

## 一、实验题目

实验一 基于感知机的鸢尾花分类

## 二、实验目的

利用感知机算法对鸢尾花种类进行分类，要求熟悉感知机算法，掌握利用Python实现机器学习算法的一般流程，了解scikit-learn机器学习库的使用。

## 三、实验内容

### 实验原理

感知机(Perceptron)在1957年由Rosenblatt提出，是神经网络和支持向量机的基础。

感知机是一种二类分类的线性分类模型，其输入为实例的特征向量，输出为实例的类别，+1代表正类，-1代表负类。感知机属于判别模型，它的目标是要将输入实例通过分离超平面将正负二类分离。感知机模型的假设空间是定义在特征空间中的所有线性分类模型，即函数集合 $\{f|f(x) = w \cdot x + b\}$ 。

本实验示例程序的感知机是一个由输入空间 $X \subset R^n$ 到输出空间 $y = \{+1, -1\}$ 的函数：

$$f(x) = \text{sign}(w \cdot x + b), \quad \text{sign}(x) = \begin{cases} +1, & x > 0 \\ -1, & x \leq 0 \end{cases}$$

其中 $w, b$ 是感知机模型的参数， $w \in R^n$ 叫做权值或权值向量， $b \in R$ 叫做偏置， $w \cdot x$ 表示 $w$ 和 $x$ 的内积， $\text{sign}$ 是激活函数，很明显是一个离散感知机模型。

### 算法/程序流程图、步骤和方法

感知机算法：

假设给定一个增广的训练式集 $\{x_1 + x_2 + \dots + x_N\}$ ，其中每个模式类别已知，它们分属于 $w_1$ 类和 $w_2$ 类。

(1) 迭代次数 $k=1$ ，令校正增量 $p$ =某正常数，分别赋给初始增广权向量 $w_1$ 的各分量较小的任意值。

(2) 输入训练模式 $x_k$ ，计算判别函数值 $w_k$ ， $x_k$

(3) 调整增广权矢量，规则是

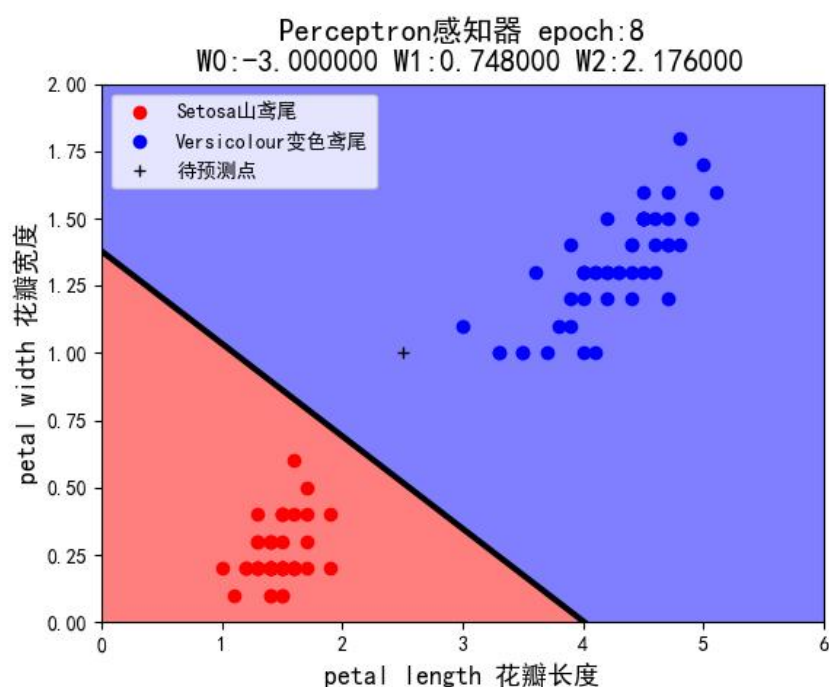
1. 如果 $x_k \in w_1$ 和 $w(k) x_k \leq 0$ ，则 $w(k+1) = w(k) + p x_k$
2. 如果 $x_k \in w_2$ 和 $w(k) x_k \geq 0$ ，则 $w(k+1) = w(k) - p x_k$
3. 如果 $x_k \in w_1$ 和 $w(k) x_k > 0$ 或如果 $x_k \in w_2$ 和 $w(k) x_k < 0$ 则 $w(k+1) = w(k)$ 。
4. 如果 $k < N$ ，令 $k=k+1$ ，返至2。如果 $k=N$ ，检验判别函数是否都能正确分类。若是，结束；若不是，令 $k=1$ ，返至2。

## 四、实验结果和分析

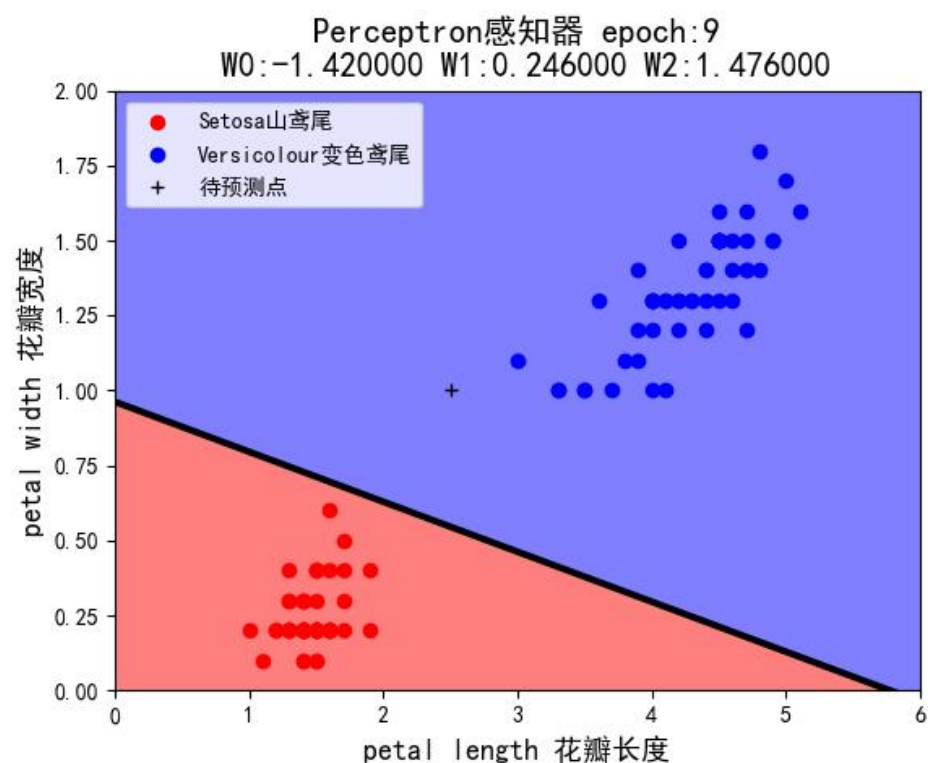
请回答下列问题：

1) 考虑学习率的作用。修改示例代码，固定初始权值=(1, 1, 1)，将学习率分别设定为1、0.5、0.1（组合1~3），程序在epoch 等于多少时实现分类？

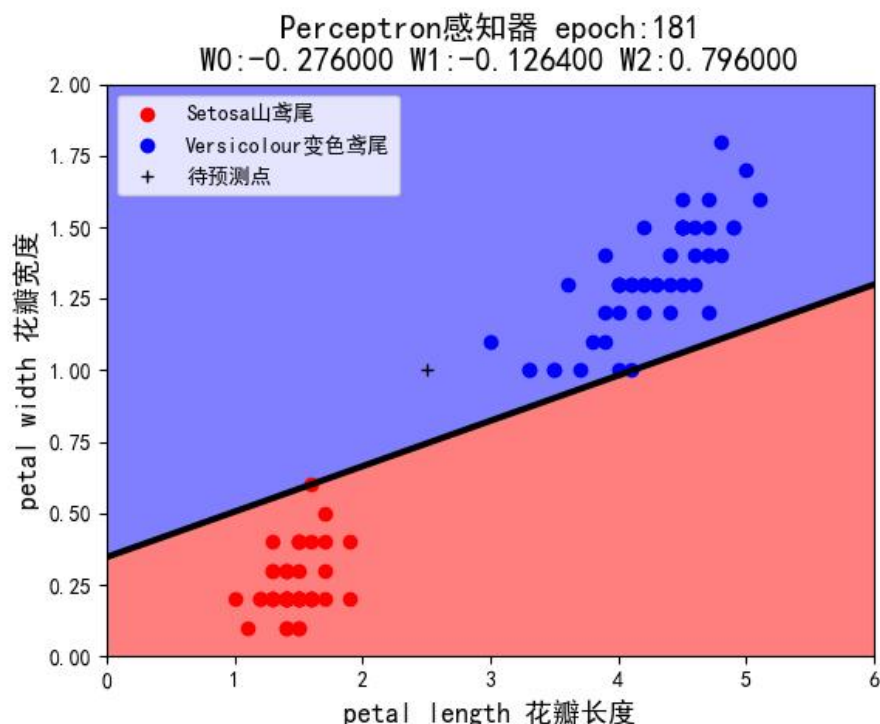
当学习率为1时，程序在epoch等于8时实现分类



当学习率为0.5时，程序在epoch等于9时实现分类



当学习率为0.1时，程序在epoch等于181时实现分类



学习率决定着目标函数能否收敛到局部最小值以及何时收敛到最小值。学习率越大，就可以在较小的步数内完成分类，同时数据的边界线离分界线越远。由图可知，学习率为0.1时数据边界线与分界线非常密切

2) 考虑初始权值的作用。修改示例代码，固定学习率=0.1，将初始权值分别设定为(-1, 1, 1)、(+1, -1, -1)、(1, -1, +1)、(-1, +1, -1) (组合4~7)，程序在epoch 等于多少时实现分类？

将初始权值设定为组合4，程序在epoch等于13 实现分类

将初始权值设定为组合5，程序在epoch等于 27 实现分类

将初始权值设定为组合6，程序在epoch等于 217 实现分类

将初始权值设定为组合7，程序在epoch等于 27 实现分类

3) 示例程序使用的是离散感知机还是连续感知机？如何判断？

使用的是离散感知机。示例程序是单输入感知器，只用到了一个神经元，是一种最简单形式的前馈式网络，同层内无互连，不同层间无反馈，神经元对输入信号加权求和后，由激活函数决定其输出。而使用的激活函数是

$\text{np.sign}, \text{sign}(x) = \begin{cases} +1, & x > 0 \\ -1, & x \leq 0 \end{cases}$  输出的值为+1, -1。所以 $f(0+) \neq f(0-)$ ，函数不连续，从而可判断该感知机为离散感知机。

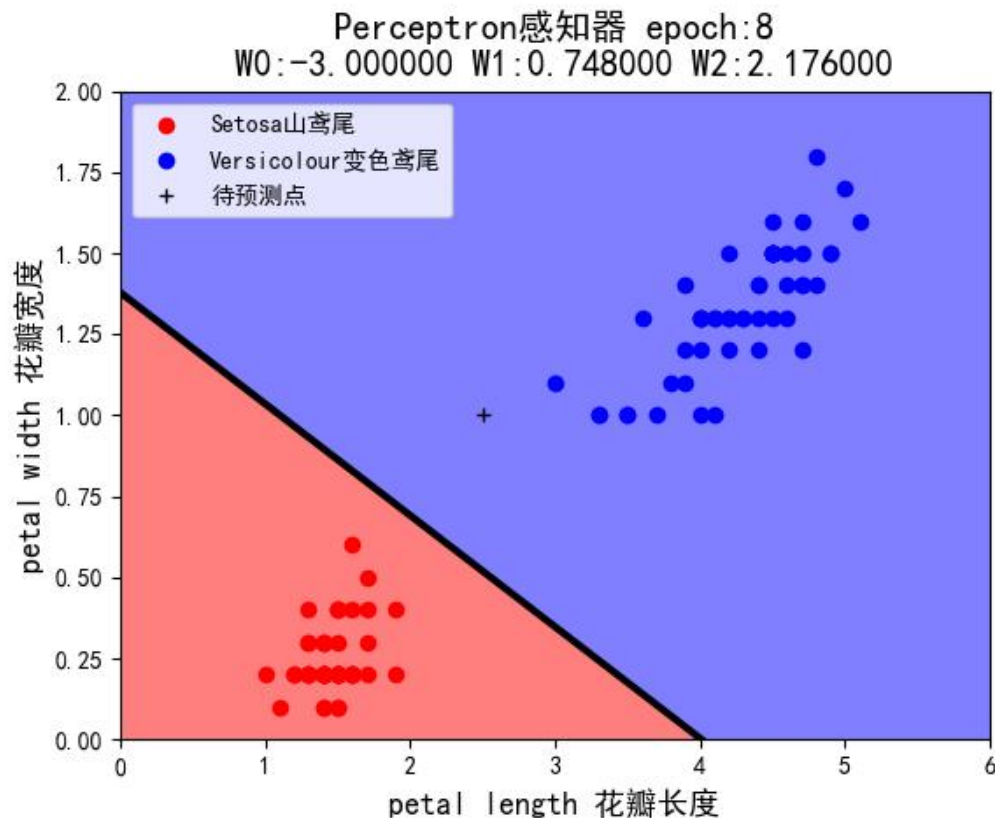
4) 为什么在学习算法中要除以X.shape[0]？示例程序采用的是批量下降还是逐一下降？是否属于随机下降？是否属于梯度下降？

X.shape[0]得到X的行数，表示有多少个数据，除以X.shape[0]得到权值的变化量；采用的是批量下降属于梯度下降。

5) 假设你在自然界找到了一朵鸢尾花，并测得它的花瓣长度为2.5cm，花瓣宽度为1cm，

它属于哪一类？在draw()中已用plt.plot 画出这个‘待预测点’。请观察1~7 这7 种组合中，感知机的判断始终一致么？这说明它受到什么因素的影响？

属于变色鸢尾。



组合1~3，固定权值 $w=(1, 1, 1)$ 不变更学习率，都属于变色鸢尾  
固定学习率为0.1

组合4的权值 $w=(-1, 1, 1)$ ，属于变色鸢尾

组合5的权值 $w=(1, -1, -1)$ ，属于山鸢尾

组合6的权值 $w=(1, -1, 1)$ ，属于变色鸢尾

组合7的权值 $w=(-1, 1, -1)$ ，属于山鸢尾

感知机的判断并不始终保持一致，当权值固定时，学习率变化，分类不会发生变化，当学习率固定时，权值变化分类也会变化

6) 修改示例代码，将变色鸢尾的数据替换为维吉尼亚鸢尾，再进行分类。  
即横轴为花瓣长度，纵轴为花瓣宽度，数据为Setosa 山鸢尾+Virginica 维吉尼亚鸢尾。  
只需修改初始化部分代码即可，将数据先进行切割再拼接

```
1. # 数据为 Setosa 山鸢尾+Virginica 维吉尼亚鸢尾-----
2. # 先按行切割数据再按列拼接数据
3. X_1 = np.c_[np.ones(50), iris.data[:50, 2:4]]
4. X_2 = np.c_[np.ones(50), iris.data[100:150, 2:4]]
5. X = np.r_[X_1, X_2]
6. # 得到真值 T，从数据集中得到真值
7. T_1 = iris.target[:50]
8. T_2 = iris.target[100:150]
```

```

9. T = np.r_[T_1, T_2].reshape(100, 1)
10. # 将 T 中等于 0 的赋值为 -1, 等于 2 的赋值为 1, 以契合 sign 函数
11. T[T == 0] = -1
12. T[T == 2] = 1

```

画图函数的修改, 只需修改标签 label 即可

```

1. # 更改画图-----
-----
2. plt.plot(X[50:100, 1], X[50:100, 2], 'bo', color='blue', label='V
    irginica 维吉尼亚鸢尾')

```

7) 【可选】目前感知机只有两个输入+偏置, 如果有三个输入 (比如增加萼片长度作为输入), 程序应如何修改 (可以不画图)?

```

1. # -----三个输入: Setosa 山鸢尾+Virginica 维吉尼亚鸢尾的花瓣长度、花
    瓣宽度、花萼长度-----
2. X_1 = np.c_[np.ones(50), iris.data[:50, 2:4], iris.data[:50, 0:2]]
3. X_2 = np.c_[np.ones(50), iris.data[100:150, 2:4], iris.data[100:150,
    0:2]]
4. X = np.r_[X_1, X_2]
5. # 得到真值 T, 从数据集中得到真值
6. T_1 = iris.target[:50]
7. T_2 = iris.target[100:150]
8. # 将 T 中等于 0 的赋值为 -1, 等于 2 的赋值为 1, 以契合 sign 函数
9. T = np.r_[T_1, T_2]
10. T[T == 0] = -1
11. T[T == 2] = 1
12. # 直接赋初值, 不使用随机数
13. W = np.array([[1],
14.                [1],
15.                [1],
16.                [1]])

```

## 五、小结与心得体会

通过此次实验, 进一步熟悉了感知机算法的原理和算法过程。同时在本次实验中, 在数据选择上优先选择有较好线性关系的数据集, 可见感知机适合用于解决线性可分的问题, 对非线性可分问题并不适用。

而对于判断感知机是离散感知机还是连续感知机, 要看激活函数是离散的还是连续的

## 一、实验题目

实验二以人事招聘为例的误差反向传播算法

## 二、实验目的

理解多层神经网络的结构和原理，掌握反向传播算法对神经元的训练过程，了解反向传播公式。通过构建BP 网络实例，熟悉前馈网络的原理及结构。

## 三、实验内容

### 实验原理/运用的理论知识

误差反向传播算法即BP 算法，是一种适合于多层神经网络的学习算法。其建立在梯度下降方法的基础之上，主要由激励传播和权重更新两个环节组成，经过反复迭代更新、修正权值从而输出预期的结果。

BP 算法整体上可以分成正向传播和反向传播，原理如下：

正向传播过程：信息经过输入层到达隐含层，再经过多个隐含层的处理后到达输出层。

反向传播过程：比较输出结果和正确结果，将误差作为一个目标函数进行反向传播：

对每一层依次求对权值的偏导数，构成目标函数对权值的梯度，网络权重再依次完成更新调整。依此往复、直到输出达到目标值完成训练。

该算法可以总结为：利用输出误差推算前一层的误差，再用推算误差算出更前一层的误差，直到计算出所有层的误差估计。

1986 年，Hinton 在论文《Learning Representations by Back-propagating Errors》中首次系统地描述了如何利用BP 算法来训练神经网络。

从此，BP 算法开始占据有监督神经网络算法的核心地位。它是迄今最成功的神经网络学习算法之一，现实任务中使用神经网络时，大多是在使用BP 算法进行训练。

### 算法/程序流程图

给定数据集 $D=\{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$ ,  $x_i \in R^d$ ,  $y_i \in R^d$ , 输入样本有 $d$ 个属性描述，输出 $n^l$ 维实向量。其他多个符号的表示含义如下：

$L$ : 神经网络层数;  $n^l$ : 第 $l$ 层神经元的个数;  $f(\cdot)$ :  $l$ 层的神经元的激活函数;  $W^l$ :  $l-1$ 层到第 $l$ 层的权重矩阵;

$b^l$ :  $l-1$ 层到第 $l$ 层的偏置;  $z^l$ : 第 $l$ 层的神经元的加权输入;  $a^l$ : 第 $l$ 层的神经元的输出

反向传播以前向传播为基础，从前向传播得到的参数向前反向推导更新每一层的权重和偏置。

假定以误差平方作为损失函数，将该目标函数作为优化目标：

$J(W, b) = \frac{1}{2} \sum_{j=1}^{n^L} (\hat{y}_j - y_j)^2$ , 其中 $\hat{y}_j$ 为神经网络的输出，是 $n^L$ 维向量， $y_j$ 为数据集给出的真实结果，是 $n^L$ 维向量

## 四、实验结果与分析

1) 如果去掉总裁这一层, 相应张三的样本修改为(1.0, 0.1, 1.0, 1.0), 分别对应张三的学习成绩、张三的实践成绩、张三的工作能力真值、张三的工作态度真值, 代码应该如何修改?

**数据初始化模块:** 需要修改真值

```
1. # 需要修改真值
2. T = np.array([[1]]) 改为 T = np.array([[1, 1]])
```

**更新权值和偏置值模块:**

这部分需要将原有的输出层的部分代码删除, 并修改求隐藏层2误差的代码

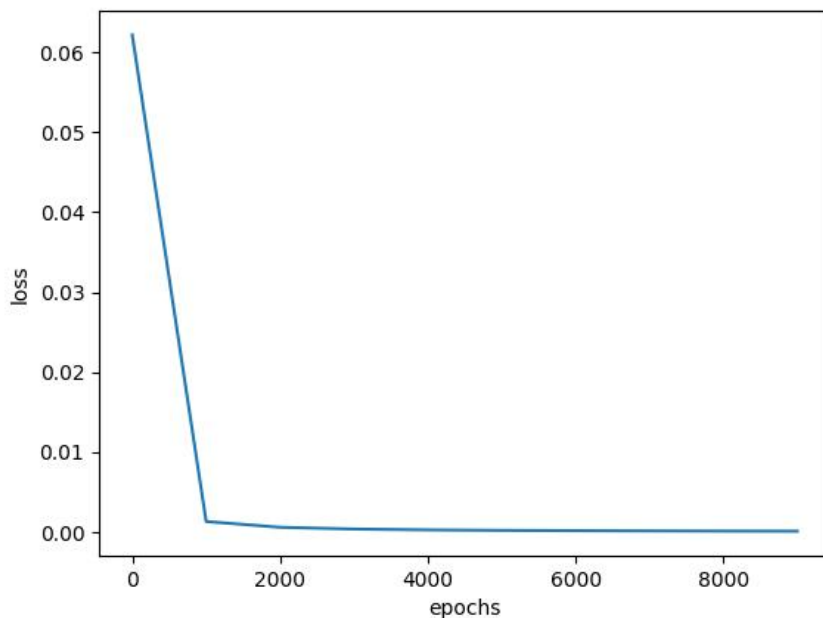
```
1. # 更新权值和偏置值
2. # -----解决问题 1, 修改后的代码-----
-
3. def update():
4.     global batch_X, batch_T, W1, W2, lr, b1, b2
5.
6.     # 隐藏层 1 输出
7.     Z1 = np.dot(batch_X, W1) + b1
8.     A1 = sigmoid(Z1)
9.
10.    # 隐藏层 2 输出
11.    Z2 = (np.dot(A1, W2) + b2)
12.    A2 = sigmoid(Z2)
13.    # 求隐藏层 2 的误差
14.    delta_A2 = (batch_T - A2)
15.    delta_Z2 = delta_A2 * dsigmoid(A2)
16.
17.    # 利用隐藏层 2 的误差, 求出三个偏导 (即隐藏层 1 到隐藏层 2 的权值改
    变)    # 由于一次计算了多个样本, 所以需要求平均
18.    delta_W2 = A1.T.dot(delta_Z2) / batch_X.shape[0]
19.    delta_B2 = np.sum(delta_Z2, axis=0) / batch_X.shape[0]
20.
21.    # 求隐藏层 1 的误差
22.    delta_A1 = delta_Z2.dot(W2.T)
23.    delta_Z1 = delta_A1 * dsigmoid(A1)
24.
25.    # 利用隐藏层 1 的误差, 求出三个偏导 (即输入层到隐藏层 1 的权值改
    变)    # 由于一次计算了多个样本, 所以需要求平均
26.    delta_W1 = batch_X.T.dot(delta_Z1) / batch_X.shape[0]
27.    delta_B1 = np.sum(delta_Z1, axis=0) / batch_X.shape[0]
28.
29.    # 更新权值
30.    W2 = W2 + lr * delta_W2
31.    W1 = W1 + lr * delta_W1
32.
33.    # 改变偏置值
34.    b2 = b2 + lr * delta_B2
35.    b1 = b1 + lr * delta_B1
```



训练模型模块:

取消原有输出层的输出, 将输出改为隐藏层2的输出

实验结果:



```
epochs: 6000 loss: 0.00017884613966145948
A2: [[0.98268007 0.98253276]]
epochs: 7000 loss: 0.00015127107169001484
A2: [[0.98388943 0.98374826]]
epochs: 8000 loss: 0.0001309173245098213
A2: [[0.9848827 0.98474705]]
epochs: 9000 loss: 0.00011529633498690287
D:\GZH\Desktop\计算智能课程上传QQ\实验2反向传播\实验二以人事招聘为例的误差反向传播算法.py:219: Matplotlib
plt.plot(range(0, epochs, report), loss)
D:\GZH\Desktop\计算智能课程上传QQ\实验2反向传播\实验二以人事招聘为例的误差反向传播算法.py:222: Matplotlib
plt.show()
output:A2
[[0.98571613 0.98558545]]
predict:
1
```

2) 如果增加一个样本, 李四 (0.1, 1.0, 0), 分别对应李四的学习成绩, 李四的实践成绩, 李四被招聘可能性的真值, 代码应该如何修改? 此时是一个样本计算一次偏导、更新一次权值, 还是两个样本一起计算一次偏导、更新一次权值? (提示: 注意 batch\_size 的作用)

数据初始化模块:

加入李四的数据和真值

将  $X = \text{np.array}([[1.0, 0.1]])$  改为  $X = \text{np.array}([[1.0, 0.1], [0.1, 1.0]])$

将  $T = \text{np.array}([[1]])$  改为  $T = \text{np.array}([[1], [0]])$



每组大小为1，一个样本计算一次偏导、更新一次权值

`batch_size = 1`

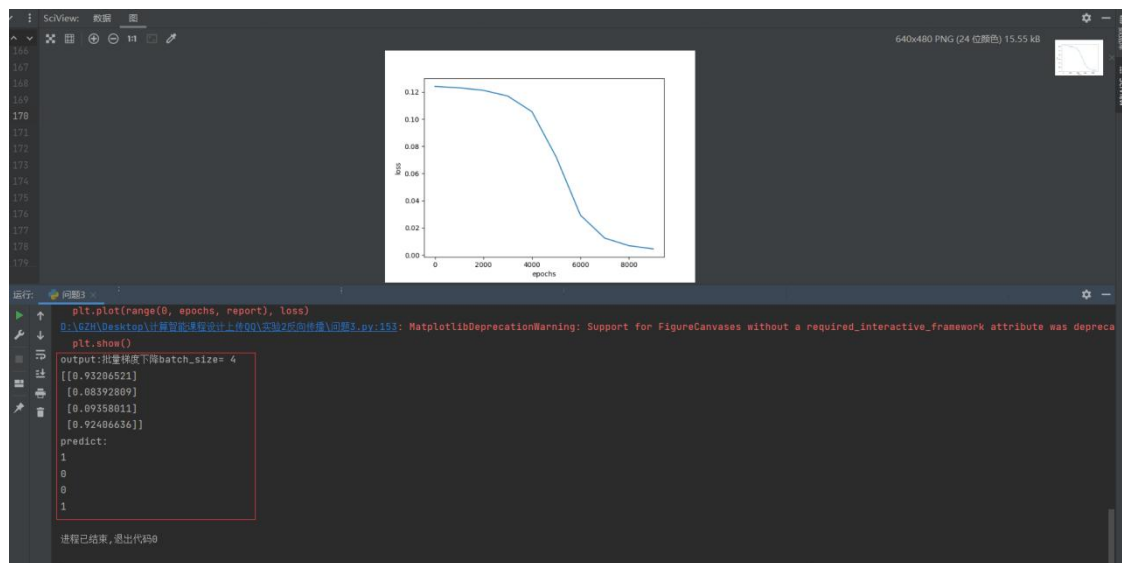
每组的大小为2，两个样本一起计算一次偏导、更新一次权值

`batch_size = 2`

3) 样本为张三 $[1, 0.1, 1]$ 、李四 $[0.1, 1, 0]$ 、王五 $[0.1, 0.1, 0]$ 、赵六 $[1, 1, 1]$ ，请利用 `batch_size` 实现教材279 页提到的“批量梯度下降”、“随机梯度下降”和“小批量梯度下降”，请注意“随机梯度下降”和“小批量梯度下降”要体现随机性。

**批量梯度下降：**需要修改的代码

`batch_size=4`



**随机梯度下降：**需要修改的代码

数据初始化模块：

1. `batch_size = 1`

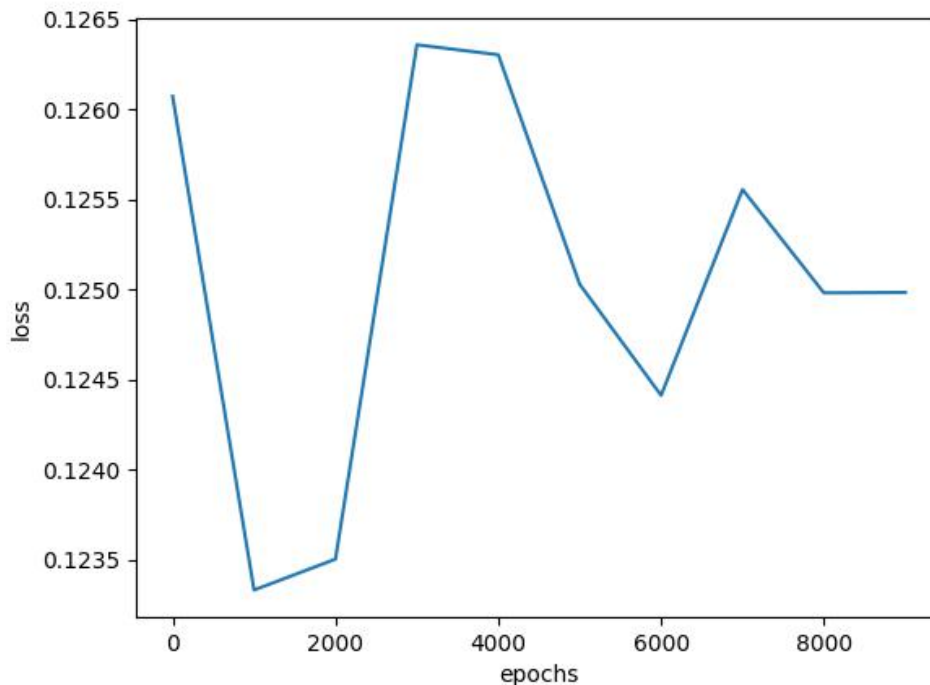
训练模型模块：

```
1. for idx_batch in range(max_batch):
2.     # 更新权值
3.     # 随机梯度下降
4.     # 打乱 X 中的数据，保证随机性
5.     np.random.shuffle(X)
6.     print("观察样本的随机性 idx_batch:", idx_batch)
7.     batch_X = X[idx_batch * batch_size:(idx_batch + 1) * batch_size,
8.         :]
9.     batch_T = T[idx_batch * batch_size:(idx_batch + 1) * batch_size,
10.         :]
11.     print("观察权值的变化")
12.     print("batch_X:", batch_X)
13.     print("batch_T", batch_T)
14.     update()
```

### 实验结果:

随机性的体现: 训练模块 `np.random.shuffle(X)`

训练周期数与loss的关系图:



小批量梯度下降:

数据初始化模块:

```
1. batch_size = 2
```

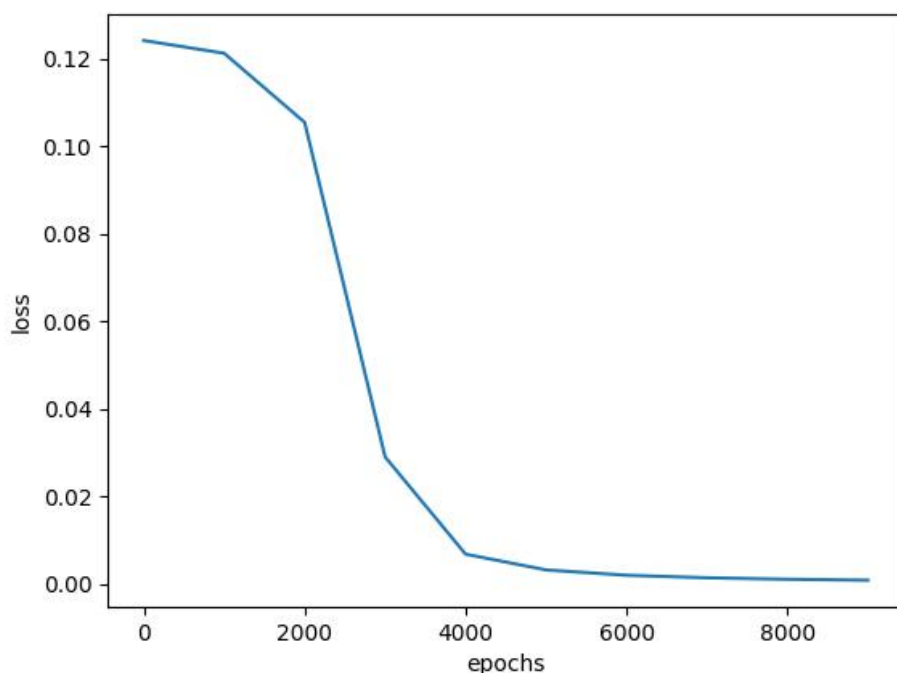
训练模块:

```
1. for idx_batch in range(max_batch):
2.     # 小批量梯度下降
3.     idx_batch = random.randint(0, max_batch - 1)
4.     print("观察样本的随机性 idx_batch:", idx_batch)
5.     batch_X = X[idx_batch * batch_size:(idx_batch + 1) * batch_size,
6.                 :]
7.     batch_T = T[idx_batch * batch_size:(idx_batch + 1) * batch_size,
8.                 :]
9.     print("观察权值的变化")
10.    print("batch_X:", batch_X)
11.    print("batch_T", batch_T)
12.    update()
```

### 实验结果:

随机性体现: `idx_batch = random.randint(0, max_batch - 1)`, 对索引取随机数

训练周期数与loss的关系图：



4 ) 【选做】本例中输入向量、真值都是行向量， 请将它们修改为列向量， 如  $X = \text{np.array}([[1, 0.1]])$  改为  $X = \text{np.array}([[1], [0.1]])$ ，请合理修改其它部分以使程序得到与行向量时相同的结果。（不允许直接使用X的转置进行全局替换）

**数据初始化模块:**将真值T和权值W加上转置，其他的不变

```
1. # 标签，也叫真值，1 行 1 列，张三的真值：一定录用
2. T = np.array([[1]]).T
3.
4. # 定义一个 2 隐层的神经网络：2-2-2-1
5. # 输入层 2 个神经元，隐藏 1 层 2 个神经元，隐藏 2 层 2 个神经元，输出层 1 个神经元
6.
7. # 输入层到隐藏层 1 的权值初始化，2 行 2 列
8. W1 = np.array([[0.8, 0.2],
9.                 [0.2, 0.8]]).T
10. # 隐藏层 1 到隐藏层 2 的权值初始化，2 行 2 列
11. W2 = np.array([[0.5, 0.0],
12.                [0.5, 1.0]]).T
13. # 隐藏层 2 到输出层的权值初始化，2 行 1 列
14. W3 = np.array([[0.5],
15.                 [0.5]]).T
```

**update() 函数:**

由于要求列向量的输出，batch\_x需要转置变为维度（2，1）的向量，权值W1也进行了转置，但转置前后维度都为(2, 2)不发生变化，所以需要交换两者的位置进行矩阵相乘。

代码修改如下：

```
1. # 隐藏层 1 输出
2. Z1 = np.dot(W1, batch_X.T) + b1
```

```

3.     A1 = sigmoid(Z1)
4.
5.     # 隐藏层 2 输出
6.     Z2 = (np.dot(W2, A1) + b2)
7.     A2 = sigmoid(Z2)
8.
9.     # 输出层输出
10.    Z3 = (np.dot(W3, A2) + b3)
11.    A3 = sigmoid(Z3)

```

对于求输出层的误差不作任何修改

利用输出层的误差求偏导，需要取delta\_Z3与A2的转置相dot，并且分母取列数batch\_x.shape[1], 对于axis=1的解释，numpy中sum默认的axis=0，就是普通相加，axis=1，是将矩阵的每一行向量相加。

利用隐藏层1, 2的误差求出三个偏导的实现过程与上同理

```

1. # 求输出层的误差
2.     delta_A3 = (batch_T - A3)
3.     delta_Z3 = delta_A3 * dsigmoid(A3)
4.
5.     # 利用输出层的误差，求出三个偏导（即隐藏层 2 到输出层的权值改
    变）    # 由于一次计算了多个样本，所以需要求平均
6.     delta_W3 = delta_Z3.dot(A2.T) / batch_X.shape[1]
7.     delta_B3 = np.sum(delta_Z3, axis=1) / batch_X.shape[1]
8.
9.     # 求隐藏层 2 的误差
10.    delta_A2 = W3.T.dot(delta_Z3)
11.    delta_Z2 = delta_A2 * dsigmoid(A2)
12.
13.    # 利用隐藏层 2 的误差，求出三个偏导（即隐藏层 1 到隐藏层 2 的权值改
    变）    # 由于一次计算了多个样本，所以需要求平均
14.    delta_W2 = delta_Z2.dot(A1.T) / batch_X.shape[1]
15.    delta_B2 = np.sum(delta_Z2, axis=1) / batch_X.shape[1]
16.
17.    # 求隐藏层 1 的误差
18.    delta_A1 = W2.T.dot(delta_Z2)
19.    delta_Z1 = delta_A1 * dsigmoid(A1)
20.
21.    # 利用隐藏层 1 的误差，求出三个偏导（即输入层到隐藏层 1 的权值改
    变）    # 由于一次计算了多个样本，所以需要求平均
22.    delta_W1 = delta_Z1.dot(batch_X.T) / batch_X.shape[1]
23.    delta_B1 = np.sum(delta_Z1, axis=1) / batch_X.shape[1]
24.
25.    # 更新权值
26.    W3 = W3 + lr * delta_W3
27.    W2 = W2 + lr * delta_W2
28.    W1 = W1 + lr * delta_W1

```

偏置的更新：对于偏置的更新没有想到更好的解决办法，因为求平均之后，得到的是一个或一行数据，在相加的时候无法与形状为(1, 2)的偏置b相加，所以就需要先转置再相加

```

29.

```

```

30.     b1 = b1.T
31.     b2 = b2.T
32.     b3 = b3.T
33.
34.     # 改变偏置值
35.     b3 = b3 + lr * delta_B3
36.     b2 = b2 + lr * delta_B2
37.     b1 = b1 + lr * delta_B1
38.
39.     b1 = b1.T
40.     b2 = b2.T
41.     b3 = b3.T

```

计算loss值：需要将隐藏层1输出将原来的`np.dot(x, W1)`改为`np.dot(W1, x.T)`，隐藏层2，输出层需要将A，W的位置互换

```

1. # 每训练 5000 次计算一次 loss 值
2. if idx_epoch % report == 0:
3.     # 隐藏层 1 输出
4.     A1 = sigmoid(np.dot(W1, X.T) + b1)
5.     # 隐藏层 2 输出
6.     A2 = sigmoid(np.dot(W2, A1) + b2)
7.     # 输出层输出
8.     A3 = sigmoid(np.dot(W3, A2) + b3)
9.     # 计算 loss 值
10.    print('A3:', A3)
11.    print('epochs:', idx_epoch, 'loss:', np.mean(np.square(T -
    A3) / 2))
12.    # 保存 loss 值
13.    loss.append(np.mean(np.square(T - A3) / 2))

```

最后的结果输出：和计算loss值一样的修改

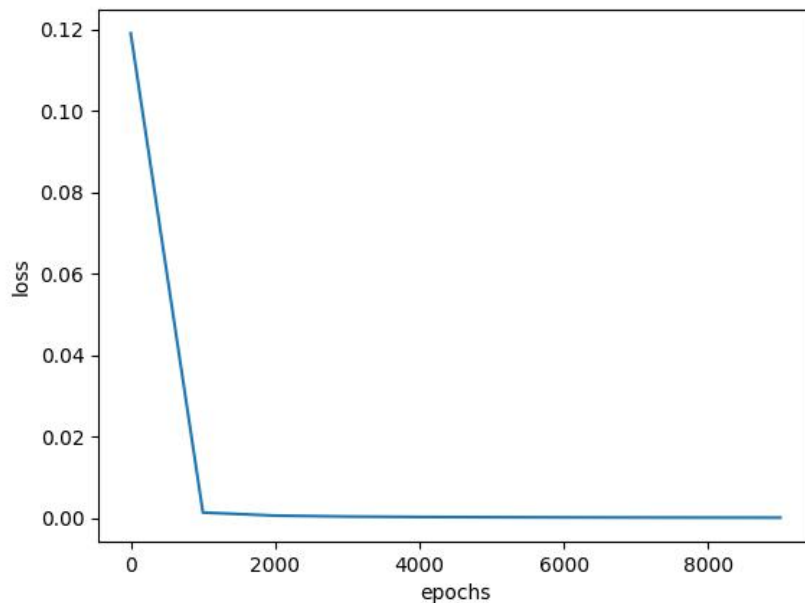
```

1. # 隐藏层 1 输出
2. A1 = sigmoid(np.dot(W1, X.T) + b1)
3. # 隐藏层 2 输出
4. A2 = sigmoid(np.dot(W2, A1) + b2)
5. # 输出层输出
6. A3 = sigmoid(np.dot(W3, A2) + b3)
7. print('output:')
8. print(A3)

```

实验结果：

训练周期数与loss的关系图：



```
问题4 x
↑
↓
↶
↷
A3: [[0.97942139 0.98218732]]
epochs: 6000 loss: 0.0001851926465778963
A3: [[0.98104781 0.98363868]]
epochs: 7000 loss: 0.00015671956116513665
A3: [[0.98234811 0.98479536]]
epochs: 8000 loss: 0.00013569251387888473
A3: [[0.9834179 0.9857444]]
epochs: 9000 loss: 0.00011954703503537342
D:\GZH\Desktop\计算智能课程设计上\QQ\实验2反向传播\问题4.py:149: Matplotlib
plt.plot(range(0, epochs, report), loss)
D:\GZH\Desktop\计算智能课程设计上\QQ\实验2反向传播\问题4.py:152: Matplotlib
plt.show()
output:
[[0.9843169 0.98654005]]
predict:
1
```

运行结果和原始行向量的结果一致

## 五、小结与心得体会

通过本次实验了解了连续多层感知器的实现，以及多感知器和单感知器的输出处理，同时学到了批量梯度下降、随机梯度下降、小批量梯度下降的代码实现，以及它们的收敛性和将行向量转换为列向量的等价程序的实现

## 一、实验题目

### 实验三 基于神经网络的手写数字识别

## 二、实验目的

掌握神经网络的设计原理，熟练掌握神经网络的训练和使用方法，能够使用Python语言，针对手写数字分类的训练和使用，实现一个三层全连接神经网络模型。具体包括：

- 1) 实现三层神经网络模型来进行手写数字分类，建立一个简单而完整的神经网络工程。通过本实验理解神经网络中基本模块的作用和模块间的关系，为后续建立更复杂的神经网络实验奠定基础。
- 2) 利用Python实现神经网络基本单元的前向传播（正向传播）和反向传播，加深对神经网络中基本单元的理解，包括全连接层、激活函数、损失函数等基本单元。
- 3) 利用Python实现神经网络的构建和训练，实现神经网络所使用的梯度下降算法，加深对神经网络训练过程的理解。

## 三、实验内容

### 总体设计

设计一个三层神经网络实现手写数字图像分类。该网络包含两个隐层和一个输出层，其中输入神经元个数由输入数据维度决定，输出层的神经元个数由数据集包含的类别决定，两个隐层的神经元个数可以作为超参数自行设置。对于手写数字图像的分类问题，输入数据为手写数字图像，原始图像一般可表示为二维矩阵（灰度图像）或三维矩阵（彩色图像），在输入神经网络前会将图像矩阵调整为一维向量作为输入。待分类的类别数一般是提前预设的，如手写数字包含0至9共10个类别，则神经网络的输出神经元个数为10。

## 四、实验结果与分析

1) 请在代码中有TODO的地方填空，将程序补充完整，在报告中写出相应代码，并给出自己的理解。

全连接层的概念：对 $n-1$ 层和 $n$ 层而言， $n-1$ 层的任意一个节点，都和第 $n$ 层所有节点有连接。在mlp多层感知机模型中输入层，隐藏层，输出层，不同层之间是全连接的。

它的作用：输入 $m$ 维向量与权重相乘后再与偏置相加得到 $n$ 维输出向量。

隐藏层的作用：增加网络的深度和复杂度，使训练得出的结果更为准确。

ReLU层的作用：根据relu激活函数的特点，当输入小于0则输出等于0，输入大于0输出就等于输入值。

Softmax损失层在本示例中计算的是前向传播的损失，具体体现在mnist\_mlp\_cpu.py中的forward函数有返回值，backward函数没有返回值，因此可以确定计算的是前向传播的损失。

### 数据加载和预处理模块

```
1. def load_mnist(self, file_dir, is_images='True'):  
2.     # Read binary data  
3.     bin_file = open(file_dir, 'rb')  
4.     bin_data = bin_file.read()
```



```

5.     bin_file.close()
6.     # Analysis file header
7.     if is_images:
8.         # Read images
9.         fmt_header = '>iiii'
10.        magic, num_images, num_rows, num_cols = struct.unpack_from(f
mt_header, bin_data, 0)
11.    else:
12.        # Read labels
13.        fmt_header = '>ii'
14.        magic, num_images = struct.unpack_from(fmt_header, bin_data,
0)
15.        num_rows, num_cols = 1, 1
16.        data_size = num_images * num_rows * num_cols
17.        mat_data = struct.unpack_from('>' + str(data_size) + 'B', bin_da
ta, struct.calcsize(fmt_header))
18.        mat_data = np.reshape(mat_data, [num_images, num_rows * num_cols]
)
19.        print('Load images from %s, number: %d, data shape: %s' % (file_
dir, num_images, str(mat_data.shape)))
20.        return mat_data
21.

```

按照指导书提示，直接按照train\_images的代码套用，load\_mnist()根据True或False 辨认images和labels

```

22. def load_data(self):
23.     # TODO: 调用函数 load_mnist 读取和预处理 MNIST 中训练数据和测试数据
    的图像和标记,True 为图像, False 为标记
24.     print('Loading MNIST data from files...')
25.     train_images = self.load_mnist(os.path.join(MNIST_DIR, TRAIN_DAT
A), True)
26.     train_labels = self.load_mnist(os.path.join(MNIST_DIR, TRAIN_LAB
EL), False)
27.     test_images = self.load_mnist(os.path.join(MNIST_DIR, TEST_DATA),
True)
28.     test_labels = self.load_mnist(os.path.join(MNIST_DIR, TEST_LABEL)
, False)
29.     self.train_data = np.append(train_images, train_labels, axis=1)
30.     self.test_data = np.append(test_images, test_labels, axis=1)
31.     print("train_images,train_labels,test_images,test_labels,self.tr
ain_data.shape,self.test_data.shape",
32.           train_images.shape, train_labels.shape, test_images.shape,
test_labels.shape, self.train_data.shape,
33.           self.test_data.shape)
34.     # self.test_data = np.concatenate((self.train_data, self.test_da
ta), axis=0)

```

基本单元模块:

全连接层: 根据指导书背景知识的全连接层介绍的数学推导公式，完善了下面代码

```

1. class FullyConnectedLayer(object):
2.     def __init__(self, num_input, num_output): # 全连接层初始化
3.         self.num_input = num_input

```

```

4.         self.num_output = num_output
5.         print('\tFully connected layer with input %d, output %d.' %
      (self.num_input, self.num_output))
6.
7.         def init_param(self, std=0.01): # 参数初始化
8.             self.weight = np.random.normal(loc=0.0, scale=std, size=(sel
          f.num_input, self.num_output))
9.             self.bias = np.zeros([1, self.num_output])
10.
11.        def forward(self, input): # 前向传播计算
12.            start_time = time.time()
13.            self.input = input
14.            # TODO: 全连接层的前向传播, 计算输出结果
15.            # 根据指导书公式(3.3)
16.            self.output = np.matmul(self.input, self.weight) + self.bias
17.
18.            return self.output
19.
20.        def backward(self, top_diff): # 反向传播的计算
21.            # TODO: 全连接层的反向传播, 计算参数梯度和本层损失
22.            # 根据指导书公式(3.4)
23.            self.d_weight = np.dot(self.input.T, top_diff)
24.            self.d_bias = top_diff
25.            bottom_diff = np.dot(top_diff, self.weight.T)
26.            return bottom_diff
27.
28.        def update_param(self, lr): # 参数更新
29.            # TODO: 对全连接层参数利用参数进行更新
30.            # 七字真言  $X = X - \text{学习率} * \text{导数}$ 
31.            self.weight = self.weight - lr * self.d_weight
32.            self.bias = self.bias - lr * self.d_bias
33.
34.        def load_param(self, weight, bias): # 参数加载
35.            assert self.weight.shape == weight.shape
36.            assert self.bias.shape == bias.shape
37.            self.weight = weight
38.            self.bias = bias
39.
40.        def save_param(self): # 参数保存
41.            return self.weight, self.bias

```

ReLU层: 根据指导书背景知识的ReLU层介绍的数学推导公式, 完善了下面代码

```

1. class ReLULayer(object):
2.     def __init__(self):
3.         print('\tReLU layer.')
4.
5.     def forward(self, input): # 前向传播的计算
6.         start_time = time.time()
7.         self.input = input
8.         # TODO: ReLU 层的前向传播, 计算输出结果
9.         # 根据指导书公式(3.5)
10.        output = np.maximum(0, self.input)
11.        return output
12.
13.    def backward(self, top_diff): # 反向传播的计算
14.        # TODO: ReLU 层的反向传播, 计算本层损失

```

```

15.         bottom_diff = top_diff
16.         bottom_diff[self.input < 0] = 0
17.         return bottom_diff

```

Softmax损失层：根据指导书背景知识的Softmax损失层介绍的数学推导公式，完善了下面代码

```

1. class SoftmaxLossLayer(object):
2.     def __init__(self):
3.         print('\tSoftmax loss layer.')
4.
5.     def forward(self, input): # 前向传播的计算
6.         # TODO: softmax 损失层的前向传播，计算输出结果
7.         input_max = np.max(input, axis=1, keepdims=True)
8.         input_exp = np.exp(input - input_max)
9.         # 根据指导书公式(3.11)
10.        self.prob = input_exp / np.tile(np.sum(input_exp, axis=1), (
11.            10, 1)).T
12.        return self.prob
13.
14.    def get_loss(self, label): # 前向传播的计算损失
15.        self.batch_size = self.prob.shape[0]
16.        self.label_onehot = np.zeros_like(self.prob)
17.        self.label_onehot[np.arange(self.batch_size), label] = 1.0
18.        loss = -
19.            np.sum(np.log(self.prob) * self.label_onehot) / self.batch_size
20.        return loss
21.
22.    def backward(self): # 反向传播的计算损失
23.        # TODO: softmax 损失层的反向传播，计算本层损失
24.        # 根据指导书公式(3.13)
25.        bottom_diff = (self.prob -
26.            self.label_onehot) / self.batch_size
27.        return bottom_diff

```

### 网络结构模块：

建立网络结构：按照指导书背景知识的图3.1，三层全连接神经网络结构图，完善了下面的代码

```

1. def build_model(self): # 建立网络结构
2.     # TODO: 建立三层神经网络结构
3.     print('Building multi-layer perception model...')
4.     self.fc1 = FullyConnectedLayer(self.input_size, self.hidden1)
5.     self.relu1 = ReLULayer()
6.     self.fc2 = FullyConnectedLayer(self.hidden1, self.hidden2)
7.     self.relu2 = ReLULayer()
8.     self.fc3 = FullyConnectedLayer(self.hidden2, self.out_classes)
9.     self.softmax = SoftmaxLossLayer()
10.    self.update_layer_list = [self.fc1, self.fc2, self.fc3]

```

网络训练模块：按照指导书背景知识的图3.1，三层全连接神经网络结构图，正向推导，反向推导，完善了下面的代码：

```

1. def forward(self, input): # 神经网络的前向传播

```

```

2.      # TODO: 神经网络的前向传播
3.      h1 = self.fc1.forward(input)
4.      h1 = self.relu1.forward(h1)
5.      h2 = self.fc2.forward(h1)
6.      h2 = self.relu2.forward(h2)
7.      h3 = self.fc3.forward(h2)
8.      prob = self.softmax.forward(h3)
9.      return prob
10.
11.     def backward(self): # 神经网络的反向传播
12.         # TODO: 神经网络的反向传播
13.         dloss = self.softmax.backward()
14.         dh3 = self.fc3.backward(dloss)
15.         dh2 = self.relu2.backward(dh3)
16.         dh2 = self.fc2.backward(dh2)
17.         dh1 = self.relu1.backward(dh2)
18.         dh1 = self.fc1.backward(dh1)

```

2) `mlp.load_data()` 执行到最后时, `train_images`、`train_labels`、`test_images`、`test_labels` 的维度是多少? 即多少行多少列, 用  $(x, y)$  来表示。`self.train_data` 和 `self.test_data` 的维度是多少?

```

train_images : (60000, 784)
train_labels : (60000, 1)
test_images : (10000, 784)
test_labels : (10000, 1)
self.train_data : (60000, 785)
self.test_data : (10000, 785)

```

3) 本案例中的神经网络一共有几层? 每层有多少个神经元? 如果要增加或减少层数, 应该怎么做 (简单描述即可不用编程)? 如果要增加或减少某一层的节点, 应该怎么做 (简单描述)? 如果要把 `softmax` 换成 `sigmoid`, 应该怎么做 (简单描述)? 这种替换合理么?

- a) 1个输入层, 2个隐藏层, 1个输出层一共有4层神经网络
- b) 输入层784个神经元, 隐藏层`hidden1`有32个神经元, 隐藏层`hidden2`有16个神经元, 输出层有10个神经元;
- c) 要增加或者减少层数, 在`init`函数的参数里增加层数的参数, 然后再在`build_model`通过`self.fc = FullyConnecte dLayer(参数)`把新增加的层数构建到网络; 删除就先在`init`里把要删除的层的参数删除, 再在`build_model`把构建的相应层删掉。
- d) 增加或减少某一层的节点, 只需要修改`init`函数的参数

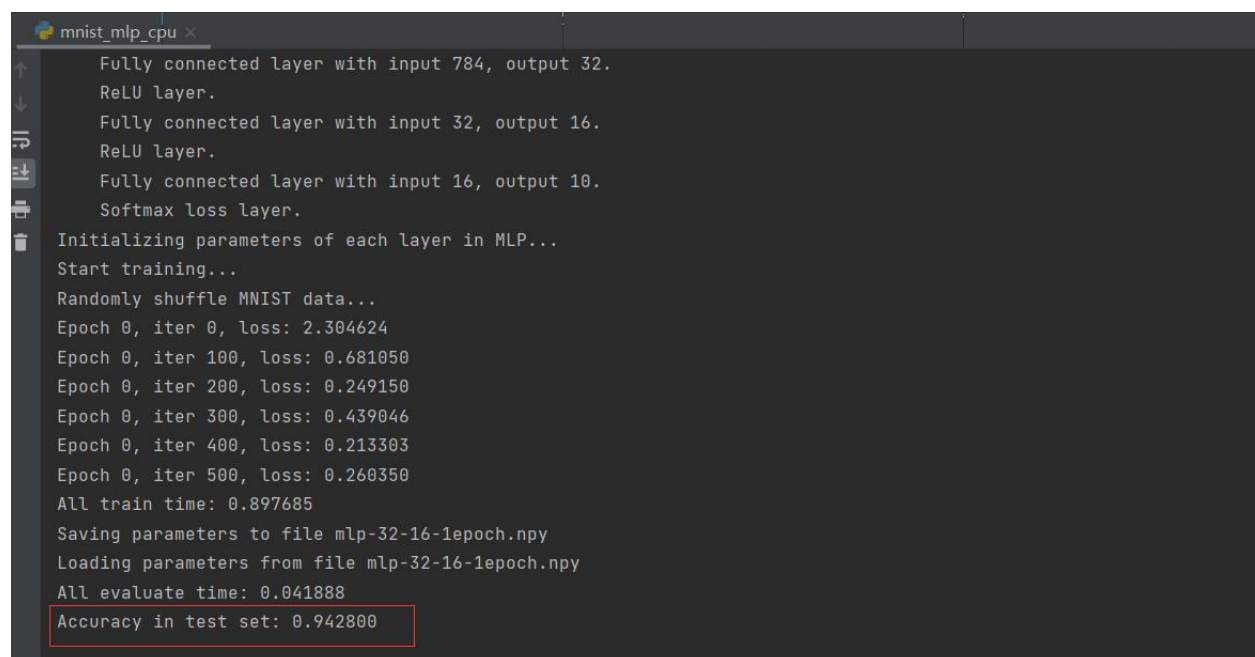
e) 把softmax 换成 sigmoid, 首先要在layers\_1文件中把SoftmaxLossLayer代码改成sigmoid求损失的SigmoidLossLayer代码; 再在minst\_mlp\_cpu文件里导入的SigmoidLossLayer, 替换掉SoftmaxLossLayer; 最后在build\_model里把self.softmax = SoftmaxLossLayer()改为self.sigmoid = SigmoidLossLayer

不合理, Softmax对于像手写数字识别这种多类别分类问题的处理方式比sigmoid的处理方式要好, 多类别分类的正确结果只有一个, 如果用sigmoid来进行处理, 正确结果可能出现多个

4)在train()函数中, max\_batch = self.train\_data.shape[0] // self.batch\_size这一句的意义是什么? self.shuffle\_data()的意义是什么?

- self.train\_data.shape[0]=60000, self.batch\_size=100将数据分为600个组。
- 调用shuffle\_data()将数据的顺序打乱, 让实验结果更具可靠性

5)最终evaluate()函数输出的Accuracy in test set是多少? 请想办法提高该数值。一次运行的值0.942800, 多次运行发现这个值恒定在0.92-0.95之间, 有80分的标准



```
mnist_mlp_cpu x
Fully connected layer with input 784, output 32.
ReLU layer.
Fully connected layer with input 32, output 16.
ReLU layer.
Fully connected layer with input 16, output 10.
Softmax loss layer.
Initializing parameters of each layer in MLP...
Start training...
Randomly shuffle MNIST data...
Epoch 0, iter 0, loss: 2.304624
Epoch 0, iter 100, loss: 0.681050
Epoch 0, iter 200, loss: 0.249150
Epoch 0, iter 300, loss: 0.439046
Epoch 0, iter 400, loss: 0.213303
Epoch 0, iter 500, loss: 0.260350
All train time: 0.897685
Saving parameters to file mlp-32-16-1epoch.npy
Loading parameters from file mlp-32-16-1epoch.npy
All evaluate time: 0.041888
Accuracy in test set: 0.942800
```

增加隐藏层神经元的个数和迭代次数

将隐藏层1, 2的神经元个数分别增加到512, 256, 迭代次数增加到10, 多次运行后正确率均高于98%, 有100分的标准

```
if __name__ == '__main__':  
    h1, h2, e = 512, 256, 10  
    mlp = MNIST_MLP(hidden1=h1, hidden2=h2, max_epoch=e)  
    mlp.load_data()  
    mlp.build_model()  
    mlp.init_model()  
  
    MNIST_MLP > evaluate() > for idx in range(self.test_data...  
mnist_mlp_cpu  
Epoch 9, iter 100, loss: 0.002931  
Epoch 9, iter 200, loss: 0.002931  
Epoch 9, iter 300, loss: 0.002066  
Epoch 9, iter 400, loss: 0.001531  
Epoch 9, iter 500, loss: 0.000886  
All train time: 65.145766  
Saving parameters to file mlp-512-256-10epoch.npy  
Loading parameters from file mlp-512-256-10epoch.npy  
All evaluate time: 0.196553  
Accuracy in test set: 0.983200
```

## 五、小结与心得体会

这个实验是整个课程设计中最难的一个, 刚开始做毫无思路, 在仔细阅读指导书后, 发现需要补充的代码是根据指导书中的数学公式进行编写, 而且整个过程会遇到很多问题, 非常考验人的意志。补充好代码后, 并没有对每个代码模块的功能进行思考, 只当自己完成了, 导致在测试时, 面对老师的提问, 对于细节问题, 我回答不上。课下, 高祥玉同学向我请教实验三的做法, 我和她一起讨论并对实验三进行了深度思考, 在本示例代码中全连接层的作用是什么, 为什么有两个隐藏层? ReLR层的作用是什么? Softmax损失层求前向传播的损失还是反向传播的损失? (在实验报告的问题一中已说明)

## 一、实验题目

实验四基于传递闭包的模糊聚类

## 二、实验目的

掌握建立模糊等价矩阵的方法，会求传递闭包矩阵；掌握利用传递闭包进行模糊聚类的一般方法；会使用Python进行模糊矩阵的有关运算。

## 三、实验内容

### 实验原理/运用的理论知识

“聚类”就是按照一定的要求和规律对事物进行区分和分类的过程，在这一过程中没有任何关于分类的先验知识，仅靠事物间的相似性作为类属划分的准则，属于无监督分类的范畴。传统的聚类分析是基于等价关系的一种硬划分，它把每个待辨识的对象根据等价关系严格地划分到某个类中，具有非此即彼的性质，因此这种分类类别界限是分明的。但是，现实的分类往往伴随着模糊性，即考虑的不是有无关系，而是关系的深浅程度，它们在形态和类属方面存在着中介性，适合进行软划分。人们用模糊的方法来处理聚类问题，并称之模糊聚类分析。常用的模糊聚类方法有(1)基于模糊等价矩阵的聚类方法(传递闭包法、Bool矩阵法)；(2)基于模糊相似矩阵的直接聚类法(直接聚类法、最大树法、编网法)；(3)基于目标函数的模糊聚类分析法。三种方法各有优缺点：直接聚类法不用计算模糊等价矩阵，计算量较小，可以比较直接地进行聚类。基于模糊等价矩阵的聚类方法较之编网法理论上更成熟，当矩阵阶数较高时，手工计算量较大，但在计算机上还是容易实现的。第三种聚类提出“聚类中心”的概念，这种方法可以得出某一对象在多大程度上属于某一类，但一般需要知道聚类数。

### 步骤和方法

传递闭包法聚类首先需要通过标定的模糊相似矩阵，然后求出包含矩阵的最小模糊传递矩阵，即的传递闭包，最后依据进行聚类。主要是以下几个步骤：

1. 得到特征指标矩阵 $X = (x_{ij})_{n \times m}$
2. 采用最大值规格化法将数据规格化，就是将 $x_{ij}$ 转换为 $\hat{x}_{ij}$ 即 $\hat{x}_{ij} = \frac{x_{ij}}{M_j}$ ，其中 $M_j = \max(x_{1j}, x_{2j}, \dots, x_{nj})$ 对原始数据进行正规化处理以后，变量的最大值为1，最小值为0，即数据在区间 $[0, 1]$ 内
3. 采用最大最小法构造模糊相似矩阵

$$r_{ij} = \frac{\sum_{k=1}^m (x_{ik} \wedge x_{jk})}{\sum_{k=1}^m (x_{ik} \vee x_{jk})}$$

4. 采用平方法合成传递闭包
5. 计算截集，得到模糊聚类结果

## 四、实验结果与分析

- 1) 为什么按最大最小法得到的一定是一个方阵？且一定是自反方阵？且一定是对称方



阵？最大最小法的数学表示：

$$r_{ij} = \frac{\sum_{k=1}^m (x_{ik} \wedge x_{jk})}{\sum_{k=1}^m (x_{ik} \vee x_{jk})}$$

自反方阵：方阵的主对角线的元素都为1

(1) 当  $i$  等于  $j$  时，元素与自身的关系相等， $r_{ij}$  等于1，满足方阵的自反性，则一定是自反方阵

对称方阵：方阵的转置与方阵的本身相等

(2) 当  $i, j$  的值互换时，计算方法的值不变， $r_{ij} = r_{ji}$ ，满足方阵的对称性，则一定是对称方阵

$$r_{ij} = \frac{\sum_{k=1}^m (x_{ik} \wedge x_{jk})}{\sum_{k=1}^m (x_{ik} \vee x_{jk})} = \frac{\sum_{k=1}^m (x_{jk} \wedge x_{ik})}{\sum_{k=1}^m (x_{jk} \vee x_{ik})} = r_{ji}$$

2) 为什么可以根据水平截集对数据进行分类？（提示：一个等价关系唯一确定一个划分）

模糊相似矩阵  $R$  满足自反性、对称性。又存在传递闭包矩阵，满足对称性，则  $R$  一定是模糊等价矩阵，对于  $\forall 0 \leq \lambda < \mu \leq 1$ ， $R_\mu$  所决定的分类中的每一个类是  $R_\lambda$  所决定的分类中的某一个子类。

$\alpha$  水平截集是指隶属度大于等于  $\alpha$  的元素组成的集合，是所有关系的集合。关系做了分类，就可以根据  $\alpha$  的特征函数来确定划分。由于一个等价关系确定一个划分，对模糊等价矩阵进行水平截集划分，划分后的数据集的并必定为全集，数据集的交必定为空集。

模糊等价矩阵的定义：

设  $R = (r_{ij})_{n \times n}$  是  $n$  阶模糊方阵， $I$  是  $n$  阶单位方阵，则  $R$  满足

(1) 自反性： $I \leq R \Leftrightarrow r_{ii} = 1$

(2) 对称性： $R^T = R \Leftrightarrow r_{ij} = r_{ji}$

(3) 传递性： $R^2 \leq R \Leftrightarrow \max\{(r_{ik} \wedge r_{kj}) | 1 \leq k \leq n\} \leq r_{ij}$

3) 请解释代码72行中两个-1的含义：

```
return np.sort(np.unique(a).reshape(-1))[:, -1]
```

第一个-1是将方阵  $a$  的维度改为一维。

第二个-1是将排序后的水平截集逆序输出。

4) 在平方法的代码实现中，如何判断平方后的矩阵是否满足传递性？为什么可以这么判断？

```
def TransitiveClosure(a):  
    """  
    平方法合成传递闭包  
    """  
    a = FuzzySimilarMatrix(a) # 用模糊相似矩阵  
    c = a  
    while True:  
        m = c  
        c = MatrixComposition(MatrixComposition(a, c), MatrixComposition(a, c))  
        if (c == m).all(): # 闭包条件  
            return np.around(c, decimals=2) # 返回传递闭包, 四舍五入, 保留两位小数
```

```

        break
    else:
        continue

```

是根据原理  $R^2 \leq R \Leftrightarrow \max\{(r_{ik} \wedge r_{kj}) | 1 \leq k \leq n\} \leq r_{ij}$  进行判断，是否满足传递性的

5) 请修改代码，将最大最小法替换为算术平均最小法。这会改变最终的聚类结果么？

```

1. def ArrageSimilarMatrix(a):
2.     """
3.     用算术平均最小法构造得到模糊相似矩阵
4.     """
5.     a = MaxNormalization(a) # 用标准化后的数据
6.     c = np.zeros((a.shape[0], a.shape[0]), dtype=float)
7.     ssum = []
8.     mmin = []
9.     for i in range(c.shape[0]): # 遍历 c 的行
10.        for j in range(c.shape[0]): # 遍历 c 的行
11.            ssum.extend([(a[i, :] + a[j, :])]) # 求 i 行和 j 行的和
12.            mmin.extend([np.fmin(a[i, :], a[j, :])]) # 取 i 和 j 行
            的最大值,即求 i 行和 j 行的交
13.        for i in range(len(ssum)):
14.            ssum[i] = np.sum(ssum[i]) # 求累加
15.            mmin[i] = np.sum(mmin[i]) # 求交的和
16.        ssum = np.array(ssum).reshape(c.shape[0], c.shape[1]) # 变换为与
            c 同型的矩阵
17.        mmin = np.array(mmin).reshape(c.shape[0], c.shape[1]) # 变换为与
            c 同型的矩阵
18.        for i in range(c.shape[0]): # 遍历 c 的行
19.            for j in range(c.shape[1]): # 遍历 c 的列
20.                c[i, j] = 2 * mmin[i, j] / ssum[i, j] # 赋值相似度
21.        return c

```

最大最小法的结果：

```
实验四基于传递闭包的模糊聚类 ×
[0.89 1. 0.84 0.88 1. ]

用最大最小法构造得到模糊相似矩阵
[[1. 0.9 0.86 0.91]
 [0.9 1. 0.81 0.84]
 [0.86 0.81 1. 0.92]
 [0.91 0.84 0.92 1. ]]

平方法合成传递闭包
[[1. 0.9 0.91 0.91]
 [0.9 1. 0.9 0.9 ]
 [0.91 0.9 1. 0.92]
 [0.91 0.9 0.92 1. ]]
1 [0.9 0.91 0.92 1. ]
2 [0.9 0.91 0.92 1. ]

水平截集为
[1. 0.92 0.91 0.9 ]
1 [0.9 0.91 0.92 1. ]
2 [0.9 0.91 0.92 1. ]

模糊聚类结果
[[[0], [1], [2], [3]], [[0], [1], [2, 3]], [[0, 2, 3], [1]], [0, 1, 2, 3]]
```

算术平均最小法的结果：

```
问题5 ×
用算术平均最小构造得到模糊相似矩阵
[[1. 0.95 0.92 0.95]
 [0.95 1. 0.9 0.91]
 [0.92 0.9 1. 0.96]
 [0.95 0.91 0.96 1. ]]

平方法合成传递闭包
[[1. 0.95 0.95 0.95]
 [0.95 1. 0.95 0.95]
 [0.95 0.95 1. 0.96]
 [0.95 0.95 0.96 1. ]]
1 [0.95 0.96 1. ]
2 [0.95 0.96 1. ]

水平截集为
[1. 0.96 0.95]
1 [0.95 0.96 1. ]
2 [0.95 0.96 1. ]

模糊聚类结果
[[[0], [1], [2], [3]], [[0], [1], [2, 3]], [0, 1, 2, 3]]
```

两者的模糊聚类结果有差异，说明将模糊相似矩阵的构造方法改变，会改变最终的聚类结果

## 五、小结与心得体会

熟悉了模糊聚类分析的步骤：

1. 建立数据矩阵，需要将数据矩阵规格化，掌握了最大值规格化的代码实现；
2. 建立模糊相似矩阵，掌握了用最大最小法和算术平均最小法去建立模糊相似矩阵、采用平方法合成传递闭包，以及水平截集能够对数据进行分类的原因。

通过理论与实践相结合，让我对模糊聚类的理解更加深刻

在完成实验四的过程中，遇到的困扰，都源自于自己的线代知识和离散数学功底不牢固

## 一、实验题目

实验五遗传算法求解无约束单目标优化问题

## 二、实验目的

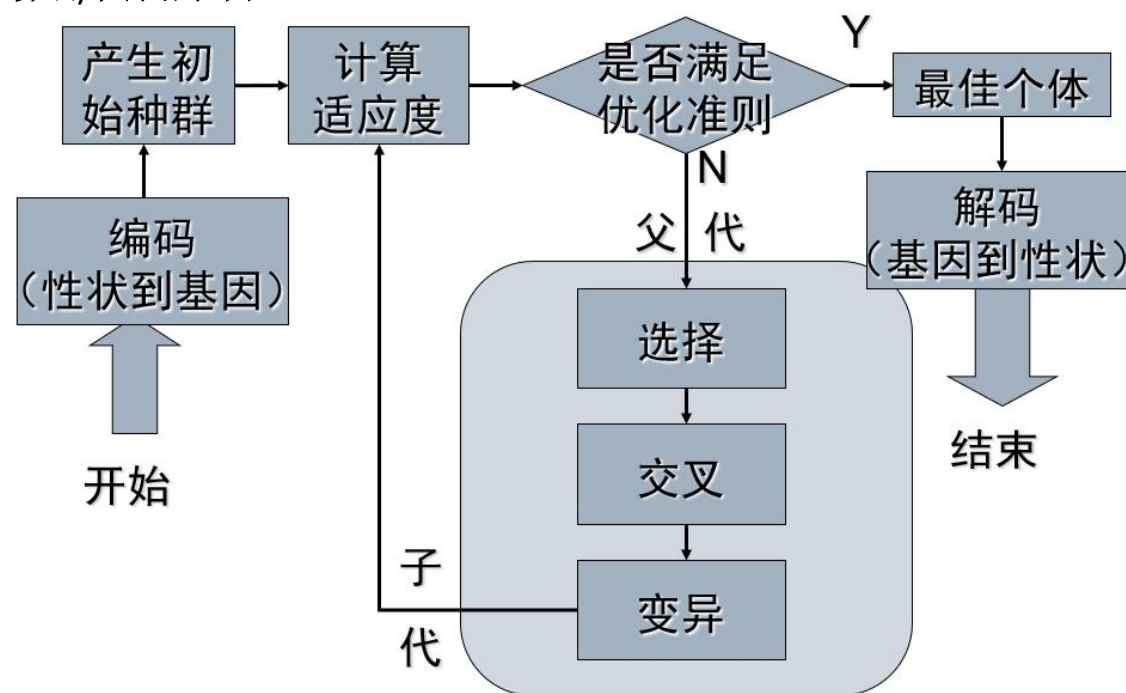
理解遗传算法原理，掌握遗传算法的基本求解步骤，包括选择、交叉、变异等，学会运用遗传算法求解无约束单目标优化问题。

## 三、实验内容

实验原理/运用的理论知识、

遗传算法(Genetic Algorithm)是借鉴生物界自然选择、适者生存遗传机制的一种随机搜索方法。遗传算法模拟了进化生物学中的遗传、突变、自然选择以及杂交等现象，是进化算法的一种。对于一个最优化问题，一定数量的候选解(每个候选解称为一个个体)的抽象表示(也称为染色体)的种群向更好的方向解进化，通过一代一代不断繁衍，使种群收敛于最适应的环境，从而求得问题的最优解。进化从完全随机选择的个体种群开始，一代一代繁殖、进化。在每一代中，整个种群的每个个体的适应度被评价，从当前种群中随机地选择多个个体(基于它们的适应度)，通过自然选择、优胜劣汰和突变产生新的种群，该种群在算法的下一次迭代中成为当前种群。传统上，解一般用二进制表示(0和1组成的串)。

算法/程序流程图



## 四、实验结果与分析

1) 代码第64行的语义是什么？两个[0]各自代表什么？最后newX有几个元

素？

```
1.         # 轮盘赌，根据随机概率选择出新的基因编码
2.         # 对于 each_rand 中的每个随机数，找到被轮盘赌中的那个染色体
3.         newX = np.array([chroms[np.where(probs_cum > rand)[0][0]]
4.                           ]
                           for rand in each_rand])
```

代码第64行的语义：

根据累积概率和轮盘赌的概率分布来选择要参与交叉的染色体

两个[0] 代表的含义：

对于一个二维数组，np.where(probs\_cum > rand)[0][0] 第一个[0]表示二维数组中的第一个一维数组，第二个[0]是该一维数组的首个元素。

最后newX有100个元素

2) 代码第70行的语义是什么？为什么要除以2再乘以2？reshape中的-1表示什么？

```
1. pairs = np.random.permutation(
2.     int(len(newX)*prob//2*2)).reshape(-1, 2) # 产生 6 个
    随机数，乱排一下，分成二列
```

在种群中选择随机配对率为0.6以下的六条染色体进行两两配对进行交叉  
确保pairs为偶数，为交叉作准备

reshape(-1, 2)中的-1表示一维长度自适应，对行数没有限制

3) 请结合Mutate函数的内容，详述变异是如何实现的。

- (1) 首先设置变异的概率
- (2) 利用随机数生成随机变异率
- (3) 遍历所有的染色体

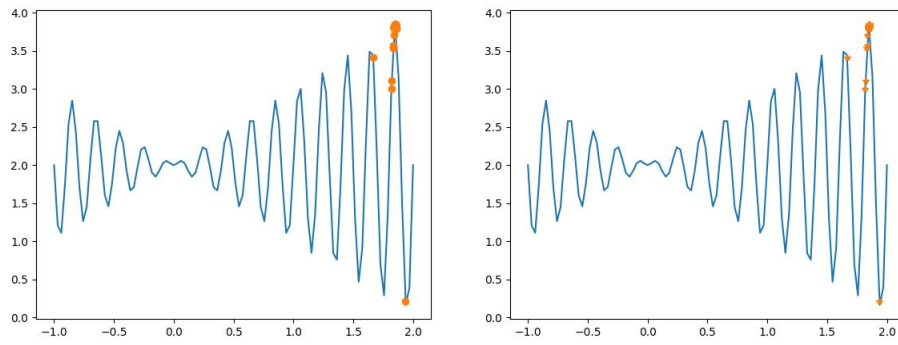
如果随机变异率<设定的变异率，则发生变异

在染色体上随机选择变异位置，对其进行取非运算得到新的染色体

- (4) 无论是否变异都要将染色体添加到新种群

4) 将代码第145行修改为newchroms = Select\_Crossover(chroms, fitness)，  
即不再执行变异，执行结果有什么不同，为什么会出现这种变化？

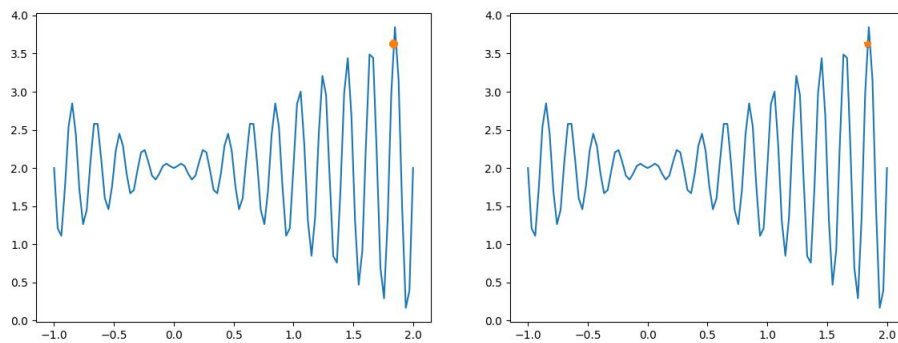
修改前的最终结果：



最大值是3.850268494119351

```
chrom=100011000111111100, dec= 0.65, fit=2.64
chrom=100011000111111100, dec= 0.65, fit=2.64
chrom=100011000111111100, dec= 0.65, fit=2.64
chrom=111101101011001010, dec= 1.89, fit=2.53
chrom=111001000100101100, dec= 1.68, fit=3.17
chrom=111101101011001010, dec= 1.89, fit=2.53
D:\GZH\Desktop\计算智能课程设计上传QQ\实验5遗传算法\实验五遗传算法求解无约束单目
plt.show()
D:\GZH\Desktop\计算智能课程设计上传QQ\实验5遗传算法\实验五遗传算法求解无约束单目
fig, (axs1, axs2) = plt.subplots(1, 2, figsize=(14, 5))
D:\GZH\Desktop\计算智能课程设计上传QQ\实验5遗传算法\实验五遗传算法求解无约束单目
plt.show()
3.850268494119351
```

修改后的最终结果:



最大值是3.634042084808603



```
chrom=100011000111111100, dec= 0.65, fit=2.64
chrom=111101101011001010, dec= 1.89, fit=2.53
chrom=111001000100101100, dec= 1.68, fit=3.17
chrom=111101101011001010, dec= 1.89, fit=2.53
D:\GZH\Desktop\计算智能课程设计上传QQ\实验5遗传算法\实验
plt.show()
D:\GZH\Desktop\计算智能课程设计上传QQ\实验5遗传算法\实验
fig, (axs1, axs2) = plt.subplots(1, 2, figsize=
D:\GZH\Desktop\计算智能课程设计上传QQ\实验5遗传算法\实验
plt.show()
3.634042084808603
```

修改后不再执行变异，选择的种群是比上次优秀而且也是根据相同的规则进行选择，没有经过变异无法向最优值逼近，导致执行结果无法取得最大值。

5) 轮盘让个体按概率被选择，对于适应度最高的个体而言，虽然被选择的概率高，但仍有可能被淘汰，从而在进化过程中失去当前最优秀的个体。一种改进方案是，让适应度最高的那个个体不参与选择，而是直接进入下一轮（直接晋级），这种方案被称为精英选择(elitist selection)。请修改Select部分的代码，实现这一思路。

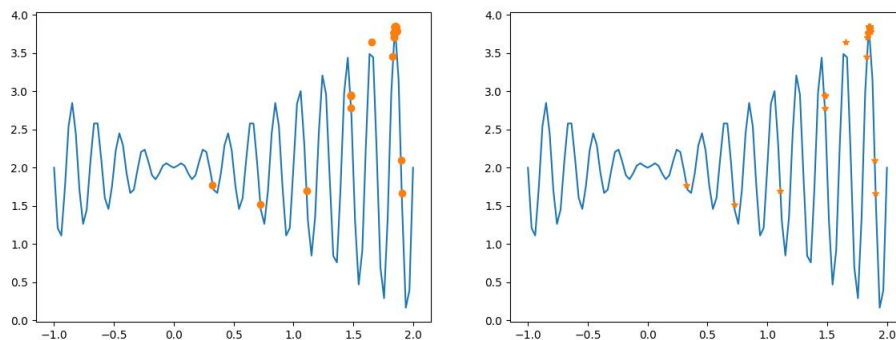
```
1. # 精英选择
2. def ElitistSelect_Crossover(chroms, fitness, prob=0.6): #
   选择和交叉
3.     probs = fitness / np.sum(fitness) # 各个个体被选择的概率
4.     probs_cum_maxIndex = np.where(probs == np.max(probs))[0]
   [0] # 求得适应度最大的个体的索引
5.     chroms_new = np.delete(chroms, probs_cum_maxIndex) # 去除适应度最高后的新种群
6.     fitness_new = np.delete(fitness, probs_cum_maxIndex) # 去除适应度最高后的新种群的适应度
7.     probs = fitness_new / np.sum(fitness_new) # 求新种群各个个体被选择的概率
8.     probs_cum = np.cumsum(probs) # 新种群的概率累加分布
9.     each_rand = np.random.uniform(size=len(fitness_new)) # 得到 9 个随机数, 0 到 1 之间
10.    # 轮盘赌, 根据随机概率选择出新的基因编码
11.    # 对于 each_rand 中的每个随机数, 找到被轮盘赌中的那个染色体
12.    newX = np.array([chroms_new[np.where(probs_cum > rand)[0][0]]
13.                      for rand in each_rand])
```

```

14.     newX = np.append(newX, chroms[probs_cum_maxIndex]) # 将
      适应度最大的个体添加到选择过后的种群
15.     # 繁殖，随机配对（概率为 0.6）
16.     # 6 这个数字怎么来的，根据遗传算法，假设有 10 个数，交叉概率为
      0.6，0 和 1 一组，2 和 3 一组。。。8 和 9 一组，每组扔一个 0 到 1 之间
      的数字
17.     # 这个数字小于 0.6 就交叉，则平均下来应有三组进行交叉，即 6 个
      染色体要进行交叉
18.     # -----交叉部分的代码未改动-----
19.     pairs = np.random.permutation(
20.         int(len(newX) * prob // 2 * 2)).reshape(-1, 2) # 产
      生 6 个随机数，乱排一下，分成二列
21.     center = len(newX[0]) // 2 # 交叉方法采用最简单的，中心交
      叉法
22.     for i, j in pairs:
23.         # 在中间位置交叉
24.         x, y = newX[i], newX[j]
25.         newX[i] = x[:center] + y[center:] # newX 的元素都是
      字符串，可以直接用+号拼接
26.         newX[j] = y[:center] + x[center:]
27.     return newX

```

实验结果：



最大值是3.85008835560526

```

chrom=111101101111010010, dec= 1.89, fit=2.35
chrom=011011000111111100, dec= 0.27, fit=2.21
chrom=011011000011001010, dec= 0.27, fit=2.23
chrom=011000100111111100, dec= 0.15, fit=1.85
chrom=100011000111010010, dec= 0.65, fit=2.64
chrom=100011000010100100, dec= 0.64, fit=2.62
chrom=100011000111111100, dec= 0.65, fit=2.64
chrom=111101101011001010, dec= 1.89, fit=2.53
chrom=111101101011001010, dec= 1.89, fit=2.53
chrom=111001000100101100, dec= 1.68, fit=3.17
D:\GZH\Desktop\计算智能课程设计上\实验5遗传算法\问题5.py:98: Matplo
plt.show()
D:\GZH\Desktop\计算智能课程设计上\实验5遗传算法\问题5.py:125: Matpl
fig, (axs1, axs2) = plt.subplots(1, 2, figsize=(14, 5))
D:\GZH\Desktop\计算智能课程设计上\实验5遗传算法\问题5.py:135: Matpl
plt.show()
最大值 3.85008835560526

```

6) 【选做】请借鉴示例代码，实现教材P57的例2.6.1，即用遗传算法求解下列二元函数的最大值。注意：不允许分开求解 $x_1 \cdot \sin(4\pi \cdot x_1)$ 和 $x_2 \cdot \sin(20\pi \cdot x_2)$ 的最大值再合并。

$$\begin{aligned}
 \max f(x_1, x_2) &= 21.5 + x_1 \cdot \sin(4\pi \cdot x_1) + x_2 \cdot \sin(20\pi \cdot x_2) \\
 \text{s. t. } -2.9 &\leq x_1 \leq 12.0 \\
 4.2 &\leq x_2 \leq 5.7
 \end{aligned}$$

完整代码：

```

1. import numpy as np
2.
3.
4. def fun(x1, x2):
5.     return 21.5 + x1 * np.sin(4 * np.pi * x1) + x2 * np.sin
   (20 * np.pi * x2)
6.
7.
8. X1s = np.linspace(-2.9, 12.0, 100)
9. X2s = np.linspace(4.2, 5.7, 100)
10. # 令随机数种子=0，确保每次取得相同的随机数
11. np.random.seed(0)
12.
13. # 初始化原始种群
14. # 在[-2.9,12.0]上以均匀分布生成 10 个浮点数，做为初始种群
15. x1_population = np.random.uniform(-2.9, 12.0, 10)
16. # 在[4.2,5.7]上以均匀分布生成 10 个浮点数，做为初识种群
17. x2_population = np.random.uniform(4.2, 5.7, 10)

```

```

18. for x1_pop, x2_pop, fit in zip(x1_population, x2_population,
    fun(x1_population, x2_population)):
19.     print("x1=%5.2f, x2=%5.2f, fit=%.2f" % (x1_pop, x2_pop,
        fit))
20.
21.
22. # 对 x1 进行编码
23. def x1_encode(population, _min=2.9, _max=12.0, scale=2 ** 2
    1, binary_len=21):
24.     normalized_data = (population - _min) / (_max -
        _min) * scale
25.     # 转成二进制编码
26.     binary_data = np.array([np.binary_repr(x, width=binary_
        len)
27.                             for x in normalized_data.astype
        (int)])
28.     return binary_data
29.
30.
31. # 对 x2 进行编码
32. def x2_encode(population, _min=4.2, _max=5.7, scale=2 ** 18,
    binary_len=18):
33.     normalized_data = (population - _min) / (_max -
        _min) * scale
34.     # 转成二进制编码
35.     binary_data = np.array([np.binary_repr(x, width=binary_
        len)
36.                             for x in normalized_data.astype
        (int)])
37.     return binary_data
38.
39.
40. # 对两个编码进行合并
41. def encode(population_x1, population_x2):
42.     choms_x1 = x1_encode(population_x1)
43.     choms_x2 = x2_encode(population_x2)
44.     binary_data = np.char.add(choms_x1, choms_x2)
45.     return binary_data
46.
47.
48. choms = encode(x1_population, x2_population) # 染色体英文
    (chromosome)
49.
50. for x1_pop, x2_pop, chrom, fit in zip(x1_population, x2_pop
    ulation, choms, fun(x1_population, x2_population)):
51.     print("x1=%.2f, x2=%.2f, chrom=%s, fit=%.2f" % (x1_pop,
        x2_pop, chrom, fit))
52.
53. # 对 X1 进行解码
54. def x1_decode(popular_gene, _min=-
    2.9, _max=12.0, scale=2 ** 21):
55.     return np.array([(int(x, base=2) / scale * 14.9) + _min
        for x in popular_gene])
56.
57. # 对 X2 进行解码

```

```

58. def x2_decode(popular_gene, _min=4.2, _max=5.7, scale=2 **
    18):
59.     return np.array([(int(x, base=2) / scale * 1.5) + _min
    for x in popular_gene])
60.
61. # 对种群进行切割，并调用 x1_decode()、x2_decode() 分别解码得到
    x1,x2
62. def decode(chroms_v):
63.     chroms_x1 = []
64.     chroms_x2 = []
65.     for arr in chroms_v:
66.         tchroms_x1 = arr[:21]
67.         tchroms_x2 = arr[21:]
68.         chroms_x1.append(tchroms_x1)
69.         chroms_x2.append(tchroms_x2)
70.     dechroms_x1 = x1_decode(chroms_x1)
71.     dechroms_x2 = x2_decode(chroms_x2)
72.     return dechroms_x1, dechroms_x2
73.
74.
75. # 将编码后的 chroms 解码并代入到适应度函数中
76. # 得到结果后进行个体评价
77. dechroms_x1, dechroms_x2 = decode(chroms)
78. fitness = fun(dechroms_x1, dechroms_x2)
79. for x1_pop, x2_pop, chrom, fit in zip(x1_population, x2_pop
    ulation, chroms,
80.                                     fitness):
81.     print("x1=%5.2f, x2=%5.2f, chrom=%s, fit=%.2f" %
82.           (x1_pop, x2_pop, chrom, fit))
83. fitness = fitness - fitness.min() + 0.000001
84. print("fitness-fitness.min()=", fitness)
85.
86.
87. def Select_Crossover(chroms, fitness, prob=0.6):
88.     probs = fitness / np.sum(fitness)
89.     probs_cum = np.cumsum(probs)
90.     each_rand = np.random.uniform(size=len(fitness))
91.
92.     newX = np.array([chroms[np.where(probs_cum > rand)[0][0]]
93.                       ])
94.     for rand in each_rand:
95.         pairs = np.random.permutation(
96.             int(len(newX) * prob // 2 * 2)).reshape(-1, 2) # 产
            生 6 个随机数，乱排一下，分成二列
97.         center = len(newX[0]) // 2 # 交叉方法采用最简单的，中心交
            叉法
98.         for i, j in pairs:
99.             # 在中间位置交叉
100.            x, y = newX[i], newX[j]
101.            newX[i] = x[:center] + y[center:] # newX 的元
            素都是字符串，可以直接用+号拼接
102.            newX[j] = y[:center] + x[center:]
103.            return newX
104.

```

```

105.     chroms = Select_Crossover(chroms, fitness)
106.
107.     dechroms = decode(chroms)
108.     fitness = fun(dechroms_x1, dechroms_x2)
109.
1. def Mutate(chroms: np.array):
2.     prob = 0.1 # 变异的概率
3.     clen = len(chroms[0]) # chroms[0]="111101101 000010110
    " 字符串的长度=18
4.     m = {'0': '1', '1': '0'} # m是一个字典, 包含两对: 第一对
    0是key而1是value; 第二对1是key而0是value
5.     newchroms = [] # 存放变异后的新种群
6.     each_prob = np.random.uniform(size=len(chroms)) # 随机
    10个数
7.
8.     for i, chrom in enumerate(chroms): # enumerate的作用是
    整一个i出来
9.         if each_prob[i] < prob: # 如果要进行变异(i的用处在这
    里)
10.             pos = np.random.randint(clen) # 从18个位置随机
    中找一个位置, 假设是7
11.             # 0~6保持不变, 8~17保持不变, 仅将7号翻转, 即0改
    为1, 1改为0。注意chrom中字符不是1就是0
12.             chrom = chrom[:pos] + m[chrom[pos]] + chrom[pos
    + 1:]
13.             newchroms.append(chrom) # 无论if是否成立, 都在
    newchroms中增加chroms的这个元素
14.     return np.array(newchroms) # 返回变异后的种群
15.
16.
17. newchroms = Mutate(chroms)
18.
19. # 上述代码只是执行了一轮, 这里反复迭代
20. np.random.seed(0) #
21. x1_population = np.random.uniform(-2.9, 12.0, 100) # 这次多
    找一些点
22. x2_population = np.random.uniform(4.2, 5.7, 100)
23. chroms = encode(x1_population, x2_population)
24.
25. for i in range(1000):
26.     dechroms_x1, dechroms_x2 = decode(chroms)
27.     fitness = fun(dechroms_x1, dechroms_x2)
28.     fitness = fitness - fitness.min() + 0.000001 # 保证所有
    的都为正
29.     newchroms = Mutate(Select_Crossover(chroms, fitness))
30.     chroms = newchroms
31.
32. dechroms_x1, dechroms_x2 = decode(chroms)
33. fitness = fun(dechroms_x1, dechroms_x2)
34. print("最大值: ", np.max(fitness))

```

实验结果：最大值是38.6825476826344

```
x1= 6.72, x2= 4.33, chrom=101001010101100101010000101100100111000,, fit=27.73
x1= 3.62, x2= 4.23, chrom=011100000000010110110000001010010110100,, fit=29.11
x1=10.39, x2= 5.45, chrom=111001000100101100111110101010010011010,, fit=11.61
x1=11.46, x2= 5.37, chrom=111101101011001010011110001110011010101,, fit=11.06
x1= 2.81, x2= 5.51, chrom=011000100010100100111110111101011100100,, fit=21.20
fitness-fitness.min()= [4.87404710e+00 7.79591741e+00 1.49688020e+01 8.71304459e+00
9.12164349e+00 1.66692528e+01 1.80488886e+01 5.45714715e-01
1.00000000e-06 1.01379107e+01]
最大值: 38.6825476826344
```

## 五、小结与心得体会

通过此次实验熟悉了如何利用遗传算法求一元函数最大值的代码实现，同时也理解了精英选择策略的算法思想，以及如何利用遗传算法求解多元函数的最大值。