**** 成 绩

模式识别与机器学习实验报告

**实验四 神经网络和反向传播算法**

**院（系）名称自动化科学与电气工程学院**

**专 业 名 称 自动控制与模式识别**

**学 生 姓 名 雷诗叶**

**学 生 学 号 15031148**

**2018年6月18日**

**目录**

[一、 实验绪论 1](#_Toc17584_WPSOffice_Level1)

[二、 实验原理 1](#_Toc18920_WPSOffice_Level1)

[三、 实验内容 2](#_Toc26916_WPSOffice_Level1)

[3.1步骤一 2](#_Toc18920_WPSOffice_Level2)

[3.2步骤二 6](#_Toc26916_WPSOffice_Level2)

[3.3讨论 8](#_Toc25050_WPSOffice_Level2)

[四、 实验结论与收获 8](#_Toc25050_WPSOffice_Level1)

GitHub：https://github.com/LeavesLei/Pattern-Recognition.git

1. 实验绪论

人工神经网络（ANN）的学习模型（或者只是一个神经网络（NN））是一种受生物神经系统启发的方法，在现实世界中生物神经系统需要非常复杂的计算而不需要明确的量化操作。该技术的最初灵感来自于对神经元形成的大脑中生物电网络的研究及其突触。在神经网络模型中，简单的节点（称为各种“神经元”或“单位”）连接在一起形成节点网络，因此称为“神经网络”。

每个节点都有一组输入线，类似于生物神经元中的输入突触。每个节点还有一个“激活功能”，告诉节点何时激活，类似于生物神经元。最简单的形式就是激活函数只需要在输入总和大于某个值时生成'1'否则为'0'。但是，激活功能并不一定非常简单，事实上，创造可以做有用的事情的网络，他们几乎总是要更复杂，至少对于网络中的一些节点而言是复杂的。通常至少有三层前馈网络——输入层，隐藏层和输出层。输入层不进行处理，它只是数据向量进入网络的地方。输入层后是隐藏层，隐藏层，输出到输出层。网络中的实际处理发生在隐藏层和输出层的节点上。

1. 实验原理

任何监督学习算法的目标是找到一个输入到其正确输出的最佳映射集合的函数。一个简单的分类任务，输入是动物的图像，正确的输出将是名字这个动物。对于一个直观的例子，神经网络的第一层可能是负责使用来自个体图像中的像素输入来学习线的方向。第二层可能结合了第一层学到的功能层，并学会识别简单的形状，如圆圈。每个更高层学习越来越多的抽象特征，如上面提到的那些可以用来图像分类的特征。每个图层都在它下面的图层中找到图案，这就是这种能力创建独立于外部输入的内部表示多层次网络的力量。发展反向传播算法的目标和动力就是寻找一种训练多层神经网络的方法，以便可以学习适当的内部表示以允许它学习任何内容输入到输出的任意映射。

在数学上，神经元的网络函数f(x)被定义为其他的可以进一步定义为其他函数组合的组成函数gi(x)的组合。这个可以方便地表示为一个网络结构，用箭头表示变量之间的依赖关系。一种广泛使用的合成类型是非线性加权求和，其中：



其中K（通常称为激活函数）是一些预定义的函数，如双曲正切。为了方便，以下会将函数集合gi简单地称为向量g =（g 1，g 2，...，g n）。反向传播需要输入值都对应已知的、期望的输出，以为了计算损失函数梯度。因此通常被认为是一种有监督的学习方法。

平方误差函数是：



其中E是平方误差，t是训练样本的目标输出，并且y是输出神经元的实际输出。对于每个神经元j，定义其输出o j



神经元的输入网络是先前神经元输出的加权和。如果神经元是在输入层之后的第一层中，输入层的输出就是该网络的输入xk。神经元的输入单位数是n。变量w ij表示神经元i和j之间的权重。

激活函数φ通常是非线性和可微的。一般使用的激活函数是逻辑函数，例如：



其中有一个很好的衍生物：



计算误差相对于权重wij的偏导数就是使用两次链式规则：



我们最终可以得到：



和



1. 实验内容

3.1步骤一

1. （Python环境下）输入IRIS数据集，采用双隐藏层结构，通过BP算法进行模型训练以及数据分类预测。步骤流程图如下：

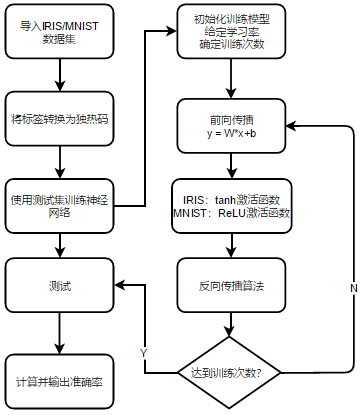


图3.1 程序实现流程图

1. 以训练集与测试集之比为7:3，学习率为0.01运行10次，结果如下：

accuracy: 0.9555555555555556

accuracy: 1.0

accuracy: 0.9777777777777777

accuracy: 0.9555555555555556

accuracy: 0.9333333333333333

accuracy: 0.9111111111111111

accuracy: 1.0

accuracy: 0.9333333333333333

accuracy: 0.9555555555555556

accuracy: 0.9555555555555556

Average: 0.9577777777777778

1. 结果分析与探究

（1）原程序结果如上

I. 双隐藏层，隐藏节点数目分别为10、10

II. 训练次数50\*150\*0.7=5250次，每次随机处理150\*0.3=45个训练集数据

III. 学习率0.01，BP算法训练因子0.001

IV. 训练集105组数据，测试集45组数据

程序运行时，实时打印训练次数及交叉熵，最终对30个测试集进行分类，打印对比结果并计算准确率，为95.78%。

1. 将学习率增加到0.05时，平均准确率为90%左右；将学习率降低到0.005，准确率明显提高，甚至可达到100%，当然测试集数据较少，存在偶然性。

可见，其他参数相同的情况下，学习率较小逼近最优解的速度变慢，增加训练次数可达到较好的效果，但过度增大学习率会导致准确率下降或错过最优值造成振荡（由损失函数变化可得），在学习率适中，迭代次数较大时变量初始化方式对最终准确率的影响不大。

（3）简单的前馈神经网络就可以认为是一种多层感知器，而大多数神经网络均有反向传播算法。

反向传播（backpropagation）算法经常用来连接优化算法求解成本函数最小化问题，比如梯度下降法。理论上，反向传播可以用于训练具有任意层、任意数量隐藏单元的前馈人工神经网络，但是计算能力的实际限制会约束反向传播的能力。反向传播与梯度下降法类似，根据成本函数的梯度变化不断更新模型参数。

1. 输入MNIST数据集，采用BP双隐藏层神经网络

由于MNIST数据集相较于IRIS数据集而言要大得多，因此我们将分别将隐藏层的神经元个数10,10增加为500,100，并将激活函数更换为ReLU。并且减小学习率。运行结果如下（重复运行5次）：

accuracy: 0.93

accuracy: 0.91

accuracy: 0.97

accuracy: 0.93

accuracy: 0.95

Average: 0.94

可以看到训练后神经网络的准确率基本可以保持在90%以上。而在训练神经网络的过程中也曾使用tanh和sigmoid作为激活函数，但结果不是特别理想。通过IRIS与MNIST两次实验的过程来看，我认为tanh和sigmoid对于小型数据集的训练比较适用。而对于于中大型数据集，则使用ReLU作为激活函数较好。

3.2步骤二

1. 激活函数与隐藏层数对效率和准确率的影响
2. I. 在sigmoid函数中，其输出是在(0,1)这个开区间内，可以联想到概率，曾经比较流行。但当输入稍微远离了坐标原点，函数的梯度就变得很小了，在反向传播的过程中会出现梯度饱和；且函数输出不是以0为中心的，这样会使权重更新效率降低。

II. tanh是双曲正切函数，和sigmoid函数比较相近。不同的是输出区间在(-1,1)之间，整个函数是以0为中心的，这个特点比sigmoid的好。一般二分类问题中，隐藏层用tanh函数，输出层用sigmoid函数。

III. ReLU(Rectified Linear Unit)函数是目前比较火的一个激活函数，相比于以上两个的优点是：在输入为正数的时候，不存在梯度饱和问题；计算速度要快很多；ReLU函数只有线性关系，不管是前向传播还是反向传播，都比sigmoid和tanh要快很多。当然，输入是负数时是完全不被激活的。

1. 一般认为，增加隐层数可以降低网络误差，提高精度，但也使网络复杂化，从而增加了网络的训练时间和出现“过拟合”的倾向。一般来讲应设计神经网络应优先考虑3层网络（即有1个隐层）。一般地，靠增加隐层节点数来获得较低的误差，其训练效果要比增加隐层数更容易实现。本实验中，也曾将隐藏层设为一层，效果发现不如两层的效果好（排除增加节点数的情况）。
2. 生物神经网络的工作方式与我们的NN模型相同吗？

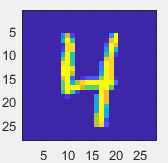
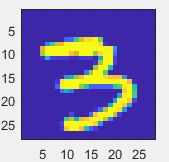
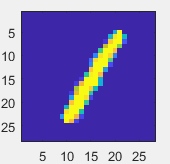
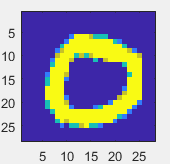
生物神经网络与人工神经网络模型工作方式是类似的。人工神经网络（ANN）旨在以一种抽象的方式模拟神经元处理新刺激的过程，但是规模变得更小、更简单。ANN由互连神经元的层构成，这种互连神经元可以接收一组输入和一组权重。然后进行数学操作，并将结果以“激活码”的方式输出，这与生物神经元中的突触十分相似，均需要输入刺激，并存在激活条件。在通常情况下ANN由数百到数千个神经元组成，而人脑的生物神经网络中的神经元要多得多。

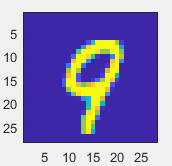
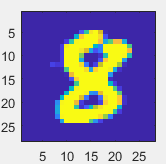
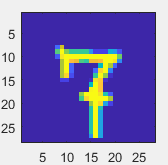
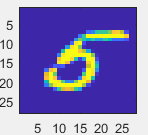
1. 自编码器

自动编码器是神经网络的一种，经过训练后能尝试将输入复制到输出。

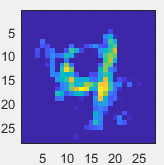
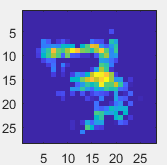
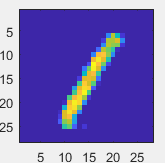
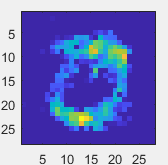
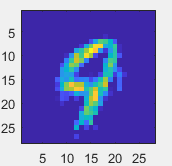
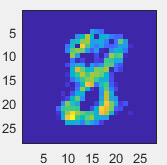
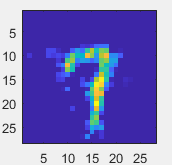
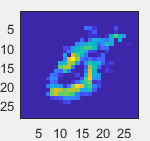
1. 本程序在python环境下采用单隐藏层实现

原始图像如下：

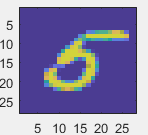
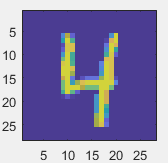
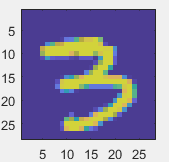




自编码器隐藏层节点数为64时输出图像如下：

通过对于可以看到当节点数较少时自编码器的输出较差，对于一些笔画较为简单的数字有较好的复现效果。但是对于一些笔画较为复杂的数组，则编码后得到的图片较为模糊。而将节点数提高到200以上时，自编码器的效果增强了不少，下图为自编码器隐藏层节点数为500时，数字3,4,5输出图像。可以看到相较于节点数为64时输出有了极大好转



（2）对于自编码器，如果隐藏节点比可视节点（输入、输出）少的话，由于被迫的降维，自编码器会自动习得训练样本的特征（变化最大，信息量最多的维度）。但是如果隐藏节点数目过多，甚至比可视节点数目还多的时候，自编码器不仅会丧失这种能力，更可能会习得一种“恒等函数”——直接把输入复制过去作为输出。这时候，我们需要对隐藏节点进行稀疏性限制。

所谓稀疏性，就是对一对输入图像，隐藏节点中被激活的节点数（输出接近1）远远小于被抑制的节点数目（输出接近0）。那么使得神经元大部分的时间都是被抑制的限制则被称作稀疏性限制。

使用比较少的隐含节点数，模型简单，但表现能力有限。 增加节点的数量，模型表示能力变强，但容易过拟合。所以网络被迫用少量的节点来学习到网络的整体信息，这些被学习到的特征都非常有代表性。

3.3讨论

神经网络在作为一种搜索策略的时候，受搜索算法的控制，当解空间函数存在局部最小值时，如果搜索步长较小（动量较小），那么有可能在这个局部求解时求得的所有解都指向了极小值的方法，就是陷入的局部最小。以BP为例，它是按照误差下降的最大梯度方向搜索，有可能由步长引起局部最优。所以，实际中人们通常采用自适应学习率调整、模拟退火技术、多组不同参数初始化等方法尽量避免陷入局部最优。

自动编码器是一种数据的压缩算法，其中数据的压缩和解压缩函数是数据相关的、有损的、从样本中自动学习的。在大部分提到自动编码器的场合，压缩和解压缩的函数是通过神经网络实现的。自编码器是一个3层或者大于3层的神经网络，将输入表达X编码为一个新的表达Y，然后再将Y解码回X。这是一个非监督学习算法，使用反向传播算法来训练网络使得输出等于输入。自编码训练遵循的是BP算法，但是与传统的BP算法不同的是输出级并不是待定给出的标签信号，而是与输入一致的信号。而一般浅层网络的结构可以用BP 实现，但是层数多了容易陷入极值点。

1. 实验结论与收获

本次实验使用python编写双隐藏层的BP神经网络，并使用iris数据集以及MNIST数据集对其进行训练。通过Neural Networks and Back Propagation代码的编写以及不断的debug，使得我对于交叉熵、激活函数等基本概念有了更加深入的了解。

在调试代码的过程中，发现自己的代码写的并没有问题，但是准确率一直上不去。通过不断的尝试以及调整发现BP神经网络的准确率与许多参数都息息相关。在尝试不同激活函数（ReLU、sigmoid、tanh）的过程中，发现使用tanh激活函数的效果最好。最后通过不断调整学习率和初始化矩阵的随机值，以及使用可变步长和尝试了多种激活函数之后，终于将准确率调整到了95%以上。

在machine learning中很多时候调参往往起着决定性的作用。而这次BP神经网络的实验让我深刻明白的调整参数的重要性。尽管在debug的时候花了很多时间，走了很多弯路。但是在最后看到accuracy = 1.0的那一次，觉得一切都是值得的。一次又一次debug，其实也是不断学习的过程。这次BP神经网络实验在帮助我理解machine learning方面真的使我受益匪浅。

附录1：针对IRIS的双隐藏层BP神经网络代码

1. **import** numpy as np
2. **from** sklearn.datasets **import** load\_iris
3. **from** sklearn.cross\_validation **import** train\_test\_split
5. #前项传播计算函数
6. **def** layerComputing(inputNum,weightNum,biaseNum):
7. outputNum = tanh(np.dot(inputNum,weightNum) + biaseNum)
8. **return** outputNum
10. #sigmoid激活函数
11. **def** sigmoid(n):
12. **for** i **in** range(np.size(n)):
13. n[0,i] = 1/(1+np.exp(-n[0,i]))
14. **return** n
16. #ReLU激活函数
17. **def** ReLU(n):
18. **for** i **in** range(np.size(n)):
19. **if**(n[0,i]<=0):
20. n[0,i] = 0
21. **return** n
23. #tanh激活函数
24. **def** tanh(n):
25. **for** i **in** range(np.size(n)):
26. n[0,i] = np.tanh(n[0,i])
27. **return** n
29. #损失函数
30. **def** Loss(output,target):
31. loss = target - output
33. loss = loss \* loss
34. n = 0
35. **for** i **in** range(np.size(loss)):
36. n = n + loss[0,i]
37. n = n/2
38. **return** n
40. #信号函数
41. **def** sign(n):
42. **for** i **in** range(np.size(n)):
43. **if**(n[0,i]<=0):
44. n[0,i] = 0
45. **else**:
46. n[0,i] = 1
47. **return** n
49. #将输出转化为属于某一类的形式
50. **def** out2class(n):
51. #找出最大数的索引值
52. order = np.argmax(n)
53. **return** order
55. #独热码计算函数
56. **def** oneHot(n,outputNum):
57. m = np.zeros((1,outputNum))
58. m[0,n] = 1
59. **return** m
61. **if** \_\_name\_\_ == '\_\_main\_\_':
62. #导入iris数据集
63. iris=load\_iris()
64. attributes=iris.data  #获取属性数据
65. target=iris.target  #获取类别数据
66. labels=iris.feature\_names #获取列属性值
67. #划分数据集和测试集
68. test\_proportion = 0.3#测试集所占比例
69. train\_proportion = 1 - test\_proportion #训练集所占比例
70. #X\_train,Y\_train分别是数据集的data和target，X\_test，Y\_target分别是测试集的data和target
71. X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(attributes, target, test\_size=test\_proportion)
72. #定义步长
73. step = 0.01
74. #定义可变步长
75. step1 = 0.008
76. #输入层神经元数目
77. input\_num = 4
78. #输出层神经元数目
79. output\_num = 3
80. #调整矩阵分布
81. distribution = 0.1
82. #定义第一层神经元
83. layer1\_num = 10
84. layer1 = (2\*np.random.randn(1,layer1\_num)-1)\*distribution
85. layer1\_weight = (2\*np.random.randn(input\_num,layer1\_num)-1)\*distribution
86. layer1\_biase = (2\*np.random.randn(1,layer1\_num)-1)\*distribution
88. #定义第二层神经元
89. layer2\_num = 10
90. layer2 =  (2\*np.random.randn(1,layer2\_num)-1)\*distribution
91. layer2\_weight = (2\*np.random.randn(layer1\_num,layer2\_num)-1)\*distribution
92. layer2\_biase = (2\*np.random.randn(1,layer2\_num)-1)\*distribution
94. #定义输出层神经元
95. layer\_out = (2\*np.random.randn(1,output\_num)-1)\*distribution
96. out\_weight = (2\*np.random.randn(layer2\_num,output\_num)-1)\*distribution
97. out\_biase = (2\*np.random.randn(1,output\_num)-1)\*distribution
99. #定义一个数组实现可变步长
100. change\_past = [0,0,0,0,0,0]
101. #change\_now = [0,0,0,0,0,0]
103. #train
104. **for** i **in** range(50):
105. **for** j **in** range((int)(150\*train\_proportion)):
106. layer\_in  = np.zeros((1,input\_num))
107. **for** k **in** range(input\_num):
108. layer\_in[0,k] = X\_train[j,k]
109. #layer\_in[0,k] = 1
111. #正向计算
112. layer1 = layerComputing(layer\_in,layer1\_weight,layer1\_biase)
113. layer2 = layerComputing(layer1,layer2\_weight,layer2\_biase)
114. layer\_out = layerComputing(layer2,out\_weight,out\_biase)
116. #计算损失函数
117. one\_hot\_target = oneHot(Y\_train[j],output\_num)
118. loss = Loss(layer\_out,one\_hot\_target)
120. out\_loss = (1 - layer\_out \* layer\_out) \* (layer\_out - one\_hot\_target)
121. layer2\_loss = np.dot(out\_loss,out\_weight.T) \* (1 - layer2 \* layer2)
122. layer1\_loss = np.dot(layer2\_loss,layer2\_weight.T) \* (1- layer1 \* layer1)
124. #计算权重的偏导数
125. layer2\_W\_out\_loss = np.dot(layer2.T,out\_loss)
126. layer1\_W\_layer2\_loss = np.dot(layer1.T,layer2\_loss)
127. in\_W\_layer1\_loss = np.dot(layer\_in.T,layer1\_loss)
128. #计算biase的偏导数
129. layer2\_B\_out\_loss = out\_loss
130. layer1\_B\_layer2\_loss = layer2\_loss
131. in\_B\_layer1\_loss = layer1\_loss
132. #更新weight和biase
133. layer1\_weight = layer1\_weight - step \* in\_W\_layer1\_loss - step1 \* change\_past[0]
134. layer2\_weight = layer2\_weight - step \* layer1\_W\_layer2\_loss - step1 \* change\_past[1]
135. out\_weight = out\_weight - step \* layer2\_W\_out\_loss - step1 \* change\_past[2]
137. layer1\_biase = layer1\_biase - step \* in\_B\_layer1\_loss - step1 \* change\_past[3]
138. layer2\_biase = layer2\_biase - step \* layer1\_B\_layer2\_loss - step1 \* change\_past[4]
139. out\_biase = out\_biase - step \* layer2\_B\_out\_loss - step1 \* change\_past[5]
140. #更新可变步长调整值
141. change\_past[0] = in\_W\_layer1\_loss
142. change\_past[1] = layer1\_W\_layer2\_loss
143. change\_past[2] = layer2\_W\_out\_loss
144. change\_past[3] = in\_B\_layer1\_loss
145. change\_past[4] = layer1\_B\_layer2\_loss
146. change\_past[5] = layer2\_B\_out\_loss
147. #test
148. pass\_num = 0
149. all\_num = 150\*test\_proportion
151. #正向计算
152. **for** j **in** range((int)(150\*test\_proportion)):
153. layer\_in  = np.zeros((1,input\_num))
154. **for** k **in** range(input\_num):
155. layer\_in[0,k] = X\_test[j,k]
156. layer1 = layerComputing(layer\_in,layer1\_weight,layer1\_biase)
157. layer2 = layerComputing(layer1,layer2\_weight,layer2\_biase)
158. layer\_out = layerComputing(layer2,out\_weight,out\_biase)
159. class\_num = out2class(layer\_out)
160. **if**(class\_num==Y\_test[j]):
161. pass\_num = pass\_num + 1
163. accuracy = pass\_num / all\_num
164. **print**("accuracy:",accuracy)

附录2：针对MNIST的双隐藏层BP神经网络代码

1. **import** numpy as np
2. **import** struct
4. **def** loadImageSet(filename):
5. **print**("load image set",filename)
6. binfile= open(filename, 'rb')
7. buffers = binfile.read()
9. head = struct.unpack\_from('>IIII' , buffers ,0)
10. **print**("head,",head)
12. offset = struct.calcsize('>IIII')
13. imgNum = head[1]
14. width = head[2]
15. height = head[3]
16. #[60000]\*28\*28
17. bits = imgNum \* width \* height
18. bitsString = '>' + str(bits) + 'B' #like '>47040000B'
20. imgs = struct.unpack\_from(bitsString,buffers,offset)
22. binfile.close()
23. imgs = np.reshape(imgs,[imgNum,1,width\*height])
24. **print**("load imgs finished")
25. **return** imgs
27. **def** loadLabelSet(filename):
29. **print**("load label set",filename)
30. binfile = open(filename, 'rb')
31. buffers = binfile.read()
33. head = struct.unpack\_from('>II' , buffers ,0)
34. **print**("head,",head)
35. imgNum=head[1]
37. offset = struct.calcsize('>II')
38. numString = '>'+str(imgNum)+"B"
39. labels = struct.unpack\_from(numString , buffers , offset)
40. binfile.close()
41. labels = np.reshape(labels,[imgNum,1])
43. **print**('load label finished')
44. **return** labels
46. #前项传播计算函数
47. **def** layerComputing(inputNum,weightNum,biaseNum):
48. outputNum = ReLU(np.dot(inputNum,weightNum) + biaseNum)
49. **return** outputNum
51. #sigmoid激活函数
52. **def** sigmoid(n):
53. **for** i **in** range(np.size(n)):
54. n[0,i] = 1/(1+np.exp(-n[0,i]))
55. **return** n
57. #ReLU激活函数
58. **def** ReLU(n):
59. **for** i **in** range(np.size(n)):
60. **if**(n[0,i]<=0):
61. n[0,i] = 0
62. **return** n
64. #tanh激活函数
65. **def** tanh(n):
66. **for** i **in** range(np.size(n)):
67. n[0,i] = np.tanh(n[0,i])
68. **return** n
70. #损失函数
71. **def** Loss(output,target):
72. loss = target - output
74. loss = loss \* loss
75. n = 0
76. **for** i **in** range(np.size(loss)):
77. n = n + loss[0,i]
78. n = n/2
79. **return** n
81. #信号函数
82. **def** sign(n):
83. **for** i **in** range(np.size(n)):
84. **if**(n[0,i]<=0):
85. n[0,i] = 0
86. **else**:
87. n[0,i] = 1
88. **return** n
90. #将输出转化为属于某一类的形式
91. **def** out2class(n):
92. #找出最大数的索引值
93. order = np.argmax(n)
94. **return** order
96. #独热码计算函数
97. **def** oneHot(n,outputNum):
98. m = np.zeros((1,outputNum))
99. m[0,n] = 1
100. **return** m
102. **if** \_\_name\_\_ == '\_\_main\_\_':
103. #导入MNIST数据集
104. imgs = loadImageSet("train-images-idx3-ubyte")
105. labels = loadLabelSet("train-labels-idx1-ubyte")
106. test\_imgs = loadImageSet("t10k-images-idx3-ubyte")
107. test\_labels = loadLabelSet("t10k-labels-idx1-ubyte")
108. #定义步长
109. step = 0.01
110. #训练集以及测试集的个数
111. train\_num = 60000
112. test\_num = 10000
113. #输入层神经元数目
114. input\_num = 784
115. #输出层神经元数目
116. output\_num = 10
117. #调整矩阵分布
118. distribution = 0.008
119. #定义第一层神经元
120. layer1\_num = 500
121. layer1 = (2\*np.random.randn(1,layer1\_num)-1)\*distribution
122. layer1\_weight = (2\*np.random.randn(input\_num,layer1\_num)-1)\*distribution
123. layer1\_biase = (2\*np.random.randn(1,layer1\_num)-1)\*distribution
125. #定义第二层神经元
126. layer2\_num = 100
127. layer2 =  (2\*np.random.randn(1,layer2\_num)-1)\*distribution
128. layer2\_weight =(2\*np.random.randn(layer1\_num,layer2\_num)-1)\*distribution
129. layer2\_biase = (2\*np.random.randn(1,layer2\_num)-1)\*distribution
131. #定义输出层神经元
132. layer\_out = (2\*np.random.randn(1,output\_num)-1)\*distribution
133. out\_weight = (2\*np.random.randn(layer2\_num,output\_num)-1)\*distribution
134. out\_biase = (2\*np.random.randn(1,output\_num)-1)\*distribution
136. #train
137. **for** i **in** range(3):
138. **for** j **in** range(train\_num):
139. layer\_in  = np.zeros((1,input\_num))
140. **for** k **in** range(input\_num):
141. layer\_in[0,k] = imgs[j,0,k]
143. #正向计算
144. layer1 = layerComputing(layer\_in,layer1\_weight,layer1\_biase)
145. layer2 = layerComputing(layer1,layer2\_weight,layer2\_biase)
146. layer\_out = layerComputing(layer2,out\_weight,out\_biase)
148. #计算损失函数
149. one\_hot\_target = oneHot(labels[j,0],output\_num)
151. loss = Loss(layer\_out,one\_hot\_target)
153. out\_loss = sign(layer\_out) \* (layer\_out - one\_hot\_target)
154. layer2\_loss = np.dot(out\_loss,out\_weight.T) \*sign(layer2)
155. layer1\_loss = np.dot(layer2\_loss,layer2\_weight.T) \*sign(layer1)
156. #in\_loss = np.dot(layer1\_loss,layer1\_weight.T) \* sign(layer\_in)
158. #计算权重的偏导数
159. layer2\_W\_out\_loss = np.dot(layer2.T,out\_loss)
160. layer1\_W\_layer2\_loss = np.dot(layer1.T,layer2\_loss)
161. in\_W\_layer1\_loss = np.dot(layer\_in.T,layer1\_loss)
162. #计算biase的偏导数
163. layer2\_B\_out\_loss = out\_loss
164. layer1\_B\_layer2\_loss = layer2\_loss
165. in\_B\_layer1\_loss = layer1\_loss
166. #更新weight和biase
168. layer1\_weight = layer1\_weight - step \* in\_W\_layer1\_loss
169. layer2\_weight = layer2\_weight - step \* layer1\_W\_layer2\_loss
170. out\_weight = out\_weight - step \* layer2\_W\_out\_loss
172. layer1\_biase = layer1\_biase - step \* in\_B\_layer1\_loss
173. layer2\_biase = layer2\_biase - step \* layer1\_B\_layer2\_loss
174. out\_biase = out\_biase - step \* layer2\_B\_out\_loss
175. #test
176. pass\_num = 0
177. all\_num = test\_num
179. #正向计算
180. **for** j **in** range(test\_num):
181. layer\_in  = np.zeros((1,input\_num))
182. **for** k **in** range(input\_num):
183. layer\_in[0,k] = test\_imgs[j,0,k]
184. layer1 = layerComputing(layer\_in,layer1\_weight,layer1\_biase)
185. layer2 = layerComputing(layer1,layer2\_weight,layer2\_biase)
186. layer\_out = layerComputing(layer2,out\_weight,out\_biase)
187. class\_num = out2class(layer\_out)
188. **if**(class\_num==test\_labels[j,0]):
189. pass\_num = pass\_num + 1
191. accuracy = pass\_num / all\_num
192. **print**("accuracy",accuracy)

附录三：针对MINIST的自编码器代码

1. **import** numpy as np
2. **import** struct
4. **def** loadImageSet(filename):
5. **print**("load image set",filename)
6. binfile= open(filename, 'rb')
7. buffers = binfile.read()
9. head = struct.unpack\_from('>IIII' , buffers ,0)
10. **print**("head,",head)
12. offset = struct.calcsize('>IIII')
13. imgNum = head[1]
14. width = head[2]
15. height = head[3]
16. #[60000]\*28\*28
17. bits = imgNum \* width \* height
18. bitsString = '>' + str(bits) + 'B' #like '>47040000B'
20. imgs = struct.unpack\_from(bitsString,buffers,offset)
22. binfile.close()
23. imgs = np.reshape(imgs,[imgNum,1,width\*height])
24. **print**("load imgs finished")
25. **return** imgs
27. **def** loadLabelSet(filename):
29. **print**("load label set",filename)
30. binfile = open(filename, 'rb')
31. buffers = binfile.read()
33. head = struct.unpack\_from('>II' , buffers ,0)
34. **print**("head,",head)
35. imgNum=head[1]
37. offset = struct.calcsize('>II')
38. numString = '>'+str(imgNum)+"B"
39. labels = struct.unpack\_from(numString , buffers , offset)
40. binfile.close()
41. labels = np.reshape(labels,[imgNum,1])
43. **print**('load label finished')
44. **return** labels
46. #前项传播计算函数
47. **def** layerComputing(inputNum,weightNum,biaseNum):
48. outputNum = ReLU(np.dot(inputNum,weightNum) + biaseNum)
49. **return** outputNum
51. #sigmoid激活函数
52. **def** sigmoid(n):
53. **for** i **in** range(np.size(n)):
54. n[0,i] = 1/(1+np.exp(-n[0,i]))
55. **return** n
57. #ReLU激活函数
58. **def** ReLU(n):
59. **for** i **in** range(np.size(n)):
60. **if**(n[0,i]<=0):
61. n[0,i] = 0
62. **return** n
64. #tanh激活函数
65. **def** tanh(n):
66. **for** i **in** range(np.size(n)):
67. n[0,i] = np.tanh(n[0,i])
68. **return** n
70. #损失函数
71. **def** Loss(output,target):
72. loss = target - output
74. loss = loss \* loss
75. n = 0
76. **for** i **in** range(np.size(loss)):
77. n = n + loss[0,i]
78. n = n/2
79. **return** n
81. #信号函数
82. **def** sign(n):
83. **for** i **in** range(np.size(n)):
84. **if**(n[0,i]<=0):
85. n[0,i] = 0
86. **else**:
87. n[0,i] = 1
88. **return** n
90. #将输出转化为属于某一类的形式
91. **def** out2class(n):
92. #找出最大数的索引值
93. order = np.argmax(n)
94. **return** order
96. **if** \_\_name\_\_ == '\_\_main\_\_':
97. #导入MNIST数据集
98. imgs = loadImageSet("train-images-idx3-ubyte")
99. labels = loadLabelSet("train-labels-idx1-ubyte")
100. test\_imgs = loadImageSet("t10k-images-idx3-ubyte")
101. test\_labels = loadLabelSet("t10k-labels-idx1-ubyte")
102. #定义步长
103. step = 0.01
104. #训练集以及测试集的个数
105. train\_num = 60000
106. test\_num = 10000
107. #输入层神经元数目
108. input\_num = 784
109. #输出层神经元数目
110. output\_num = 784
111. #调整矩阵分布
112. distribution = 0.008
113. #定义隐藏层神经元
114. layer1\_num = 64
115. layer1 = (2\*np.random.randn(1,layer1\_num)-1)\*distribution
116. layer1\_weight = (2\*np.random.randn(input\_num,layer1\_num)-1)\*distribution
117. layer1\_biase = (2\*np.random.randn(1,layer1\_num)-1)\*distribution

120. #定义输出层神经元
121. layer\_out = (2\*np.random.randn(1,output\_num)-1)\*distribution
122. out\_weight = (2\*np.random.randn(layer1\_num,output\_num)-1)\*distribution
123. out\_biase = (2\*np.random.randn(1,output\_num)-1)\*distribution

126. #train
127. **for** i **in** range(3):
128. **for** j **in** range(train\_num):
129. layer\_in  = np.zeros((1,input\_num))
130. **for** k **in** range(input\_num):
131. layer\_in[0,k] = imgs[j,0,k]
133. #正向计算
134. layer1 = layerComputing(layer\_in,layer1\_weight,layer1\_biase)
135. layer\_out = layerComputing(layer1,out\_weight,out\_biase)
137. #计算损失函数
138. loss = Loss(layer\_out,layer\_in)
140. out\_loss = sign(layer\_out) \* (layer\_out - layer\_in)
141. layer1\_loss = np.dot(out\_loss,out\_weight.T) \*sign(layer1)
142. #in\_loss = np.dot(layer1\_loss,layer1\_weight.T) \* sign(layer\_in)
144. #计算权重的偏导数
145. layer1\_W\_out\_loss = np.dot(layer1.T,out\_loss)
146. in\_W\_layer1\_loss = np.dot(layer\_in.T,layer1\_loss)
147. #计算biase的偏导数
148. layer1\_B\_out\_loss = out\_loss
149. in\_B\_layer1\_loss = layer1\_loss
150. #更新weight和biase
152. layer1\_weight = layer1\_weight - step \* in\_W\_layer1\_loss
153. out\_weight = out\_weight - step \* layer1\_W\_out\_loss
155. layer1\_biase = layer1\_biase - step \* in\_B\_layer1\_loss
156. out\_biase = out\_biase - step \* layer1\_B\_out\_loss
158. #自编码器输出
159. #选择测试集中的第1000副图片输出
160. n = 1000
161. layer\_in  = np.zeros((1,input\_num))
162. **for** k **in** range(input\_num):
163. layer\_in[0,k] = test\_imgs[n,0,k]
164. layer1 = layerComputing(layer\_in,layer1\_weight,layer1\_biase)
165. layer\_out = layerComputing(layer1,out\_weight,out\_biase)
166. image\_in = layer\_in
167. image\_out = layer\_out
168. **print**(image\_in)
169. **print**(image\_out)