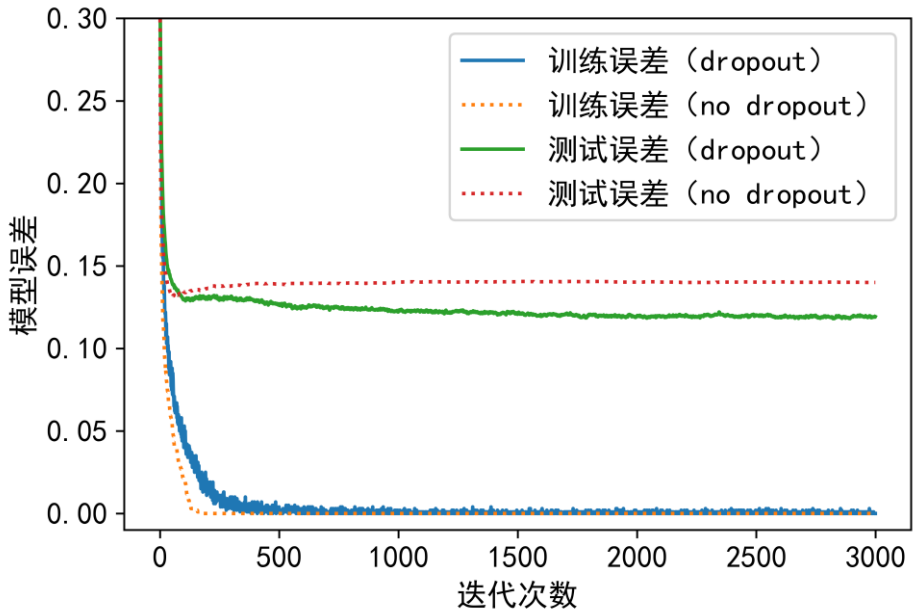

图 6-12 训练误差和测试误差（考虑使用 Dropout 和没有使用 Dropout 两种情况）

6.3.4 增加观测点

在实际应用中，增加观测点也可以减少过拟合。沿用 6.2 节的类比例子，学生通过 100 道习题来学习函数相关知识，然后参加考试来测试知识的掌握程度。

假设一个学生在 10 个小时内学习 100 道习题，结果过度学习了。一个简单的解决过度学习方法是增加习题数量，并且增加习题的多样性。例如，如果新增的习题里出现了$b = g(a)$这样表示函数的形式，学生就会知道，只要满足$y = f(x)$的形式就可以称做函数，而符号并不是很重要。因此，在一定的学习时间内，增加习题数量和多样性可以减少学生过度学习的程度。

在神经网络中，其原理是类似的。在同样的神经网络结构下，用同样的时间训练模型，更多观测点可以减少过拟合，提高模型的预测准确度。但是，在现实中，因为有更多观测点，我们可能会尝试更复杂的神经网络模型。而更复杂的神经网络模型就可以造成较大的过拟合。因此，在现实中，增加观测点有可能让模型表现更好，但是通常也需要结合其他的正则化方法。另一个增加观测点的缺点是，在现实中，收集更多的数据意味着需要投入更高昂的资金和人力成本。总的来说，增加观测点可以减少过拟合现象。但是严格的说，增加观测点不算是一种正则化方法。

6.4 本章小结

在本章中，我们学习了评估神经网络模型预测效果的方法，知道了训练误差和测试误差的区别。我们还学习了神经网络模型经常出现的两类问题：欠拟合和过拟合。通过观察训练误差和验证误差，可以判断神经网络模型是欠拟合还是过拟合，并且改进神经网络的表现。欠拟合和过拟合的判断依据和改进措施可以总结如表 6-2。

表 6-2 欠拟合和过拟合的判断依据和改进措施

两类问题	判断方法	改进措施
欠拟合	1.训练误差和验证误差差距较小 2.训练误差和验证误差都比较大	1.增加隐藏层 2.增加隐藏层节点
过拟合	1.训练误差和验证误差差距较大 2.训练误差较小，验证误差较大，且验证误差开始上升	1.早停法 2.L_2范数 3. Dropout 4. 增加观测点

习题

1. 分析 mnist 数据，建立具有 1 个隐藏层的神经网络，尝试不同隐藏层节点个数，观察欠拟合和过拟合现象。
2. 分析 Fasion-mnist 数据，建立具有 1 个隐藏层的神经网络，使用早停法控制过拟合。在该例子中，早停法会过早的停止训练模型，使我们没能得到更优的神经网络模型吗？如果会，我们可以怎么改进早停法？
3. 分析 Fasion-mnist 数据，建立具有两个隐藏层的神经网络，使用L_2惩罚法控制过拟合。尝试多个不同的惩罚项参数λ。
4. 分析 Fasion-mnist 数据，建立具有两个隐藏层的神经网络，使用 Dropout 控制过拟合。尝试多个 Dropout 方法保留节点的比例。
5. 分析 Fasion-mnist 数据，建立具有两个隐藏层的神经网络，同时使用L_2惩罚和 Dropout 控制过拟合。
6. 分析 Default 数据，建立具有 1 个隐藏层的神经网络，得到误差矩阵，计算错误率、假阳性率和假阴性率。