BP 神经网络分类问题

一、实验目的

根据 BP 神经网络算法（AP-ANN）的算法思想和步骤，编写程序实现算法，理解 BP-ANN 算法的基本原理和基本的程序实现方法。以带类标的数据：titanic.dat 为文本文件（以@开头的行为注释行，剩下每一行为一个带类标的训练样例，由空格隔开的属性值构成，最后一个属性值为类标，取值为 1.0 或者-1.0，分别表示两个不同的类别）作为输入的数据集。

二、算法描述和分析

2.1 读入数据

使用 python 的 readlines()读入操作将每一行的数据（每个样例）读入到字符串中，然后使用 split(',')方法将数据通过逗号分隔开，存入到一个字典中。字典的结构为：diary{"attribute":list[3]","label":int}。attribute 表示是属性，而 label 则是类标号。每一行首先判断是否已@开头，以@开头的就忽略，否则就存入到字典中。最后组成一个由每一个乘客所对应的的字典所组成的一个列表 people 中。

因此，people 的结构为：

[diary{"attribute":list[3]","label":int}, diary{"attribute":list[3]","label":int}, …]

2.2 数据规范化

首先将读入的信息列表 people 中每一个乘客的 class，age，sex 属性抽提出来分别存入一个列表中，求出每一个列表的最大值和最小值，然后按照最大-最小规范化（映射到 0-1 之间）将每一个属性值都按照一下公式进行转换：

$$x = \frac{x - x_{\min}}{x_{max} - x_{\min}}$$

转换后的数据重新存入一个字典列表中。

除此之外，由于在本 BP 神经网络算法中选用的激活函数是 sigmoid 函数，其取值空间是[0,1]，这时分类的类标为 1.0 和-1.0 就不合适了，因此需要将类标为-1.0 的全部重新标注为 0.0，这样便于之后 BP 神经网络的学习过程。

2.3 数据集随机划分

利用随机数生成 0-len(people)范围内的 0.7* len(people)个随机数，并保证每一个随机数

之间没有重复，利用这些随机数作为索引将 people 中的字典划分到另一个列表中，作为训练数据集，而将剩下的作为测试数据集。

（0.7 的划分参数可调）

2.4 设定 BP 神经网络结构和权值矩阵结构

由于本项目限定三层**（包括输入层，不算入输入层则是两层神经网络）**，且隐藏层的神经元个数为 4，因此该 BP 神经网络为 3*4*1 的三层神经网络。其中，输入层 3 个神经元，输出层 1 个神经元。

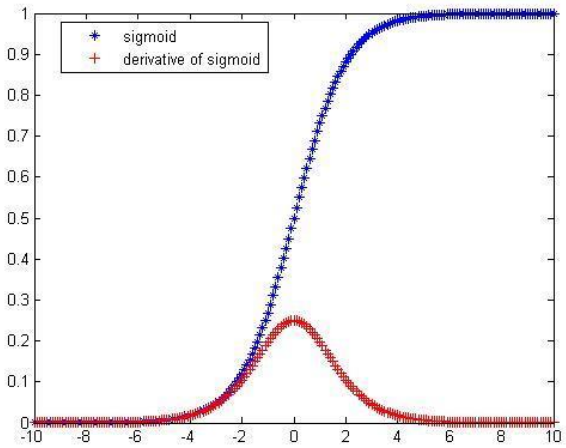
由于每个输入层神经元与隐藏层神经元之间都有一个权值，而每一个隐藏层神经元都与输出层神经元有权值，因此，一共有 3*4+4*1 个权值。因此，定义变量 weight 保存各层的权重参数，weight 的结构为：weight = [matrix([3*4]),matrix([4*1])]，即 weight 是一个包含两个矩阵的列表，第一个矩阵是 3*4 的矩阵，存储了每个输入层神经元到每个隐藏层神经元的权重；而第二个矩阵是 4*1 的矩阵，储存了每个隐藏层神经元到每个输出层神经元的权重。

另外，由于神经元存在激活阈值，因此需要定义一个 bias 列表，用来存储隐藏层神经元和输出层神经元的偏移量（输入层不需要 bias），因此定义 bias 结构为：bias = [matrix([1*4]),matrix([1*1])]。

2.5 定义 sigmoid 函数和 sigmoid_derivate 函数

本项目选择的激活函数是 sigmoid 函数，因为 sigmoid 函数可以引入非线性，从而提高神经网络的学习能力，并且 sigmoid 函数的输出范围为(0, 1)，所以数据在传递的过程中不容易发散。除此之外，sigmoid 函数的求导非常容易，因此很容易操作。

但是，sigmoid 函数也有缺点，那就是其在饱和状态时的梯度很小，在那时训练速度很低。



Sigmoid_derivate 函数就是 sigmoid 函数的导数，这里直接定义 sigmoid 的导数函数可以方便之后的后向传播算法的运算。

2.6 正向传播算法

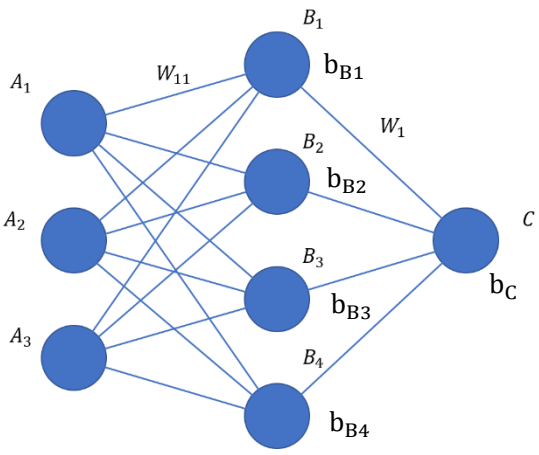
输入数据是一个三元组，即 matrix([1*3])，因此，在正向传播过程中，从输入层到隐藏层传递过程就是 matrix([1*3]) * matrix([3*4])的矩阵乘法运算，然后得到一个 matrix([1*4])的结果，这就是第一层未激活的输出。在经过 sigmoid 函数激活后，得到了激活后的第一层输出，也就是隐藏层的输入，sigmoid(matrix([1*4]))。

同理，隐藏层到输出层的正向传播过程就是 matrix([1*4]) * matrix([4*1])的过程，最后得到没有激活的输出，再经过 sigmoid 激活后得到最终的输出值。

7. 后向传播算法

BP 神经网络后向传播算法的思想来源于数学推导。我的所有算法思路来源于上课的 BP 神经网络内容以及一篇博客对 BP 神经网络的公式推导[1]。

在正向传播算法得到输出值之后，需要通过后向传播算法将输出值和类标之间的误差沿着 BP 神经网络逆向传播回去。



本项目的损失函数使用的是二次损失函数，也就是：

$$E = \frac{1}{2} \cdot (Label_c - out_c)^2$$

2.7 反向传播算法

这里简单描述一下后向传播算法的基本过程。首先，通过 E 我们可以求出正向传播过程中得到的输出值和类标的差别。但是我们关心的并不是这个差别 E，而是这个 E 对权值的梯度。

2.7.1 对 weight 进行反向传播

输出层：

为了方便介绍，输出层只以w_1为例。我们需要求 E 对w_1的偏导数。根据链式求导法则：

[1] https://www.sohu.com/a/235924191_633698

$$\frac{\partial \mathrm{E}}{\partial w_1}=\frac{\partial E}{\partial out_C}\cdot\frac{\partial out_C}{\partial in_c}\cdot\frac{\partial in_c}{\partial w_1}$$

其中，out_C是激活后得到的神经网络输出值，而in_c是未激活的输出值，w_1是隐藏层节点B_1到输出层节点 C 的权重。

$$\frac{\partial E}{\partial out_C}=\frac{\partial\left(\frac{1}{2}\cdot(Label_c-out_c)^2\right)}{\partial out_C}=-(Label_c-out_c)=out_c-Label_c$$

$$\frac{\partial out_C}{\partial in_c}=\frac{\mathrm{sigmoid}(in_c)}{\partial in_c}=sigmoid(in_c)\cdot[1-sigmoid(in_c)]$$

$$\frac{\partial in_c}{\partial w_1}=\frac{\mathrm{out_{b1}}*w_1+\cdots}{w_1}=\mathrm{out_{b1}}$$

因此，E 对w_1的偏导数为：

$$\frac{\partial \mathrm{E}}{\partial w_1}=(out_c-Label_c)\cdot sigmoid(in_c)\cdot[1-sigmoid(in_c)]\cdot\mathrm{out_{b1}}$$

隐藏层：

隐藏层的后向传播过程和输出层有点不一致，因为输出层使用了类标号 label 的值，而因为隐藏层没有类标号，因此公式不一样。

为了方便介绍，输出层只以w_{11}为例。我们需要求 E 对w_{11}的偏导数。根据链式求导法则：

$$\frac{\partial \mathrm{E}}{\partial \mathrm{w_{11}}}=\frac{\partial \mathrm{E}}{\partial \mathrm{out_{B1}}}\cdot\frac{\partial \mathrm{out_{B1}}}{\partial \mathrm{in_{B1}}}\cdot\frac{\partial \mathrm{in_{B1}}}{\partial \mathrm{w_{11}}}$$

其中，$\mathrm{out_{B1}}$是激活后得到的隐藏层节点B_1输出值，而$\mathrm{in_{B1}}$是未激活的输出值，w_{11}是输入层节点A_1到隐藏层节点B_1的权重。

$$\frac{\partial \mathrm{E}}{\partial \mathrm{out_{B1}}}=\frac{\partial E}{\partial out_C}\cdot\frac{\partial out_C}{\partial in_C}\cdot\frac{\partial in_C}{out_{B1}}=(out_c-Label)\cdot sigmoid(in_c)\cdot[1-sigmoid(in_c)]\cdot\mathrm{w}_1$$

$$\frac{\partial \mathrm{out_{B1}}}{\partial \mathrm{in_{B1}}}=\frac{\mathrm{sigmoid}(in_{B1})}{\partial in_{B1}}=sigmoid(in_{B1})\cdot[1-sigmoid(in_{B1})]$$

$$\frac{\partial \mathrm{in_{B1}}}{\partial \mathrm{w_{11}}}=\frac{\mathrm{input_{A1}}*w_{11}+\cdots}{w_{11}}=\mathrm{input_{A1}}$$

因此，E 对w_{11}的偏导数为：

$$\frac{\partial \mathrm{E}}{\partial \mathrm{w_{11}}}=(out_c-Label)\cdot sigmoid(in_c)\cdot[1-sigmoid(in_c)]\cdot\mathrm{w_1}\cdot(sigmoid(in_{B1})\cdot[1-sigmoid(in_{B1})])\cdot\mathrm{input_{A1}}$$

简化得到：

$$\frac{\partial \mathrm{E}}{\partial \mathrm{w_{11}}}=(out_c-Label)\cdot sigmoid\_dervatei(in_C)\cdot\mathrm{w_1}\cdot sigmoid\_derivate(in_{B1})\cdot\mathrm{input_{A1}}$$

由此就求得了损失函数对w_{11}的梯度，然后就可以更新w_{11}和w_1的值。其中，α是学习率 Learn Ratio。

$$w_1^+ = w_1 - \alpha \cdot \frac{\partial E}{\partial w_1}$$

$$w_{11}^+ = w_{11} - \alpha \cdot \frac{\partial E}{\partial w_{11}}$$

2.7.2 对 bias 进行反向传播

输出层：

我们需要求 E 对b_c的偏导数。根据链式求导法则：

$$\frac{\partial E}{\partial b_c} = \frac{\partial E}{\partial out_C} \cdot \frac{\partial out_C}{\partial in_c} \cdot \frac{\partial in_c}{\partial b_c}$$

其中，out_C是激活后得到的神经网络输出值，而in_c是未激活的输出值，b_1是输出层节点 C 的偏移量。

$$\frac{\partial E}{\partial out_C} = \frac{\partial \left(\frac{1}{2} \cdot (Label_c - out_c)^2\right)}{\partial out_C} = -(Label_c - out_c) = out_c - Label$$

$$\frac{\partial out_C}{\partial in_c} = \frac{\mathrm{sigmoid}(in_c)}{\partial in_c} = sigmoid(in_c) \cdot [1 - sigmoid(in_c)]$$

$$\frac{\partial in_c}{\partial b_c} = \frac{\mathrm{out_{b1}} * w_1 + \cdots}{b_c} = 1$$

因此，E 对b_c的偏导数为：

$$\frac{\partial E}{\partial b_c} = (out_c - Label) \cdot sigmoid(in_c) \cdot [1 - sigmoid(in_c)]$$

隐藏层：

为了方便介绍，输出层只以w_{11}为例。我们需要求 E 对w_{11}的偏导数。根据链式求导法则：

$$\frac{\partial E}{\partial b_{B1}} = \frac{\partial E}{\partial \mathrm{out_{B1}}} \cdot \frac{\partial \mathrm{out_{B1}}}{\partial \mathrm{in_{B1}}} \cdot \frac{\partial \mathrm{in_{B1}}}{\partial \mathrm{b_{B1}}}$$

其中，$\mathrm{out_{B1}}$是激活后得到的隐藏层节点B_1输出值，而$\mathrm{in_{B1}}$是未激活的输出值，$\mathrm{b_{B1}}$是隐藏层节点B_1的偏移量。

$$\frac{\partial E}{\partial \mathrm{out_{B1}}} = \frac{\partial E}{\partial out_C} \cdot \frac{\partial out_C}{\partial in_C} \cdot \frac{\partial in_C}{out_{B1}} = (out_c - Label) \cdot sigmoid(in_c) \cdot [1 - sigmoid(in_c)] \cdot w_1$$

$$\frac{\partial \mathrm{out_{B1}}}{\partial \mathrm{in_{B1}}} = \frac{\mathrm{sigmoid}(in_{B1})}{\partial in_{B1}} = sigmoid(in_{B1}) \cdot [1 - sigmoid(in_{B1})]$$

$$\frac{\partial \mathrm{in_{B1}}}{\partial \mathrm{b_{B1}}} = \frac{\mathrm{input_{A1}} * w_{11} + \cdots}{b_{B1}} = 1$$

因此，E 对$\mathrm{b_{B1}}$的偏导数为：

$$\frac{\partial E}{\partial b_{B1}} = (out_c - Label) \cdot sigmoid(in_c) \cdot [1 - sigmoid(in_c)] \cdot w_1 \cdot (sigmoid(in_{B1}) \cdot [1 - sigmoid(in_{B1})])$$

简化得到：

$$\frac{\partial \mathrm{E}}{\partial b_{B1}} = (out_c - Label) \cdot sigmoid\_dervatei(in_C) \cdot \mathrm{w}_1 \cdot sigmoid\_derivate(in_{B1})$$

从上述公式中，不难发现，权值 weight 和偏移量 bias 的梯度计算公式非常相似，差别只在于最后一个量的有无。即，bias 不需要最后乘上上一层的输出值（或输入层的输入值），而 weight 需要。

由此就求得了损失函数对w_{11}的梯度，然后就可以更新w_{11}和w_1的值。其中，α是学习率 Learn Ratio。

$$\mathrm{b}_c^+ = \mathrm{b}_c - \alpha \cdot \frac{\partial \mathrm{E}}{\partial b_c}$$

$$\mathrm{b}_{B1}^+ = \mathrm{b}_{B1} - \alpha \cdot \frac{\partial \mathrm{E}}{\partial b_{B1}}$$

2.7 梯度下降法和随机梯度下降法

在训练神经网络的过程中，有批量梯度下降法和随机梯度下降法和 mini-batch 随机梯度下降法。这两种方法都有各自的优点和缺点，下面分别介绍。

2.7.1 梯度下降法

梯度下降法，也叫批量梯度下降法（BGD）指的是每次的权值更新都使用了所有样本的信息。简单来说，就是，将每次权值的改变值看作每个样本对权值改变的总和的平均值。这样做的好处是每次权值更新都是全局最优解。但是因为每次权值的改变都需要计算整个样本的正向传播和后向传播，因此计算时间比较长，训练过程很慢。

2.7.2 随机梯度下降法

随机梯度下降法（SGD）指的是每次的权值更新不是使用了所有样本的信息，而是从训练集中随机抽取一个样本进行训练，得到权值的更新。因此每次权值更新的过程只使用了一个样本，导致随机梯度下降法的训练轮数会大大增加。

如果样本量很大的情况（例如几十万），那么可能只用其中几万条或者几千条的样本，就已经将 theta 迭代到最优解了，对比上面的批量梯度下降，迭代一次需要用到十几万训练样本，一次迭代不可能最优，如果迭代 10 次的话就需要遍历训练样本 10 次。但是，SGD 伴随的一个问题是噪音较 BGD 要多，使得 SGD 并不是每次迭代都向着整体最优化方向。

2.7.3 小样本随机梯度下降法

小样本随机梯度下降法（mini-batch SGD）是随机梯度下降法和梯度下降法的折中方法，

即每次权值的更新既不是依赖于整个训练集的信息，也不是只依赖于随机一个样本的信息，而是依赖于一个小样本的信息。由此看，小样本随机梯度下降法的效率会低于随机梯度下降法，但是高于梯度下降法。因此，其算法的训练过程比较快，也保证了最终参数训练的准确率

2.8 神经网络分类

根据 GD、SGD 或者 mini-batch SGD 进行神经网络训练，然后最终使用正向传播过程对测试集的样本进行分类，再比较分类结果和类标是否一致从而判断分类是否正确。由此，可以计算测试集的分类正确率。

由于 GD、SGD 和 minii-batchh SGD 在训练过程中的抽样方式不同，因此使用这三种方法训练神经网络使用的轮数也不同。这里，在只关注神经网络的准确度而不专注训练时间的前提下，针对三种方法选择相应能使得预测结果比较稳定的训练轮数，比较三种方法对分类准确率的影响。

三、程序实现技术技巧的介绍和分析

3.1 数据读取、储存和预处理

1. 数据的读取。因为数据集一行就是一个样例，因此可以每次读取一行，然后采用列表套字典套列表的方式对整个数据集的数据进行储存。
2. 数据规范化。使用了 python 特有的自定义匿名函数快速简洁地对数据进行了规范化。

```
def ReadplusStandard(filename):
    people,pclass,age,sex = readDat(filename)
    maxclass,minclass = max(pclass),min(pclass)
    maxage,minage = max(age),min(age)
    maxsex,minsex = max(sex),min(sex)
    standclass = lambda x: (x-minclass)/(maxclass-minclass)
    standage = lambda x: (x-minage)/(maxage-minage)
    standsex = lambda x: (x-minsex)/(maxsex-minsex)
    for person in people:
        a,b,c = person["attribute"]
        d = person["label"]
        person["attribute"] = [standclass(a),standage(b),standsex(c)]
        if d == -1.0:
            person["label"] = 0.0
    #print(people)
    return people
```

由于本项目使用的激活函数是 sigmoid 函数，因此激活后的输出在[0,1]范围内。原本的分类类标是-1.0 和+1.0，并不在 sigmoid 函数的值域内，因此需要将-1.0 的类标全部转化为

0.0，这样才能得到比较准确的训练结果。

本项目也曾尝试过使用-1.0 和+1.0 作为类标进行训练，但是结果显而易见，在同样的方法（SGD），同样的参数下，使用-1.0 和+1.0 作为类标的准确率大幅度下降。在 SGD 和 GD 方法下的测试如下（只测试一次作为表征）

	Label = [+1.0,-1.0]	Label = [+1.0,0.0]
LR=0.1,k=4,num=1000000,SGD	0.296969696969697	0.7848484848484848
LR=0.1,k=4,num=500,GD	0.3237518910741301	0.7776096822995462

3.2 数据划分

本项目使用随机数划分对样本集进行划分。利用随机数生成 **样本集大小*划分比例** 的个数来生成从[0,样本集大小-1]的随机数，将这些随机数存入一个列表中，重复生成的随机数将被去除。因此，这一步的结果是生成 **样本集大小*划分比例** 大小的各不相同的随机数。

以随机数列表中的随机数作为索引，将抽取的样本划分到训练集中，剩下的归为测试集。代码如下：

```
def DataDevide(people,ratio=0.7):
    randomintlist = []
    traindata = []
    testdata = []
    for i in range(int(floor(len(people)*ratio))):
        index = randint(0,len(people)-1)
        while index in randomintlist:
            index = randint(0,len(people)-1)
        randomintlist.append(index)
        traindata.append(people[index])
    for i in range(len(people)-1):
        if i not in randomintlist:
            testdata.append(people[i])
    return traindata,testdata
```

3.3 正向传播&反向传播

根据 2.6 和 2.7 部分的推导，可以完成对正向传播算法和反向传播算法的实现。这由函数 ForwardABackward 来完成。由于后续的训练方法 GD、SGD、mini-batch SGD 对权重的偏移量向量的更新方式不同，因此此函数的输出只是权重和偏移量的更新差，而不是更新后的值。

首先进行正向传播过程，利用矩阵的运算可以得出。由于 weight = [matrix([3*4]),matrix([4*1])]，而 bias = [matrix([1*4]),matrix([1*1])]，而输出的样本数据为 matrix[1*3]，因此 inputdata*(weight[0]) + bias[0]得到隐藏层的输入，经激活函数激活后隐藏层的输出，为 matrix[1*4]。而在隐藏层到输出层的过程和之前相同，最后得到激活后的 matrix[1*1]的输出值，将其转换为 float 形式得到神经网络的前向传播

过程的输出。按照 2.7 部分后向传播算法的公式，计算输出值和类标的误差，然后将误差后向传播到前面去，从而得到权值向量 weight 和偏移量向量 bias 的更新差。

代码如下：

```
def ForwardABackward(inputdata,label,weight,bias,k,LearnRatio):
    '''
    Forward propagation function and Back propagation function together
    input: inputdata, label, weight, bias, k, LeanRatio
    inputdata: the 3-element tuple of the attributes, (class, age, sex)
    label: the label of the train data, used to calculate the error
    weight: the weight list containing all the weight of the BP neural network
    bias: the bias list containing all the bias of the BP neural network
    k: the number of neuron in hidden layer
    LearnRatio: the learn ratio of the neural network

    The funtion is used to update the weight list and bias list
    '''
    #进行正向传播
    L1array = inputdata*(weight[0]) + bias[0] #output: 1*k
    actiL1array = []
    for i in range(k):
        actiL1array.append(sigmoid(L1array[0,i]))
    actiL1array = np.mat(actiL1array) #output: 1*4
    L2value = float(actiL1array*(weight[1]) + bias[1])  #由此得到了正向传播的输出值
    actiL2value = sigmoid(L2value) #float

    #进行反向传播
    ##计算误差
    change_b = [np.zeros(b.shape) for b in bias]
    change_w = [np.zeros(w.shape) for w in weight]
    L2error = actiL2value-label #float
    L2Delta = L2error*sigmod_derivate(L2value) #float

    ##计算第二层权值变化
    change_b[-1] = L2Delta
    # 乘于前一层的输出值
    change_w[-1] = (actiL1array.T)*L2Delta #output: 4*1

    ##计算第一层权值变化
    sigmoid_prime = np.multiply(actiL1array,(1-actiL1array))
    L1Delta = np.multiply(L2Delta*(weight[1].T),sigmoid_prime) #output: 1*4
    change_b[-2] = L1Delta
    change_w[-2] = (np.mat(inputdata).T)*L1Delta

    return change_w,change_b
```

3.4 BP 神经网络训练

实现前向传播算法和后向传播算法后，根据不同的方法（GD、SGD、mini-batch SGD）使用不同的训练方法。

对于 GD（梯度下降法），也叫批量梯度下降法，在每一轮的训练过程中，计算每一个训练样本对权值和偏移量的更新差，最后计算所有训练样本更新差的平均值，使用这个平均值更新权值向量和偏移量向量。这种方法的好处是每轮的更新都是全局最优的，但是坏处是计算量大幅提高。经过测试，训练次数达到 500 次后，GD 的分类效果比较好。

代码如下：

```
def GD(traindata,k,LearnRatio,num):
    '''
    Gradient Descent algorithm
    '''
    attrinum = len(traindata[0]["attribute"])
    bias = [np.mat(np.random.normal(size=(1,y)),dtype='float64') for y in [k,1] ]
    weight = [np.mat(np.random.normal(size=(x,y)),dtype='float64') for x,y in [(attrinum,k),(k,1)] ]
    #[ 0 0 0 0    0  ]
    #[ 0 0 0 0  ,  0  ]
    #[ 0 0 0 0    0  ]
    #[           0  ]   #每个矩阵最后一行是阈值节点的权值

    for j in range(num):
        #初始化累加权值和偏移量
        nabla_b = [np.zeros(b.shape) for b in bias]
        nabla_w = [np.zeros(w.shape) for w in weight]
        for i in traindata:
            inputdata,label = i["attribute"],i["label"]
            #进行正向传播和反向传播
            change_w,change_b = ForwardABackward(inputdata,label,weight,bias,k,LearnRatio)
            #累加权值偏差
            nabla_b[0] += change_b[-2]
            nabla_b[1] += change_b[-1]
            nabla_w[0] += change_w[-2]
            nabla_w[1] += change_w[-1]
        #更新权值
        datalens = len(traindata)
        weight[0] -= LearnRatio/datalens*nabla_w[0]
        weight[1] -= LearnRatio/datalens*nabla_w[1]
        bias[0] -= LearnRatio/datalens*nabla_b[0]
        bias[1] -= LearnRatio/datalens*nabla_b[1]

    return weight,bias
```

对于 SGD（随机梯度下降法），在每一轮的训练中，仅仅从训练集中随机抽取一个样本训练神经网络，计算出权值和偏移量的更新差，使用这个更新差去更新权值向量和偏移量向量。由于每次只是随机抽取一个样本，因此每轮训练不是全局最优，但是至少是局部最优。经过多轮迭代后权值也会稳定在一个值。这种方法的好处是计算量大大减少，特别是在样本集很大的时候，但是坏处是可能陷入局部最优也不是全局最优。由于 SGD 每次只抽取 1 个样本，因此需要更多的轮数才能够训练到比较好的神经网络。

代码如下：

```
def SGD(traindata,k,LearnRatio,num):
    '''
    Gradient Descent algorithm
    '''
    attrinum = len(traindata[0]["attribute"])
    bias = [np.mat(np.random.normal(size=(1,y)),dtype='float64') for y in [k,1] ]
    weight = [np.mat(np.random.normal(size=(x,y)),dtype='float64') for x,y in [(attrinum,k),(k,1)] ]
    #[ 0 0 0 0    0  ]
    #[ 0 0 0 0  ,  0  ]
    #[ 0 0 0 0    0  ]
    #[           0  ]   #每个矩阵最后一行是阈值节点的权值

    for j in range(num):
        #初始化累加权值和偏移量
        nabla_b = [np.zeros(b.shape) for b in bias]
        nabla_w = [np.zeros(w.shape) for w in weight]
        index = randint(0,len(traindata)-1)
        i = traindata[index]
        inputdata,label = i["attribute"],i["label"]
        #进行正向传播和反向传播
        change_w,change_b = ForwardABackward(inputdata,label,weight,bias,k,LearnRatio)

        #更新权值

        weight[1] -= LearnRatio * change_w[-1]
        bias[1] -= LearnRatio * change_b[-1]
        ##更新第一层权值
        weight[0] -= LearnRatio * change_w[-2]
        bias[0] -= LearnRatio * change_b[-2]

    return weight,bias
```

对于 mini-batch SGD（小样本集随机梯度下降法），这是 SGD 和 GD 中间的一种折中的方法，相对于 SGD 来说，每次不是只抽取一个样本，而是抽取一批样本，作为小样本集去进行训练。一般来说，mini-batch SGD 是比较推荐的一种做法。本项目中 mini-batch 的大小是 10。

代码如下：

```
def miniSGD(traindata,k,LearnRatio,num,minibatch):
    '''
```

```
    Stochastic Gradient Descent algorithm
    '''
    attrinum = len(traindata[0]["attribute"])
    bias = [np.mat(np.random.normal(size=(1,y)),dtype='float64') for y in [k,1] ]
    weight = [np.mat(np.random.normal(size=(x,y)),dtype='float64') for x,y in [(attrinum,k),(k,1)] ]
    #[ 0 0 0 0    0  ]
    #[ 0 0 0 0  ,  0  ]
    #[ 0 0 0 0    0  ]
    #[           0  ]   #每个矩阵最后一行是阈值节点的权值
    #print(weight)
    #print(bias)

    #利用小样本集进行训练
    for j in range(num):
        nabla_b = [np.zeros(b.shape) for b in bias]
        nabla_w = [np.zeros(w.shape) for w in weight]

        #划分小样本集
        randomlist = []
        for i in range(minibatch):
            index = randint(0,len(traindata)-1)
            while index in randomlist:
                index = randint(0,len(traindata)-1)
            randomlist.append(index)

        for i in randomlist:
            person = traindata[i]
            inputdata,label = person["attribute"],person["label"]
            #进行正向传播和反向传播
            change_w,change_b = ForwardABackward(inputdata,label,weight,bias,k,LearnRatio)
            #累加权值偏差
            #nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, change_b)]
            #nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, change_w)]
            nabla_b[0] += change_b[-2]
            nabla_b[1] += change_b[-1]
            nabla_w[0] += change_w[-2]
            nabla_w[1] += change_w[-1]
        #更新权值
        datalens = minibatch
        weight[0] -= LearnRatio/datalens*nabla_w[0]
        weight[1] -= LearnRatio/datalens*nabla_w[1]
        bias[0] -= LearnRatio/datalens*nabla_b[0]
        bias[1] -= LearnRatio/datalens*nabla_b[1]
        #weight = [w-(LearnRatio/datalens)*nw for w, nw in zip(weight, nabla_w)]
```

```
        #bias = [b-(LearnRatio/datalens)*nb for b, nb in zip(bias, nabla_b)]


    return weight,bias
```

最后，是调控使用哪种神经网络训练方法的函数，BPNetwork。

```
def BPNetwork(traindata,k,LearnRatio,method,num):
    '''
    BPNetwork
    input:
    traindata;
    intermediate layer unit number, k;
    LearnRatio;
    method: GD or SGD
    output: updated weight, updated bias
    '''
    #初始化各个权值
    ##节点个数
    attrinum = len(traindata[0]["attribute"])
    input_node = 3
    output_node = 1
    if method == "GD":
        weight,bias = GD(traindata,k,LearnRatio,num)
    elif method == "SGD":
        weight,bias = SGD(traindata,k,LearnRatio,num)
    elif method == "miniSGD":
        minibatch = 10
        weight,bias = miniSGD(traindata,k,LearnRatio,num,minibatch)
    else:
        print("method error, please check your method.\n")
    return weight,bias
```

3.5 测试神经网络分类准确度

定义正向传播算法 Forward 函数，将测试集的数据输入到训练好的神经网络中，得到预测的类。将预测的类和类标进行比对从而判断分类是否成功。

```
def TestAccurary(traindata,testdata,k,LearnRatio,method,num):
    weight,bias = BPNetwork(traindata,k,LearnRatio,method,num)
    #print(weight)
    #print(bias)
    Truenum = 0
    for i in testdata:
        output = 0
        inputdata,label = i["attribute"],i["label"]
        output = Forward(inputdata,weight,bias,k)
```

```
        #print(output,label)
        if output >= 0.5:
            output = 1.0
        else:
            output = 0.0
        if output == label:
            Truenum+=1
    return Truenum/len(testdata)
```

四、实验数据和实验方法

1. 数据分析

查看 titanic.dat 文件的数据，每一个元组都有四个属性。每一个属性都是离散的，Class，Age，Sex，Survived。进行简单的统计分析后发现，Class 有四个取值，分别是-1.87，-0.923，0.0214，0.965；Age 有两个取值，-0.228 和 4.38；Sex 有两个取值，-1.92 和 0.521，survived 也是两个取值，-1 和 1。

Attribute	Value	Value	Value	Value	Tips
Class	-1.87	-0.923	0.0214	0.965	
Age	-0.228	4.38			老/少？
Sex	-1.92	0.521			男/女
Survived	-1	1			死/活

五、实验结果、结果分析和实验结论

1. BPANN 算法分类：

经过多次尝试，寻找合适的训练轮数：

GD：num=500 时比较合适

miniSGD：num=5000 时比较合适

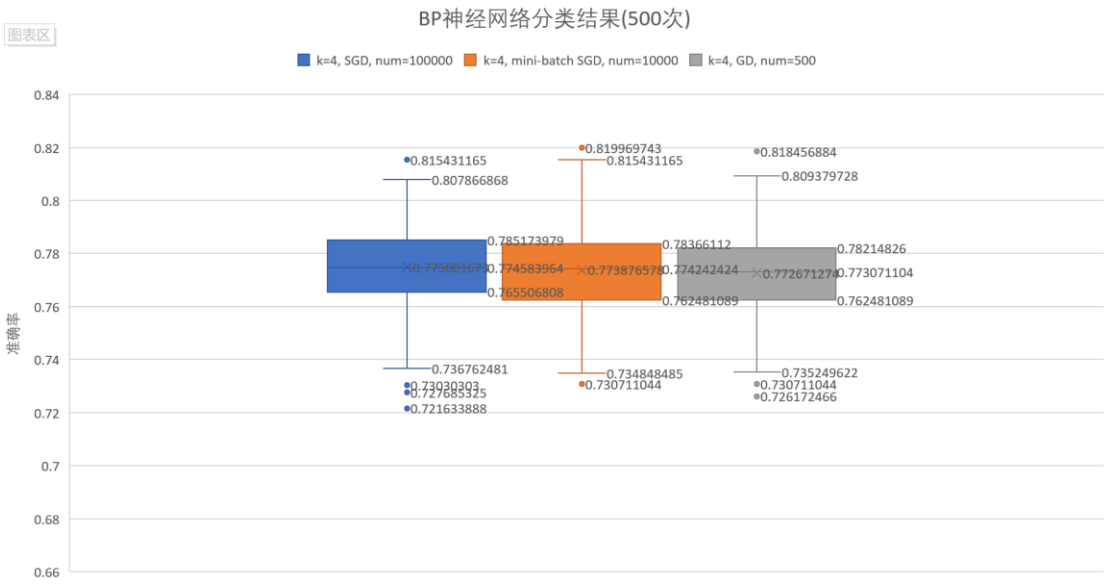
SGD：num=500000 时比较合适

```
C:\Users\mxdwa\Documents\hmc\Data Mining\BP_ANN\BP_ANN>python BPANN.py titanic.dat 4 GD 0.1 500
Truevalue:  0.7776096822995462
```

```
C:\Users\mxdwa\Documents\hmc\Data Mining\BP_ANN\BP_ANN>python BPANN.py titanic.dat 4 SGD 0.1 500000
Truevalue:  0.793939393939394
```

```
C:\Users\mxdwa\Documents\hmc\Data Mining\BP_ANN\BP_ANN>python BPANN.py titanic.dat 4 miniGD 0.1 5000
Truevalue:  0.7957639939485628
```

2. 准确率统计



由上图可知，在选择合适的训练轮数时，三种训练方法（SGD、GD、mini-batch SGD）的分类准确率类似，平均值都在 0.77 左右，即 77%。但是，三种方法的计算时间却不相同。根据计算结果和统计结构可以得出，要得到近似的准确度，mini-batch SGD 的训练时间最短，然后是 SGD，而 GD 的训练时间最长。

这也证明了 mini-batch 是一种优选的神经网络训练方法。

六、小结

本项目构建了 3*4*1 的三层神经网络（若不算入输入层则是两层），输入层有三个神经元，隐藏层只有一层，有四个神经元，输出层只有一个神经元。本项目使用 sigmoid 函数为激活函数，使用二次损失函数作为损耗函数。本项目完成了正向传播算法和反向传播算法的公式推导和函数实现，并使用梯度下降法（GD）、随机梯度下降法（SGD）和小样本集随机梯度下降法（mini-batch SGD）作为神经网络的训练方法，并对比了三种方法的分类准确度和计算时间。最后，本项目得出结论：mini-batch SGD 相对于其他两种方法效果更好，在相同准确率的前提下训练的时间最少。