模式识别实验二：感知器、神经网络

1. 实验目的：

1.1. 利用感知器算法对 Iris 数据集进行分类；

1.2. 利用 BP 算法解决异或问题，对 Iris 数据集进行分类。

2. 实验原理：

2.1. 感知器算法：

Perception 是一种直接获得完整的线性判别函数$g(x)=\alpha^T y$的方法，其中$\alpha=[w_0\,,w_1,w_2,\dots,w_d]^T, y=[1,x_1,x_2,\dots,x_4]^T$。

决策规则为： $g(y)>0, y\,\epsilon\, w_1; g(y)<0, y\,\epsilon\, w_2$。

Perception 应用于线性可分的样本数据。从线性可分的两类数据中，取得第一类数据时，$y\ = y_1$，取得第二类数据时，$y=-y_2$，从而将线性分类问题转换为在权值空间中求解向量α^*使得$\alpha^T y_i>0$的问题。

求解方法是：利用梯度下降法来迭代求解最小化感知器准则函数式。

$$\alpha(t+1)=\ \alpha(t)-\ \rho_t \nabla J_p(\alpha)=\alpha(t)+\rho_t \sum_{\alpha^T y_k \le 0}(-y_k)$$

$$J_p(\alpha)=\sum_{\alpha^T y_k \le 0}(-\alpha^T y_k)$$

其中，ρ_t为权值更新的修正步长，为了减少迭代步数，可以使用绝对修正法：$\rho_t=\frac{|\alpha(k)|^T y_i}{||y_i||^2}$来动态调整步长，以减少迭代步数。

总的来说，增量式Perception算法步骤为：

1. 随机初始化权值向量；
2. 考察一个样本，若$g(y) \leq 0$，则更新权值向量，否则继续；
3. 考察另一个样本，重复2直至所有样本都有$g(y) > 0$.

Iris 数据集要求我们搭建一个多分类感知器对三类数据集进行分类，所以我们采用数据集两两分类的方式来训练得到权值向量：$\alpha_{12}, \alpha_{13}, \alpha_{23}$,判定规则为：

$$\alpha_{12} > 0 \; and \; \alpha_{13} > 0, x \in w_1$$

$$\alpha_{12} < 0 \; and \; \alpha_{23} > 0, x \in w_2$$

$$\alpha_{13} < 0 \; and \; \alpha_{23} < 0, x \in w_3$$

2.2. BP算法：

BP算法即反向传播算法，用于对三层及以上的前馈网络的权值修正。主要思想是：从后向前逐层传播输出层的误差，以计算出隐层误差。算法主要分为两个阶段：

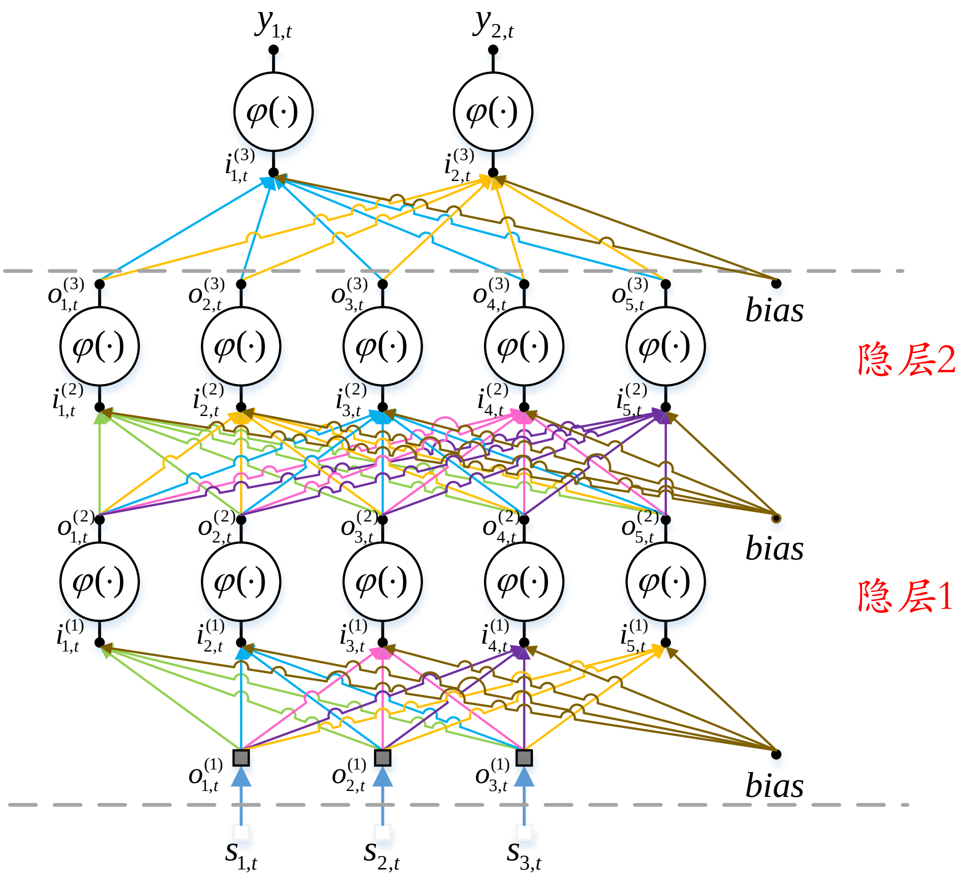
1. 正向过程：输入信息从输入层经隐层逐层计算各单元的输出值；
2. 反向过程：输出误差逐层向前算出隐层各单元的误差，并用此误差修正前层权值。

在BP算法中通常采用SGD随机梯度下降方法修正权值，为此要求输出函数可微，并且考虑到函数的非线性，通常采用 Sigmoid 函数作为输出函数：

Sigmoid函数：$f(x) = \frac{1}{1+e^{-x}} \in (0,1)$

其导数为：$f'(x) = f(x)(1 - f(x))$

神经网络的结果大概如下：



正向过程：

数据从输入层添加 bias 偏置单元之后，与权值矩阵进行矩阵乘法进而得到隐层神经元结点的输入。

此过程的矩阵表示形式为：$i^1 = w^1 * o^1$

在隐层结点内，经过 sigmoid 函数，得到神经元结点的输出。

此过程的举证表达形式为：$o^2 = [\,sigmoid(i^1), 1]$

重复以上步骤，知道计算出最终输出层的输出结果，即完成了正向传播过程。

误差函数：

$$
\begin{aligned}
E(\boldsymbol{w}) &= \frac{1}{2}\sum_{t=1}^{n}\left\|\boldsymbol{e}_t\right\|_2 = \frac{1}{2}\sum_{t=1}^{n}\left\|\boldsymbol{d}_t - \boldsymbol{y}_t\right\|_2 \\
&= \frac{1}{2}\sum_{t=1}^{n}\left\|\begin{bmatrix} d_{1,t} - y_{1,t} \\ d_{2,t} - y_{2,t} \end{bmatrix}\right\|_2 = \frac{1}{2}\sum_{t=1}^{n}\left[(d_{1,t} - y_{1,t})^2 + (d_{2,t} - y_{2,t})^2\right]
\end{aligned}
$$

反向过程:

反向过程为误差 E(w)对各层权值矩阵求偏导的过程，以矩阵形式写出 E(w)对各层权值矩阵的偏导数为:

$$\frac{\partial E}{\partial \boldsymbol{w}^{(3)}} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(3)}} \cdot \frac{\partial \boldsymbol{i}_t^{(3)}}{\partial \boldsymbol{w}^{(3)}} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(3)}} \cdot \frac{\partial\left(\boldsymbol{w}^{(3)} \cdot \boldsymbol{o}_{t+bias}^{(3)}\right)}{\partial \boldsymbol{w}^{(3)}} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(3)}} \cdot \boldsymbol{o}_{t+bias}^{(3)T}$$

$$\frac{\partial E}{\partial \boldsymbol{w}^{(2)}} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(2)}} \cdot \frac{\partial \boldsymbol{i}_t^{(2)}}{\partial \boldsymbol{w}^{(2)}} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(2)}} \cdot \frac{\partial\left(\boldsymbol{w}^{(2)} \cdot \boldsymbol{o}_{t+bias}^{(2)}\right)}{\partial \boldsymbol{w}^{(2)}} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(2)}} \cdot \boldsymbol{o}_{t+bias}^{(2)T}$$

$$\frac{\partial E}{\partial \boldsymbol{w}^{(1)}} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(1)}} \cdot \frac{\partial \boldsymbol{i}_t^{(1)}}{\partial \boldsymbol{w}^{(1)}} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(1)}} \cdot \frac{\partial\left(\boldsymbol{w}^{(1)} \cdot \boldsymbol{o}_{t+bias}^{(1)}\right)}{\partial \boldsymbol{w}^{(1)}} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(1)}} \cdot \boldsymbol{o}_{t+bias}^{(1)T}$$

$$\boldsymbol{\delta}^{(3)} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(3)}} = \frac{\partial E}{\partial \boldsymbol{y}_t} \circ \varphi'(\boldsymbol{i}_t^{(3)})$$

$$\boldsymbol{\delta}^{(2)} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(2)}} = \boldsymbol{w}_{nobias}^{(3)T} \cdot \boldsymbol{\delta}^{(3)} \circ \varphi'(\boldsymbol{i}_t^{(2)})$$

$$\boldsymbol{\delta}^{(1)} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(1)}} = \boldsymbol{w}_{nobias}^{(2)T} \cdot \boldsymbol{\delta}^{(2)} \circ \varphi'(\boldsymbol{i}_t^{(1)})$$

$$\nabla^{(3)} = \frac{\partial E}{\partial \boldsymbol{w}^{(3)}} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(3)}} \cdot \boldsymbol{o}_{t+bias}^{(3)T} = \boldsymbol{\delta}^{(3)} \cdot \boldsymbol{o}_{t+bias}^{(3)T}$$

$$\nabla^{(2)} = \frac{\partial E}{\partial \boldsymbol{w}^{(2)}} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(2)}} \cdot \boldsymbol{o}_{t+bias}^{(2)T} = \boldsymbol{\delta}^{(2)} \cdot \boldsymbol{o}_{t+bias}^{(2)T}$$

$$\nabla^{(1)} = \frac{\partial E}{\partial \boldsymbol{w}^{(1)}} = \frac{\partial E}{\partial \boldsymbol{i}_t^{(1)}} \cdot \boldsymbol{o}_{t+bias}^{(1)T} = \boldsymbol{\delta}^{(1)} \cdot \boldsymbol{o}_{t+bias}^{(1)T}$$

到这里我们就得到了，针对每一层神经元权值的更新矩阵 $\nabla^1\ \nabla^2\ \nabla^3$.BP 算法通常使用步长$\alpha$作为学习率，通常在 0.1~3 之间试探，来控制权值每一步更新的速度。一般地，学习率越大，权值更新的越快,但是过大的权值会导致函数在极值点出振荡,所以一般为了求得函数的极值点，一般不采用大学习率。

对于多分类问题,我们需要将标签转化为多维矩阵,比如说我们有

三类数据进行分类，那么第一类就定义为：[1,0,0]，第二类为：[0,1,0]，第三类为[0,0,1].也就是说，我们的标签是哪一类，我们就去这里的向量第几个元素为1，其余为0.

而我们sigmoid函数的输出取值在0~1之间，神经网络的输出必定为3维向量，这个时候我们取向量最大值处为1，其余地方为0，就可以和标签向量对应起来了。

3. 实验过程及步骤：

<u>本实验采用python语言，Jupyter编辑</u>

3.1. 感知器算法实验过程：

3.1.1. 实验数据预处理

```python
#导入库文件
from sklearn import datasets
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

```python
#加载数据并显示
iris=datasets.load_iris()

#data对应了样本的4个特征，150行4列
[length,width] = iris.data.shape
print('iris dataset length:%d, width: %d' % (length,width))

#target对应样本的标签，150行1列
long = iris.target.shape
print('iris.target long : %d ' % long)

#为数据集整体添加偏置项
dataset = np.hstack((iris.data,np.ones((length,1))))
print('iris dataset with bias size: ',dataset.shape)
```

```
iris dataset length:150, width: 4
iris.target long : 150
iris dataset with bias size:  (150, 5)
```

```python
#数据集按照标签分开，用于两两分类训练权值
dataset1 = dataset[0:50,:]
labels1 = iris.target[0:50]

dataset2 = dataset[50:100,:]
labels2 = iris.target[50:100]

dataset3 = dataset[100:150,:]
labels3 = iris.target[100:150]
```

3.1.2. 定义 check_fit 函数来检验是否所有样本都分类正确

Alpha 为权值向量

返回的 Error 是平均误差的绝对值，error=0 则所有的样本被正确分类

```python
#检查样本是否都被正确分类
def check_fit(dataset,labels,alpha):
    value = dataset @ alpha
    value[value > 0] = labels[0]
    value[value < 0] = labels[-1]
    errors = abs(value - labels)
    error = np.mean(errors)
    return error
```

3.1.3. 定义 train_function

Iteration 定义最大训练次数

```python
def train_function(dataset1,labels1,dataset2,labels2,iteration):
    #合并数据集dataset1 dataset2
    dataset = np.vstack((dataset1,dataset2))
    labels = np.hstack((labels1,labels2))
    #权值初始化为（-0.5, 0.5）之间的随机数
    alpha = np.random.random((5))- 0.5
    [length,width] = dataset.shape
    #取x_label为最后一个label 用于数据变号
    x_label = labels[-1]
    for i in range(iteration):
        indice = np.random.randint(0,length)
        data = dataset[indice]
        #如果为第二类样本中的数据，data取负
        if labels[indice] == x_label:
            data = -data
        value = data @ alpha
```

```
        if value <= 0:
            #step = abs(value) / sum(data * data)
            step = 1
            alpha = alpha + step * data
        else:
            error = check_fit(dataset, labels, alpha)
            #控制训练集的误差在0.02以内
            if error > 0.02:
                continue
            else:
                print('train process finished')
                break
    print('iteration: ',i)
    print('alpha: ', alpha)
    return alpha
```

3.1.4. 多分类感知器算法分类准则定义 predict 函数

Data 为一个待测试的样本；

Alpha12,alpha13,alpha23 分别对应权值向量

```
def predict_function(data,alpha12,alpha13,alpha23):
    value = np.zeros((3,))
    value[0] = data @ alpha12
    value[1] = data @ alpha13
    value[2] = data @ alpha23
    # value12 and vaule13 > 0
    if value[0] > 0 and value[1] > 0:
        predict = 0
    # value12<0 and value23 > 0
    elif value[0] < 0 and value[2] > 0:
        predict = 1
    # value13 and value 23 < 0
    elif value[1] < 0 and value[2] < 0:
        predict = 2
    #如果为IR区域，定为第4类
    else:
        predict = 3
    return predict
```

3.1.5. 定义误差计算函数

```
#传入分类结果和标签用于计算分类错误率
def error_ratio(result,labels):
    error = abs(result - labels)
    error[error > 0] = 1
    error_ratio = np.mean(error)
    print('error_ratio = ',error_ratio)
```

3.2. BP 算法实验过程：

3.2.1. 定义 sigmoid 函数及其导数函数

```python
#定义sigmoid函数
def sigmoid(data):
    return 1.0 /(1.0 + np.exp(-data))
```

```python
#定义sigmoid函数的导数
def derive_sigmoid(data):
    return sigmoid(data)*(1 - sigmoid(data))
```

3.2.2. 定义 BPNN 类并定义其子函数：

```python
class BPNN:
    #定义初始化函数
    #n_input,n_nodes,n_output分别为输入层、隐层、输出层节点个数
    def init_function(self,n_input,n_nodes,n_output):
        self.n_output = n_output
        self.value1 = np.random.random((n_nodes,n_input+1))*0.1
        self.value2 = np.random.random((self.n_output,n_nodes+1))*0.1
        self.hiddenlayer_input = np.zeros((n_nodes,1))
        self.outputlayer_input = np.zeros((n_output,1))
        self.predict = np.zeros((self.n_output,1))
        self.error = self.predict
```

前向传播函数，单个数据 data 作为输入

```python
def forward_function(self,data):
    self.hiddenlayer_input = self.value1 @ data
    hiddenlayer_output = np.vstack((sigmoid(self.hiddenlayer_input),[1]))

    self.outputlayer_input = self.value2 @ hiddenlayer_output
    self.predict = sigmoid(self.outputlayer_input)
```

后向传播函数，单个数据 data,其标签 label 和学习率 alpha 作为输入

```python
def backward_function(self,data,label,alpha):
    self.error = self.predict - label

    sigma2 = self.error * derive_sigmoid(self.outputlayer_input)
    sigma1 = np.transpose(self.value2[:,:-1]) @ sigma2 * derive_sigmoid(self.hiddenlayer_input)

    delta2 = sigma2 @ np.vstack((sigmoid(self.hiddenlayer_input),[1])).T

    delta1 = sigma1 @ data.T

    self.value2 -= delta2 * alpha
    self.value1 -= delta1 * alpha
```

在给定循环次数条件下，训练神经网络

```python
def train_function(self, dataset, labels, alpha, iteration):
    [length, width] = dataset.shape

    for i in range(iteration):
        indice = np.random.randint(length)
        data = dataset[indice, :].reshape((width, 1))
        label = labels[indice, :].reshape((self.n_output, 1))
        self.forward_function(data)
        self.backward_function(data, label, alpha)
    print('training process finished')
```

基于训练后的权值，对测试集进行预测

```python
def predict_function(self, dataset):
    [length, width] = dataset.shape
    predict = np.zeros((length, self.n_output))

    for i in range(length):
        data = dataset[i, :].reshape((width, 1))
        self.forward_function(data)
        indice = np.argmax(self.predict)
        predict[i, indice] = 1
    return predict
```

计算测试集预测结果的错误率，并输出误差向量 errors

```python
def loss_function(self, predict, labels):
    errors = predict - labels
    errors = np.sum(abs(errors), 1)/2
    error_rate = np.mean(errors)
    print('error_rate: ', error_rate)
    return errors
```

对数据集批量添加偏置单元，并将向量的标签转化为符合 BP 算法输出的矩阵形式

```python
def pretrait_function(self, dataset, pre_labels):
    [length, width] = dataset.shape
    dataset = np.hstack((dataset, np.ones((length, 1))))

    labels = np.zeros((length, self.n_output))
    for i in range(length):
        indice = int(pre_labels[i])
        labels[i, indice] = 1
    return dataset, labels
```

4. 实验结果分析：

4.1. 感知器算法实验结果：

利用 train_test_split 函数对数据集进行划分，其中三折属于用于测试集，其余数据用于训练集

```
#将数据集分为训练集和测试集
X_train1,X_test1,y_train1,y_test1 = train_test_split(dataset1, labels1, test_size = 0.3, random_state = 0)
X_train2,X_test2,y_train2,y_test2 = train_test_split(dataset2, labels2, test_size = 0.3, random_state = 0)
X_train3,X_test3,y_train3,y_test3 = train_test_split(dataset3, labels3, test_size = 0.3, random_state = 0)
```

```
#训练得到三个权值向量
alpha12 = train_function(X_train1,y_train1,X_train2,y_train2,1000)
alpha13 = train_function(X_train1,y_train1,X_train3,y_train3,1000)
alpha23 = train_function(X_train2,y_train2,X_train3,y_train3,1000)
```

```
train process finished
iteration:  25
alpha:  [ 1.09788663  6.28964454 -8.8571606  -2.98949689  1.18888483]
train process finished
iteration:  16
alpha:  [  3.01911417   6.99082169 -10.19139253  -6.04451416   1.64422894]
train process finished
iteration:  27
alpha:  [ 2.52452252  1.59116369 -3.65704625 -2.02606035  0.87933523]
```

```
predict = np.zeros((length))
for i in range(length):
    predict[i] = predict_function(predict_dataset[i,:],alpha12,alpha13,alpha23)
#errors = abs(predict - predict_labels)
#print(errors)
error_ratio(predict,predict_labels)
```

```
error_ratio =  0.06666666666666667
```

4.2. BP 算法实验结果：

4.2.1. 异或问题：

构造数据

```
data = np.array([[0,0],[0,1],[1,1],[1,0]]).reshape(4,2)
label = np.array([[0],[1],[0],[1]])
dataset1 = (np.random.random((50,2)) - 0.5) * 0.2 + data[0,:]
dataset2 = (np.random.random((50,2)) - 0.5) * 0.2 + data[1,:]
dataset3 = (np.random.random((50,2)) - 0.5) * 0.2 + data[2,:]
dataset4 = (np.random.random((50,2)) - 0.5) * 0.2 + data[3,:]
pre_dataset = np.vstack((dataset1,dataset2,dataset3,dataset4))

pre_labels = np.zeros((200,1))
pre_labels[50:100] = 1
pre_labels[150:200] = 1

print('dataset size: ', pre_dataset.shape)
print('labels size: ',pre_labels.shape)
```

```
dataset size:  (200, 2)
labels size:  (200, 1)
```

a. 创造 bpnn 类;

b. 初始化神经网络模型输入数据 2 维 5 个隐层结点输出 2 维;

c. 对初始数据集和标签进行预处理，添加偏置单元转换标签格式;

d. 利用 train_test_split 得到数据集中 8 成数据用于训练，剩余数据用于测试;

e. 学习率 alpha=1，循环次数 iteration=5000 训练 bpnn 模型;

f. 对 X_test 集预测;

g. Loss_function 计算错误率输出错误率已经误差向量 errors

```
bpnn = BPNN()
bpnn.init_function(2,5,2)
[dataset, labels] = bpnn.pretrait_function(pre_dataset,pre_labels)
X_train,X_test,y_train,y_test = train_test_split(dataset, labels, test_size = 0.2, random_state = 0)
bpnn.train_function(X_train,y_train,alpha=1,iteration=5000)
predict = bpnn.predict_function(X_test)
errors = bpnn.loss_function(predict,y_test)
print(errors)
```

```
training process finished
error_rate:  0.0
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

4.2.2. Iris 数据集

```
#加载数据并显示
iris=datasets.load_iris()

#data对应了样本的4个特征，150行4列
print(iris.data.shape)

#target对应样本的标签，150行1列
print(iris.target.shape)
```

```
(150, 4)
(150,)
```

h. 创造 iris_bpnn 类;

i. 初始化神经网络模型输入数据 4 维 10 个隐层结点输出 3 维;

j. 对初始数据集和标签进行预处理，添加偏置单元转换标签格式;

k. 利用 train_test_split 得到数据集中 8 成数据用于训练，剩余数

据用于测试；

l. 学习率 alpha=0.5,循环次数 iteration=5000 训练 iris_bpnn；

m. 对 X_test 集预测；

n. Loss_function 计算错误率输出错误率已经误差向量 errors

```
iris_bpnn = BPNN()
iris_bpnn.init_function(4,10,3)
[dataset, labels] = iris_bpnn.pretrait_function(iris.data,iris.target)
X_train,X_test,y_train,y_test = train_test_split(dataset, labels, test_size = 0.2, random_state = 0)
iris_bpnn.train_function(X_train,y_train,alpha=0.5,iteration=5000)
predict = iris_bpnn.predict_function(X_test)
errors = iris_bpnn.loss_function(predict,y_test)
print(errors)
```

```
training process finished
error_rate:  0.0
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
```

5. 实验结论与讨论：

5.1. 结论：

5.1.1. 感知器算法在对各个权值各训练 1000 次之后，3 折数据进行测试，得到的准确率为 93%；

5.1.2. BP 神经网络经过不断尝试学习率 alpha 和循环次数 iteration 之后,也找到了使所有测试集都分类正确的合适参数，准确率为 100%

5.2. 讨论：

5.2.1. 两种分类方法的精度和复杂度都与初值的选取有关,如果要提高算法的训练速度的话，可以尝试优化算法中参数的初始化方法。

5.2.2. 通过对 iris 数据，以及异或问题的数据分析，发现数据相差并不大，所以在感知器算法和 BP 神经网络算法中并没有定义归一化函数；

5.2.3. BP 神经网络训练时长和初始化权值的选取有很大关系，因为考虑到权值目标权值都为极小的小数，所以对其初始化是在 0~0.1 内取随机值；

5.2.4. 感知器算法和BP神经网络总的来说,没有很大的可比性。感知器算法针对线性数据进行分类，BP 神经网络主要用于对非线性数据分类。这里，我主观地认为感知器算法通常被用来解决线性可分问题,BP 神经网络用于复杂非线性问题。在具体实验过程中，BPNN 的参数调整依靠经验，简单线性可分问题也要重复此过程，相比于感知器算法，反而更麻烦。

5.2.5. 关于 BP 神经网络调参的实验很复杂，网上有很多分享神经网络调参的经验和方法，比如说加入正则项提高泛化能力，提升训练速度；使用自适应学习率来提升训练速度和精度等等。在本实验中，我就不在做验证性的实验了。