第五周学习笔记（算法实现）

2022-05-12

第一章　多类分类

首先，导入我们需要的包。

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.io import loadmat
from scipy.optimize import minimize
from sklearn.metrics import classification_report
```

1.1　数据集的初步认识

在逻辑回归和正则化的分类问题中，我们曾经尝试使用决策边界划分 0 和 1，现在将完成多类分类 (即多个 *logistic* 回归)。

在本练习中，我们将去实现一个 $one-vs-all$ 逻辑回归和神经网络来识别手写数字（0-9）。

在之前的学习中数据均为文本数据可以直接打开看，此次作业的数据是 *matlab* 的数据文件格式，需要使用 *scipy* 工具（*loadmat*）来加载数据。

我们先看一下数据集的数据：

```
data = loadmat('ex3data1.mat')
print(data)
```

数据展示如下：

```
{'__header__': b'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Sun
                                    Oct 16 13:09:09 2011', '__version__':
                                     '1.0', '__globals__': [], 'X': array
                                    ([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
```

```
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]]), 'y': array([[10],
       [10],
       [10],
       ...,
       [ 9],
       [ 9],
       [ 9]], dtype=uint8)}
```

从上面的数据中我们可以看出，y 是标签，y 的数值从 10 到 1，分别代表数字 9 到 0。

同时查看一下 X 矩阵、y 矩阵的维度以及标签分类的数目：

```
def loaddata(path):
    data=loadmat(path)
    X = data['X']
    y = data['y']
    return X, y
X, y = loaddata('ex3data1.mat')
print(X.shape,y.shape,np.unique(y))
```

结果展示如下：

```
(5000, 400) (5000, 1) [ 1  2  3  4  5  6  7  8  9 10]
```

根据数据集的介绍文件，我们知道共计有 5000 个训练样本，每个样本是 20*20 像素的数字的灰度图像。每个像素代表一个浮点数，表示该位置的灰度强度。20×20 的像素网格被展开成一个 400 维的向量。在我们的数据矩阵 X 中，每一个样本都变成了一行，这给了我们一个 5000×400 矩阵 X，每一行都是一个手写数字图像的训练样本，手写数字的像素点有值为 255，空白像素点的位置为 0。

$$X = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \cdots \\ (x^{(m)})^T \end{bmatrix} \tag{1.1}$$

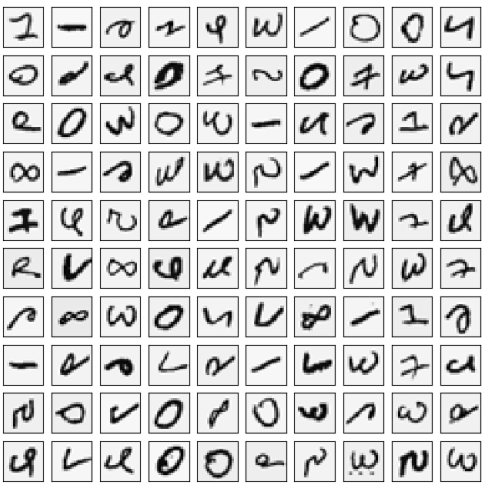
命题 1.1.1. *单个像素点通过 8 位（二进制数）的灰度值（0-255）来表示。例如一幅 500*500 像素的单通道灰度图是由 500X500=250000 个不同灰度的像素点组成。*

1.2　数据的可视化

数据可视化对观察数据有很好的帮助。代码从 X 中随机选择 100 行并将这些行传递给我们定义的函数。此函数将每一行映射到 20×20 像素的灰度图像，并将图像一起显示。我们提供了如下的函数。

```
def plot_random_100_image(X):
    sample_index = np.random.choice(np.arange(X.shape[0]), 100)#随机采样函
                                                数
    sample_images = X[sample_index, :]#选取该行，所有列
    fig, ax_array = plt.subplots(nrows=10, ncols=10, sharey=True, sharex=
                                            True, figsize=(8, 8))#10×10共100
                                            张图，总窗口大小为8×8，所有图像共
                                            享x,y轴属性
for row in range(10):#行
    for column in range(10):#列
        ax_array[row, column].matshow(sample_images[10 * row + column].
                                                reshape((20, 20)),cmap='
                                                gray_r')
    plt.xticks([])#隐藏x,y轴坐标刻度
    plt.yticks([])
    plt.show()
plot_random_100_image(X):
```

运行此步骤后，我们将会看到如下图所示的图像：



1.3　向量化逻辑回归

我们接下来将使用多个 $one-vs-all$(一对多)$logistic$ 回归模型来构建一个多类分类器。由于有 10 个类，需要训练 10 个独立的分类器。为了提高训练效率，重要的是向量化。所以，我们将实现一个向量化的 $logistic$ 回归版本。

首先回顾一下之前学习的 $logistic$ 回归的假设函数：

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T X}}$$

其中的 g 为 $sigmoid$ 函数，$g(z) = \frac{1}{1+e^{-z}}$。

正则化的 $logistic\quad regression$ 的代价函数的定义如下：

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[-y^{(i)}log(h_\theta(x^{(i)})) - (1-y^{(i)})log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

对于每个样本 i 都需要求 $sigmoid$ 和 $h_\theta(x)$，事实上我们可以对所有的样本用矩阵乘法来快速的计算。让我们如下来定义 X 和 θ：

$$X = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \cdots \\ (x^{(m)})^T \end{bmatrix}, \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \cdots \\ \theta_n \end{bmatrix} \tag{1.2}$$

然后通过计算矩阵积 $X \times \theta$，我们可以得到：

$$X\theta = \begin{bmatrix} (x^{(1)})^T\theta \\ (x^{(2)})^T\theta \\ \cdots \\ (x^{(m)})^T\theta \end{bmatrix} = \begin{bmatrix} \theta^T(x^{(1)}) \\ \theta^T(x^{(2)}) \\ \cdots \\ \theta^T(x^{(m)}) \end{bmatrix} \tag{1.3}$$

在最后一个等式中，我们用到了一个定理:

定理 1.3.1. 如果 a 和 b 都是向量，那么 $a^T \times b = b^T \times a$。

同正则化逻辑回归一样，我们定义 $sigmoid$ 函数以及代价函数 $Cost$。

首先是 $sigmoid$ 函数，代码如下：

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

接下来是代价函数 $Cost$：

```
def Cost(theta, X, y, l):
    theta_reg = theta[1:]
    first = (-y * np.log(sigmoid(X @ theta))) + (y - 1) * np.log(1 -
                                            sigmoid(X @ theta))
    reg = (theta_reg @ theta_reg) * l / (2 * len(X))
    return np.mean(first) + reg
```

1.4　向量化梯度下降

1.4.1　非正则化

回想一下，（非正则化的）逻辑回归成本的梯度是一个向量，第 j 个元素被定义为 $\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}((h_\theta(x^{(i)}) - y_i)x_j^{(i)})$。

为了在数据集上向量化这个操作，我们首先显式地写出所有 θ_j 的偏导数:

$$\begin{aligned}
\begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{bmatrix} &= \frac{1}{m}\begin{bmatrix} \sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}\right) \\ \sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)}\right) \\ \sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_2^{(i)}\right) \\ \vdots \\ \sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_n^{(i)}\right) \end{bmatrix}^{\top} \\
&= \frac{1}{m}\sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x^{(i)}\right) \\
&= \frac{1}{m}X^T(h_\theta(x) - y).
\end{aligned}$$

$$h_\theta(x) - y = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(1)}) - y^{(m)} \end{bmatrix}.$$

其中：

注意 $x^{(i)}$ 是向量，而 $h_\theta((x^{(i)}) - y^{(i)})$ 是单一的数。

为了理解推导的最后一步，令 $\beta_i = h_\theta((x^{(i)}) - y^{(i)})$，并且注意：

$$\sum_i \beta_i x^{(i)} = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} = X^T\beta,$$

上面的表达式允许我们计算所有的偏导数，而没有任何循环。

1.4.2　带正则化

在实现了逻辑回归的向量化后，现在将向成本函数中添加正则化。回想一下，对于正则化的逻辑回归，成本函数被定义为:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log(h_\theta(x^{(i)})) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2.$$

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \qquad \text{for } j = 0$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j \qquad \text{for } j \geq 1$$

因为不惩罚 θ_0，所以分为两种情况：

Repeat until convergence{

$$\theta_0 := \theta_0 - a\frac{1}{m}\sum_{i=1}^{m}((h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)})$$

$$\theta_j := \theta_j - a[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j]$$

$$for\ j = 1, 2, \dots n$$

}

```
def gradient(theta, X, y, l):
        theta_reg = theta[1:]
        grand = (X.T @ (sigmoid(X @ theta) - y)) / len(X)
        # 这里人为插入一维0，使得对theta_0不惩罚，方便计算
        reg = np.concatenate([np.array([0]), (l / len(X)) * theta_reg])
        return grand + reg # 注意返回的是一个向量，维度与 theta一致
```

现在我们已经完成了定义代价函数和梯度函数，接下来是构建分类器的时候了。

1.5　一对多分类

这部分我们将实现一对多分类，通过训练多个正则化 *logistic* 回归分类器。

对于这个任务，我们有 10 个可能的类，并且由于逻辑回归只能一次在 2 个类之间进行分类，我们需要多类分类的策略。在本练习中，我们的任务是实现一对一全分类方法，其中具有 k 个不同类的标签就有 k 个分类器，每个分类器在“类别 i”和“不是 i”之间决定。我们将把分类器训练包含在一个函数中，该函数计算 10 个分类器中的每个分类器的最终权重，并将权重返回为 $k * n + 1$ 数组，其中 n 是参数数量。

之前的两次作业中，在定义代价函数和梯度下降函数后，我们都是手动定义学习率 α 和梯度下降的迭代次数 *iters* 来得到最终优化的参数集 θ 和 θ_0。

本练习中我们利用 *scipy* 库中的一个优化函数来自动优化模型参数，该函数可以根据输入的参数返回最终优化的参数集，其定义如下如下：

```
scipy.optimize.minimize(fun, x0, args=(), method=None, jac=None, hess=None,
                                        hessp=None, bounds=None, constraints
                                       =(), tol=None, callback=None, options
                                       =None)
```

输入参数有一大堆，本练习中我们只用到以下几个：

```
fun:要优化的函数,即代价函数Cost
x_0: 初始的参数值, 即theta
args=(): 其他参数, 例如X XX, y yy 以及正则化项系数lambda等
method: 求极值的方法, 官方文档给了很多种, 我们选择TNC算法（截断牛顿法）
jac: 梯度下降算法中的梯度向量
```

接下来我们去实现模型：

```
def one_vs_all(X, y, l, num_labels):  # num_labels = 10
    all_theta = np.zeros((num_labels, X.shape[1]))
    # 大theta矩阵包含theta_1-theta_10,维度为(10, 401)
    for i in range(1, num_labels + 1):
        theta = np.zeros(X.shape[1])  # 求theta_i
        y_i = np.array([1 if label == i else 0 for label in y])
        ret = minimize(fun=Cost, x0=theta, args=(X, y_i, l), method='TNC',
                                            jac=gradient, options={'disp'
                                            : True})
```

```
        all_theta[i - 1, :] = ret.x  # 将求好的theta_i赋值给theta矩阵
    return all_theta
```

此时的 all theta 为 10×401 维，每一行对应一个分类器的 θ_i 如（判别是不是数字 1 的 θ_1，判别是不是数字 2 的 θ_2）初始化 X，y，求出 all theta。

```
X1, y1 = loaddata('ex3data1.mat')
X = np.insert(X1, 0, 1, axis=1)  # 在第0列, 插入数字1, axis控制维度为列(
                                    5000, 401)
y = y1.flatten()  # 这里消除了一个维度, 方便后面的计算 or .reshape(-1) (
                                    5000, )
all_theta = one_vs_all(X, y, 1, 10)
print(all_theta) # 每一行是一个分类器的一组参数
```

```
[[-2.38187334e+00  0.00000000e+00  0.00000000e+00 ...  1.30433279e-03
  -7.29580949e-10  0.00000000e+00]
 [-3.18303389e+00  0.00000000e+00  0.00000000e+00 ...  4.46340729e-03
  -5.08870029e-04  0.00000000e+00]
 [-4.79638233e+00  0.00000000e+00  0.00000000e+00 ... -2.87468695e-05
  -2.47395863e-07  0.00000000e+00]
 ...
 [-7.98700752e+00  0.00000000e+00  0.00000000e+00 ... -8.94576566e-05
   7.21256372e-06  0.00000000e+00]
 [-4.57358931e+00  0.00000000e+00  0.00000000e+00 ... -1.33390955e-03
   9.96868542e-05  0.00000000e+00]
 [-5.40542751e+00  0.00000000e+00  0.00000000e+00 ... -1.16613537e-04
   7.88124085e-06  0.00000000e+00]]
```

接下来通过 $theta$ 来求预估的 $h_\theta(x)$，便于后面与真实值做对比求模型的准确度。这里的 h 共 5000 行，10 列，每行代表一个样本，每列是预测对应数字的概率。我们取概率最大对应的下标加 1，就是我们分类器最终预测出来的类别。

此外，最终返回的 h_{argmax} 是一个数组，包含 5000 个样本对应的预测值。程序实现如下：

```
def predict(X, all_theta):
    h = sigmoid(X @ all_theta.T)# 注意的这里的all_theta需要转置
    h_argmax = np.argmax(h, axis=1)# 找到每行的最大索引（0-9）
    h_argmax = h_argmax + 1# +1后变为每行的分类器 （1-10）
    return h_argmax
y_predict = predict(X, all_theta)
```

```
accuracy = np.mean(y_predict == y)
print ('accuracy = {0}%'.format(accuracy * 100))
```

```
accuracy = 94.46%
```

我们可以生成报告如下：

```
print(classification_report(y, y_predoct))
```

```
              precision    recall  f1-score   support
           1       0.95      0.99      0.97       500
           2       0.95      0.92      0.93       500
           3       0.95      0.91      0.93       500
           4       0.95      0.95      0.95       500
           5       0.92      0.92      0.92       500
           6       0.97      0.98      0.97       500
           7       0.95      0.95      0.95       500
           8       0.93      0.92      0.92       500
           9       0.92      0.92      0.92       500
          10       0.97      0.99      0.98       500
    accuracy                           0.94      5000
   macro avg       0.94      0.94      0.94      5000
weighted avg       0.94      0.94      0.94      5000
```

从上面的评估结果我们可以看出来，准确率为 94%，模型较好。

第二章　神经网络-前向传播算法

首先，导入我们需要的包。

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.io import loadmat
from scipy.optimize import minimize
from sklearn.metrics import classification_report
```

在上面的第一章节中，我们使用了多个 *logistic* 回归来取得对应的预测结果，但是我们知道 *logistic* 回归不能形成更复杂的假设，因为它只是一个线性分类器。

接下来我们用神经网络来尝试解决问题，神经网络可以实现非常复杂的非线性模型。我们将利用已经训练好了的权重进行预测。

权重数据存储在文件 *ex3weights.mat* 中。

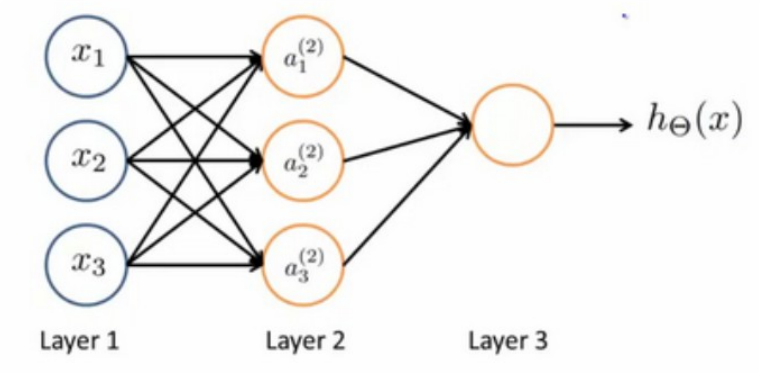
2.1　数据展示

```
data = loadmat('ex3data1.mat')
x = data['X']
 y = data['y']
x = np.insert(x, 0, values=1, axis=1)
print('x.shape=',x.shape)
y = y.flatten()
print('y.shape=',y.shape)
theta = loadmat('ex3weights.mat')
theta1 = theta['Theta1']
print('theta1.shape=',theta1.shape)
theta2 = theta['Theta2']
print('theta2.shape=',theta2.shape)
```

输出结果为：

```
x.shape= (5000, 401)
y.shape= (5000,)
theta1.shape= (25, 401)
theta2.shape= (10, 26)
```

2.2　模型展示



其中 x_1, x_2, x_3 是输入单元，有必要的时候，我们会额外增加一个额外的节点 x_0，这个额外的节点我们称为偏置单元或者偏置神经元，我们将原始数据输入给它们。a_1, a_2, a_3 是中间单元，它们负责将数据进行处理，然后呈递给下一层。最后是输出单元，它负责计算 $h_\Theta(x)$。

结合之前的逻辑回归，我们知道 $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$。

神经网络模型是许多逻辑单元按照不同层级组织起来的网络，每一层的输出变量都是下一层的输入变量。

对于上图所示的模型，激活单元和输出分别表示为：

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$

$$h_\theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

我们可以知道，每一个 a 都是由上一层所有的 x 和对应的参数 θ 所决定的。

总结一下，其中的神经网络定义了函数 $h_\theta(x)$，从输入 x 到输出 y 的映射，这些假设被参数化，我们可以将参数记为 θ，这样一来改变 θ 就能得到不同的假设，就能得到不同的函数，比如从 x 到 y 的映射。

我们把这样的算法称为前向传播算法。

其中，x, θ, a 的矩阵分别表示为：

$$X = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}, \theta = \begin{bmatrix} \theta_{10}, \theta_{11}, \theta_{12}, \theta_{13} \\ \theta_{20}, \theta_{21}, \theta_{22}, \theta_{23} \\ \theta_{30}, \theta_{31}, \theta_{32}, \theta_{33} \end{bmatrix}, a = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{2.1}$$

通过上面的方程，我们计算出了三个隐藏单元的激活值，然后利用这些值计算出最终输出假设函数 $h_\theta(x)$ 的值，接下来，我们定义额外的项：

$$z_1^{(2)} = \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3$$

$$z_2^{(2)} = \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3$$

$$z_3^{(2)} = \Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3$$

这些 z 值都是线性组合，将某个特定的神经元的输入值 x_0, x_1, x_2, x_3 的加权线性组合构成。所以我们得到的算法为：$z^{(2)}$=$\Theta^{(1)} x$，$a^{(2)} = g(z^{(2)})$，$z^{(3)} = \Theta^{(2)} a^{(2)}$，$h_\theta(x) = a^{(3)} = g(z^{(3)})$。

2.3　前向传播算法

因为权重的数据已经存在，所以我们直接调用就可以。

前向传播就是按照网络顺序计算出最后的结果，计算的具体方式，核心思想是上一层的每个节点相对于下一层的某个节点都有对应的权重，所有权重组合起来就是参数 θ。

需要注意每层需要添加偏置单元，一般是在该层其他单元结果计算完后再加。

实现之前我们要理解 θ 是怎么存储的，第 i 行就是对应计算第 i 个单元的全部参数（参数个数等于前一层的单元数），因此在实现时需要将 θ 转置。

实现前向传播加最后的预测，其中 θ 接受元组或列表型参数，其中存储每层对应的参数 θ，最后求输出层值最大的单元号作为分类结果。

代码实现结果如下，我们可以看一下数据的维度如何：

```
# 输入层
a1=x
print('a1.shape=',a1.shape)
# 隐藏层
z2 = x @ theta1.T
print('z2.shape=',z2.shape)
a2 = sigmoid(z2)
print('a2.shape=',a2.shape)
# 输出层
a2 = np.insert(a2, 0, values=1, axis=1)
print('a2.shape=',a2.shape)
z3 = a2 @ theta2.T
print('z3.shape=',z3.shape)
a3=sigmoid(z3)
print('a3.shape=',a3.shape)
```

我们得到每一层数据的维度如下：

```
a1.shape= (5000, 401)
z2.shape= (5000, 25)
a2.shape= (5000, 25)
a2.shape= (5000, 26)
z3.shape= (5000, 10)
a3.shape= (5000, 10)
```

这个网络模型除输入层和输出层外，只有一层隐藏层，输入层为 400 个单元，隐藏层为 25 个单元，输出层为 10 个单元。我对隐藏层的作用的理解就是自动组合生成更好的特征。

接下来使用之前读入的已经训练好的参数，直接预测。

```
y_pre = np.argmax(a3, axis=1)
y_pre = y_pre + 1
acc = np.mean(y_pre == y)
print('accuracy = {}%'.format(acc * 100))
```

我们可以得到的结果如下：

```
accuracy = 97.52%
```

可以看到拟合的效果比逻辑回归的效果更好，我认为就是自动训练出来得特征起得效果，这是隐藏层做的事，可以通过增加隐藏层来提高准确度。但是神经网络计算量也

是非常的大。

最后，我们可以生成准确率的报告：

```
report=classification_report(y_pre,y)
print(report)
```

```
              precision    recall  f1-score   support

           1       0.98      0.97      0.98       507
           2       0.97      0.98      0.98       494
           3       0.96      0.98      0.97       491
           4       0.97      0.97      0.97       499
           5       0.98      0.97      0.98       506
           6       0.99      0.98      0.98       504
           7       0.97      0.98      0.97       496
           8       0.98      0.98      0.98       502
           9       0.96      0.97      0.96       496
          10       0.99      0.98      0.99       505

    accuracy                           0.98      5000
   macro avg       0.98      0.98      0.98      5000
weighted avg       0.98      0.98      0.98      5000
```

第三章　神经网络-反向传播算法

首先，导入我们需要的包。

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.io import loadmat
from scipy.optimize import minimize
from sklearn.metrics import classification_report
from sklearn.preprocessing import OneHotEncoder
```

3.1　可视化数据

对于这个练习，我们将再次处理手写数字数据集，这次使用反向传播的前馈神经网络。我们将通过反向传播算法实现神经网络成本函数和梯度计算的非正则化和正则化版本。我们还将实现随机权重初始化和使用网络进行预测的方法。

由于我们在上个练习中使用的数据集是相同的，所以我们将重新使用代码来加载数据。

```
data = loadmat('ex4data1.mat')
print(data)
```

数据结果展示如下：

```
{'__header__': b'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Sun
                                    Oct 16 13:09:09 2011', '__version__':
                                     '1.0', '__globals__': [], 'X': array
                                    ([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
```

```
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]]), 'y': array([[10],
       [10],
       [10],
       ...,
       [ 9],
       [ 9],
       [ 9]], dtype=uint8)}
```

由于我们以后需要这些（并将经常使用它们），我们先来创建一些有用的变量，并展示一下维度大小。

```
X = data['X']
y = data['y'].flatten() #降维
print('X.shape=',X.shape)
print('y.shape=',y.shape)
```

结果如下：

```
X.shape= (5000, 400)
y.shape= (5000,)
```

首先我们要将标签值（1，2，3，4，…，10）转化成非线性相关的向量，向量对应位置（$y[i-1]$）上的值等于 1，例如 $y[0]=6$ 转化为 $y[0]=[0,0,0,0,0,1,0,0,0,0]$。

我们也需要对我们的 y 标签进行一次 $one-hot$ 编码。$one-hot$ 编码将类标签 n（k 类）转换为长度为 k 的向量，其中索引 n 为 “hot”（1），而其余为 0。$Scikit-learn$ 有一个内置的实用程序，我们可以使用这个。

定义 3.1.1. $One-hot$ $Encoding$ 是一种向量表示，其中向量中的所有元素都是 0，除了一个，它的值是 1，其中 1 表示指定元素类别的布尔值。

我们的代码实现如下：

```
encoder=OneHotEncoder(sparse=False)
y_onehot=encoder.fit_transform(y)
print('y_onehot=\n',y_onehot)
print('y_onehot.shape=',y_onehot.shape)
```

展示的结果为：

```
y_onehot=
 [[0. 0. 0. ... 0. 0. 1.]
```

```
 [0. 0. 0. ... 0. 0. 1.]
 [0. 0. 0. ... 0. 0. 1.]
 ...
 [0. 0. 0. ... 0. 1. 0.]
 [0. 0. 0. ... 0. 1. 0.]
 [0. 0. 0. ... 0. 1. 0.]]
y_onehot.shape= (5000, 10)
```

我们要为此练习构建的神经网络具有与我们的实例数据（400 + 偏置单元）大小匹配的输入层，25 个单位的隐藏层（带有偏置单元的 26 个），以及一个输出层，10 个单位对应我们的一个 $one-hot$ 编码类标签。

我们需要实现的第一件是评估一组给定的网络参数的损失的代价函数。

3.2 *sigmoid* 函数

g 代表一个常用的逻辑函数（$logistic\quad function$）为 S 形函数（$Sigmoid\quad function$），公式为：$g(z)=\frac{1}{e^{-z}}$，合起来，我们得到模型的假设函数 $h_\theta(x)=\frac{1}{e^{-\theta^T x}}$。

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

3.3 前向传播函数

我们由之前的神经网络传播图得知，输入层的特征为 400 个，加上一个偏置单元为 401 个，同理隐藏层为（25+1）个，输出层为 10 个。

```
def foward_propagate(X,theta1,theta2):
    m=X.shape[0] #m=5000
    a1=np.insert(X,0,values=np.ones(m),axis=1)
    z2=a1*theta1.T
    a2=np.insert(sigmoid(z2),0,values=np.ones(m),axis=1)
    z3=a2*theta2.T
    h=sigmoid(z3)
    return a1,z2,a2,z3,h
```

3.4　代价函数

接下来我们需要定义代价函数，公式如下：

$$h_\Theta(x) = a^{(L)} = g(\Theta^{(L-1)}a^{(L-1)}) = g(z^{(L)})$$

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)}\log((h_\Theta(x^{(i)}))_k) + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2$$

> L: 神经网络的总层数
>
> s_l: 第 l 层激活单元的数量（不包含偏置单元）
>
> $h_\Theta(x)_k$: 分为第 k 个分类(k^{th})的概率 $P(y=k|x;\Theta)$
>
> K: 输出层的输出单元数量，即类数 - 1
>
> $y_k^{(i)}$: 第 i 个训练样本的第 k 个分量值
>
> y: K 维向量

在神经网络的代价函数中，左边的变化实际上是为了求解 K 分类问题，即公式会对每个样本特征都运行 K 次，并依次给出分为第 k 类的概率，即 $h_\Theta(x) \in R^K$ $y \in R^K$。

右边的正则化项比较容易理解，每一层有多维矩阵 $\Theta^{(l)} \in R^{(s_l+1)\times s_{l+1}}$，从左到右看这个三次求和式 $\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}$，就是对每一层间的多维矩权重 $\Theta^{(l)}$，依次平方后求取其除了偏置权重部分的和值，并循环累加即得结果。

再次可见，神经网络背后的思想是和逻辑回归一样的，但由于计算复杂，实际上神经网络的代价函数 $J(\Theta)$ 是一个非凸 $(non-convex)$ 函数。

我们使用 $Python$ 实现如下：

```
def cost(param,input_size,hide_size,label_num,X,y,learning_rate):
    m=X.shape[0]
    X=np.matrix(X)
    y=np.matrix(y)
    # 将参数数组解开为每个层的参数矩阵，reshape重新定义维度
    theta1=np.matrix(np.reshape(param[:hide_size * (input_size + 1)], (
                                        hide_size, (input_size + 1))))
    theta2=np.matrix(np.reshape(param[hide_size*(input_size+1):],(label_num
                                        ,(hide_size+1))))
    a1,z2,a2,z3,h=foward_propagate(X,theta1,theta2)
    J=0
    for i in range(m):
        first=np.multiply(-y[i,:],np.log(h[i,:]))
```

```
        second=np.multiply((1-y[i,:]),np.log((1-h[i,:])))
        J=J+np.sum(first-second)
    J=J/m
    return J
```

前向传播函数计算给定当前参数的每个训练实例的假设。它的输出形状应该与 y 的一个 $one-hot$ 编码相同。

3.5　初始化设置

3.5.1　随机生成 $theta1$,$theta2$

```
input_size=400
hide_size=25
label_num=10
learning_rate=1
param=(np.random.random(size=hide_size*(input_size+1)+label_num*(hide_size+
                                        1))-0.5)*0.25
# np.random.random生成（-0.5~0.5）*0.25的浮点型随机数组
m=X.shape[0]
X=np.matrix(X)
y=np.matrix(y)
theta1 = np.matrix(np.reshape(param[:hide_size * (input_size + 1)], (
                                        hide_size, (input_size + 1))))
theta2 = np.matrix(np.reshape(param[hide_size * (input_size + 1):], (
                                        label_num, (hide_size + 1))))
```

3.5.2　根据数据集所给权重来生成 $theta1$,$theta2$

这里我们提供了已经训练好的参数 θ_1 θ_2，存储在 $ex4weight.mat$ 文件中。这些参数的维度由神经网络的大小决定，第二层有 25 个单元，输出层有 10 个单元 (对应 10 个数字类)。

我们可以直接读取 θ_1 和 θ_2 的值。

```
weight = loadmat('ex4weights.mat')
Theta1, Theta2 = weight['Theta1'], weight['Theta2']
```

当我们用这两个权重来计算代价函数的值时，代码实现可以为：

```
def cost_1(Theta1, Theta2, input_size, hide_size, num_labels, X, y,
                                        learning_rate):
    m = X.shape[0]
    X = np.matrix(X)
    y = np.matrix(y)
    a1, z2, a2, z3, h = foward_propagate(X, Theta1, Theta2)
    J = 0
    for i in range(m):
        first = np.multiply(-y[i, :], np.log(h[i, :]))
        second = np.multiply(1 - y[i, :], np.log(1 - h[i, :]))
        J = J + np.sum(first - second)
    J = J / m
    return J
```

我们接下来看一下权重参数的维度，代码及结果如下显示：

```
print('theta1.shape=',theta1.shape)
print('theta2.shape=',theta2.shape)
```

```
theta1.shape= (25, 401)
theta2.shape= (10, 26)
```

接下来我们来看一下前向传播算法计算的各个参数的维度：

```
a1,z2,a2,z3,h=foward_propagate(X,theta1,theta2)
print('a1.shape=',a1.shape)
print('z2.shape=',z2.shape)
print('a2.shape=',a2.shape)
print('z3.shape=',z3.shape)
print('h.shape=',h.shape)
```

结果如下所示：

```
a1.shape= (5000, 401)
z2.shape= (5000, 25)
a2.shape= (5000, 26)
z3.shape= (5000, 10)
h.shape= (5000, 10)
```

3.5.3　计算代价函数

我们还可以计算上面定义的代价函数的值，结果如下：

```
print('cost=',cost(param,input_size,hide_size,label_num,X,y_onehot,
                                        learning_rate))
```

```
cost= 7.200986460136575
```

```
print('cost_1=',cost_1(Theta1, Theta2, input_size, hide_size, label_num, X,
                                        y_onehot, learning_rate))
```

我们计算结果如下：

```
cost_1= 0.2876291651613187
```

综上所述，第一种随机生成结果可能不同，第二种结果：0.2876291651613187。

3.6　正则化代价函数

我们的下一步是增加代价函数的正则化。它实际上并不像看起来那么复杂，事实上，正则化术语只是我们已经计算出的代价的一个补充。

下面是修改后的代价函数。

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}(\Theta_{j,k}^{(1)})^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}(\Theta_{j,k}^{(2)})^2\right].$$

注意不要将每层的偏置项正则化。

我们用 *Python* 代码实现如下：

```
def reg_cost(param,input_size,hide_size,label_num,X,y,learning_rate):
    m=X.shape[0]
    X=np.matrix(X)
    y=np.matrix(y)
    theta1=np.matrix(np.reshape(param[:hide_size * (input_size + 1)], (
                                        hide_size, (input_size + 1))))
    theta2 = np.matrix(np.reshape(param[hide_size * (input_size + 1):], (
                                        label_num, (hide_size + 1))))
```

```
    a1, z2, a2, z3, h = foward_propagate(X, theta1, theta2)
    J = 0
    for i in range(m):
        first = np.multiply(-y[i, :], np.log(h[i, :]))
        second = np.multiply((1 - y[i, :]), np.log(1 - h[i, :]))
        J =J+ np.sum(first - second)
    J = J / m
    J += (float(learning_rate) / (2 * m)) * (np.sum(np.power(theta1[:, 1:],
                                          2)) + np.sum(np.power(theta2[:,
                                         1:], 2)))
    return J
```

我们继续计算此时的代价函数的值：

```
print('reg_cost=',reg_cost(param, input_size, hide_size, label_num, X,
                                    y_onehot, learning_rate))
```

```
reg_cost= 7.2062719095901775
```

当我们利用题目给出的权重时，带正则化的代价函数和代价函数的值为：

```
def reg_cost_1(Theta1, Theta2, input_size, hide_size, num_labels, X, y,
                                    learning_rate):
    J = cost_1(Theta1, Theta2, input_size, hide_size, num_labels, X, y,
                                        learning_rate)
    J += (float(learning_rate) / (2 * m)) * (
          np.sum(np.power(Theta1[:, 1:], 2)) + np.sum(np.power(Theta2[:, 1:
                                            ], 2)))
    return J
```

```
print('reg_cost_1=', reg_cost_1(Theta1, Theta2, input_size, hide_size,
                                    label_num, X, y_onehot, learning_rate
                                    ))
```

```
reg_cost_1= 0.3837698590909234
```

3.7　反向传播算法

类似于回归模型中的梯度下降算法，为了求解神经网络最优化问题，我们也要计算$\frac{\partial}{\partial\Theta}J(\Theta)$，以此 $minimize_{\Theta}J(\Theta)$。

在神经网络中，代价函数看上去虽然不复杂，但要注意到其中 $h_\Theta(x)$ 的求取实际上是由前向传播算法求得，即需从输入层开始，根据每层间的权重矩阵 Θ 依次计算激活单元的值 a。在最优化代价函数时，我们必然也需要最优化每一层的权重矩阵，再次强调一下，算法最优化的是权重，而不是输入。

Backpropagation algorithm

Training set $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$

Set $\triangle_{ij}^{(l)} = 0$ (for all l, i, j).

For $i = 1$ to m

Set $a^{(1)} = x^{(i)}$

Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \dots, L$

Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$

$\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \text{ if } j \neq 0$$
$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} \text{ if } j = 0$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

反向传播算法用于计算每一层权重矩阵的偏导 $\frac{\partial}{\partial \Theta} J(\Theta)$，算法实际上是对代价函数求导的拆解。

1. 对于给定训练集 $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})$，初始化每层间的误差和矩阵 Δ，即令所有的 $\Delta_{i,j}^{(l)} = 0$，使得每个 $\Delta^{(l)}$ 为一个全零矩阵。

2. 接下来遍历所有样本实例，对于每一个样本实例，有下列步骤：

2.1. 运行前向传播算法，得到初始预测 $a^{(L)} = h_\Theta(x)$。

2.2. 运行反向传播算法，从输出层开始计算每一层预测的误差 (error)，以此来求取偏导。

输出层的误差即为预测与训练集结果的之间的差值：$\delta^{(L)} = a^{(L)} - y$

对于隐藏层中每一层的误差，都通过上一层的误差来计算：

$\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} .* \frac{\partial a^{(l)}}{\partial z^{(l)}}$，$for \quad l := L-1, L-2, \cdots, 2$。

隐藏层中，$a^{(l)}$ 即为增加偏置单元后的 $g(z^{(l)})$，$a^{(l)}$ 与 $\Theta^{(l)}$ 维度匹配，得以完成矩阵运算。

即对于隐藏层，有 $a^{(l)} = g(z^{(l)})$ 添加偏置单元 $a_0^{(l)} = 1$。

解得 $\frac{\partial}{\partial z^{(l)} g(z^{(l)})} = g'(z^{(l)}) = g(z^{(l)})(1 - g(z^{(l)}))$。

则有 $\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} .* a^{(l)} .* (1 - a^{(l)})$，$a_0^{(l)} = 1$。

根据以上公式计算依次每一层的误差 $\delta^{(L)}, \delta^{(L-1)}, \cdots, \delta^{(2)}$。

2.3. 依次求解并累加误差 $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)}\delta_i^{(l+1)}$，向量化实现即 $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$。

3. 遍历全部样本实例，求解完 Δ 后，最后则求得偏导 $\frac{\partial}{\partial\Theta_{i,j}^{(l)}}J(\Theta) = D_{i,j}^{(l)}$。

$D_{i,j}^{(l)} := \frac{1}{m}\Delta_{i,j}^{(l)} + \frac{\lambda}{m}\Theta_{i,j}^{(l)}, if \quad j \neq 0$;

$D_{i,j}^{(l)} := \frac{1}{m}\Delta_{i,j}^{(l)}\, if \quad j = 0$。（对应于偏置单元）

字母解释如下：

定义 3.7.1. $\delta^{(l)}$: 第 l 层的误差向量

$\delta_i^{(l)}$: 第 l 层的第 i 个激活单元的误差

$\Delta_{i,j}^{(l)}$: 从第 l 层的第 j 个单元映射到第 $l+1$ 层的第 i 个单元的权重代价的偏导（所有样本实例之和）

$D_{i,j}^{(l)}$:$\Delta_{i,j}^{(l)}$ 的样本均值与正则化项之和

注：无需计算 $\delta^{(1)}$，因为输入没有误差。

这就是反向传播算法，即从输出层开始不断向前迭代，根据上一层的误差依次计算当前层的误差，以求得代价函数的偏导。

代码实现如下：

```
# 计算我们之前创建的Sigmoid函数的梯度的函数。
def sigmoid_gradient(z):
    return np.multiply(sigmoid(z),(1-sigmoid(z)))


def back_propagate(param,input_size,hide_size,label_num,X,y,learning_rate):
    m=X.shape[0]
    y=np.matrix(y)
    theta1 = np.matrix(np.reshape(param[:hide_size * (input_size + 1)], (
                                          hide_size, (input_size + 1))))
    theta2 = np.matrix(np.reshape(param[hide_size * (input_size + 1):], (
                                          label_num, (hide_size + 1))))
    a1, z2, a2, z3, h = foward_propagate(X, theta1, theta2)
    J = 0
    delta1=np.zeros(theta1.shape)
    delta2=np.zeros(theta2.shape)
    for i in range(m):
        first=np.multiply(-y[i,:],np.log(h[i,:]))
        second=np.multiply((1-y[i,:]),np.log((1-h[i,:])))
        J=J+np.sum(first-second)
```

```
    J=J/m
    J+=(float(learning_rate)/(2*m))*(np.sum(np.power(theta1[:,1:],2))+np.
                                        sum(np.power(theta2[:,1:],2)))
    for t in range(m):
        a1_t=a1[t,:]
        z2_t=z2[t,:]
        a2_t=a2[t,:]
        z3_t=z3[t,:]
        h_t=h[t,:]
        y_t=y[t,:]
        d3_t=h_t-y_t
        z2_t = sigmoid_gradient(np.insert(z2_t, 0, values=np.ones(1)))
        d2_t=np.multiply((theta2.T*d3_t.T).T,z2_t)
        delta1=delta1+(d2_t[:,1:]).T*a1_t
        delta2=delta2+d3_t.T*a2_t
    delta1=delta1/m
    delta2=delta2/m
    grad=np.concatenate((np.ravel(delta1),np.ravel(delta2)))
    return J,grad
```

反向传播计算的最难的部分（除了理解为什么我们正在做所有这些计算）是获得正确矩阵维度。顺便说一下，你容易混淆了 $A*B$ 与 $np.multiply\ A\ B$ 使用。基本上前者是矩阵乘法，后者是元素乘法（除非 A 或 B 是标量值，在这种情况下没关系）。无论如何，让我们测试一下，以确保函数返回我们期望的。

无论如何，让我们测试一下，以确保函数返回我们期望的。

```
J,grad=back_propagate(param,input_size,hide_size,label_num,X,y_onehot,
                                        learning_rate)
print('J=',J)
print('grad.shape=',grad.shape)
```

结果如下：

```
J= 7.2062719095901775
grad.shape= (10285,)
```

我们还需要对反向传播函数进行一个修改，即将梯度计算加正则化。

```
#梯度函数加正则化
def reg_back_propagate(param,input_size,hide_size,label_num,X,y,
                                        learning_rate):
```

```
    m=X.shape[0]
    X=np.matrix(X)
    y=np.matrix(y)
    theta1 = np.matrix(np.reshape(param[:hide_size * (input_size + 1)], (
                                        hide_size, (input_size + 1))))
    theta2 = np.matrix(np.reshape(param[hide_size * (input_size + 1):], (
                                        label_num, (hide_size + 1))))
    a1, z2, a2, z3, h = foward_propagate(X, theta1, theta2)
    J = 0
    delta1=np.zeros(theta1.shape)
    delta2=np.zeros(theta2.shape)
    for i in range(m):
        first=np.multiply(-y[i,:],np.log(h[i,:]))
        second=np.multiply((1-y[i,:]),np.log((1-h[i,:])))
        J=J+np.sum(first-second)
    J=J/m
    J+=(float(learning_rate)/(2*m))*(np.sum(np.power(theta1[:,1:],2))+np.
                                        sum(np.power(theta2[:,1:],2)))
    for t in range(m):
        a1_t=a1[t,:]
        z2_t=z2[t,:]
        a2_t=a2[t,:]
        z3_t=z3[t,:]
        h_t=h[t,:]
        y_t=y[t,:]
        d3_t=h_t-y_t
        z2_t = sigmoid_gradient(np.insert(z2_t, 0, values=np.ones(1)))
        d2_t=np.multiply((theta2.T*d3_t.T).T,z2_t)
        delta1=delta1+(d2_t[:,1:]).T*a1_t
        delta2=delta2+d3_t.T*a2_t
    delta1=delta1/m
    delta2=delta2/m
    #添加正则项
    delta1[:,1:]=delta1[:,1:]+(theta1[:,1:]*learning_rate)/m
    delta2[:,1:]=delta2[:,1:]+(theta2[:,1:]*learning_rate)/m
    grad=np.concatenate((np.ravel(delta1),np.ravel(delta2)))
    return J,grad
```

```
J_reg, grad_reg = reg_back_propagate(param, input_size, hide_size,
```

```
                                        label_num, X, y_onehot, learning_rate
                                        )
print('J_reg=',J_reg)
print('grad_reg.shape=',grad_reg.shape)
```

结果如下：

```
J_reg= 7.2062719095901775
grad_reg.shape= (10285,)
```

3.8　模型预测

我们终于准备好训练我们的网络，并用它进行预测。这与以往的具有多类逻辑回归的练习大致相似。使用工具库计算参数最优解。

```
#进行预测
fmin=minimize(fun=reg_back_propagate,x0=param,args=(input_size,hide_size,
                                        label_num,X,y_onehot,learning_rate),
                                        method='TNC',jac=True,options={'
                                        maxiter':250})
print('fmin=\n',fmin)
```

```
fmin=
      fun: 0.34199285981108274
      jac: array([ 1.15054579e-03,  1.13161417e-06, -1.72390065e-06, ...,
       -1.34590279e-04, -1.46789669e-04, -9.19384290e-05])
  message: 'Max. number of function evaluations reached'
     nfev: 250
      nit: 20
   status: 3
  success: False
        x: array([ 1.54284856,  0.00565807, -0.0086195 , ...,  1.81640105,
        0.59994396, -2.78914351])
```

由于目标函数不太可能完全收敛，我们对迭代次数进行了限制。我们的总代价已经下降到 0.5 以下，这是算法正常工作的一个很好的指标。让我们使用它发现的参数，并通过网络转发，以获得一些预测。

让我们使用它找到的参数，并通过网络前向传播以获得预测。

```
X = np.matrix(X)
theta1 = np.matrix(np.reshape(fmin.x[:hide_size * (input_size + 1)], (
                                    hide_size, (input_size + 1))))
theta2 = np.matrix(np.reshape(fmin.x[hide_size * (input_size + 1):], (
                                    label_num, (hide_size + 1))))
a1, z2, a2, z3, h = foward_propagate(X, theta1, theta2)
y_predict = np.array(np.argmax(h, axis=1) + 1)
print('y_predict=\n',y_predict)
```

最后，我们可以计算准确度，看看我们训练完毕的神经网络效果怎么样。

```
accuracy = np.mean(y_predict == y)
print('accuracy = {}%'.format(accuracy * 100))
print(classification_report(y, y_predict))
```

```
y_predict=
 [[10]
 [10]
 [10]
 ...
 [ 9]
 [ 9]
 [ 9]]
accuracy = 99.14%
              precision    recall  f1-score   support

           1       0.98      1.00      0.99       500
           2       0.99      0.99      0.99       500
           3       0.99      0.98      0.99       500
           4       0.99      0.99      0.99       500
           5       1.00      0.99      0.99       500
           6       1.00      1.00      1.00       500
           7       0.99      0.99      0.99       500
           8       0.99      1.00      0.99       500
           9       0.99      0.98      0.98       500
          10       0.99      1.00      0.99       500

    accuracy                           0.99      5000
   macro avg       0.99      0.99      0.99      5000
weighted avg       0.99      0.99      0.99      5000
```

第四章　源代码

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.io import loadmat
from scipy.optimize import minimize
from sklearn.metrics import classification_report
from sklearn.preprocessing import OneHotEncoder

def model_1():
    data = loadmat('ex3data1.mat')
    print('-----打印出数据展示-----')
    print(data)

    def loaddata(path):
        data=loadmat(path)
        X = data['X']
        y = data['y']
        return X, y

    X, y = loaddata('ex3data1.mat')
    print(X.shape,y.shape,np.unique(y))

    def plot_random_100_image(X):
        sample_index = np.random.choice(np.arange(X.shape[0]), 100)
        sample_images = X[sample_index, :]
        fig, ax_array = plt.subplots(nrows=10, ncols=10, sharey=True,
                                                sharex=True, figsize=(8, 8))
        for row in range(10):
            for column in range(10):
                ax_array[row, column].matshow(sample_images[10 * row +
```

```
                                                    column].reshape((20,
                                                    20)),cmap='gray_r')
        plt.xticks([])
        plt.yticks([])
        plt.show()

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

    def Cost(theta, X, y, l):
        theta_reg = theta[1:]
        first = (-y * np.log(sigmoid(X @ theta))) + (y - 1) * np.log(1 -
                                                sigmoid(X @ theta))
        reg = (theta_reg @ theta_reg) * l / (2 * len(X))
        return np.mean(first) + reg

    def gradient(theta, X, y, lam):
        theta_reg = theta[1:]
        grand = (X.T @ (sigmoid(X @ theta) - y)) / len(X)
        # 这里人为插入一维0, 使得对theta_0不惩罚, 方便计算
        reg = np.concatenate([np.array([0]), (lam / len(X)) * theta_reg])
        return grand + reg

    def one_vs_all(X, y, lam, num_labels):  # num_labels = 10,标签数
        all_theta = np.zeros((num_labels, X.shape[1]))
        # 大 矩阵包含 1-theta10维度为(10, 401)
        for i in range(1, num_labels + 1):  # 获取每一个数字的y向量
            theta = np.zeros(X.shape[1])  # 求 _i
            y_i = np.array([1 if label == i else 0 for label in y])  # 逐个
                                                    训练theta向量
            final_result = minimize(fun=Cost, x0=theta, args=(X, y_i, lam),
                                                     method='TNC',jac=
                                                    gradient, options={'disp'
                                                    : True})
            all_theta[i - 1, :] = final_result.x  # 将求好的 i赋值给大 矩阵
        return all_theta
    X1, y1 = loaddata('ex3data1.mat')
    X = np.insert(X1, 0, 1, axis=1)  # 在第0列, 插入数字1, axis控制维度为列
                                            (5000, 401)
```

```
    y = y1.flatten()  # 这里消除了一个维度，方便后面的计算 or .reshape(-1)
                                                （5000，）
    all_theta = one_vs_all(X, y, 1, 10)
    print(all_theta) # 每一行是一个分类器的一组参数

    def predict(X, all_theta):
        h = sigmoid(X @ all_theta.T) # 注意的这里的all_theta需要转置
        h_argmax = np.argmax(h, axis=1) # 找到每行的最大索引（0-9）
        h_argmax = h_argmax + 1 # +1后变为每行的分类器 （1-10）
        return h_argmax

    h_argmax=predict(X,all_theta)
    print(h_argmax.shape)
    y_predict = predict(X, all_theta)
    accuracy = np.mean(y_predict == y)
    print('accuracy = {0}%'.format(accuracy * 100))
    print(classification_report(y, y_predict))
    return None

def model_2():
    data = loadmat('ex3data1.mat')
    x = data['X']
    y = data['y']
    x = np.insert(x, 0, values=1, axis=1)
    print('x.shape=',x.shape)
    y = y.flatten()
    print('y.shape=',y.shape)
    theta = loadmat('ex3weights.mat')
    theta1 = theta['Theta1']
    print('theta1.shape=',theta1.shape)
    theta2 = theta['Theta2']
    print('theta2.shape=',theta2.shape)

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

    # 输入层
    a1=x
    print('a1.shape=',a1.shape)
```

```
    # 隐藏层
    z2 = x @ theta1.T
    print('z2.shape=',z2.shape)
    a2 = sigmoid(z2)
    print('a2.shape=',a2.shape)
    # 输出层
    a2 = np.insert(a2, 0, values=1, axis=1)
    print('a2.shape=',a2.shape)
    z3 = a2 @ theta2.T
    print('z3.shape=',z3.shape)
    a3=sigmoid(z3)
    print('a3.shape=',a3.shape)
    y_pre = np.argmax(a3, axis=1)
    y_pre = y_pre + 1
    acc = np.mean(y_pre == y)
    print('accuracy = {}%'.format(acc * 100))
    report=classification_report(y_pre,y)
    print(report)

def model_3():
    data = loadmat('ex4data1.mat')
    X = data['X']
    y = data['y']
    print('X.shape=',X.shape)
    print('y.shape=',y.shape)
    encoder=OneHotEncoder(sparse=False)
    y_onehot=encoder.fit_transform(y)
    print('y_onehot=\n',y_onehot)
    print('y_onehot.shape=',y_onehot.shape)

    #sigmoid函数
    def sigmoid(z):
        return 1/(1+np.exp(-z))

    #前向传播函数
    def foward_propagate(X,theta1,theta2):
       m=X.shape[0] #m=5000
       a1=np.insert(X,0,values=np.ones(m),axis=1)
       z2=a1*theta1.T
```

```
    a2=np.insert(sigmoid(z2),0,values=np.ones(m),axis=1)
    z3=a2*theta2.T
    h=sigmoid(z3)
    return a1,z2,a2,z3,h

#代价函数
def cost(param,input_size,hide_size,label_num,X,y,learning_rate):
    m=X.shape[0]
    X=np.matrix(X)
    y=np.matrix(y)
    # 将参数数组解开为每个层的参数矩阵，reshape重新定义维度
    theta1=np.matrix(np.reshape(param[:hide_size * (input_size + 1)], (
                                        hide_size, (input_size + 1)))
                                        )
    theta2=np.matrix(np.reshape(param[hide_size*(input_size+1):],(
                                        label_num,(hide_size+1))))
    a1,z2,a2,z3,h=foward_propagate(X,theta1,theta2)
    J=0
    for i in range(m):
        first=np.multiply(-y[i,:],np.log(h[i,:]))
        second=np.multiply((1-y[i,:]),np.log((1-h[i,:])))
        J=J+np.sum(first-second)
    J=J/m
    return J

#初始化设置
input_size=400
hide_size=25
label_num=10
learning_rate=1
param=(np.random.random(size=hide_size*(input_size+1)+label_num*(
                                    hide_size+1))-0.5)*0.25
# np.random.random生成（-0.5~0.5）*0.25的浮点型随机数组
m=X.shape[0]
X=np.matrix(X)
y=np.matrix(y)
theta1 = np.matrix(np.reshape(param[:hide_size * (input_size + 1)], (
                                    hide_size, (input_size + 1))))
theta2 = np.matrix(np.reshape(param[hide_size * (input_size + 1):], (
```

```
                                        label_num, (hide_size + 1))))
print('theta1.shape=',theta1.shape)
print('theta2.shape=',theta2.shape)
a1,z2,a2,z3,h=foward_propagate(X,theta1,theta2)
print('a1.shape=',a1.shape)
print('z2.shape=',z2.shape)
print('a2.shape=',a2.shape)
print('z3.shape=',z3.shape)
print('h.shape=',h.shape)
print('cost=',cost(param,input_size,hide_size,label_num,X,y_onehot,
                                        learning_rate))

#正则化代价函数
def reg_cost(param,input_size,hide_size,label_num,X,y,learning_rate):
    m=X.shape[0]
    X=np.matrix(X)
    y=np.matrix(y)
    theta1=np.matrix(np.reshape(param[:hide_size * (input_size + 1)], (
                                            hide_size, (input_size + 1)))
                                            )
    theta2 = np.matrix(np.reshape(param[hide_size * (input_size + 1):],
                                             (label_num, (hide_size + 1))
                                            ))
    a1, z2, a2, z3, h = foward_propagate(X, theta1, theta2)
    J = 0
    for i in range(m):
        first = np.multiply(-y[i, :], np.log(h[i, :]))
        second = np.multiply((1 - y[i, :]), np.log(1 - h[i, :]))
        J =J+ np.sum(first - second)
    J = J / m
    J += (float(learning_rate) / (2 * m)) * (np.sum(np.power(theta1[:,
                                            1:], 2)) + np.sum(np.power(
                                            theta2[:, 1:], 2)))
    return J
print('reg_cost=',reg_cost(param, input_size, hide_size, label_num, X,
                                        y_onehot, learning_rate))

# 计算我们之前创建的Sigmoid函数的梯度的函数。
def sigmoid_gradient(z):
```

```
        return np.multiply(sigmoid(z),(1-sigmoid(z)))

    def back_propagate(param,input_size,hide_size,label_num,X,y,
                                            learning_rate):
        m=X.shape[0]
        X=np.matrix(X)
        y=np.matrix(y)
        theta1 = np.matrix(np.reshape(param[:hide_size * (input_size + 1)],
                                              (hide_size, (input_size + 1)
                                             )))
        theta2 = np.matrix(np.reshape(param[hide_size * (input_size + 1):],
                                              (label_num, (hide_size + 1))
                                             ))
        a1, z2, a2, z3, h = foward_propagate(X, theta1, theta2)
        J = 0
        delta1=np.zeros(theta1.shape)
        delta2=np.zeros(theta2.shape)
        for i in range(m):
            first=np.multiply(-y[i,:],np.log(h[i,:]))
            second=np.multiply((1-y[i,:]),np.log((1-h[i,:])))
            J=J+np.sum(first-second)
        J=J/m
        J+=(float(learning_rate)/(2*m))*(np.sum(np.power(theta1[:,1:],2))+
                                              np.sum(np.power(theta2[:,1:],
                                              2)))
        for t in range(m):
            a1_t=a1[t,:]
            z2_t=z2[t,:]
            a2_t=a2[t,:]
            z3_t=z3[t,:]
            h_t=h[t,:]
            y_t=y[t,:]
            d3_t=h_t-y_t
            z2_t = sigmoid_gradient(np.insert(z2_t, 0, values=np.ones(1)))
            d2_t=np.multiply((theta2.T*d3_t.T).T,z2_t)
            delta1=delta1+(d2_t[:,1:]).T*a1_t
            delta2=delta2+d3_t.T*a2_t
        delta1=delta1/m
        delta2=delta2/m
```

```
        grad=np.concatenate((np.ravel(delta1),np.ravel(delta2)))
        return J,grad
    J,grad=back_propagate(param,input_size,hide_size,label_num,X,y_onehot,
                                            learning_rate)
    print('J=',J)
    print('grad.shape=',grad.shape)

    #梯度函数加正则化
    def reg_back_propagate(param,input_size,hide_size,label_num,X,y,
                                            learning_rate):
        m=X.shape[0]
        X=np.matrix(X)
        y=np.matrix(y)
        theta1 = np.matrix(np.reshape(param[:hide_size * (input_size + 1)],
                                        (hide_size, (input_size + 1)
                                        )))
        theta2 = np.matrix(np.reshape(param[hide_size * (input_size + 1):],
                                        (label_num, (hide_size + 1))
                                        ))
        a1, z2, a2, z3, h = foward_propagate(X, theta1, theta2)
        J = 0
        delta1=np.zeros(theta1.shape)
        delta2=np.zeros(theta2.shape)
        for i in range(m):
            first=np.multiply(-y[i,:],np.log(h[i,:]))
            second=np.multiply((1-y[i,:]),np.log((1-h[i,:])))
            J=J+np.sum(first-second)
        J=J/m
        J+=(float(learning_rate)/(2*m))*(np.sum(np.power(theta1[:,1:],2))+
                                        np.sum(np.power(theta2[:,1:],
                                        2)))
        for t in range(m):
            a1_t=a1[t,:]
            z2_t=z2[t,:]
            a2_t=a2[t,:]
            z3_t=z3[t,:]
            h_t=h[t,:]
            y_t=y[t,:]
            d3_t=h_t-y_t
```

```
            z2_t = sigmoid_gradient(np.insert(z2_t, 0, values=np.ones(1)))
            d2_t=np.multiply((theta2.T*d3_t.T).T,z2_t)
            delta1=delta1+(d2_t[:,1:]).T*a1_t
            delta2=delta2+d3_t.T*a2_t
        delta1=delta1/m
        delta2=delta2/m
        #添加正则项
        delta1[:,1:]=delta1[:,1:]+(theta1[:,1:]*learning_rate)/m
        delta2[:,1:]=delta2[:,1:]+(theta2[:,1:]*learning_rate)/m
        grad=np.concatenate((np.ravel(delta1),np.ravel(delta2)))
        return J,grad
    J_reg, grad_reg = reg_back_propagate(param, input_size, hide_size,
                                         label_num, X, y_onehot,
                                         learning_rate)
    print('J_reg=',J_reg)
    print('grad_reg.shape=',grad_reg.shape)

    #进行预测
    fmin=minimize(fun=reg_back_propagate,x0=param,args=(input_size,
                                         hide_size,label_num,X,y_onehot,
                                         learning_rate),method='TNC',jac=
                                         True,options={'maxiter':250})
    print('fmin=\n',fmin)

    X = np.matrix(X)
    theta1 = np.matrix(np.reshape(fmin.x[:hide_size * (input_size + 1)], (
                                         hide_size, (input_size + 1))))
    theta2 = np.matrix(np.reshape(fmin.x[hide_size * (input_size + 1):], (
                                         label_num, (hide_size + 1))))
    a1, z2, a2, z3, h = foward_propagate(X, theta1, theta2)
    y_predict = np.array(np.argmax(h, axis=1) + 1)
    print('y_predict=\n',y_predict)

    accuracy = np.mean(y_predict == y)
    print('accuracy = {}%'.format(accuracy * 100))
    print(classification_report(y, y_predict))

if __name__=='__main__':
    model_3()
```