实现BP神经网络

在这个练习中，你将实现一个反向传播（BP）两层神经网络，以对比利时的交通标志数据集进行分类。

**输入：**

本次练习中的输入来自比利时的交通标志数据集。因为数据量较大，所以请自行从网址进行下载：<http://btsd.ethz.ch/shareddata/>。这个网页中有很多数据集，本次实验使用的是：BelgiumTSC\_Training (171.3MBytes) <http://btsd.ethz.ch/shareddata/BelgiumTSC/BelgiumTSC_Training.zip>

BelgiumTSC\_Testing (76.5MBytes)

<http://btsd.ethz.ch/shareddata/BelgiumTSC/BelgiumTSC_Testing.zip>

在这两个数据包中，都包含62个子目录，从00000到00061，也即该数据集总共有62中图片。其中训练集中共有4575张图片，测试集中共有2534张图片。目录名就是分类标签，目录下的图片就是该分类的样本。

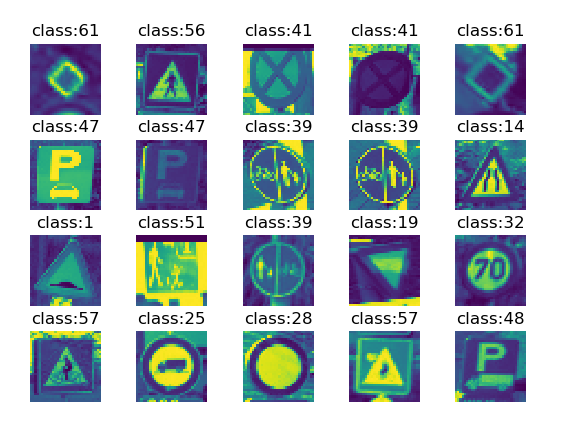
数据中图片的格式是.ppm，这个格式现在不常见，一般不能直接打开。但是Scikit Image支持这种格式，可以直接使用imread读取图片。

以下是部分图片。



在数据集中，图片本身的大小不同，譬如有94\*105\*3、128\*139\*3等。其中，\*3是因为图片本身是彩色的，有RGB三个色彩通道。这样的数据格式明显不能直接传递给神经网络，所以我们首先要对这些图片进行处理。

为了降低作业难度，提供了一份参考的图片处理，处理之后图片的数据统一为32\*32，以下是处理之后的效果。



**任务：**

手写实现反向传播难度较大，因此，提供部分关键代码。

1. 实现BP\_classifier.py中的main函数。
2. 调整参数，测试对预测准确率的影响。

**提示：**

可以参考课程中所提供的反向传播代码。

**代码检查标准：**

1. 代码能否正确运行。要求没有语法和逻辑错误，同时便于他人理解。请写好相应的注释，便于老师理解。
2. 请尽可能提高预测正确率。

**import** numpy **as** np  
**import** random  
**from** get\_data **import** read\_data, to\_grayscale, normalise\_images  
**import** matplotlib.pyplot **as** plt  
  
  
**class** dense(object):  
 **def** \_\_init\_\_(self, x, neurons, activation):  
 *"""  
 全连接层的实现* **:param** *x: 当前层的输入* **:param** *neurons: 神经元数目* **:param** *activation: 神经元激活函数  
 """* self.x = x  
 self.activation = activation  
 *# 初始化bias，shape=(neurons, 1)* self.biases = np.random.randn(neurons, 1)  
 *# 初始化权重，shape=(x, neurons)* self.weights = np.random.randn(len(self.x), neurons)  
 self.a = np.zeros(neurons)  
 self.z = np.zeros(neurons)  
  
 **def** feedforward(self, x):  
 *"""  
 前向传播实现* **:param** *x: 当前层的输入* **:return***: 当前层的激活值  
 """* self.x = x  
 self.z = np.dot(self.weights.transpose(), self.x) + self.biases  
 self.a = self.activation.activate(self.z)  
 **return** self.a  
  
 **def** backprop(self):  
 *"""  
 计算梯度* **:return***:  
 """* grad = self.activation.prime(self.z)  
 **return** grad  
  
 **def** back(self, delta\_w, delta\_b):  
 *"""  
 更新weights和bias* **:param** *delta\_w: 权重变化大小矩阵* **:param** *delta\_b: 偏置变化大小矩阵* **:return***:  
 """* self.weights = self.weights - delta\_w  
 self.biases = self.biases - delta\_b  
  
  
**class** sigmoid(object):  
  
 @staticmethod  
 **def** activate(z):  
 *"""The sigmoid function."""* **return** 1.0 / (1.0 + np.exp(-z))  
  
 @staticmethod  
 **def** prime(z):  
 *"""sigmoid函数的微分."""* **return** sigmoid.activate(z) \* (1 - sigmoid.activate(z))  
  
  
**class** QuadraticCost(object):  
  
 @staticmethod  
 **def** fn(a, y):  
 *"""Return the cost associated with an output ``a`` and desired output  
 ``y``.  
  
 """* **return** 0.5 \* np.linalg.norm(a - y) \*\* 2  
  
 @staticmethod  
 **def** delta(a, y):  
 *"""Return the error delta from the output layer."""* **return** (a - y)  
  
  
**class** Network(object):  
 **def** \_\_init\_\_(self, sizes, cost=QuadraticCost):  
 *"""  
 神经网络结构定义* **:param** *sizes: 神经网络的大小；（784, 30, 10）即表示输入层是784个节点；隐藏层是30个节点；输出是10个节点* **:param** *cost: 代价函数  
 """* self.num\_layers = len(sizes)  
 self.cost = cost  
 self.sizes = sizes  
 self.x = np.zeros(self.sizes[0])  
 self.fc\_layers = []  
 x = self.x  
 *# 创建隐藏层和输出层* **for** neurons **in** self.sizes[1:]:  
 self.fc\_layers.append(dense(x, neurons, sigmoid))  
 x = self.fc\_layers[-1].a  
  
  
  
 **def** feedforward(self, a):  
 *"""  
 前向传播，计算网络的输出* **:param** *a: 网络输入* **:return***: 网络输出  
 """* x = a  
 **for** layer **in** self.fc\_layers:  
 x = layer.feedforward(x)  
 **return** self.fc\_layers[-1].a  
  
 **def** SGD(self, training\_data, epochs, mini\_batch\_size, eta,  
 evaluation\_data=**None**):  
 *"""  
 随机梯度下降* **:param** *training\_data: 训练集* **:param** *epochs: 训练周期* **:param** *mini\_batch\_size: 批次大小* **:param** *eta: 学习率参数 ，真实学习率lr=eta/len(mini\_batch\_size)* **:param** *evaluation\_data: 验证集* **:return***:  
 """* train\_accs = []  
 train\_costs = []  
  
 test\_accs = []  
 test\_costs = []  
 **if** evaluation\_data: n\_data = len(evaluation\_data)  
 n = len(training\_data)  
 **for** j **in** range(epochs):  
 *# 数据随机化* random.shuffle(training\_data)  
 mini\_batches = [  
 training\_data[k:k + mini\_batch\_size]  
 **for** k **in** range(0, n, mini\_batch\_size)]  
 *# 针对每一个批次进行梯度下降* **for** mini\_batch **in** mini\_batches:  
 self.update\_mini\_batch(  
 mini\_batch, eta)  
  
 train\_acc = self.evaluate(training\_data, **True**) / n  
 train\_accs.append(train\_acc)  
 train\_cost = self.total\_cost(training\_data)  
 train\_costs.append(train\_cost)  
 print(**"Epoch {} training, Acc: {}, cost: {}:"**.format(j, train\_acc, train\_cost))  
 *# 打印出正确分类的个数* **if** evaluation\_data:  
 test\_correct\_count = self.evaluate(evaluation\_data)  
 test\_acc = test\_correct\_count/n\_data  
 test\_cost = self.total\_cost(evaluation\_data, convert=**True**)  
 test\_accs.append(test\_acc)  
 test\_costs.append(test\_cost)  
 print(**"Epoch {} evaluate: {} / {}, Acc: {}, cost: {}"**.format(j, self.evaluate(evaluation\_data), n\_data, test\_acc, test\_cost))  
 **else**:  
 print(**"Epoch {} complete"**.format(j))  
 **return** train\_accs, train\_costs, test\_accs, test\_costs  
  
 **def** update\_mini\_batch(self, mini\_batch, eta):  
 *"""  
 梯度下降，向后传播更新参数* **:param** *mini\_batch: 批次数据* **:param** *eta: 学习率参数* **:return***:  
 """* lr = eta / len(mini\_batch)  
 **for** x, y **in** mini\_batch:  
 *# 前向传播计算每层的输出* self.feedforward(x)  
 *# 计算输出层的梯度* delta = self.cost.delta(self.fc\_layers[-1].a, y) \* self.fc\_layers[-1].backprop()  
 delta\_w = np.dot(self.fc\_layers[-2].a, delta.transpose())  
 self.fc\_layers[-1].back(lr \* delta\_w, eta / lr \* delta)  
 *# 当网络结构多一个隐藏层* **if** self.num\_layers > 3:  
 **for** layer **in** range(2, self.num\_layers - 1):  
 delta = np.dot(self.fc\_layers[-layer + 1].weights, delta) \* self.fc\_layers[  
 -layer].backprop()  
 delta\_w = np.dot(self.fc\_layers[-layer - 1].a, delta.transpose())  
 self.fc\_layers[-layer].back(lr \* delta\_w, lr \* delta)  
  
 *# 三层结构* **else**:  
 delta = np.dot(self.fc\_layers[-1].weights, delta) \* self.fc\_layers[  
 -2].backprop()  
 delta\_w = np.dot(x, delta.transpose())  
 self.fc\_layers[-2].back(lr \* delta\_w, lr \* delta)  
  
 **def** evaluate(self, test\_data, convert=**False**):  
 *"""统计出来正确分类的个数;argmax表示数组中最大的值的位置；因为最终output的结果是0,1向量，只有一个值为1，也就是判别的种类"""* **if not** convert:  
 test\_results = [(np.argmax(self.feedforward(x)), y)  
 **for** (x, y) **in** test\_data]  
 *# print(test\_results)* **else**:  
 test\_results = [(np.argmax(self.feedforward(x)), np.argmax(y))  
 **for** (x, y) **in** test\_data]  
 **return** sum(int(x == y) **for** (x, y) **in** test\_results)  
  
 **def** total\_cost(self, data, convert=**False**):  
 *"""  
 所有的损失值  
 """* cost = 0.0  
 **for** x, y **in** data:  
 a = self.feedforward(x)  
 **if** convert: y = vectorized\_result(y)  
 cost += self.cost.fn(a, y) / len(data)  
 **return** cost  
  
  
**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...9) into a corresponding desired output from the neural  
 network."""* e = np.zeros((62, 1))  
 e[j] = 1.0  
 **return** e  
  
  
**def** main():  
 train\_x, train\_y = read\_data(**'./data/BelgiumTSC\_Training.zip'**, **'Training/'**)  
  
  
  
  
  
  
  
   
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()