**华东师范大学数据学院实践报告**

|  |  |  |
| --- | --- | --- |
| **课程名称**： 专业英语 | **年级**：大二 |  |
| **指导教师**：周烜 | **姓名**：陈诺 |  |
| **上机实践名称**： 手搭神经网络 | **学号**： 10175501112 | **上机实践日期**：2019.6 |

1. **目的**

利用python手搭神经网络

1. **内容与设计思想**

本次作业名称：Neural Network Practice

作业内容：本次作业主要是希望通过实践，加深大家对于神经网络的理解。作业的数据集：[https://www.kaggle.com/uciml/mushroom-classification#mushrooms.csv](https://www.kaggle.com/uciml/mushroom-classification" \l "mushrooms.csv)

Classification data Sources，作业内容主要是对收集的Mushroom数据进行分类，判断是否有毒，具体的数据格式

以及详细要求参考以上链接。

作业要求：使用自己搭建的人工神经网络模型完成Classification任务，并且对模型进行不断优化，将准确率提升到极

限。希望同学们自己动手搭建神经网络。代码实现可以参考教科书中的手写数字识别样例代码neural-networksand-

deep-learning。不推荐使用已有的集成Machine Learning库，⽐如：keras，sklearn等。目的是让大家深入

体会神经网络的构造细节。

提交材料：

1. 报告一份，描述神经网络的构建过程、优化过程、参数选择的考虑、以及心得体会。

2. 代码一份。

提交时间：7月4日

参考资料：<http://neuralnetworksanddeeplearning.com/index.html>

1. **使用环境**

Python3.7

1. **实验过程**



首先要进行数据处理，这里将毒蘑菇置0不毒置1，并将参数映射到0-1范围 内，使参数相对联系较为紧密。

由于其中有一列缺失‘？’值较多，故不失一般性，将此参数删去，数据大小变为8124\*22。

data=data.drop('stalk-root',axis=1)#缺省太多的特征不去考虑

顺便提一下，它共有8124个样本。实际上，可以用稍微不同的⽅法对数据进⾏划分：

将样本按3：1的比例分割为6093个训练集和2031个测试集：

from sklearn.model\_selection import train\_test\_split

X, y = datanp[:, 1:], datanp[:, 0]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=0)



以下是加在数据的细节：

import pickle

import gzip

# def load\_data():

# f = gzip.open('../data/mnist.pkl.gz', 'rb')

# training\_data, validation\_data, testing\_data = pickle.load(f,encoding='bytes')

def load\_data\_wrapper(train\_data,test\_data):

"""Return a tuple containing ``(training\_data, validation\_data,

test\_data)``. Based on ``load\_data``, but the format is more

convenient for use in our implementation of neural networks.

In particular, ``training\_data`` is a list containing 50,000

2-tuples ``(x, y)``. ``x`` is a 784-dimensional numpy.ndarray

containing the input image. ``y`` is a 10-dimensional

numpy.ndarray representing the unit vector corresponding to the

correct digit for ``x``.

``validation\_data`` and ``test\_data`` are lists containing 10,000

2-tuples ``(x, y)``. In each case, ``x`` is a 784-dimensional

numpy.ndarry containing the input image, and ``y`` is the

corresponding classification, i.e., the digit values (integers)

corresponding to ``x``.

Obviously, this means we're using slightly different formats for

the training data and the validation / test data. These formats

turn out to be the most convenient for use in our neural network

code."""

tr\_d, te\_d = train\_data,test\_data

training\_inputs = [np.reshape(x, (21, 1)) for x in tr\_d[0]]

training\_results = [vectorized\_result(y) for y in tr\_d[1]]

training\_data = zip(training\_inputs, training\_results)

test\_inputs = [np.reshape(x, (21, 1)) for x in te\_d[0]]

test\_results = [vectorized\_result(y) for y in te\_d[1]]

test\_data = zip(test\_inputs, test\_results)

training\_data=list(training\_data)

test\_data=list(test\_data)

return (training\_data,test\_data)

def vectorized\_result(j):

"""Return a 10-dimensional unit vector with a 1.0 in the jth

position and zeroes elsewhere. This is used to convert a digit

(0,1) into a corresponding desired output from the neural

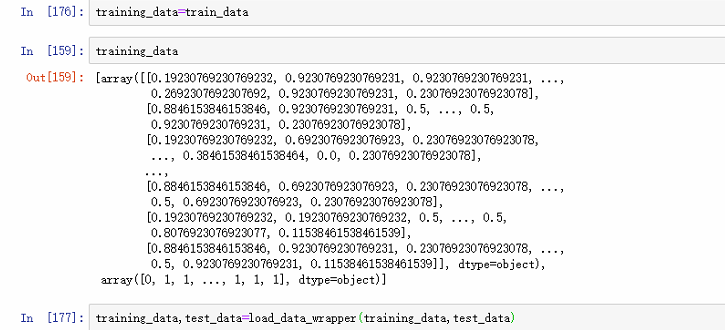
network."""

e = np.zeros((2, 1))

e[j] = 1.0

return e

其将train\_data,test\_data变换成为我们需要的格式。



除了这些数据，之后还需要用到Numpy，用来做快速线性代数。

代码的核心片段是一个Network 类，我们用来表示一个神经网络。这是用来初始化一个Network 对象的代码：

class Network(object):

def \_\_init\_\_(self, sizes):

self.num\_layers = len(sizes)

self.sizes = sizes

self.biases = [np.random.randn(y, 1) for y in sizes[1:]]

self.weights = [np.random.randn(y, x)

for x, y in zip(sizes[:-1], sizes[1:])]

在这段代码中，列表sizes 包含各层神经元的数量。

Network 对象中的偏置和权重都是被随机初始化的，使用Numpy 的np.random.randn 函数来

⽣成均值为0，标准差为1 的高斯分布。这样的随机初始化给了我们的随机梯度下降算法一个起点。注意Network 初始化代码假设第一层神经元是一个输⼊层，并对这些神经元不设置任何偏置，因为偏置仅在后⾯的层中用于计算输出。

然后对Network 类添加一个feedforward ⽅法，对于网络给定一个输⼊a，返回对应的输

出。这个⽅法所做的是对每一层应用⽅程：

def feedforward(self, a):

*"""Return the output of the network if "a" is input."""*

for b, w in zip(self.biases, self.weights):

a = sigmoid(np.dot(w, a)+b)

return a

当然，我们想要Network 对象做的主要事情是学习。为此给它们一个实现随机梯度下降算法。代码如下

import random

import json

class Network(object):

#sizes 是一个数组，其中每一个数值表示每一层的神经元的数量

def \_\_init\_\_(self, sizes):#[21,15,2]

self.num\_layers = len(sizes)#3层

self.sizes = sizes

self.biases = [np.random.randn(y,1) for y in sizes[1:]]#初始化偏置

#weights 的shape 是 （n+1）\* n的形式

self.weights = [np.random.randn(y,x) for x, y in zip(sizes[:-1],sizes[1:])]#21\*15 15\*2

def sigmoid(self,z):

#randn(x,y) 会生成一个(x\*y)的随机数组

z=z.astype(float)

return 1.0/(1.0+np.exp(-z))

def sigmoid\_prime(self,z):#sigmoid的导数

return self.sigmoid(z) \* (1 - self.sigmoid(z))

def feedforward(self,a):#前馈神经网络 a input 把output输出

""" return the output of the network if "a" is input """

for b, w in zip(self.biases,self.weights):

a = self.sigmoid(np.dot(w,a)+b)#按元素点乘

return a

def SGD(self, training\_data, epochs, mini\_batch\_size, eta, test\_data = None):

#training\_data,30,10,3.0,test\_data=test\_data

'''

training\_data 是训练集，epochs是训练周期

mini\_batch\_size 是mini\_batch\_SGD 算法中 一个组的size

eta 是学习速率

'''

'''

用小批量随机梯度下降训练神经网络。' training\_data ' '是一个列表' ' (x, y) ' '表示训练输入和所需输出。其他非可选参数是不言自明的。

如果提供了' ' test\_data ' '，则在每个历元之后根据测试数据对网络进行评估，并打印出部分进度。这是有用的跟踪进展，但大大放慢了速度。

'''

evaluation\_accuracy = []

training\_accuracy = []

if test\_data:

n\_test = len(test\_data)#2031

n = len(training\_data)#6093

for j in range(epochs):

random.shuffle(training\_data)#随机排

mini\_batches = [

training\_data[k:k+mini\_batch\_size]

for k in range(0, n, mini\_batch\_size)]#0-10的对 11-20的对

for mini\_batch in mini\_batches:

self.update\_mini\_batch(mini\_batch, eta)

if test\_data:

print("Epoch {0}: {1} / {2}".format(

j, self.evaluate(test\_data), n\_test))

else:

print("Epoch {0} complete".format(j))

accuracy = self.evaluate(training\_data)

training\_accuracy.append(accuracy)

accuracy = self.evaluate(test\_data)

evaluation\_accuracy.append(accuracy)

return evaluation\_accuracy, training\_accuracy

def update\_mini\_batch(self, mini\_batch, eta):

'''

更新网络的权重和偏差应用梯度下降使用反向传播到一个单一的小批。' ' mini\_batch ' '是一个元组列表' ' (x, y) ' '， ' ' eta ' '是学习率。

'''

nabla\_bias = [np.zeros(b.shape) for b in self.biases]#每层大小\*1 的多个零向量

nabla\_weight = [np.zeros(w.shape) for w in self.weights]#xxx 的多个零矩阵

for x, y in mini\_batch:

# print('x.shape',x.shape,'y.shape',y.shape)

delta\_nabla\_b, delta\_nabla\_w = self.backprop(x,y) # Cx对权重和偏置的全部导数

# print('delta\_nabla\_b[0].shape',delta\_nabla\_b[0].shape,'delta\_nabla\_w[0].shape',delta\_nabla\_w[0].shape)

nabla\_bias = [nb+dnb for nb, dnb in zip(nabla\_bias,delta\_nabla\_b)]

# print('nabla\_weight[0]:',nabla\_weight[0].shape)

# print('delta\_nabla\_w[0]:',delta\_nabla\_w[0].shape)

nabla\_weight = [nw + dnw for nw, dnw in zip(nabla\_weight,delta\_nabla\_w)]

#更新参数

self.weights = [w - (eta/len(mini\_batch))\* nw for w, nw in zip(self.weights, nabla\_weight)]

self.biases = [b - (eta/len(mini\_batch))\* nb for b, nb in zip(self.biases, nabla\_bias)]#除以len(mini\_batch)求所有导数的平均值

def backprop(self,x ,y):

'''

"返回tuple ' (nabla\_b, nabla\_w) ' '表示损失函数C\_x的梯度。“nabla\_b”和“nabla\_w”是逐层的numpy array列表，类似于“self.biases and self.weights’

'''

nabla\_bias = [np.zeros(b.shape) for b in self.biases]

nabla\_weight = [np.zeros(w.shape) for w in self.weights]

activation = x

activations = [x] #list to store all activations, layer by layer

zs = [] #list to store all the z vectors, layer by layer

for b, w in zip(self.biases,self.weights):

z = np.dot(w, activation) + b

zs.append(z)

#print('z.shape',z.shape)

activation = self.sigmoid(z)

#print('activation.shape',activation.shape)

activations.append(activation)

#print('activations',activations)

# 向后传播轨迹

#print('activations[-1]',activations)

delta = self.cost\_derivative(activations[-1],y) \* self.sigmoid\_prime(zs[-1])

# print('delta:',delta.shape)

nabla\_bias[-1] = delta

nabla\_weight[-1] = np.dot(delta, activations[-2].transpose())

for l in range(2, self.num\_layers):

z = zs[-l]

sp = self.sigmoid\_prime(z)

# print(self.weights[-l+1].transpose().shape)

# print(delta.shape)

# print(sp.shape)

delta = np.dot(self.weights[-l+1].transpose(),delta) \* sp

nabla\_bias[-l] = delta

nabla\_weight[-l] = np.dot(delta, activations[-l-1].transpose())#+1改成-1

# print(self.weights[-l+1].transpose().shape)

# print(nabla\_bias[0].shape)

# print(nabla\_weight[0].shape)

return (nabla\_bias, nabla\_weight)

def evaluate(self, test\_data):

for (x,y) in test\_data:

# print('x.shape',x.shape)

# print('y.shape',y.shape)

break

test\_results = [(np.argmax(self.feedforward(x)),np.argmax(y)) for (x,y) in test\_data]#y改成np.argmax(y)

for (x,y) in test\_results:

# print('x.shape',x)

# print('y.shape',y)

break

return sum(int(x == y) for (x,y) in test\_results)

def cost\_derivative(self, output\_activations, y):

return(output\_activations - y)

def save(self, filename):

data = {"sizes": self.sizes,

"weights": [w.tolist() for w in self.weights],

"biases": [b.tolist() for b in self.biases]}

f = open(filename, "w")

json.dump(data, f) #写入json文件

f.close()

（我调了整整四天啊！！才第一次跑通 就是那天来问你数据格式对不上就已经第二天了 之后实在内心崩溃重新来 所以运行了三百多次清0了 又运行了两百多次才过）

training\_data 是一个(x, y) 元组的列表，表示训练输入和其对应的期望输出。变量epochs和mini\_batch\_size ：迭代期数量，和采样时的小批量数据的大小。eta 是学习速率，\_。如果给出了可选参数test\_data，那么程序会在每个训练器后评估网络，并打印出部分进展。这对于追踪进度很有用，但相当拖慢执行速度。

代码如下工作。在每个迭代期，它首先随机地将训练数据打乱，然后将它分成多个适当大小的小批量数据。这是一个简单的从训练数据的随机采样方法。然后对于每一mini\_batch应用一次梯度下降。这是通过代码self.update\_mini\_batch(mini\_batch, eta) 完成的，它仅仅使用mini\_batch 中的训练数据，根据单次梯度下降的迭代更新网络的权重和偏置。

大部分工作由这行代码完成：

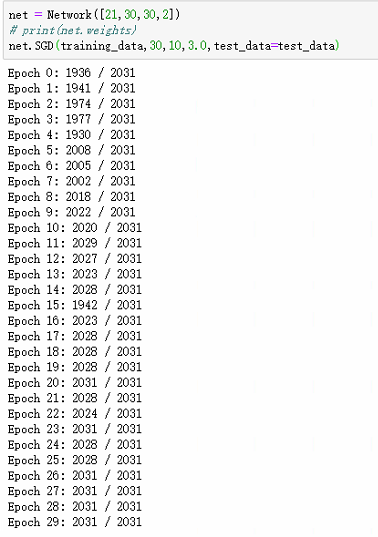
delta\_nabla\_b, delta\_nabla\_w = self.backprop(x, y)

这行调用了一个称为反向传播的算法，一种快速计算代价函数的梯度的方法。因此

update\_mini\_batch 的工作仅仅是对mini\_batch 中的每一个训练样本计算梯度，然后适当地更新self.weights 和self.biases。现在，就假设它按照我们要求的工作，返回与训练样本x 相关代价的适当梯度。

所有的繁重⼯作由self.SGD 和self.update\_mini\_batch 完成。self.backprop 方法利用一些额外的函数来帮助计算梯度，即sigmoid\_prime，它计算delta 函数的导数，以及self.cost\_derivative，这里我不会对它过多描述。因为你直接能够通过查看代码或⽂档注释来获得这些细节。

在导⼊如上所列的名为network 的Python 程序后做，

（这里因为debug多运行了几次，相当于机器作弊看了几次标准答案，所以效果出奇的好。。）

这是⼀个神经网络训练运行时的部分打印输出。打印内容显示了在每轮训练期后神经⽹络能正确识别蘑菇有无毒的数量。在仅仅一次迭代期后，达到了2031中选中的1936个。而且数目还在持续增长。

将这个合适的网络（[21,30,30,2]四层, epochs，batch\_size,eta:3.030,10,3.0）

net.save("model.json")

保存下来.

接下来我们需要对其进行调参，以使得模型最优化

首先进行隐藏层的对比

# 对比 隐藏层的对比

epochs=30

mini\_batch=10

eta=3.0

layers1 = [21,30,2]

layers2 = [21,50,2]

layers3 = [21,30,100,2]

layers4 = [21,50,100,2]

net1 = Network(layers1)

net2 = Network(layers2)

net3 = Network(layers3)

net4 = Network(layers4)

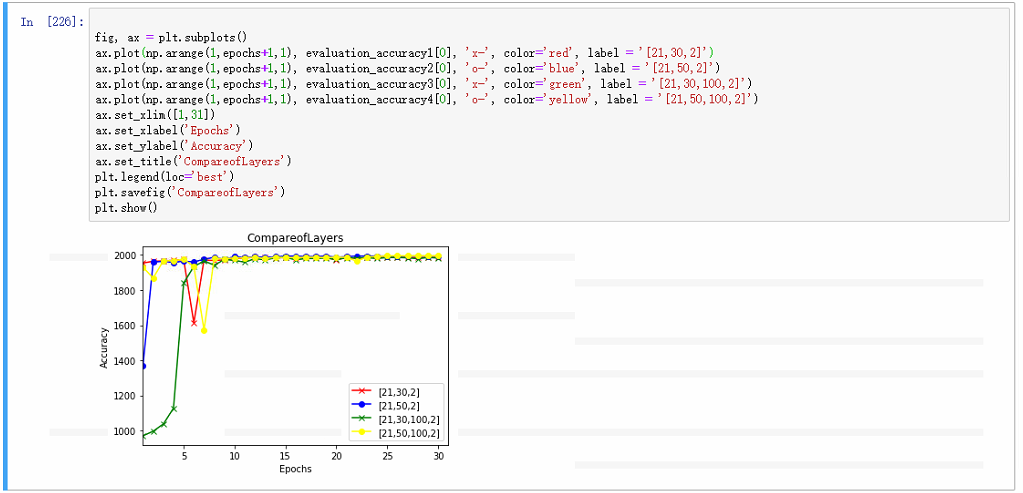
evaluation\_accuracy1 = net1.SGD(training\_data,epochs, mini\_batch, eta, test\_data=test\_data)

evaluation\_accuracy2 = net2.SGD(training\_data,epochs, mini\_batch, eta, test\_data=test\_data)

evaluation\_accuracy3 = net3.SGD(training\_data,epochs, mini\_batch, eta, test\_data=test\_data)

evaluation\_accuracy4 = net4.SGD(training\_data,epochs, mini\_batch, eta, test\_data=test\_data)

并利用matplotlib进行绘图观察四种方案的好坏



可见使用[21,50,100,2]四个layer是最好的，达到了1996 / 2031。

Epoch 0: 1954 / 2031

Epoch 1: 1964 / 2031

Epoch 2: 1966 / 2031

Epoch 3: 1970 / 2031

Epoch 4: 1974 / 2031

Epoch 5: 1614 / 2031

Epoch 6: 1972 / 2031

Epoch 7: 1968 / 2031

Epoch 8: 1973 / 2031

Epoch 9: 1980 / 2031

Epoch 10: 1983 / 2031

Epoch 11: 1986 / 2031

Epoch 12: 1986 / 2031

Epoch 13: 1984 / 2031

Epoch 14: 1986 / 2031

Epoch 15: 1984 / 2031

Epoch 16: 1986 / 2031

Epoch 17: 1986 / 2031

Epoch 18: 1985 / 2031

Epoch 19: 1973 / 2031

Epoch 20: 1986 / 2031

Epoch 21: 1981 / 2031

Epoch 22: 1986 / 2031

Epoch 23: 1986 / 2031

Epoch 24: 1986 / 2031

Epoch 25: 1986 / 2031

Epoch 26: 1986 / 2031

Epoch 27: 1985 / 2031

Epoch 28: 1986 / 2031

Epoch 29: 1986 / 2031

Epoch 0: 1370 / 2031

Epoch 1: 1959 / 2031

Epoch 2: 1964 / 2031

Epoch 3: 1957 / 2031

Epoch 4: 1965 / 2031

Epoch 5: 1960 / 2031

Epoch 6: 1976 / 2031

Epoch 7: 1987 / 2031

Epoch 8: 1975 / 2031

Epoch 9: 1991 / 2031

Epoch 10: 1985 / 2031

Epoch 11: 1991 / 2031

Epoch 12: 1986 / 2031

Epoch 13: 1990 / 2031

Epoch 14: 1993 / 2031

Epoch 15: 1994 / 2031

Epoch 16: 1991 / 2031

Epoch 17: 1991 / 2031

Epoch 18: 1994 / 2031

Epoch 19: 1987 / 2031

Epoch 20: 1991 / 2031

Epoch 21: 1994 / 2031

Epoch 22: 1994 / 2031

Epoch 23: 1994 / 2031

Epoch 24: 1994 / 2031

Epoch 25: 1991 / 2031

Epoch 26: 1994 / 2031

Epoch 27: 1994 / 2031

Epoch 28: 1994 / 2031

Epoch 29: 1994 / 2031

Epoch 0: 970 / 2031

Epoch 1: 996 / 2031

Epoch 2: 1037 / 2031

Epoch 3: 1127 / 2031

Epoch 4: 1841 / 2031

Epoch 5: 1934 / 2031

Epoch 6: 1966 / 2031

Epoch 7: 1942 / 2031

Epoch 8: 1974 / 2031

Epoch 9: 1969 / 2031

Epoch 10: 1959 / 2031

Epoch 11: 1979 / 2031

Epoch 12: 1973 / 2031

Epoch 13: 1983 / 2031

Epoch 14: 1986 / 2031

Epoch 15: 1973 / 2031

Epoch 16: 1980 / 2031

Epoch 17: 1980 / 2031

Epoch 18: 1983 / 2031

Epoch 19: 1974 / 2031

Epoch 20: 1984 / 2031

Epoch 21: 1983 / 2031

Epoch 22: 1985 / 2031

Epoch 23: 1980 / 2031

Epoch 24: 1986 / 2031

Epoch 25: 1986 / 2031

Epoch 26: 1984 / 2031

Epoch 27: 1979 / 2031

Epoch 28: 1986 / 2031

Epoch 29: 1983 / 2031

Epoch 0: 1930 / 2031

Epoch 1: 1869 / 2031

Epoch 2: 1966 / 2031

Epoch 3: 1968 / 2031

Epoch 4: 1976 / 2031

Epoch 5: 1934 / 2031

Epoch 6: 1573 / 2031

Epoch 7: 1980 / 2031

Epoch 8: 1979 / 2031

Epoch 9: 1983 / 2031

Epoch 10: 1984 / 2031

Epoch 11: 1987 / 2031

Epoch 12: 1980 / 2031

Epoch 13: 1987 / 2031

Epoch 14: 1985 / 2031

Epoch 15: 1985 / 2031

Epoch 16: 1987 / 2031

Epoch 17: 1987 / 2031

Epoch 18: 1985 / 2031

Epoch 19: 1988 / 2031

Epoch 20: 1986 / 2031

Epoch 21: 1965 / 2031

Epoch 22: 1988 / 2031

Epoch 23: 1992 / 2031

Epoch 24: 1995 / 2031

Epoch 25: 1996 / 2031

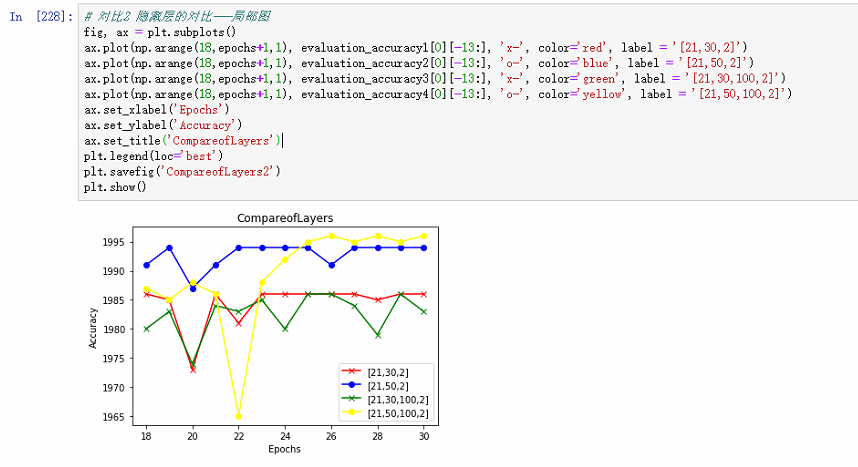
Epoch 26: 1995 / 2031

Epoch 27: 1996 / 2031

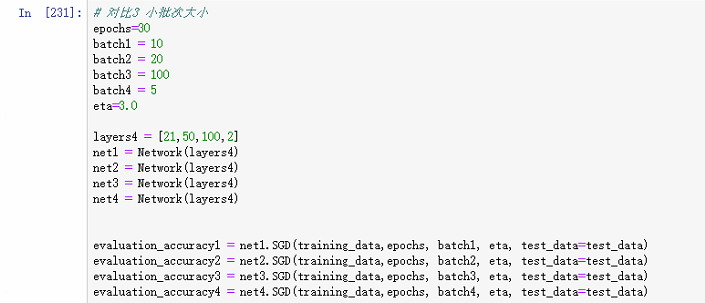
Epoch 28: 1995 / 2031

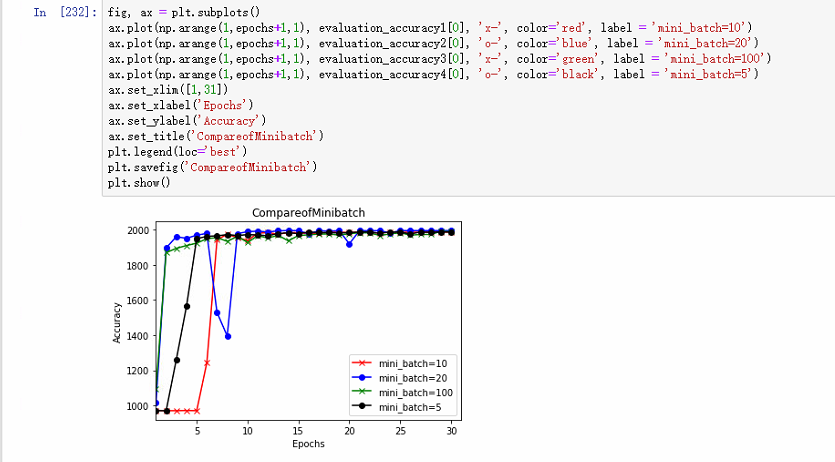
Epoch 29: 1996 / 2031

之后对比一下最后几层，可以明显看到最后[21,50,100,2]更胜一筹，故之后用这个继续优化。



接着优化小批次大小：





可以看到小批次大小为20的时候最优，达到了1994/2031(实际上大小为30时是1996/2031)

Epoch 0: 972 / 2031

Epoch 1: 970 / 2031

Epoch 2: 970 / 2031

Epoch 3: 971 / 2031

Epoch 4: 970 / 2031

Epoch 5: 1243 / 2031

Epoch 6: 1942 / 2031

Epoch 7: 1973 / 2031

Epoch 8: 1954 / 2031

Epoch 9: 1938 / 2031

Epoch 10: 1979 / 2031

Epoch 11: 1983 / 2031

Epoch 12: 1973 / 2031

Epoch 13: 1983 / 2031

Epoch 14: 1982 / 2031

Epoch 15: 1978 / 2031

Epoch 16: 1979 / 2031

Epoch 17: 1984 / 2031

Epoch 18: 1986 / 2031

Epoch 19: 1983 / 2031

Epoch 20: 1986 / 2031

Epoch 21: 1983 / 2031

Epoch 22: 1980 / 2031

Epoch 23: 1986 / 2031

Epoch 24: 1983 / 2031

Epoch 25: 1986 / 2031

Epoch 26: 1983 / 2031

Epoch 27: 1986 / 2031

Epoch 28: 1986 / 2031

Epoch 29: 1986 / 2031

Epoch 0: 1017 / 2031

Epoch 1: 1896 / 2031

Epoch 2: 1959 / 2031

Epoch 3: 1950 / 2031

Epoch 4: 1968 / 2031

Epoch 5: 1978 / 2031

Epoch 6: 1528 / 2031

Epoch 7: 1395 / 2031

Epoch 8: 1977 / 2031

Epoch 9: 1988 / 2031

Epoch 10: 1991 / 2031

Epoch 11: 1986 / 2031

Epoch 12: 1993 / 2031

Epoch 13: 1994 / 2031

Epoch 14: 1994 / 2031

Epoch 15: 1975 / 2031

Epoch 16: 1994 / 2031

Epoch 17: 1991 / 2031

Epoch 18: 1994 / 2031

Epoch 19: 1919 / 2031

Epoch 20: 1994 / 2031

Epoch 21: 1993 / 2031

Epoch 22: 1994 / 2031

Epoch 23: 1981 / 2031

Epoch 24: 1994 / 2031

Epoch 25: 1994 / 2031

Epoch 26: 1994 / 2031

Epoch 27: 1994 / 2031

Epoch 28: 1994 / 2031

Epoch 29: 1994 / 2031

Epoch 0: 1094 / 2031

Epoch 1: 1869 / 2031

Epoch 2: 1893 / 2031

Epoch 3: 1909 / 2031

Epoch 4: 1923 / 2031

Epoch 5: 1948 / 2031

Epoch 6: 1954 / 2031

Epoch 7: 1933 / 2031

Epoch 8: 1961 / 2031

Epoch 9: 1927 / 2031

Epoch 10: 1965 / 2031

Epoch 11: 1952 / 2031

Epoch 12: 1967 / 2031

Epoch 13: 1936 / 2031

Epoch 14: 1966 / 2031

Epoch 15: 1970 / 2031

Epoch 16: 1974 / 2031

Epoch 17: 1975 / 2031

Epoch 18: 1968 / 2031

Epoch 19: 1977 / 2031

Epoch 20: 1982 / 2031

Epoch 21: 1981 / 2031

Epoch 22: 1966 / 2031

Epoch 23: 1975 / 2031

Epoch 24: 1982 / 2031

Epoch 25: 1968 / 2031

Epoch 26: 1972 / 2031

Epoch 27: 1972 / 2031

Epoch 28: 1990 / 2031

Epoch 29: 1988 / 2031

Epoch 0: 970 / 2031

Epoch 1: 970 / 2031

Epoch 2: 1261 / 2031

Epoch 3: 1564 / 2031

Epoch 4: 1948 / 2031

Epoch 5: 1961 / 2031

Epoch 6: 1964 / 2031

Epoch 7: 1971 / 2031

Epoch 8: 1966 / 2031

Epoch 9: 1971 / 2031

Epoch 10: 1969 / 2031

Epoch 11: 1965 / 2031

Epoch 12: 1977 / 2031

Epoch 13: 1982 / 2031

Epoch 14: 1977 / 2031

Epoch 15: 1986 / 2031

Epoch 16: 1983 / 2031

Epoch 17: 1983 / 2031

Epoch 18: 1980 / 2031

Epoch 19: 1983 / 2031

Epoch 20: 1986 / 2031

Epoch 21: 1986 / 2031

Epoch 22: 1979 / 2031

Epoch 23: 1986 / 2031

Epoch 24: 1986 / 2031

Epoch 25: 1975 / 2031

Epoch 26: 1986 / 2031

Epoch 27: 1984 / 2031

Epoch 28: 1986 / 2031

Epoch 29: 1986 / 2031

接着取小批量为20优化学习率：

# 对比四 学习率

batch=20

epochs=30

eta1 = 0.05

eta2 = 0.5

eta3 = 10

eta4 = 50

layers4 = [21,50,100,2]

net1 = Network(layers4)

net2 = Network(layers4)

net3 = Network(layers4)

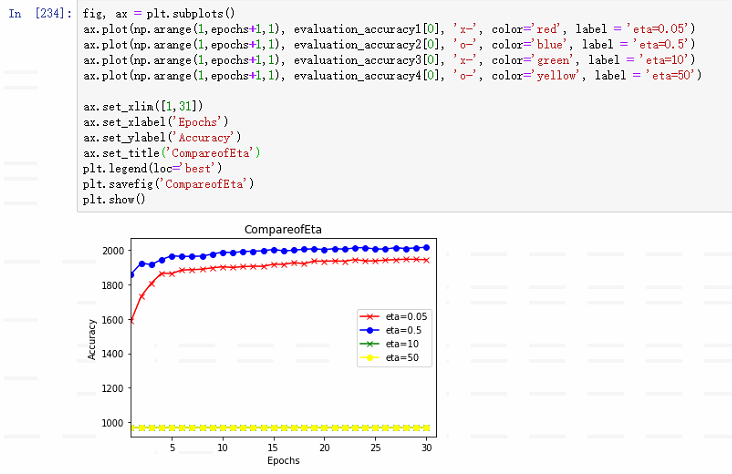
net4 = Network(layers4)

evaluation\_accuracy1 = net1.SGD(training\_data,epochs, batch, eta1, test\_data=test\_data)

evaluation\_accuracy2 = net2.SGD(training\_data,epochs, batch, eta2, test\_data=test\_data)

evaluation\_accuracy3 = net3.SGD(training\_data,epochs, batch, eta3, test\_data=test\_data)

evaluation\_accuracy4 = net4.SGD(training\_data,epochs, batch, eta4, test\_data=test\_data)



可以看到学习率为0.5时最优，50时直接走远了，0.05则太慢了，过犹不及。

Epoch 0: 1587 / 2031

Epoch 1: 1732 / 2031

Epoch 2: 1808 / 2031

Epoch 3: 1866 / 2031

Epoch 4: 1864 / 2031

Epoch 5: 1884 / 2031

Epoch 6: 1886 / 2031

Epoch 7: 1889 / 2031

Epoch 8: 1897 / 2031

Epoch 9: 1903 / 2031

Epoch 10: 1899 / 2031

Epoch 11: 1904 / 2031

Epoch 12: 1906 / 2031

Epoch 13: 1906 / 2031

Epoch 14: 1918 / 2031

Epoch 15: 1917 / 2031

Epoch 16: 1927 / 2031

Epoch 17: 1921 / 2031

Epoch 18: 1936 / 2031

Epoch 19: 1935 / 2031

Epoch 20: 1937 / 2031

Epoch 21: 1935 / 2031

Epoch 22: 1945 / 2031

Epoch 23: 1937 / 2031

Epoch 24: 1937 / 2031

Epoch 25: 1942 / 2031

Epoch 26: 1944 / 2031

Epoch 27: 1947 / 2031

Epoch 28: 1946 / 2031

Epoch 29: 1944 / 2031

Epoch 0: 1860 / 2031

Epoch 1: 1924 / 2031

Epoch 2: 1916 / 2031

Epoch 3: 1945 / 2031

Epoch 4: 1967 / 2031

Epoch 5: 1963 / 2031

Epoch 6: 1964 / 2031

Epoch 7: 1965 / 2031

Epoch 8: 1977 / 2031

Epoch 9: 1988 / 2031

Epoch 10: 1985 / 2031

Epoch 11: 1992 / 2031

Epoch 12: 1993 / 2031

Epoch 13: 1996 / 2031

Epoch 14: 2004 / 2031

Epoch 15: 1995 / 2031

Epoch 16: 1999 / 2031

Epoch 17: 2005 / 2031

Epoch 18: 2007 / 2031

Epoch 19: 2003 / 2031

Epoch 20: 2008 / 2031

Epoch 21: 2005 / 2031

Epoch 22: 2013 / 2031

Epoch 23: 2014 / 2031

Epoch 24: 2006 / 2031

Epoch 25: 2006 / 2031

Epoch 26: 2014 / 2031

Epoch 27: 2009 / 2031

Epoch 28: 2013 / 2031

Epoch 29: 2016 / 2031

Epoch 0: 970 / 2031

Epoch 1: 970 / 2031

Epoch 2: 970 / 2031

Epoch 3: 970 / 2031

Epoch 4: 970 / 2031

Epoch 5: 970 / 2031

Epoch 6: 970 / 2031

Epoch 7: 970 / 2031

Epoch 8: 970 / 2031

Epoch 9: 970 / 2031

Epoch 10: 970 / 2031

Epoch 11: 970 / 2031

Epoch 12: 970 / 2031

Epoch 13: 970 / 2031

Epoch 14: 970 / 2031

Epoch 15: 970 / 2031

Epoch 16: 970 / 2031

Epoch 17: 970 / 2031

Epoch 18: 970 / 2031

Epoch 19: 970 / 2031

Epoch 20: 970 / 2031

Epoch 21: 970 / 2031

Epoch 22: 970 / 2031

Epoch 23: 970 / 2031

Epoch 24: 970 / 2031

Epoch 25: 970 / 2031

Epoch 26: 970 / 2031

Epoch 27: 970 / 2031

Epoch 28: 970 / 2031

Epoch 29: 970 / 2031

Epoch 0: 970 / 2031

Epoch 1: 970 / 2031

Epoch 2: 970 / 2031

Epoch 3: 970 / 2031

Epoch 4: 970 / 2031

Epoch 5: 970 / 2031

Epoch 6: 970 / 2031

Epoch 7: 970 / 2031

Epoch 8: 970 / 2031

Epoch 9: 970 / 2031

Epoch 10: 970 / 2031

Epoch 11: 970 / 2031

Epoch 12: 970 / 2031

Epoch 13: 970 / 2031

Epoch 14: 970 / 2031

Epoch 15: 970 / 2031

Epoch 16: 970 / 2031

Epoch 17: 970 / 2031

Epoch 18: 970 / 2031

Epoch 19: 970 / 2031

Epoch 20: 970 / 2031

Epoch 21: 970 / 2031

Epoch 22: 970 / 2031

Epoch 23: 970 / 2031

Epoch 24: 970 / 2031

Epoch 25: 970 / 2031

Epoch 26: 970 / 2031

Epoch 27: 970 / 2031

Epoch 28: 970 / 2031

Epoch 29: 970 / 2031

最后优化到了2016/2031，成功率达到了99.3%，已经非常不错了。

五**总结**

我整整调了四天才第一次跑成功，真要了命了。

还天天建模，整的我最近多病缠身。代码硬刚算是刚完整了，还是很自豪的。

通过使用python，手搭神经网络，不断更改参数，也可将正确率达到99%以上。