模式识别与统计学习
实验六：*BackPropogation*基础神经网络实验报告

郭宇航 2016100104014

一　实验原理

1.1　神经网络基本框架

在引入神经网络之前，按照目前学习的进度首先先回顾一下线性回归，给出线性回归的一般表达式($\boldsymbol{Z} = \boldsymbol{W}\boldsymbol{X} + b$)：

$$z = w_1x_1 + w_2x_2 + \cdots + w_kx_k + \cdots + w_Nx_N + b \tag{1}$$

通过图形可视化我们可以表示为图1所示： 如果在输出Z前对加权输出进行基函数变换，基函数假

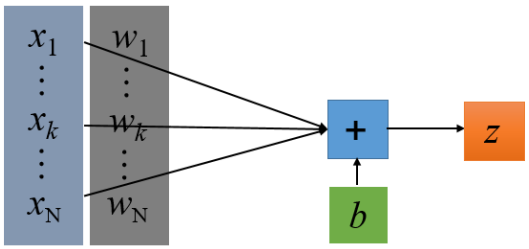

图 1: 基础线性回归

设选取为sigmoid函数即可以得到logistic regression的表达式。用图形可以很容易表示为图2： 上

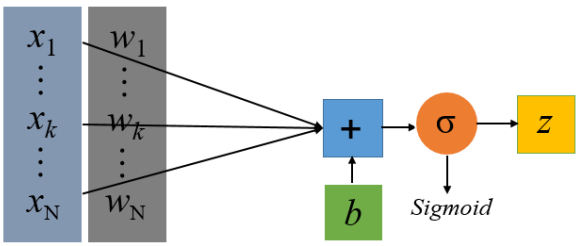

图 2: Sigmoid函数变换后的线性回归

述的模型我们可以称之为logistic regression，但同时经过简单的变形后我们也可以称之为感知机模型(Perceptron)，换一种数学公式来表达：

$$Y = \sigma(\sum_{i=1}^{N} w_ix_i + b - \theta) \tag{2}$$

其中θ表示激活阈值，$\sigma()$表示单位阶跃的函数，当内部表达式大于0则激活，小于0则不激活，由此我们发现这个模型就变为了0,1输出模型，这就是感知机模型。
由基本的感知机模型我们可以进行多层的实现得到最常见的神经网络模型：全连接的神经网络。

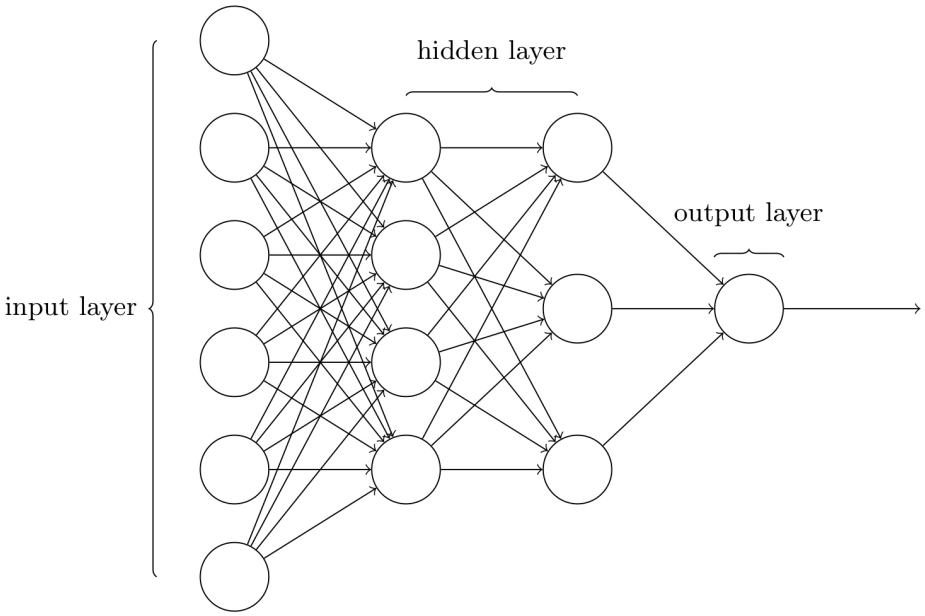

图 3: 全连接神经网络

1.2 前向传播原理

上述的图3给出了一般化的神经网络的模型，下面给出神经网络模型的前向传播过程的数学表示，首先为了方便表示，我们先给出如下的一些定义：

- w_{jk}^l表示从$(l-1)$层的第k个神经元到l层的第j个神经元的连接权重。
- b_j^l表示第l层的第j个神经元的偏置。
- z_j^l表示第l层的第j个神经元的带权输入。
- a_j^l表示第l层的第j个神经元的激活值。
- $\sigma()$表示激活函数：$a^l = \sigma(z^l)$.

考虑前向传播过程十分简单，对于一个输入数据样本来说，经过第一层网络加权求和并激活之后得到了新的维度的特征数据样本，同样的可以通过第二层网络做这样的操作，多次重复后最后得到输出的结果。通过数学符号可以表达更加直观，假设第$l-1$层共有m个神经元，第l层共有n个神经元，不难得到如下：

$$a_j^l = \sigma(z_j^l) = \sigma(\sum_{k=1}^{m} w_{jk}^l a_k^{l-1} + b_j^l) \tag{3}$$

如果使用矩阵来表示，第l层的线性系数w组成了一个$n \times m$的矩阵$\boldsymbol{W}^l$，第l层的偏置组成了一个$n \times 1$的向量b^l，第$l-1$层的输出a组成了一个$m \times 1$的向量a^{l-1}，第l层的带权输入z组成了一个$n \times 1$的向量z^l，第l层的输出a组成了一个$n \times 1$的向量a^l，则用矩阵法表示，第l层的输出为:

$$a^l = \sigma(z^l) = \sigma(W^l a^{l-1} + b^l) \tag{4}$$

1.3 反向传播原理

反向传播原理是神经网络最核心的思想，同时也是解决模型训练问题的关键所在，我们知道在参数数量如此庞大的神经网络中如何找出一组最好的权重来适配已知的数据。同样传统算法一样，模型的优化过程从损失函数出发，然后对这个损失函数求极值，大部分情况下损失函数都是找不到极值的解析解，因此需要采用搜索算法在能量面上寻找局部最优解。最常用的就是梯度下降，然后和一般模型不同的是神经网络由于有多层模型，从损失函数直接对每一层的权重求梯度无法直接求解，因此需要采用多元函数求偏导的链式法则进行运算。下面给出详细介绍：

如果采用均方误差作为损失函数，假设神经网络最后一层为第L层，则对于神经网络最终的损失函数可以表示为：

$$C(W,b,x,y)=\frac{1}{2}||a^L-y||_2^2 \tag{5}$$

其中W,b表示权重和偏置量，x表示输入数据，y表示样本的实际类别。对于最后一层输出层来说，考虑W,b满足：

$$a^L=\sigma(z^L)=\sigma(W^La^{L-1}+b^L) \tag{6}$$

因此重写损失函数：

$$C(W,b,x,y)=\frac{1}{2}||\sigma(W^La^{L-1}+b^L)-y||_2^2 \tag{7}$$

利用链式法则可以得到损失函数对输出层的权重和偏置的导数：

$$\begin{aligned}\frac{\partial C(W,b,x,y)}{\partial W^L}&=\frac{\partial C(W,b,x,y)}{\partial Z^L}\frac{\partial Z^L}{\partial W^L}=(a^L-y)\odot\sigma'(z^L)(a^{L-1})^T\\ \frac{\partial C(W,b,x,y)}{\partial b^L}&=\frac{\partial C(W,b,x,y)}{\partial Z^L}\frac{\partial Z^L}{\partial b^L}=(a^L-y)\odot\sigma'(z^L)\end{aligned} \tag{8}$$

其中$\odot$表示两个维度一致的向量或者矩阵做对应位置相乘的操作。下面计算输出层之前的其它层的梯度。对于第l层的带权输入z^l，我们将损失函数对其的偏导数定义为误差δ^l，则此误差可以表示为：

$$\delta^l=\frac{\partial C}{\partial z^L}\frac{\partial z^L}{\partial z^{L-1}}\frac{\partial z^{L-1}}{\partial z^{L-2}}\cdots\frac{\partial z^{l+1}}{\partial z^l} \tag{9}$$

结合前向传播的公式：

$$z^l=W^la^{l-1}+b^l \tag{10}$$

我们不难计算出对任意的第l层的W^l,b^l求偏导数的结果为：

$$\begin{aligned}\frac{\partial C}{\partial W^l}&=\frac{\partial C}{\partial z^l}\frac{\partial z^l}{\partial W^l}=\delta^l(a^{l-1})^T\\ \frac{\partial C}{\partial b^l}&=\frac{\partial C}{\partial z^l}\frac{\partial z^l}{\partial b^l}=\delta^l\end{aligned} \tag{11}$$

下面问题集中转化为求解δ^l，由于最后一层输出层的误差δ^L已经求得结果，因此可以假设第$l+1$层的误差δ^{l+1}已知，求解δ^l.

$$\delta^l=\frac{\partial C}{\partial z^l}=\frac{\partial C}{\partial z^{l+1}}\frac{\partial z^{l+1}}{\partial z^l}=\delta^{l+1}\frac{\partial z^{l+1}}{\partial z^l} \tag{12}$$

下面寻找z^{l+1}和z^l之间的关系：

$$z^{l+1}=W^{l+1}a^l+b^{l+1}=W^{l+1}\sigma(z^l)+b^{l+1} \tag{13}$$

这样很容易求得：

$$\delta^l=\delta^{l+1}\frac{\partial z^{l+1}}{\partial z^l}=(W^{l+1})^T\delta^{l+1}\odot\sigma'(z^l) \tag{14}$$

上式即为δ^l的递推关系式，递推求解误差后即可很容易求解出该层对应的损失函数对权重和偏置的偏导数。最终神经网络的反向传播方程可以集中表示为这样四个方程：

$$\begin{cases} \delta^L = \nabla_{a^L} C \odot \sigma'(z^L) \\ \delta^l = (W^{l+1})^T \delta^{l+1} \odot \sigma'(z^l) \\ \nabla_{b^l} C = \delta^l \\ \nabla_{W^l} C = a^{l-1} \delta^l \end{cases} \tag{15}$$

二 Python代码实现

本次实验的数据仍然是MNIST数据集上手写数字数据集，主要搭建多层的DNN基础网络实现手写体的识别，使用python编写程序，程序共分为五个部分：第一部分定义一层神经网络的类，第二部分神经网络的前向传播，第三部分神经网络的反向传播，第四部分定义了多个激活函数用于调用，第五部分主函数部分，下面一一给出展示。网络模型构造大致如下图4，共计四层网络，输入层784个神经元，两个隐藏层分别为100和50个神经元，最后一层输出层10个神经元分别对应分别取标签0-9的可能性，选择最大结果作为该样本的类标签。

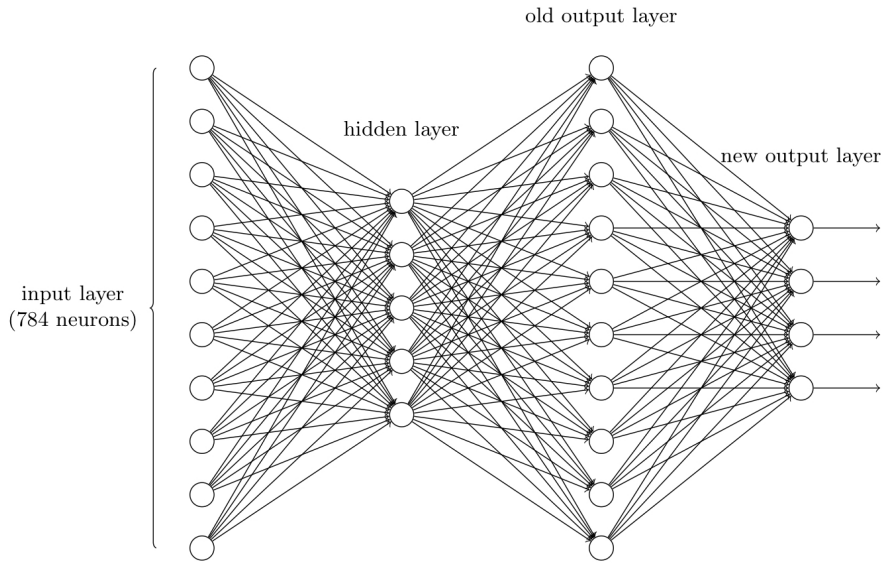

图 4: MINST数据集上的DNN模型

2.1 定义神经网络类

这一部分的实现主要是为了神经网络的封装和增强网络的泛化性。代码实现如下：

```
class input_layer:
    def __init__(self,data_size,batch_size):
        self.size =[data_size,batch_size]
        self.input =None
        self.a_value =None
        self.layer_kind ='input'

class full_connected_layer:
    def __init__(self,input,layer_size,activation):
```

```
        self.size =[layer_size,input.size[0]]
        self.weights =np.random.randn(self.size[0],self.size[1])/10
        self.bias =np.random.randn(layer_size,1)/10
        self.input =None
        self.activation =activation
        self.z_value =None
        self.a_value =None
        self.error =None
        self.delta_weights =None
        self.layer_kind ='full'
```

2.2 激活函数

这一部分给出激活函数的定义，其中主要包括'Sigmoid'和'RELU'两种激活方式，函数中分别定了其前向传播激活的函数与反向传播时导数的表达式，主要用于前向传播函数和后向传播使用。

```
def Sigmoid_forward(weighted_input):
        return 1.0 /(1.0 +np.exp(-weighted_input))

def Sigmoid_backward(output_value):
        return np.multiply(output_value,(1-output_value))

def RELU_forward(weighted_input):
        return weighted_input *(weighted_input >=0)

def RELU_backward(output_value):
        output_value[output_value>0] =1
        output_value[output_value<=0] =0
        return output_value

def softmax_forward(H_x):
        for i in H_x:
            i = np.exp(i) /np.sum(np.exp(H_x))
        return H_x
```

2.3 前向传播

前向传播函数传入参数是一个神经网络model，model中包含了所有的层以及各个参数，前向传播通过一个循环进行，代码实现如下：

```
def full_connected(model,data):
    model[0].input =data
    model[0].a_value =data
    for i in range(1,len(model)):
        model[i].input =model[i-1].a_value
        model[i].z_value =np.dot(model[i].weights,model[i].input) +model[i].bias
        if model[i].activation =='Sigmoid':
            model[i].a_value =activation_function.Sigmoid_forward(model[i].z_value)
        elif model[i].activation =='RELU':
            model[i].a_value =activation_function.RELU_forward(model[i].z_value)
```

2.4 反向传播

反向传播如前一节介绍的公式所示，其中最后一层的误差求解与损失函数的形式有关，单独

拿出来计算，前面的所有网络层通过递推的方式求解，通过python的反向索引从倒数第二层往前传播，传播公式这里不再给出，即上一节总结的反向传播的四大公式，实现代码如下：

```
def full_connected_back(last_layer,next_layer):
    if last_layer.layer_kind =='full':
        if last_layer.activation =='Sigmoid':
            last_layer.error =np.multiply(np.dot(next_layer.weights.T,next_layer.error),
                            activation_function.Sigmoid_backward(last_layer.a_value))
        elif last_layer.activation =='RELU':
            last_layer.error =np.multiply(np.dot(next_layer.weights.T,next_layer.error),
                            activation_function.RELU_backward(last_layer.z_value))
    last_layer.delta_weights =np.dot(last_layer.error,last_layer.input.T)

def last_layer_error(last_layer,real_data,loss_function_type):
    if loss_function_type =='RSE':
        if last_layer.activation =='Sigmoid':
            last_layer.error =(last_layer.a_value -real_data) *
                            activation_function.Sigmoid_backward(last_layer.a_value)
        #error = np.sum((last_layer.a_value - real_data)**2)
```

2.5 DNN主函数部分

主函数部分主要从tensorflow中加载数据，生成小批量数据函数用于MBGD算法训练模型，训练模型函数以及测试集效果测试，代码主要如下：

```
import numpy as np
import random
import time
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
import NN_class
import activation_function
import NN_forward
import NN_propagation

def load_data(filename):
    data_set =input_data.read_data_sets(filename,one_hot=True)
    train_data,train_label,test_data,test_label =
        data_set.train.images,data_set.train.labels,data_set.test.images,data_set.test.labels
    return train_data,train_label,test_data,test_label

def mini_batch_generate(data,size):
    mini_batches =[]
    total_number =len(data)
    for i in range(0,total_number,size):
        mini_batches.append(data[i:i+size])
    return mini_batches

def train_process(model,data,label,alpha):
    NN_forward.full_connected(model,data)
    cost =np.sum((model[-1].a_value -label)**2)
    #print('loss_function:',cost)
    NN_propagation.last_layer_error(model[-1],label,'RSE')
    model[-1].weights -=alpha *model[-1].delta_weights /model[-1].input.shape[1]
    #print(np.average(model[-1].error,axis=1).shape,model[-1].bias.shape)
    model[-1].bias -=alpha *np.average(model[-1].error,axis=1).reshape(len(model[-1].error),1)
    for i in range(2,len(model)):
```

```
        NN_propagation.full_connected_back(model[-i],model[-i+1])
    for j in range(1,len(model)):
        model[j].weights -=alpha *model[j].delta_weights /model[j].input.shape[1]
        model[j].bias -=alpha *np.average(model[j].error,axis=1).reshape(len(model[j].error),1)

def test_process(model,test_data,test_label):
    total_num =10000
    true_number =0
    NN_forward.full_connected(model,test_data)
    result =np.argmax(model[-1].a_value,axis=0).reshape(total_num,1)
    true_label =np.array([np.argmax(y) for y in test_label.T]).reshape(total_num,1)
    for i in range(total_num):
        if result[i] ==true_label[i]:
            true_number +=1
    accuracy =true_number /total_num
    return accuracy

if __name__ =='__main__':
    train_data,train_label,test_data,test_label =load_data('MINST')
    training_set =list(zip(train_data,train_label))
    total_number =len(training_set)
    input_number =train_data.shape[1]
    batch_size =100
    np.random.shuffle(training_set)
    train_data_shuffle =np.array(list(zip(*training_set))[0])
    train_label_shuffle =np.array(list(zip(*training_set))[1])
    mini_batches_data =mini_batch_generate(train_data_shuffle,batch_size)
    mini_batches_label =mini_batch_generate(train_label_shuffle,batch_size)
    layer_0 =NN_class.input_layer(data_size=784,batch_size=batch_size)
    layer_1 =NN_class.full_connected_layer(input=layer_0,layer_size=100,activation='Sigmoid')
    layer_2 =NN_class.full_connected_layer(input=layer_1,layer_size=50,activation='Sigmoid')
    layer_3 =NN_class.full_connected_layer(input=layer_2,layer_size=10,activation='Sigmoid')
    model =[layer_0,layer_1,layer_2,layer_3]
    error =[]
    accuracy_list =[]
    for iteration in range(100):
        for i in range(len(mini_batches_data)):
            data =mini_batches_data[i].T
            label =mini_batches_label[i].T
            error_1 =train_process(model,data,label,0.5)
            error.append(error_1)
        print(test_process(model,test_data.T,test_label.T))
```

三　结果及图形展示

第一次测试使用的是最简单的三层网络，输入层，一个隐藏层，输出层，使用RSE均方误差以及Sigmoid激活函数：网络结构如下图5： 训练迭代100次的结果如下图6：

```
    layer_0 = NN_class.input_layer(data_size=784,batch_size=batch_size)
    layer_1 = NN_class.full_connected_layer(input=layer_0,layer_size=15,activation='Sigmoid')
    layer_2 = NN_class.full_connected_layer(input=layer_1,layer_size=10,activation='Sigmoid')
    model = [layer_0,layer_1,layer_2]
```

图 5: 三层网络模型架构

第二次测试使用四层的网络结构(图7)： 同样训练迭代100次的结果如下(图8)：

```
iteration:0 epoch,accuracy:0.740900
iteration:1 epoch,accuracy:0.880000
iteration:2 epoch,accuracy:0.898300
iteration:3 epoch,accuracy:0.906900
iteration:4 epoch,accuracy:0.911700
iteration:5 epoch,accuracy:0.916100
iteration:6 epoch,accuracy:0.919800
iteration:7 epoch,accuracy:0.922800
iteration:8 epoch,accuracy:0.924200
iteration:9 epoch,accuracy:0.926200
iteration:10 epoch,accuracy:0.927900
iteration:11 epoch,accuracy:0.929300
iteration:12 epoch,accuracy:0.929800
iteration:13 epoch,accuracy:0.930500
iteration:14 epoch,accuracy:0.931200
iteration:15 epoch,accuracy:0.932300
iteration:16 epoch,accuracy:0.932700
iteration:17 epoch,accuracy:0.933300
iteration:18 epoch,accuracy:0.933000
```

```
iteration:81 epoch,accuracy:0.942900
iteration:82 epoch,accuracy:0.943100
iteration:83 epoch,accuracy:0.943300
iteration:84 epoch,accuracy:0.943400
iteration:85 epoch,accuracy:0.943300
iteration:86 epoch,accuracy:0.943400
iteration:87 epoch,accuracy:0.943500
iteration:88 epoch,accuracy:0.943400
iteration:89 epoch,accuracy:0.943400
iteration:90 epoch,accuracy:0.943400
iteration:91 epoch,accuracy:0.943400
iteration:92 epoch,accuracy:0.943400
iteration:93 epoch,accuracy:0.943500
iteration:94 epoch,accuracy:0.943600
iteration:95 epoch,accuracy:0.943900
iteration:96 epoch,accuracy:0.943800
iteration:97 epoch,accuracy:0.943900
iteration:98 epoch,accuracy:0.944000
iteration:99 epoch,accuracy:0.943900
```

图 6: 三层网络模型训练结果

```
layer_0 = NN_class.input_layer(data_size=784,batch_size=batch_size)
layer_1 = NN_class.full_connected_layer(input=layer_0,layer_size=100,activation='Sigmoid')
layer_2 = NN_class.full_connected_layer(input=layer_1,layer_size=50,activation='Sigmoid')
layer_3 = NN_class.full_connected_layer(input=layer_2,layer_size=10,activation='Sigmoid')
model = [layer_0,layer_1,layer_2,layer_3]
```

图 7: 四层网络模型架构

```
iteration:0 epoch,accuracy:0.449300
iteration:1 epoch,accuracy:0.793100
iteration:2 epoch,accuracy:0.878900
iteration:3 epoch,accuracy:0.897500
iteration:4 epoch,accuracy:0.906900
iteration:5 epoch,accuracy:0.913600
iteration:6 epoch,accuracy:0.919300
iteration:7 epoch,accuracy:0.924400
iteration:8 epoch,accuracy:0.927200
iteration:9 epoch,accuracy:0.930600
iteration:10 epoch,accuracy:0.933200
iteration:11 epoch,accuracy:0.935100
iteration:12 epoch,accuracy:0.936800
iteration:13 epoch,accuracy:0.938600
iteration:14 epoch,accuracy:0.941100
iteration:15 epoch,accuracy:0.942800
iteration:16 epoch,accuracy:0.944700
iteration:17 epoch,accuracy:0.946300
iteration:18 epoch,accuracy:0.947500
iteration:19 epoch,accuracy:0.948400
iteration:20 epoch,accuracy:0.949300
```

```
iteration:79 epoch,accuracy:0.973900
iteration:80 epoch,accuracy:0.974100
iteration:81 epoch,accuracy:0.974000
iteration:82 epoch,accuracy:0.974100
iteration:83 epoch,accuracy:0.974200
iteration:84 epoch,accuracy:0.974200
iteration:85 epoch,accuracy:0.974400
iteration:86 epoch,accuracy:0.974600
iteration:87 epoch,accuracy:0.974500
iteration:88 epoch,accuracy:0.974500
iteration:89 epoch,accuracy:0.974600
iteration:90 epoch,accuracy:0.974600
iteration:91 epoch,accuracy:0.974700
iteration:92 epoch,accuracy:0.974800
iteration:93 epoch,accuracy:0.974900
iteration:94 epoch,accuracy:0.974800
iteration:95 epoch,accuracy:0.974700
iteration:96 epoch,accuracy:0.974700
iteration:97 epoch,accuracy:0.974700
iteration:98 epoch,accuracy:0.974700
iteration:99 epoch,accuracy:0.974900
```

图 8: 四层网络模型训练结果

最后附上两个不同的全连接DNN模型的准确率结果图(图9)：

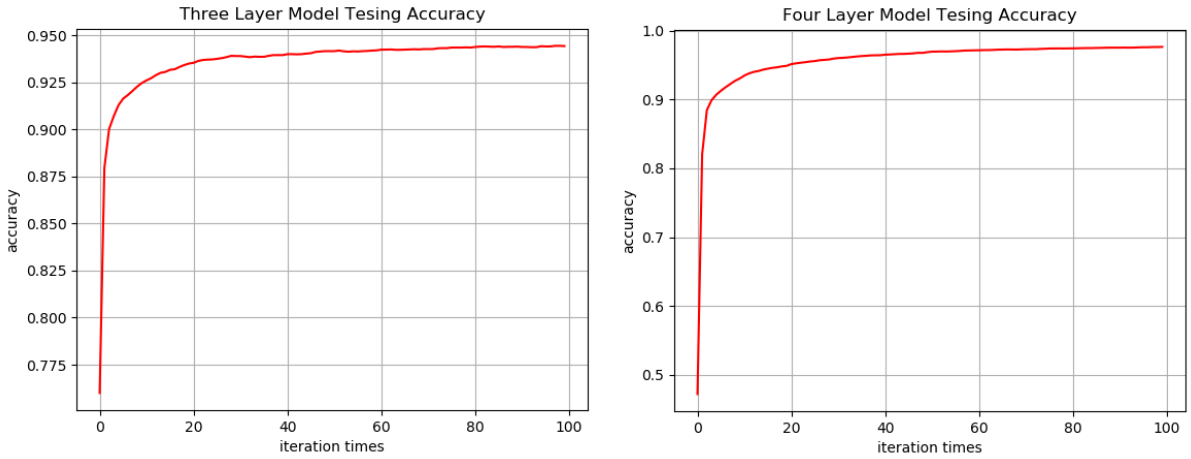

图 9: 两个模型的准确率对比图

四　总结与体会

本次实验本质上来说其实原理的理解并不是很难，但是在代码段的实现上来说自己第一次写这类进行封装好的神经网络还是很费时间的，DNN的python实现写了两次，第一次写虽然成功实现但是发现代码的可读性不强，没有将每一部分的代码都分得很清楚，而且运行的速度非常慢，迭代100次大概需要3,4个小时，运行效率较低，通过可移植性很差，尤其是想要将全连接的代码运用的之后的CNN实现中，因此第二次下定决心重新搭建网络框架，将每一个部分写成可读性更强的模块，同时优化了一些细节，是的代码的运行效率大幅提升，同时可以随便更改网络的层数等参数，代码的鲁棒性强了很多。从原理到代码的实现是一个很辛苦的过程，但是实现之后的成就感会很强。