人工神经网络：BP神经网络的实现

# 1实验目的

1. 了解BP神经网络的结构和工作方式；
2. 了解输入信息与误差信息在BP神经网络上的传播方式；
3. 学会设计BP神经网络。

## 2实验要求

本次试验后，要求学生能：

1. 理解BP神经网络输入值前向传播，误差值后向传播的特点；
2. 理解BP神经网络各层权值的更新方式；
3. 用python编写BP神经网络框架，解决简单识别问题。

# 3实验原理

## 3.1实验介绍

本实验将运用python编写一个可以自定义网络结构的BP神经网络框架，用此框架解决简单问题。

（1）数字识别

在一个7\*9的格子上用0、1描绘出0~9十个数字，用来训练BP神经网络，最后进行测试。

例如：0 ，0 ，0， 0， 0， 0 ，0，

0 ，1 ，0， 0， 0， 0 ，0，

0 ，1 ，0， 1， 0， 0 ，0，

0 ，1 ，0， 1， 0， 0 ，0，

0 ，1 ，0， 1， 0， 0 ，0，

0 ，1 ，1， 1， 1， 1 ，0，

0 ，0 ，0， 1， 0， 0 ，0，

0 ，0 ，0， 1， 0， 0 ，0，

0 ，0 ，0， 0， 0， 0 ，0，

代表数字“4”。

（2）函数逼近

对函数，获取两组数据一组作为训练集，一组作为测试集，训练一个单隐层网络，改变隐层神经元个数，观察对逼近效果的影响。

## 3.2 BP神经网络

### 3.2.1网络结构

BP神经网络是多层前向网络，具有m层，第一层为输入层，第m层为输出层，其他层为隐含层，每层有若干个节点（根据具体问题来设定）和一个偏置节点。

各层节点关系如下：

公式1

其中ak为第k层的输入，zk为第k层的输出，wk,k+1其第k到k+1层的权值矩阵，bk为k层的转置矩阵。

可以看出，输入信息是沿着神经网络，前向传播到输出层。

### 3.2.2训练学习

（1）更新weight矩阵中的权值

神经网络的学习过程就是不断调解神经网络的权值，使它的输入与输出满足需要的对应关系。

BP神经网络时通过反向学习来调整权值，使误差变小。BP学习算法可归纳为：

公式2

（2）更新阈值

在BP神经网络中，可以在每层单独设置一个没有输入、输出恒为1的偏置节点作为阈值，然后将weight矩阵扩展成weight’，只需像上一步中那样更新weight’即可，等价于下一层神经元的阈值是本层偏置节点输出的权值。

下面给出新的更新公式，这样可以不考虑阈值与权值在网络结构上的不同，用一个公式统一进行更新。

其中代表层输入向量，代表k层输出向量，代表k层的阈值向量，代表k到k+1层的权值矩阵。

## 3.3 BP定理

BP定理：给定任意ε>0，对于任意的连续函数f，存在一个三层前向神经网络，可以在任意ε评分误差精度内逼近f。

BP定理说明只要用三层BP网络就可以逼近一个任意连续函数，但可能需要大量的隐层神经元个数。为了减少神经元个数，也可以采用多层BP神经网络结构。

对于如何选择合适的BP神经网络隐层数和隐层神经元数，目前尚无有效的理论和方法。

## 3.4 BP神经网络特点分析

（1）优点

BP神经网络具有非线性映射能力、泛化能力、容错能力，允许输入样本中带有较大误差甚至个别错误。反应正确规律的知识来自全体样本，个别样本中的误差不能左右对权矩阵的调整。

（2）缺陷

需要用梯度下降法来更新权值矩阵，这就要求目标函数和损失函数必须可微，而且如果一片区域比较平坦会花费较多时间进行训练

构建网络需要的参数过多，而且参数的选择没有有效的方法。确定一个BP神经网络需要知道：网络的层数、每一层神经元的个数和权值。

如果隐层神经元数量太多会引起过拟合，如果隐层神经元个数太少又可能会难以收敛。此外学习率的选择也是需要考虑。目前来说，对于参数的确定缺少一个简单有效的方法，所以导致算法很不稳定。

属于监督学习，对于样本有较大依赖性，网络学习的逼近和推广能力与样本有很大关系，如果样本集合代表性差，样本矛盾多，存在冗余样本，网络就很难达到预期的性能；

最后，由于权值是随机给定的，所以BP神经网络具有不可重现性。

# 4实验步骤

## 4.1安装依赖

本实验需要用到python的扩展程序库Numpy进行数值计算，如果没有安装过这个库，使用国内的清华pip源安装，命令如下：

pip3 install numpy scipy matplotlib -i <https://pypi.tuna.tsinghua.edu.cn/simple>

导入需要的库。

import numpy as np

from time import time

## 4.2计算激励函数

（1）Sigmoid函数及其导数

# 激励函数sigmoid

def sigmoid(x):

return 1 / (1 + np.exp(-x))

# 激励函数导数sigmoid\_d

def sigmoid\_d(x):

return x \* (1 - x)

（2）tanh函数及其导数

# 激励函数tanh

def tanh(x):

return np.tanh(x)

# 激励函数导数tanh\_d

def tanh\_d(x):

return 1.0 - np.tanh(x) \* np.tanh(x)

## 4.3建立BP神经网络模型

建立一个BP\_NN类，包含的成员函数如下：

（1）网络模型对象的构造函数\_\_init\_\_(self, layers, w, activation=tanh, activation\_d=tanh\_d)

根据layers中保存的信息（layers），构建网络层次结构，例如，layers=[63,9,3,10]表示要构造的神经网络一共有四层，输入层有63的节点，第一隐层有9个节点，第二隐层有3个节点，输出层有10个节点。

根据网络结构中每层的节点数设置权重矩阵，并给权重矩阵以一个期望为0,范围在[-w,w]的随机分布赋初始值（w<1）。

例如，输入层有i个输入、1个偏置，第一个隐层有j个节点、1个偏置，则输入层是1\*（i+1）矩阵，第一个隐层是1\*（j+1）矩阵。故，应设置（i+1）\*（j+1）的权重矩阵，以实现输入层到第一个隐层的变换。

（2）输入值前向传播实现预测predict(self, inputs)

利用矩阵乘法运算实现输入值在各层网络中的前向传播：

用到numpy的np.dot()实现矩阵乘法

（3）误差值后向传播back\_propagate(self, label, learn\_step)

利用矩阵乘法运算计算输出误差传播到各层的delta（误差函数对每个权值的偏导数）：

需要注意的是，对1\*n的numpy.ndarray型的一维向量的转置还是1\*n，要先调用np.atleast\_2d（）将1\*n的向量变成二维，转置后才会变成n\*1的矩阵，从而成功和1\*n的deltas[i]矩阵相乘得到n\*n的权值矩阵。

（4）训练train(self, X, Y, epochs=1000, learn\_step=0.1)

在设置好的迭代次数内，循环正向传播输入值，反向传播误差值，直到训练出合适的权值矩阵。

（5）BP神经网络详细代码

class BP\_ANN:

# layers保存结构信息，w是权重初始值范围上限，可利用activation变量选择激活函数，默认用sigmoid函数

def \_\_init\_\_(self, layers, w, activation=tanh, activation\_d=tanh\_d):

self.w0 = w

self.activation = activation

self.activation\_d = activation\_d

self.weights = [] # 权重矩阵

self.nodes\_in = [] # 节点的输入值u\_(k+1) = y\_k \* w\_k + b\_k

self.nodes\_out = [] # 节点的输出值y\_(k+1) = Sigmoid(u\_(k+1))

self.deltas = [] # 误差

# 输入层 nodes\_out = Sigmoid(nodes\_in)

self.nodes\_in.append(np.zeros(layers[0] + 1)) # “+1”是因为多设了一个偏置节点

self.nodes\_out.append(np.zeros(layers[0] + 1))

# 隐层

for i in range(1, len(layers) - 1):

# 初始时，以一个期望为0,范围在[-w,w]的随机分布设置各层的权重

self.weights.append((2 \* np.random.random((layers[i - 1] + 1, layers[i] + 1)) - 1) \* self.w0)

self.nodes\_in.append(np.zeros(layers[i] + 1)) # 第i层隐层，每个节点初始为0

self.nodes\_out.append(np.zeros(layers[i] + 1))

self.deltas.append(np.zeros(layers[i] + 1))

# 输出层

self.weights.append((2 \* np.random.random((layers[-2] + 1, layers[-1])) - 1) \* self.w0) # 输出节点不需要多设偏置节点

self.nodes\_in.append(np.zeros(layers[-1]))

self.nodes\_out.append(np.zeros(layers[-1]))

self.deltas.append(np.zeros(layers[-1]))

# 输入值前向传输, inputs是'numpy.ndarray'类型

def predict(self, inputs):

inputs = np.append(inputs, 1.0)

# 计算输入层

self.nodes\_in[0] = inputs

self.nodes\_out[0] = inputs

# 计算隐层（偏置节点恒为1）

for i in range(len(self.weights)):

# a(i+1) = zi \* wi + bi

self.nodes\_in[i + 1] = np.dot(self.nodes\_out[i], self.weights[i])

self.nodes\_in[i + 1][-1] = 1.0

# z(i+1) = S(a(i+1))

self.nodes\_out[i + 1] = self.activation(self.nodes\_in[i + 1])

self.nodes\_out[i + 1][-1] = 1.0

# 计算输出层（没有偏置节点）

self.nodes\_in[-1] = np.dot(self.nodes\_out[-2], self.weights[-1])

self.nodes\_out[-1] = self.activation(self.nodes\_in[-1])

return self.nodes\_out[-1]

# 误差值反向传播

def back\_propagate(self, label, learn\_step):

# 计算输出层误差: d\_m = [y\_m - y\_real] \* f\_m'[u\_m]

loss = label - self.nodes\_out[-1]

self.deltas[-1] = loss \* self.activation\_d(self.nodes\_in[-1])

# 计算其他层误差: d\_k = f\_k'(u\_k) \* d\_(k+1)\*w\_k

for i in range(len(self.nodes\_out) - 2, 0, -1): # 这里-2是因为输入层没有deltas

# 隐含层节点误差

self.deltas[i - 1] = np.dot(self.deltas[i], self.weights[i].T) \* self.activation\_d(self.nodes\_in[i])

# 更新各层权重、偏置值

for i in range(len(self.weights)):

self.weights[i] += learn\_step \* np.atleast\_2d(self.nodes\_out[i]).T.dot(

np.atleast\_2d(self.deltas[i])) # 权重更新

def train(self, X, Y, epochs=1000, learn\_step=0.1):

for k in range(epochs):

for m in range(len(X)):

case = X[m]

label = Y[m]

self.predict(case)

self.back\_propagate(label, learn\_step)

## 4.4 数字识别

（1）确定神经网络结构

输入是在7\*9的格子内以“0”和“1”画出的数字，所以输入层设置63个节点，隐层设置12个节点，输出层设置10个节点。当输出层除了第n个节点（n从0开始计数）为1，其余全为0时，代表识别出的数字是n。

（2）创建BP神经网络模型，利用训练样本和测试样本进行训练和预测

主函数代码如下。

if \_\_name\_\_ == '\_\_main\_\_':

train\_data = [[0, 0, 0, 0, 0, 0, 0, # 0

0, 1, 1, 1, 1, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 1

0, 0, 0, 1, 0, 0, 0,

0, 0, 1, 1, 0, 0, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 1, 1, 1, 0, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 2

0, 0, 1, 1, 1, 0, 0,

0, 1, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 1, 0, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 1, 0, 0, 0, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 3

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 1, 1, 0, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 4

0, 1, 0, 0, 0, 0, 0,

0, 1, 0, 1, 0, 0, 0,

0, 1, 0, 1, 0, 0, 0,

0, 1, 0, 1, 0, 0, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 5

0, 1, 1, 1, 1, 1, 0,

0, 1, 0, 0, 0, 0, 0,

0, 1, 0, 0, 0, 0, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 6

0, 1, 1, 1, 1, 1, 0,

0, 1, 0, 0, 0, 0, 0,

0, 1, 0, 0, 0, 0, 0,

0, 1, 1, 1, 1, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 7

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 1, 0, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 1, 0, 0, 0, 0,

0, 0, 1, 0, 0, 0, 0,

0, 0, 1, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 8

0, 1, 1, 1, 1, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 1, 1, 1, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 9

0, 0, 1, 1, 1, 1, 0,

0, 0, 1, 0, 0, 1, 0,

0, 0, 1, 0, 0, 1, 0,

0, 0, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 0, 0],

]

label = [[1, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 1, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 1, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]

test\_data = [[0, 0, 0, 0, 0, 0, 0, # 0

0, 1, 1, 1, 1, 1, 0,

0, 1, 0, 0, 0, 0.9, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 0.9, 0, 0, 0, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 1

0, 0, 0, 1, 0, 0, 0,

0, 0, 1, 0.9, 0, 0, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 0, 0.9, 0, 0, 0,

0, 0, 1, 1, 1, 0, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 2

0, 0.1, 1, 1, 1, 0, 0,

0, 0.9, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 1, 0, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 1, 0, 0, 0, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 3

0, 0.9, 1, 1, 1, 1, 0,

0, 0.1, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 1, 1, 0, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 4

0, 1, 0, 0, 0, 0, 0,

0.1, 0.9, 0, 1, 0, 0, 0,

0, 1, 0, 1, 0, 0, 0,

0, 1, 0, 1, 0, 0, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 5

0, 1, 1, 1, 1, 1, 0,

0, 1, 0, 0, 0, 0, 0,

0, 1, 0, 0, 0, 0, 0,

0, 0.9, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0.1, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 6

0, 1, 1, 1, 1, 1, 0,

0, 1, 0, 0, 0, 0, 0,

0, 1, 0, 0, 0, 0, 0,

0, 1, 1, 1, 1, 1, 0,

0, 1, 0, 0, 0, 0.9, 0.1,

0, 1, 0, 0, 0, 1, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 7

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 1, 0, 0,

0, 0, 0, 1, 0, 0, 0,

0, 0, 0.9, 0.1, 0, 0, 0,

0, 0, 1, 0, 0, 0, 0,

0, 0, 1, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 8

0, 1, 1, 1, 1, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 0.9, 0, 0, 0, 1, 0,

0, 1, 1, 1, 1, 1, 0,

0, 1, 0, 0, 0, 1, 0,

0, 1, 0.1, 0, 0, 1, 0,

0, 1, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, # 9

0, 0, 1, 1, 1, 1, 0,

0, 0.1, 1, 0, 0, 1, 0,

0, 0, 1, 0, 0, 1, 0,

0, 0, 1, 1, 1, 1, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 0.9, 0,

0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 0, 0],

]

X = np.array(train\_data)

Y = np.array(label)

TEST = np.array(test\_data)

# 这里选用了tanh作为激励函数

nn = BP\_ANN([63, 12, 10], 0.25, activation=tanh, activation\_d=tanh\_d)

nn.train(X, Y, 1000, 0.1)

# 训练

print("训练样本")

for index in range(len(X)):

print("标准%d：" % index)

print(nn.predict(X[index]) \* max\_num)

print("测试样本")

for index in range(len(TEST)):

print("%d的预测结果：" % index)

result = nn.predict(TEST[index]) \* max\_num

print(result)

for n in range(result.shape[0]):

if 0.5 < np.absolute(result[n]) < 0.9:

print("可能为%d。" % n)

elif np.absolute(result[n]) > 0.9:

print("应该是%d。" % n)

## 4.5函数逼近

### 4.5.1实验思路

随机生成300组（x，y，z），其中x、y、z是[0,10]范围内的随机数，将他们作为训练样本；用function函数计算他们对应的函数值作为训练标签。

再生成100组[0,10]内随机的（x，y，z）作为测试样本，比较训练得到的BP神经网络预测出的结果与实际结果之间的差距。改变网络结构多次实验，观察隐含层节点个数n对函数逼近效果的影响。

用均方根误差来表示预测结果对实际函数的逼近效果，均方根误差计算公式为：

公式3

### 4.5.2得到训练、测试数据

定义函数function(x, y, z)用于计算的真实值。

def function(x, y, z):

return x + np.power(y, 2) + np.power(z, 3)

得到训练、测试数据。随机取300组10以内的x,y,z并计算他们对应的函数值作为训练样本，随机取100组10以内x,y,z作为测试样本。

train\_num = 300

test\_num = 100

train\_data = np.random.rand(train\_num, 3) \* 10

label = []

for x\_y\_z in train\_data:

label.append(function(x\_y\_z[0], x\_y\_z[1], x\_y\_z[2]))

test\_data = np.random.rand(test\_num, 3) \* 10

### 4.5.3归一化预处理

这里用了线性归一化，让训练样本、训练标签、测试样本分别除以各自的最大元素，从而得到一个[0,1]上的均匀分布。这样预处理简单领过，但是如果出现了极大或极小的值，会导致映射后数据质量较低。

先定义find\_max(case, label)函数，找出训练样本和标签的最大值

# 找出训练数据的最大值

def find\_max(case, label):

a = case.max()

b = label.max()

return max(a, b)

线性归一化预处理，由于样本和标签都大于零，所以直接让数据除以对应的最大值。

X = np.array(train\_data)

Y = np.array(label)

TEST = np.array(test\_data)

max\_X = 10

max\_Y = function(10, 10, 10)

max\_TEST = 10

X = X / max\_X

Y = Y / max\_Y

TEST = TEST / max\_TEST

### 4.5.4实验

设置不同的迭代次数、隐含层节点个数，进行多次实验。

# 设置不同的迭代次数

epochs = [50, 150, 500, 1000, 2000]

for epoch in epochs:

# 设置不同的隐层节点数

HNode\_nums = [2, 6, 10, 14, 18, 22, 26, 30, 34, 40, 60, 100]

for HNode\_num in HNode\_nums:

# 这里选用了sigmoid作为激励函数

nn = BP\_ANN([3, HNode\_num, 1], 0.25, activation=tanh, activation\_d=tanh\_d)

# 训练

start = time()

nn.train(X, Y, epoch, 0.1)

end = time()

print("测试样本")

error = 0 # 计算和方差

for index in range(len(TEST)):

"""print(TEST[index] \* max\_TEST)"""

pred = nn.predict(TEST[index])[0] \* max\_Y

real = function(TEST[index][0] \* max\_TEST, TEST[index][1] \* max\_TEST, TEST[index][2] \* max\_TEST)

"""print("预测值为：%.2f" % pred)

print("实际值为：%.2f" % real)"""

"""loss = np.absolute(round(pred - real, 2))

print("偏倚为：%.2f" % loss)"""

error += (real - pred) \*\* 2

RMSE = np.sqrt(error / test\_num)

print("\*\*\*\*\*迭代次数设置为%d," % epoch + "隐层节点数设置为%d时：\*\*\*\*\*" % HNode\_num)

print("均方根误差为：%.2f" % RMSE)

print("训练用时间：%s seconds" % (end - start))

# 5实验结果

## 5.1数字识别实验

BP神经网络的结构设为三层结构，包括输入层63个节点、隐含层12个节点、输出层10个节点。用梯度下降法训练神经网络的权值，学习步长设置为0.1，迭代次数设置为1000。

用训练出的BP神经网络对测试样本进行预测实验，结果如下：

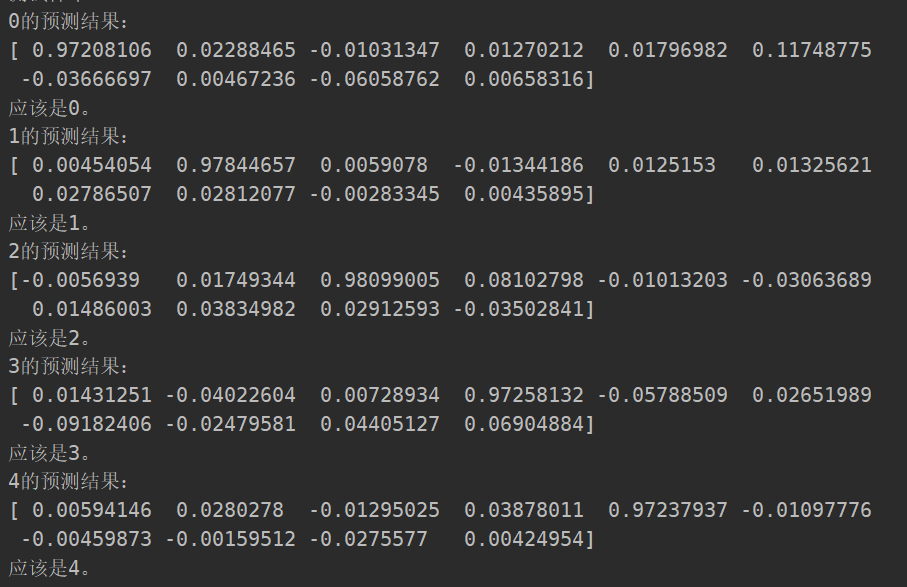


图1-1 数字识别实验结果（0~4）

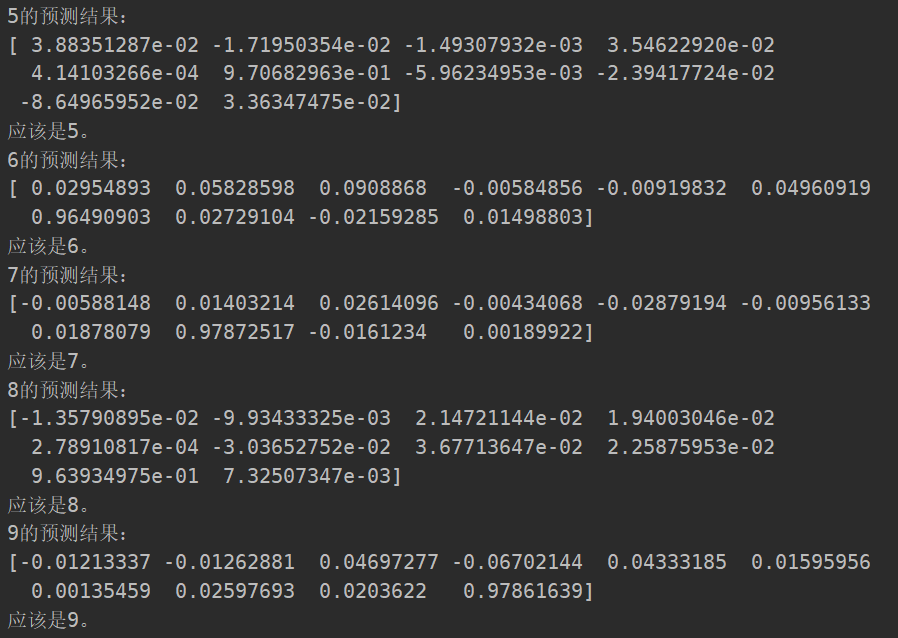


图1-2 BP 数字识别实验结果（5~9）

可以看到，预测的误差在0.1之内。改变网络结构多次实验，发现**本文题中**网络隐含层节点在大于8时才能有较好的结果。

由于训练样本过少，其实这样训练出的神经网络准确度较低，如果加上的噪音较大，则无法得到准确的预测结果。

## 5.2函数逼近实验

对于函数，训练一个BP神经网路，对于任意10以内的x、y、z，都能预测出函数值。

BP神经网络结构设置为3层，输入层节点为3个，隐含层节点为n个，输出层节点为1个。

（1）单次预测实验

随机取300组（x，y，z）作为训练样本，100组（x，y，z）作为测试样本。将迭代次数设置为500，学习步长为0.1，隐含层节点为8个。

部分实验结果如下：

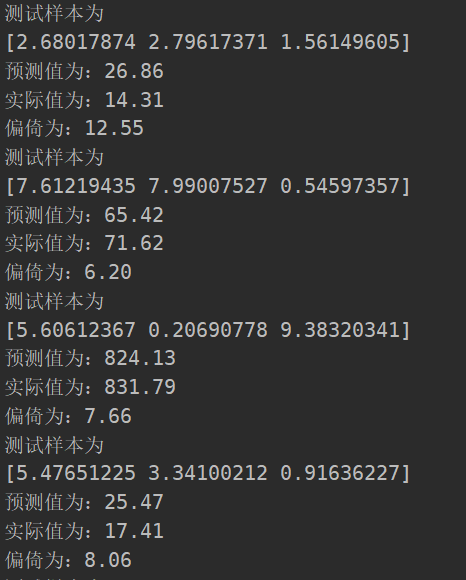


图2 隐含层节点为8

（2）探究隐含层节点数对逼近效果的影响

对相同的训练、测试样本，将隐含层节点依次设置为2、18、34、40、60、100、150、250，分别迭代100、1000次，学习步长统一设为0.1，进行数字识别实验。

实验结果统计如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| 迭代次数 | 隐含层层数 | 训练用时/s | 均根方差 |
| 100 | 2 | 1.6336 | 20.56 |
| 18 | 1.7443 | 23.66 |
| 34 | 1.8221 | 24.03 |
| 40 | 1.9967 | 25.45 |
| 60 | 1.7911 | 25.36 |
| 100 | 1.8201 | 29.02 |
| 150 | 2.0186 | 31.58 |
| 250 | 2.1861 | 121.06 |
| 1000 | 2 | 17.3460 | 16.04 |
| 18 | 17.5341 | 16.32 |
| 34 | 18.6132 | 16.30 |
| 40 | 18.4384 | 15.11 |
| 60 | 18.6970 | 15.95 |
| 100 | 19.3761 | 14.35 |
| 150 | 22.5188 | 14.31 |
| 250 | 23.0797 | 13.24 |

表1 隐含层层数对逼近效果影响实验结果