题目要求:

采用的数据集著名的“MNIST 数据集”完成一个神经网络的训练和测试，不允许使用 tensorflow 等框架。并用两种不同的 bp 模型做性能对比 （比如一个层数和神经元较少的简单模型和一个层数和神经元较多的复杂模型）。

问题分析：

MNIST 数据集是一系列的手写图片，它们是由 28*28 的像素点构成，所以这里在设计神经网络时的输入节点数就是 28*28=784 个节点，输出节点设计为 10 个每一个结点对应 0-9 的一个数字，第几个节点的输出值最大，就判定输出结果为节点对应序号。将神经网络包装为一个类，它的层数可自定义，每层的节点个数也可自定义。

问题解决：

首先合理的加载训练集的文件，将每个图片数据格式化为 784*1 的 numpy 数组，标签数据格式化为 10*1 的 numpy 数组：

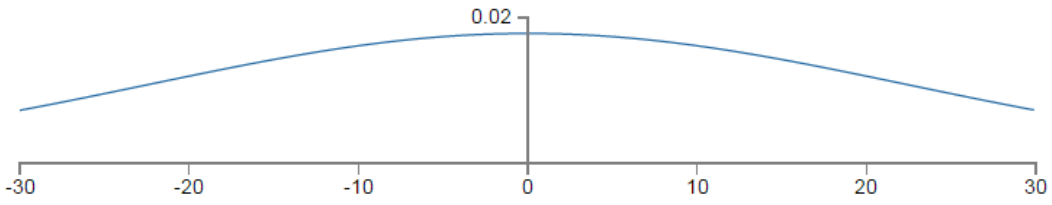
```
#数据加载函数，kind值标明了读取文件的类型
def load(path, kind='train'):
    labels_path = os.path.join(path,'%s-labels.idx1-ubyte'% kind)
    images_path = os.path.join(path,'%s-images.idx3-ubyte'% kind)
    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',lbpath.read(8))
        labels = numpy.fromfile(lbpath,dtype=numpy.uint8)
    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack('>IIII',imgpath.read(16))
        images = numpy.fromfile(imgpath,dtype=numpy.uint8).reshape(len(labels), 784)
    #读取到的labels为0-9的数字，需转化为十位的numpy数组
    ls=[]
    for i in range(labels.size):
        temp=numpy.zeros(10)
        temp[labels[i]]=1
        ls.append(temp.T)
    labels=numpy.array(ls)
    #由于源数据有些数据过大，会导致激活函数计算溢出，所以对数据集集体缩小，
    #由于图片数据每一位的值均为0-255之间，但统一除以255后发现当神经元个数达到一定数目或层数增加时还是会计算溢出，于是决定统一除以2550
    return (images/2550), labels
```

在加载 labels 数据时，需要将原本的 0-9 转化为十位的 numpy 数组，为 1 的那一位的下标就代表真实标签，其余位均为 0。在加载 images 数据时，由于原始数据中游的数据偏大，若不改变数据大小会导致计算激活函数时溢出，又因为图片数据每一位均在 0-255 之间，但统一除以 255 后发现当神经元个数达到一定数目或层数增加时还是会计算溢出，于是决定统一除以 2550。

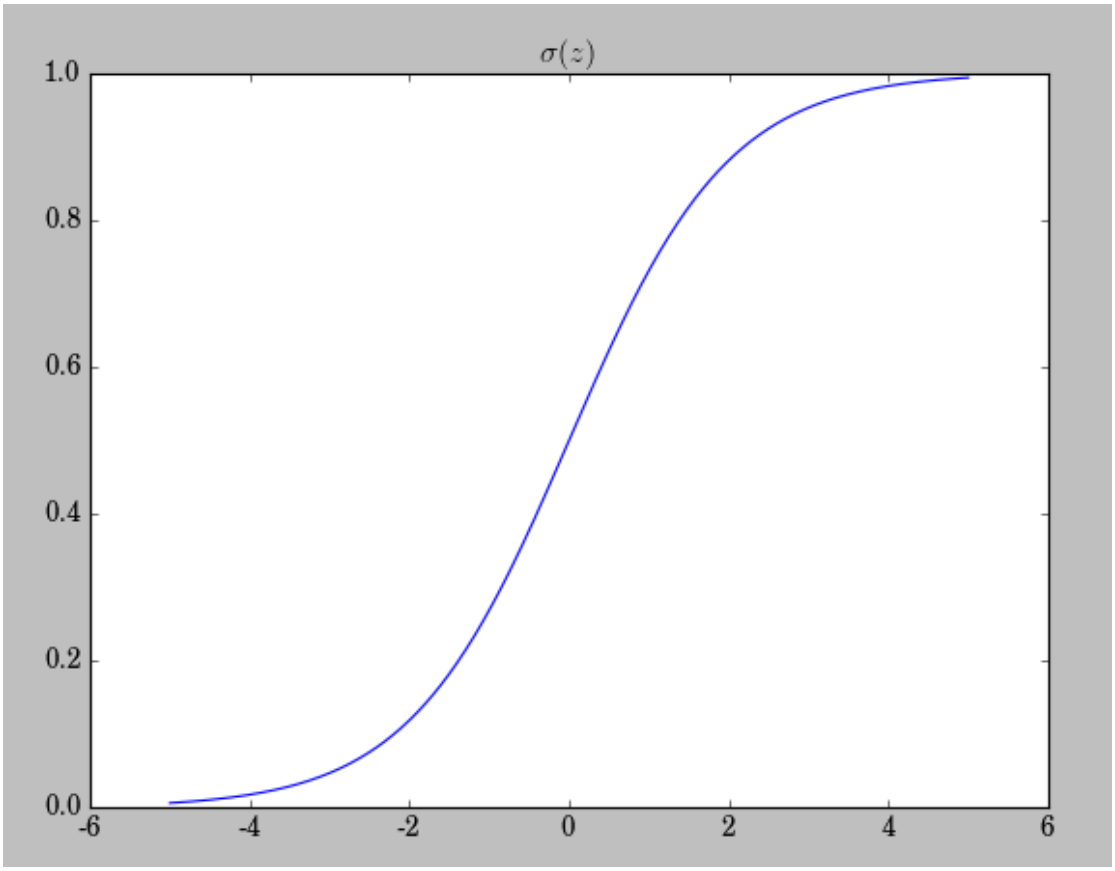
然后就可以开始编写神经网络类了。在构造函数中，我们定义 numNeuronLayers, learningrate, numNeurons_perLayer 来初始化网络层数，学习率和每层神经元个数，初始权重值为均值为 0，标准差为$1/\sqrt{n_i}$的正态分布，n_i为当前输出层的节点数，这里之所以采用这样的初始化方式，若直接采用标准正态分布的方式，则可能出现以下问题：

假设一个输入样本（特征向量 x，维度为 1000），一半为 1，一半为 0（这样的假设很特殊，但也很能说明问题），根据前向传递公式，z=500，z 即为输入层向中间隐层第一个神经元的输入。因为输入的一半为 0 的缘故，权重初始化为独立同分布的标准高斯随机变量，z 为 500 个标准正态随机变量的和，由独立随机变量和的方差等于方差的和可知，因此 z 的分布服从 0 均值，标准差为$\sqrt{500}\approx 22.4$，标准差从 1 升高到 22.4，由高斯密度函数可知，方

差越大，密度函数的分布越扁平，也即分布越均匀而不是集中在一段区域，其概率密度函数为（可见十分平坦）：

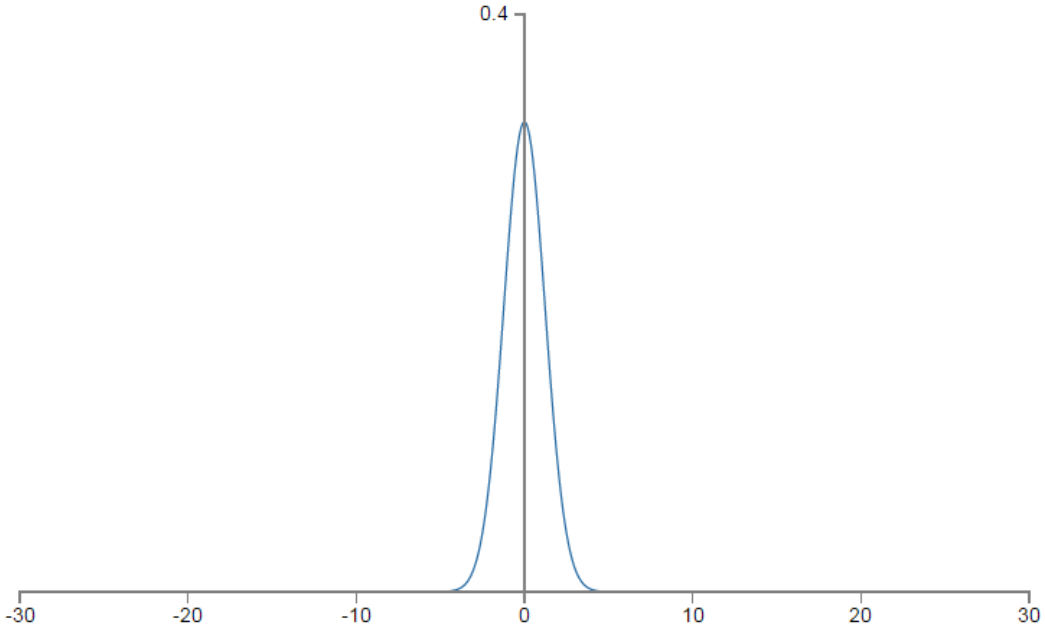


因为概率密度函数较为均匀，z 的值就不会像 N~(0,1)那样集中于均值附近（越远离均值中心，密度值会迅速衰减），而会以更大的概率取更大的值。这中初始化机制可能带来|z||z|取值很大，也即 z≫1 或者 z≪−1，相应的该神经元的 activation 值就会接近 0 或者 1，而我们知道，如下图示，激活函数越靠近 0 或者 1 其变化率越小，也即是达到一种饱和（saturate）状态。



也就意味着，在相同学习率的前提下，梯度变化更小，学习速度越慢，会导致模型收敛越慢。

而如果更改为均值为 0，标准差为$1/\sqrt{n_i}$的正态分布，在之前的例子中，概率密度函数的标准差就会变为$\sqrt{500/1000}\approx 0.7$，权值的分布就会变得陡峭：



这样模型的收敛速度也会加快。

还有就是激活函数的定义，这里激活函数采用“S”型函数。

以下是神经网络类构造函数的代码：

```
class neuralNetwork:
    def __init__(self,numNeuronLayers,numNeurons_perLayer,learningrate):
        self.numNeurons_perLayer=numNeurons_perLayer
        self.numNeuronLayers=numNeuronLayers
        self.learningrate = learningrate
        self.weight=[]
        for i in range(numNeuronLayers):
            self.weight.append(numpy.random.normal(0.0, pow(self.numNeurons_perLayer[i+1],-0.5),
                               (self.numNeurons_perLayer[i+1],self.numNeurons_perLayer[i]) )  )
        self.activation_function = lambda x: 1.0/(1.0+numpy.exp(-x))
```

接下来定义训练函数 update：

首先是前向传播代码的编写：

```
    def update(self,inputnodes,targets):
        inputs = numpy.array(inputnodes,ndmin=2).T
        targets = numpy.array(targets,ndmin=2).T
        #前向传播
        #定义输出值列表（outputs[0]为输入值）
        self.outputs=[]
        self.outputs.append(inputs)
        #用激活函数对神经网络的每一层计算输出值，并保存到outputs列表中
        for i in range(self.numNeuronLayers):
            temp_inputs=numpy.dot(self.weight[i],inputs)
            temp_outputs=self.activation_function(temp_inputs)
            inputs=temp_outputs
            self.outputs.append(temp_outputs)
```

根据定义的神经网络层数，遍历每一层节点，根据激活函数和输入值算出输出值，并保存在 outputs 列表中（outputs[0]为 inputnodes）。

有了前向传播的输出值，我们就可以计算每层的误差：

对于输出层的误差，我们可以用 targets-outputs[-1]得到，也就是目标值减输出层的输出，对于隐藏层的误差，我们可以用当前层与下一层之间的权值矩阵乘下一层的误差矩阵得到：

```
#计算每层的训练误差
self.output_errors=[]
for i in range(self.numNeuronLayers):
    #输出层的误差=目标值-输出值
    if i == 0:
        self.output_errors.append(targets - self.outputs[-1])
    #隐藏层的误差=当前隐藏层与下一层之间的权值矩阵与下一层的误差矩阵的乘积
    else:
        self.output_errors.append(numpy.dot((self.weight[self.numNeuronLayers-i]).T,
                                            self.output_errors[i-1]))
```

有了每一层的误差值，我们就可以更新各层权值了：

```
#反向传播
for i in range(self.numNeuronLayers):
    #权值更新规则为之前权值+学习率*误差*第二层输出*（1-第二层输出）*第一层输出
    #f(x)*（1-f(x)）即为激活函数f(x)的导函数
    self.weight[self.numNeuronLayers-i-1] +=
                                    self.learningrate * numpy.dot((
                                        self.output_errors[i] * self.outputs[-1-i] * (1.0 - self.outputs[-1-i])),
                                            numpy.transpose(self.outputs[-1-i-1]))
```

更新规则为之前权值+学习率*误差*第二层输出*（1-第二层输出）*第一层输出，f(x)*（1-f(x)）即为激活函数 f(x)的导函数，更新过程从后向前进行。

最后是测试函数的编写，思路就是将测试用例作为输入，让模型走一遍前向传播过程得到输出，然后返回输出结果是否与测试用例标签一致：

```
def test(self,test_inputnodes,test_labels):
    inputs = numpy.array(test_inputnodes,ndmin=2).T
    #走一遍前向传播得到输出
    for i in range(self.numNeuronLayers):
        temp_inputs=numpy.dot(self.weight[i],inputs)
        temp_outputs=self.activation_function(temp_inputs)
        inputs=temp_outputs
    #返回模型输出是否与标签一致
    return list(inputs).index(max(list(inputs)))==list(test_labels).index(1)
```

在 main 函数中载入训练集和测试集，神经网络定义为三层，输入层节点为 784，隐藏层节点为 30，隐藏层的节点个数选择，我参考了别人的经验公式 m=$\sqrt{n+l}+\alpha$，其中 n 为输入层节点数，l 为输出层节点数，α一般取 1-10，同时，我将回合数设为 30，因为在学习过程中，反向传播时一开始梯度下降应较大，让训练速度加快，随着训练的进行，训练结果越来越接近正确值，所以应逐步减少学习率，从而防止随着回合数增加，模型跳过最优值，造成模型不收敛，甚至发散的结果，但学习率太低又会造成训练速度过慢的缺陷，如果是深入研究的话，是可以根据损失函数的变化找到最优学习率的，这里由于是刚刚入门机器学习，所以简单地随着训练轮数增加降低学习率，当回合进行到 30 时，学习率减少一个数量级：

```python
if __name__ == '__main__':
    learning_rate = 0.1
    images_data,labels=load("C:\\Users\\Anonymous\\Documents\\机器学习\\作业四赵虎201600301325", kind='train')
    test_images_data,test_labels=load("C:\\Users\\Anonymous\\Documents\\机器学习\\作业四赵虎201600301325", kind='t10k')
    ls=[784,30,10]
    n=neuralNetwork(2,ls,0.3)
    for i in range(30):
        if i/6==0:
            n.learningrate=n.learningrate/2
        for i in range(len(images_data)):
            n.update(images_data[i],labels[i])
    count=0
    for i in range(len(test_images_data)):
        if n.test(test_images_data[i],test_labels[i]):
            count+=1
    print(count/10000)
```

最终得到的识别率结果为 96.42%。

```
0.9642
[Finished in 412.5s]
```

接下来我们进行调超参数的工作，首先考虑增加神经网络隐藏层的个数，将隐藏层设为两层，由于层数的增长让程序运行时间急剧增加，所以这里将训练回合将为 5 后做对比。第一层隐藏层设为 100 个节点，第二个隐藏层设为 20 个节点，学习率依然是 0.3。最终的识别结果为 96.15%。

```
0.9615
[Finished in 172.5s]
```

而同样经过 5 个回合训练的三层神经网络（隐藏层节点数为 30）的识别率为:

```
0.9515
[Finished in 59.3s]
```

可见增加神经网络层数可以增加识别率。

接着我们仍然使用三层的神经网络，但增加隐藏层节点数为 100，回合数为 5:

```
0.967
[Finished in 313.5s]
```

可见增加神经网络隐藏层节点数也可以增加识别率。

节点数增加为 500:

```
0.9636
[Finished in 1353.2s]
```

可见识别率反而出现了降低。

增加神经网络层数和神经元个数带来的影响是计算时间的增长，但是可能出现过拟合的情况，在时间的允许下可以一直增加隐藏层的节点个数（但是要小于输入层节点数），从而找到最优值。