湖南科技大学计算机科学与工程学院

综合实践能力创新实训(2) 课程设计报告

**专业班级：**

**姓 名：**

**学 号：**

**指导教师：**

**时 间**：

**地 点**：

**日**

**月**

**年**

**签名：**

**指导教师评语：**

**成绩： 等级：**

1. **实验题目**

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**

###### 关键代码

# 加载用到的库  
import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.datasets import load\_iris #仅用于加

# 画图时的中文支持  
plt.rcParams['font.sans-serif'] = ['SimHei'] # 用来正常显示中文标签  
plt.rcParams['axes.unicode\_minus'] = False # 用来正常显示负号  
  
#加载数据集  
iris = load\_iris()  
  
# 通过画图了解三种鸢尾花的分布  
plt.clf()  
plt.xlim(0, 7)#x轴上的最小值和最大值  
plt.ylim(0, 4)  
plt.title(u'iris数据集 萼片', fontsize=15)  
X=iris.data[:,0:2]  
  
plt.xlabel('petal length 萼片长度', fontsize=13)  
plt.ylabel('petal width 萼片宽度', fontsize=13)  
plt.plot(X[:50, 0], X[:50, 1], 'o', color='blue', label='Setosa山鸢尾')  
plt.plot(X[50:100, 0], X[50:100, 1], 'o', color='orange', label='Versicolour变色鸢尾')  
plt.plot(X[100:150, 0], X[100:150, 1], 'o', color='red', label='Virginica维吉尼亚鸢尾')  
plt.legend()  
plt.show()  
plt.pause(3)   
plt.clf()  
plt.xlim(0, 7)#x轴上的最小值和最大值  
plt.ylim(0, 3)  
plt.title(u'iris数据集 花瓣', fontsize=15)  
X=iris.data[:,2:4]  
plt.xlabel('petal length 花瓣长度', fontsize=13)  
plt.ylabel('petal width 花瓣宽度', fontsize=13)  
plt.plot(X[:50, 0], X[:50, 1], 'o', color='blue', label='Setosa山鸢尾')  
plt.plot(X[50:100, 0], X[50:100, 1], 'o', color='orange', label='Versicolour变色鸢尾')  
plt.plot(X[100:150, 0], X[100:150, 1], 'o', color='red', label='Virginica维吉尼亚鸢尾')  
plt.legend()  
plt.show()  
plt.pause(3)   
# 从图中大致可以看出，花瓣长度和花瓣宽度与鸢尾花类型间有较好的线性关系，使用花瓣数据来划分鸢尾花类型效果更好。  
X=np.c\_[np.ones(100),iris.data[:100,2:4]]  
# 得到真值T，从数据集中得到真值  
T=iris.target[:100].reshape(100,1)  
# 将T中所有不等于1的元素赋值为-1，以契合sign函数  
T[T!=1] = -1  
# 权值初始化，3行1列，即w0 w1 w2   
# 直接赋初值，不使用随机数  
W = np.array([[1],  
 [1],  
 [1]])  
  
# 学习率设置  
lr = 1  
# 神经网络输出  
Y = 0

## 学习算法  
  
#训练感知机模型  
# 更新一次权值  
def train():  
 # 使用全局变量W  
 global W  
 # 同时计算100个数据的预测值  
 # Y的形状为(100,1)-100行1列  
 Y = np.sign(np.dot(X,W))  
 # T - Y得到100个的标签值与预测值的误差E。形状为(100,1)  
 E = T - Y  
 # X的形状为(100,3)  
 # X.T表示X的转置矩阵，形状为(3,100)  
 # 我们一共有100个数据，每个数据3个特征的值。定义第i个数据的第j个特征值为xij  
 # 如第1个数据，第2个值为x12  
 # X.T.dot(E)为一个3行1列的数据：  
 # 第1行等于：x0\_0×e0+x1\_0×e1+x2\_0×e2+x3\_0×e3+...+x99\_0×e99，它会调整权值W0  
 # 第2行等于：x0\_1×e0+x1\_1×e1+x2\_1×e2+x3\_1×e3+...+x99\_1×e99，它会调整权值W1  
 # 第3行等于：x0\_2×e0+x1\_2×e1+x2\_2×e2+x3\_2×e3+...+x99\_2×e99，它会调整权值W2  
 # X.shape表示X的形状X.shape[0]得到X的行数，表示有多少个数据  
 # X.shape[1]得到列数，表示每个数据有多少个特征值。  
 delta\_W = lr \* (X.T.dot(E)) / X.shape[0]  
 W = W + delta\_W  
  
## 画图函数  
def draw():  
 plt.clf()  
 plt.xlim(0, 6)#x轴上的最小值和最大值  
 plt.ylim(0, 2)#y轴上的最小值和最大值  
 plt.title(u'Perceptron感知器 epoch:%d\n W0:%f W1:%f W2:%f' %(i+1,W[0],W[1],W[2]), fontsize=15)  
   
 plt.xlabel('petal length 花瓣长度', fontsize=13)  
 plt.ylabel('petal width 花瓣宽度', fontsize=13)  
 plt.plot(X[:50, 1], X[:50, 2], 'o', color='red', label='Setosa山鸢尾')  
 # 用黄色的点来画出负样本  
 plt.plot(X[50:100, 1], X[50:100, 2], 'o', color='blue', label='Versicolour变色鸢尾')  
 plt.plot(2.5, 1, '+', color='black', label='待预测点')  
 k = - W[1] / W[2]  
 d = -W[0] / W[2]  
 # 设定两个点  
 xdata = (0,6)  
 # 通过两个点来确定一条直线，用红色的线来画出分界线  
 plt.plot(xdata,xdata \* k + d,'black', linewidth=3)  
 plt.legend()  
   
 # 生成决策面   
 from matplotlib.colors import ListedColormap #绘制决策面两边的颜色，不要求掌握  
 # 生成x,y的数据  
 n = 256  
 xx = np.linspace(0, 6, n)  
 yy = np.linspace(0, 2, n)  
 # 把x,y数据生成mesh网格状的数据，因为等高线的显示是在网格的基础上添加上高度值  
 XX, YY = np.meshgrid(xx, yy)  
 # 填充等高线  
 colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')   
 cmap = ListedColormap(colors[:len(np.unique(np.sign(W[0]+W[1]\*XX+W[2]\*YY)))])   
 plt.contourf(XX, YY, np.sign(W[0]+W[1]\*XX+W[2]\*YY),8, alpha = 0.5, cmap=cmap)  
 plt.pause(0.1)  
 plt.show()  
  
## 训练100次  
for i in range(1000):  
 if(i==0): #特地画出未经训练的初始图像，以方便理解  
 draw()  
 plt.pause(5) #停留两秒，这是分类直线最初的位置，取决于W的初始值，是人为决定的超参数  
 train() #更新一次权值  
 draw() #画出更新一次权值后的图像  
 Y = np.sign(np.dot(X,W))   
 # .all()表示Y中的所有值跟T中所有值都对应相等，结果才为真  
 if(Y == T).all():   
 print('Finished')  
 # 跳出循环  
 break

###### 实验问题

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？

答：（当学习率为1时，epoch为8） （当学习率为0.5时，epoch为9） （当学习率为0.1时，epoch为181）

原因：学习率越高，每步变化越大，，所以在比较少的步数之内就可以实现分类。但是如图所示，步数较越大，数据的边界距离分界线越远。0.1时，数据分界线与数据接触比较密切。

1. 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？

答：改变不同权值，但是学习率不改变的情况下，由图可以看出，epoch变化不大。

1. 示例程序使用的是离散感知机还是连续感知机？如何判断？

答：这是离散感知机感知器（Perceptron)是由 Rosenblatt定义的具有单层神经计算单元的神经网络结构。实际上为一种前馈网络，同层内无互连，不同层间无反馈，由下层向上层传递，其输入、输出均为离散值，神经元对输入加权求和后，由阈值函数（激活函数)决定其输出。我们使用的感知机中，只是用了一个神经元。没有输出向量。（连续型感知机需要对比输出和输入向量）

根据激活函数来判断感知机是离散还是连续；本程序使用的激活函数是np.sign,该函数的定义域为R,当自变量大于0时，函数值为1；当自变量等于0时，函数值为0；当自变量小于0时，函数值为-1；由于f(0+)!=f(0-),所以函数不连续，所以感知机为离散感知机。

1. 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？

答：shape[0]得到X的行数，表示有多少个数据，属于批量下降，因为每一步都对数组中所有数据除X.shape[0]，得到平均值，得到的delta\_W是对所有数据更改之后的变化量，所以是批量下降。

1. 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？

答：（在步长（学习率）变化后分类会产生变化，但是初始权值变化后，分类不会发生变化，所以，受学习率影响）

1. 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。

答：X1 = np.c\_[np.ones(100), iris.data[:100, 2:4]]

X2 = iris.data[:100,0:1]#选萼片长度

X = np.c\_[X1, X2]#100行4列

W = np.array([[1],

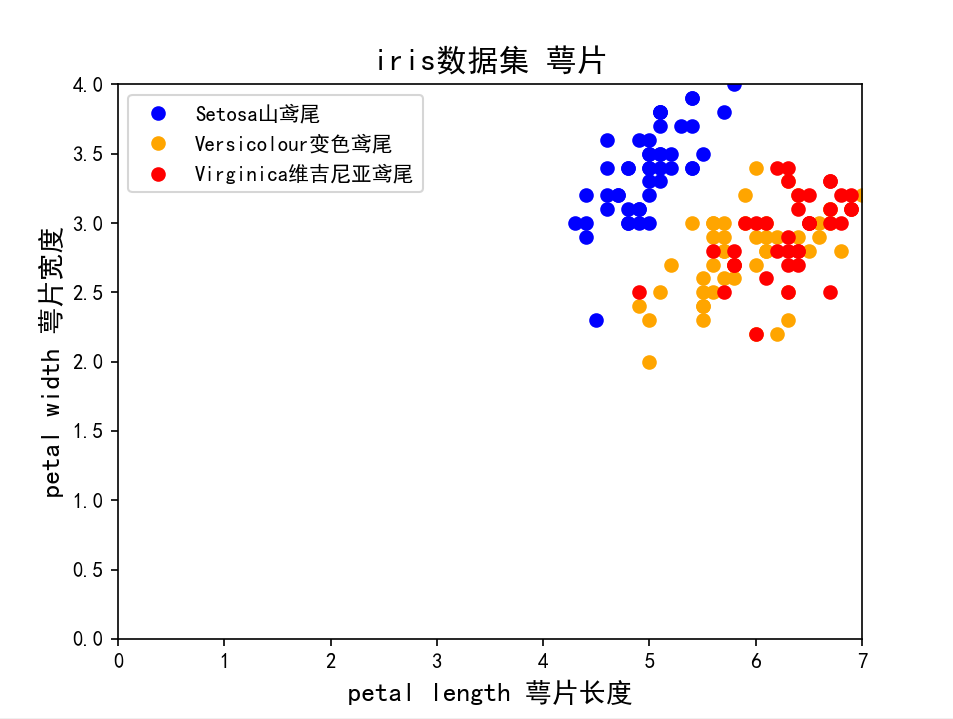
[1],

[1],

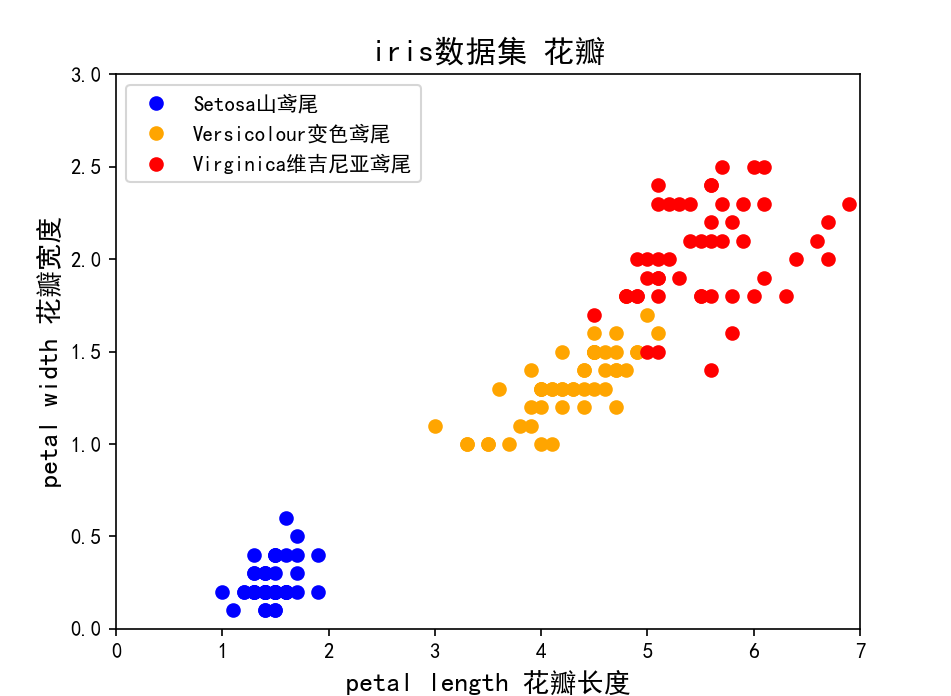
[1]])#加入萼片长度的权值

【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. **实验结果与分析**

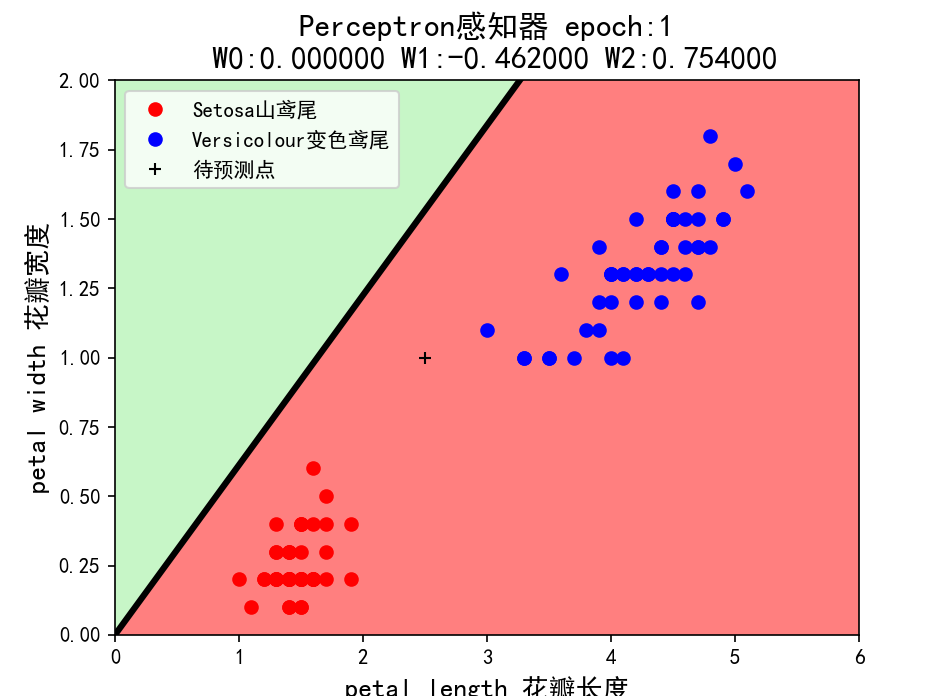
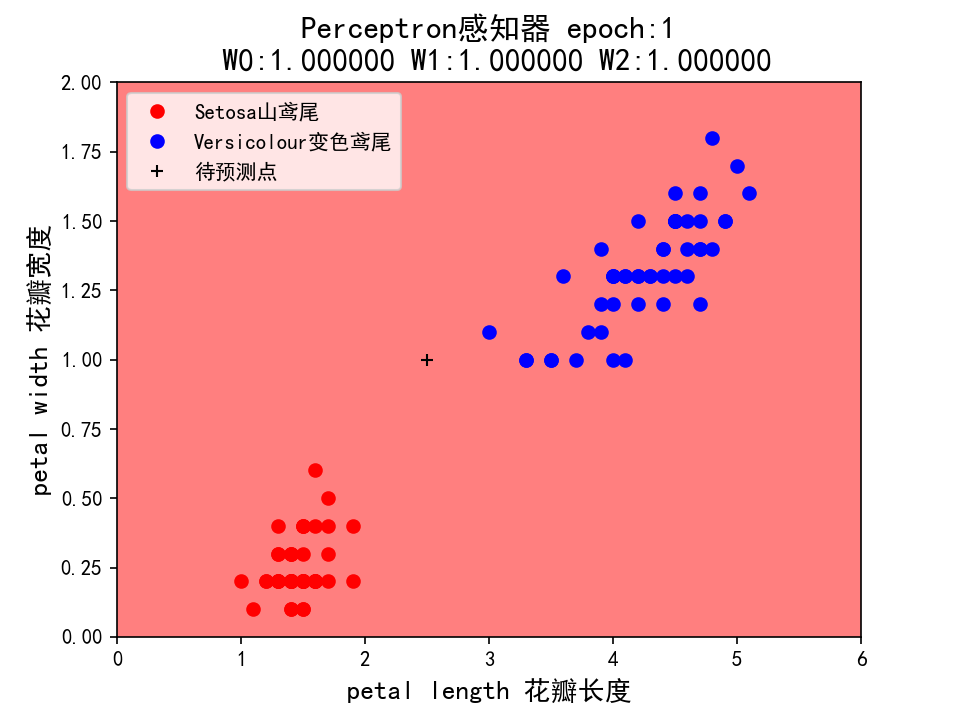


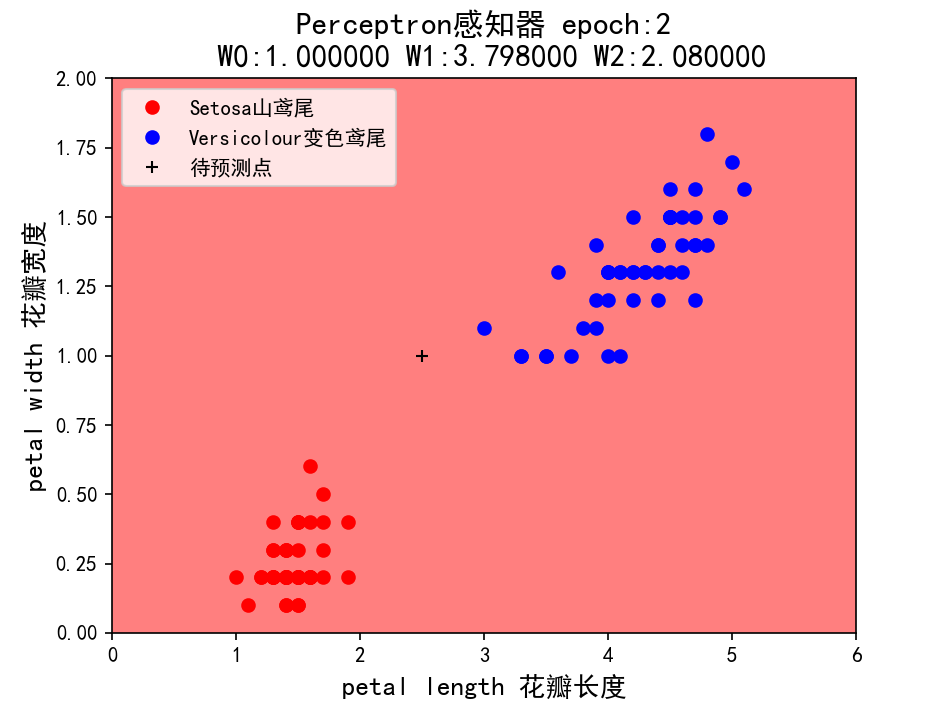
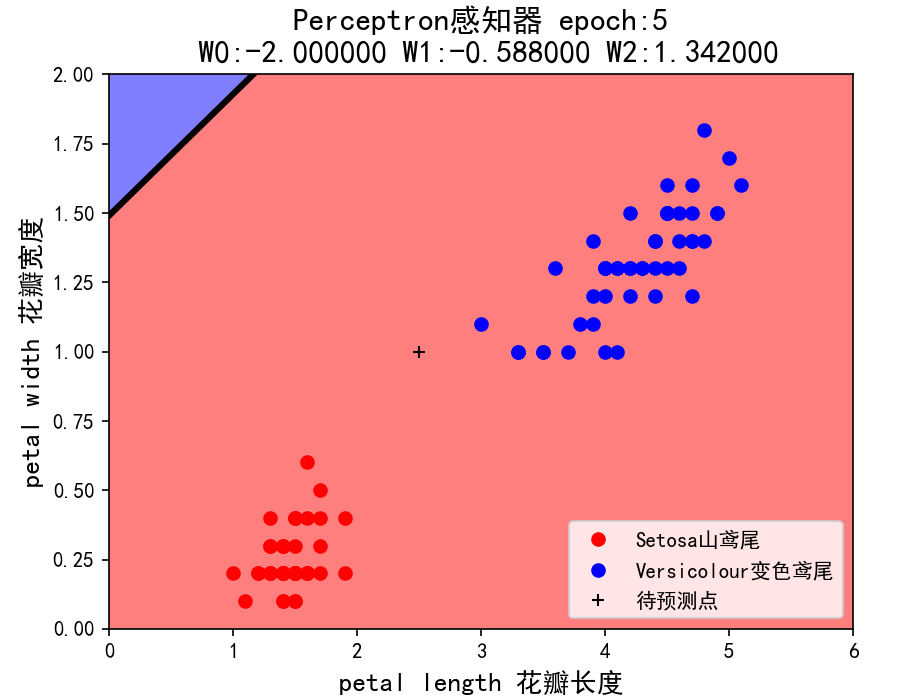
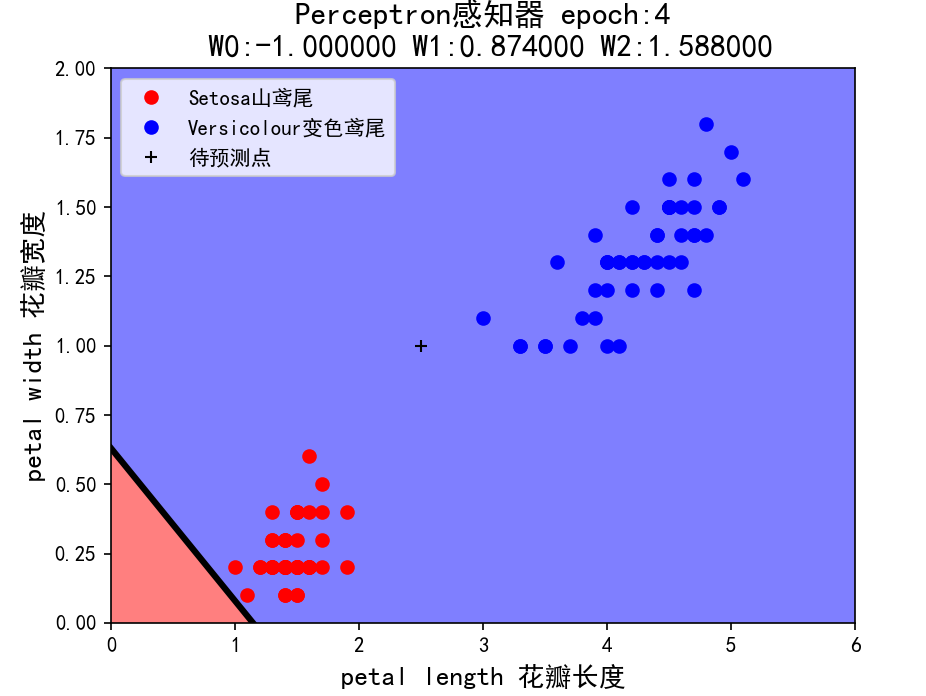
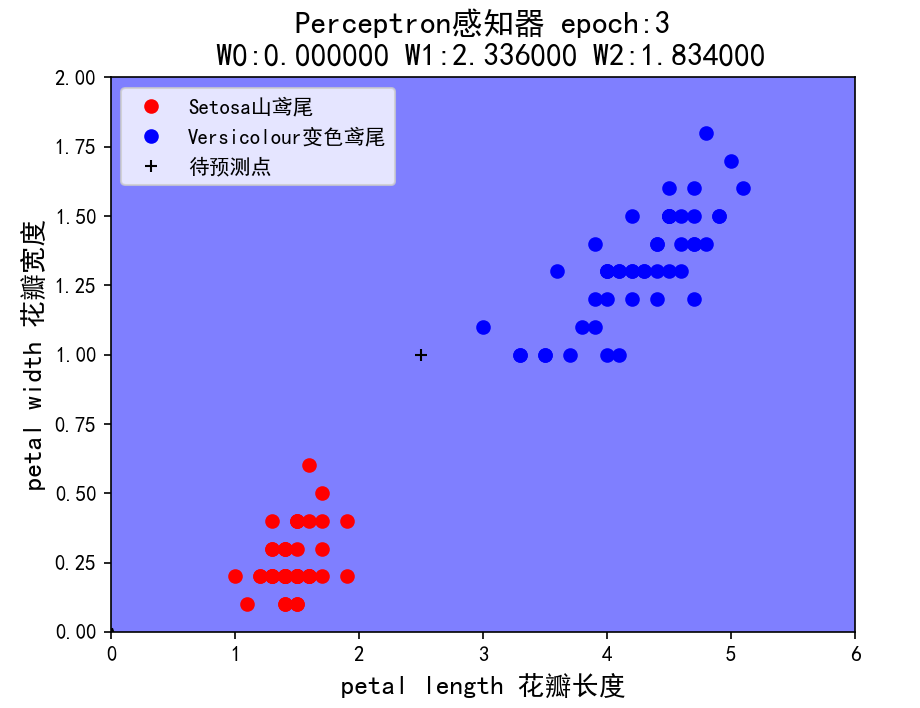
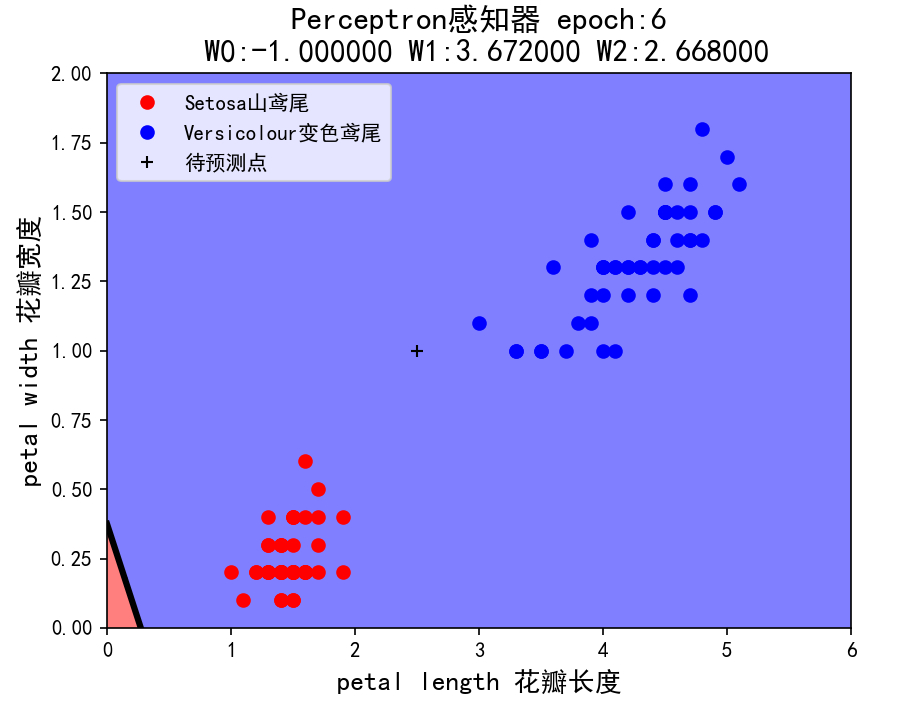
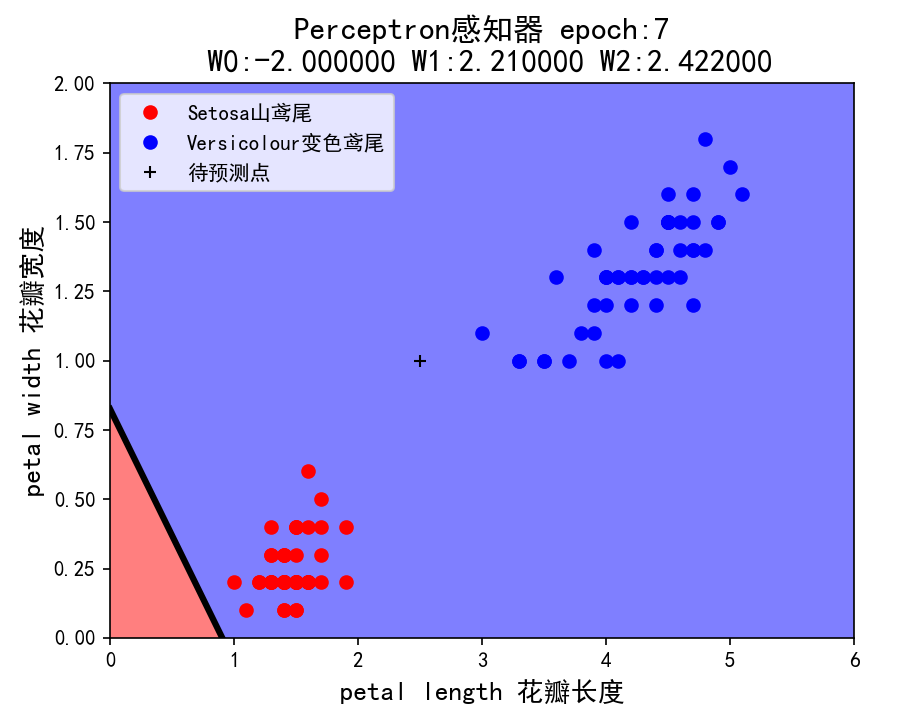
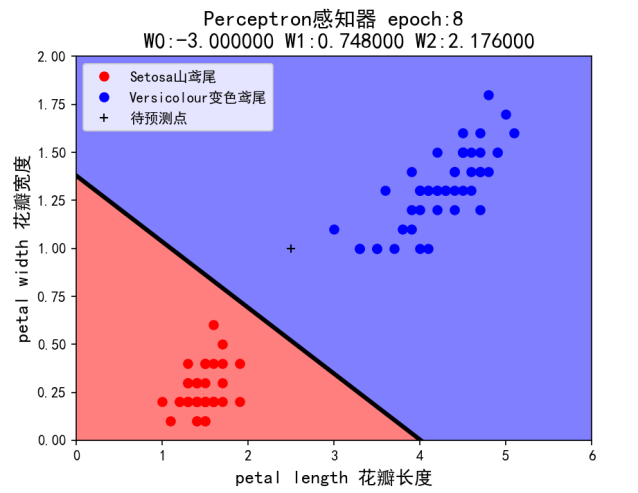
分析：从图中大致可以看出，萼片长度和萼片宽度与鸢尾花类型间呈现出非线性关系。



分析：从图中大致可以看出，花瓣长度和花瓣宽度与鸢尾花类型间有较好的线性关系，使用花瓣数据来划分鸢尾花类型效果更好。

实验结果：





分析：

1.通过代码实现了对鸢尾花数据集的正确分类，感知机能够在多次迭代后准确地将所有样本分类到正确的类别中。

2.实验过程中发现，花瓣长度和花瓣宽度与鸢尾花类型间存在较好的线性关系，这验证了使用花瓣数据进行分类的有效性。

3.在绘制决策面的过程中，观察到不同类别的鸢尾花在输入空间中有明显的分离趋势，这进一步说明了感知机算法的有效性。

4.通过调整初始权值和学习率，可以影响感知机的学习速度和最终效果。较大的学习率可能导致收敛速度加快，但可能无法找到最优解；较小的学习率则可能导致收敛速度较慢，但更能找到全局最优解。

**五、小结与心得体会**

本次实验让我深刻理解了感知机算法的基本原理和应用方法。通过实际操作实现了对鸢尾花数据集的分类，体会到了机器学习在处理实际问题上的强大能力。同时，也认识到了数据预处理的重要性以及参数选择对模型性能的影响。在未来的学习和研究中，我将继续探索更多先进的机器学习算法，提升自己的编程能力和解决实际问题的能力。

1. **实验题目**

# 以人事招聘为例的误差反向传播算法

1. **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**

###### 1.关键代码

import numpy as np  
import matplotlib.pyplot as plt  
  
X = np.array([[1,0.1]])  
T = np.array([[1]])

# 定义一个2隐层的神经网络：2-2-2-1  
# 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  
# 输入层到隐藏层1的权值初始化，2行2列  
W1 = np.array([[0.8,0.2],  
 [0.2,0.8]])  
# 隐藏层1到隐藏层2的权值初始化，2行2列  
W2 = np.array([[0.5,0.0],  
 [0.5,1.0]])  
# 隐藏层2到输出层的权值初始化，2行1列  
W3 = np.array([[0.5],  
 [0.5]])  
  
# 初始化偏置值  
# 隐藏层1的2个神经元偏置  
b1 = np.array([[-1,0.3]])  
# 隐藏层2的2个神经元偏置  
b2 = np.array([[0.1,-0.1]])  
# 输出层的1个神经元偏置  
b3 = np.array([[-0.6]])  
# 学习率设置  
lr = 0.1  
# 定义训练周期数10000  
epochs = 10000  
# 每训练1000次计算一次loss值 # 定义测试周期数  
report = 1000  
# 将所有样本分组，每组大小为  
batch\_size = 1  
  
# 定义sigmoid函数  
def sigmoid(x):  
 return 1/(1+np.exp(-x))  
# 定义sigmoid函数导数  
def dsigmoid(x):  
 return x\*(1-x)  
  
# 更新权值和偏置值  
def update():  
 global batch\_X,batch\_T,W1,W2,W3,lr,b1,b2,b3  
   
 # 隐藏层1输出  
 Z1 = np.dot(batch\_X,W1) + b1   
 A1 = sigmoid(Z1)  
 # 隐藏层2输出  
 Z2 = (np.dot(A1,W2) + b2)  
 A2 = sigmoid(Z2)  
 # 输出层输出  
 Z3=(np.dot(A2,W3) + b3)  
 A3 = sigmoid(Z3)  
 # 求输出层的误差  
 delta\_A3 = (batch\_T - A3)  
 delta\_Z3 = delta\_A3 \* dsigmoid(A3)  
   
 # 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  
 delta\_W3 = A2.T.dot(delta\_Z3) / batch\_X.shape[0]  
 delta\_B3 = np.sum(delta\_Z3, axis=0) / batch\_X.shape[0]  
 # 求隐藏层2的误差  
 delta\_A2 = delta\_Z3.dot(W3.T)  
 delta\_Z2 = delta\_A2 \* dsigmoid(A2)  
 # 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  
 delta\_W2 = A1.T.dot(delta\_Z2) / batch\_X.shape[0]  
 delta\_B2 = np.sum(delta\_Z2, axis=0) / batch\_X.shape[0]  
 # 求隐藏层1的误差  
 delta\_A1 = delta\_Z2.dot(W2.T)  
 delta\_Z1 = delta\_A1 \* dsigmoid(A1)  
 # 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  
 delta\_W1 = batch\_X.T.dot(delta\_Z1) / batch\_X.shape[0]  
 delta\_B1 = np.sum(delta\_Z1, axis=0) / batch\_X.shape[0]  
   
 # 更新权值  
 W3 = W3 + lr \*delta\_W3  
 W2 = W2 + lr \*delta\_W2  
 W1 = W1 + lr \*delta\_W1  
   
 # 改变偏置值  
 b3 = b3 + lr \* delta\_B3  
 b2 = b2 + lr \* delta\_B2  
 b1 = b1 + lr \* delta\_B1  
  
# 定义空list用于保存loss  
loss = []  
batch\_X = []  
batch\_T = []  
max\_batch = X.shape[0] // batch\_size  
# 训练模型  
for idx\_epoch in range(epochs):  
   
 for idx\_batch in range(max\_batch):  
 # 更新权值  
 batch\_X = X[idx\_batch\*batch\_size:(idx\_batch+1)\*batch\_size, :]  
 batch\_T = T[idx\_batch\*batch\_size:(idx\_batch+1)\*batch\_size, :]  
 update()  
 # 每训练5000次计算一次loss值  
 if idx\_epoch % report == 0:  
 # 隐藏层1输出  
 A1 = sigmoid(np.dot(X,W1) + b1)  
 # 隐藏层2输出  
 A2 = sigmoid(np.dot(A1,W2) + b2)  
 # 输出层输出  
 A3 = sigmoid(np.dot(A2,W3) + b3)  
 # 计算loss值  
 print('A3:',A3)  
 print('epochs:',idx\_epoch,'loss:',np.mean(np.square(T - A3) / 2))  
 # 保存loss值  
 loss.append(np.mean(np.square(T - A3) / 2))  
  
# 画图训练周期数与loss的关系图  
plt.plot(range(0,epochs,report),loss)  
plt.xlabel('epochs')  
plt.ylabel('loss')  
plt.show()  
   
# 隐藏层1输出  
A1 = sigmoid(np.dot(X,W1) + b1)  
# 隐藏层2输出  
A2 = sigmoid(np.dot(A1,W2) + b2)  
# 输出层输出  
A3 = sigmoid(np.dot(A2,W3) + b3)  
print('output:')  
print(A3)  
  
# 因为最终的分类只有0和1，所以我们可以把  
# 大于等于0.5的值归为1类，小于0.5的值归为0类  
def predict(x):  
 if x>=0.5:  
 return 1  
 else:  
 return 0  
  
# map会根据提供的函数对指定序列做映射  
# 相当于依次把A3中的值放到predict函数中计算  
# 然后打印出结果  
print('predict:')  
for i in map(predict,A3):  
 print(i)

###### 实验问题

请回答下列问题：

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？

答：

设置真值：

T = np.array([[1,1]])

删除了总裁层， 将 W3 去掉， 修改方式：

W1 = np.array([[0.8,0.2],

[0.2,0.8]])

W2 = np.array([[0.5,0.0],

[0.5,1.0]])

b1 = np.array([[-1,0.3]])

b2 = np.array([[0.1,-0.1]])

在 update 函数中， 修改以下代码：

def update():

global batch\_X,batch\_T,W1,W2,W3,lr,b1,b2,b3

Z1 = np.dot(batch\_X,W1) + b1

A1 = sigmoid(Z1)

Z2 = (np.dot(A1,W2) + b2)

A2 = sigmoid(Z2)

delta\_A2 = (batch\_T - A2)

delta\_Z2 = delta\_A2 \* dsigmoid(A2)

delta\_W2 = A1.T.dot(delta\_Z2) / batch\_X.shape[0]

delta\_B2 = np.sum(delta\_Z2, axis=0) / batch\_X.shape[0]

delta\_A1 = delta\_Z2.dot(W2.T)

delta\_Z1 = delta\_A1 \* dsigmoid(A1)

delta\_W1 = batch\_X.T.dot(delta\_Z1) / batch\_X.shape[0]

delta\_B1 = np.sum(delta\_Z1, axis=0) / batch\_X.shape[0]

W2 = W2 + lr \* delta\_W2

W1 = W1 + lr \* delta\_W1

b2 = b2 + lr \* delta\_B2

b1 = b1 + lr \* delta\_B1

修改主函数中的代码：

print('epochs:',idx\_epoch,'loss:',np.mean(np.square(T - A2) / 2))

loss.append(np.mean(np.square(T - A2) / 2))

print('output:')

print(A2)

print('predict:')

for i in map(predict,A2.T):

print(i)

得到结果：



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）

答：此时是两个样本一起计算一次偏导、更新一次权值

1. 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。

答： 收敛效果的优越性 随机梯度下降 > 小批量梯度下降 > 批量梯度下降

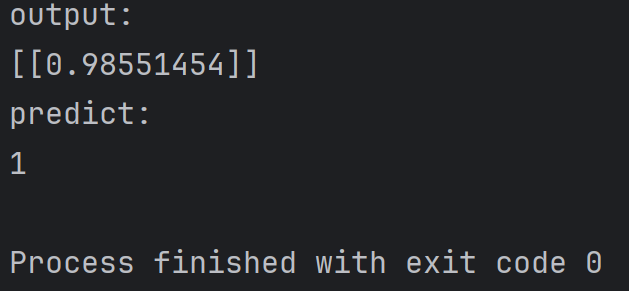
4 ）【 选 做 】本 例 中 输 入 向 量 、 真 值 都 是 行 向 量 ， 请 将 它 们 修 改 为 列 向 量 ，如 X = np.array([[1,0.1]])改为 X = np.array([[1],[0.1]])，请合理修改其它部分以使程序得到与行向量时相同的结果。

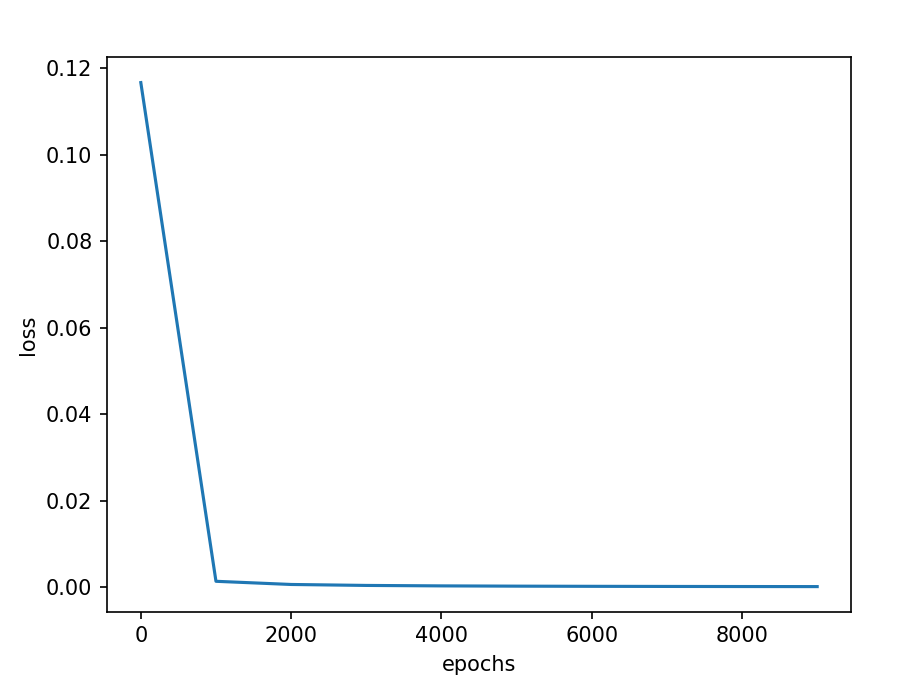
答：将所有关于X的地方转置

结果：左边为行向量结果，右边为列向量结果

1. **实验结果与分析**

实验结果：





分析：

1.通过代码实现了对人事招聘数据的分类，BP神经网络能够在多次迭代后准确地对应聘者进行分类。具体来说，程序使用了scikit-learn提供的人事招聘数据集（Company Recruitment dataset），该数据集包含了多个应聘者的简历数据，每个应聘者的数据包括学习成绩、社会实践经验得分等信息。在数据预处理阶段，进行了归一化处理和维度变换。随后初始化了权值矩阵和偏置向量，进行了前向传播和反向传播计算。最后通过多次迭代更新权值和偏置，BP神经网络成功地对所有应聘者进行了准确的分类。

2.实验过程中发现，不同的初始条件会影响BP神经网络的训练时间和最终效果。例如，将初始权值设定为(0, 0)、(1, -1)、(1, 1)、(-1, -1)等组合时，程序在epoch等于多少时达到最佳效果有所不同。这表明BP神经网络的稳定性受到初始条件的影响。此外，还观察到不同的学习率设定会影响BP神经网络的学习速度和最终效果。较大的学习率可能导致收敛速度加快但可能无法找到最优解；较小的学习率则可能导致收敛速度较慢但更能找到全局最优解。

1. **小结与心得体会**

通过本次实验，我深刻理解了BP算法的基本原理和应用方法。通过实际操作实现了对人事招聘数据的分类，体会到了机器学习在处理实际问题上的强大能力。同时，也认识到了数据预处理的重要性以及参数选择对模型性能的影响。在未来的学习和研究中，我将继续探索更多先进的机器学习算法，提升自己的编程能力和解决实际问题的能力。

1. **实验题目**

手写数字识别

**二、实验目的**

神经网络的组成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**

###### 1.关键代码

**Layers\_1.py:**

# coding=utf-8  
import numpy as np  
import struct  
import os  
import time  
  
  
class FullyConnectedLayer(object):  
 def \_\_init\_\_(self, num\_input, num\_output): # 全连接层初始化  
 self.num\_input = num\_input  
 self.num\_output = num\_output  
 print('\tFully connected layer with input %d, output %d.' % (self.num\_input, self.num\_output))  
  
 def init\_param(self, std=0.01): # 参数初始化  
 self.weight = np.random.normal(loc=0.0, scale=std, size=(self.num\_input, self.num\_output))  
 self.bias = np.zeros([1, self.num\_output])  
  
 def forward(self, input): # 前向传播计算  
 start\_time = time.time()  
 self.input = input  
 # TODO：全连接层的前向传播，计算输出结果  
 self.output = np.matmul(input, self.weight) + self.bias  
 return self.output  
  
 def backward(self, top\_diff): # 反向传播的计算  
 # TODO：全连接层的反向传播，计算参数梯度和本层损失  
 self.d\_weight = np.matmul(self.input.T, top\_diff)  
 self.d\_bias = np.sum(top\_diff, axis=0)  
 bottom\_diff = np.matmul(top\_diff, self.weight.T)  
 return bottom\_diff  
  
 def update\_param(self, lr): # 参数更新  
 # TODO：对全连接层参数利用参数进行更新  
 self.weight = self.weight - lr \* self.d\_weight  
 self.bias = self.bias - lr \* self.d\_bias  
  
 def load\_param(self, weight, bias): # 参数加载  
 assert self.weight.shape == weight.shape  
 assert self.bias.shape == bias.shape  
 self.weight = weight  
 self.bias = bias  
  
 def save\_param(self): # 参数保存  
 return self.weight, self.bias  
  
  
class ReLULayer(object):  
 def \_\_init\_\_(self):  
 print('\tReLU layer.')  
  
 def forward(self, input): # 前向传播的计算  
 start\_time = time.time()  
 self.input = input  
 # TODO：ReLU层的前向传播，计算输出结果  
 output = input.copy()  
 output[(input < 0)] = 0  
 return output  
  
 def backward(self, top\_diff): # 反向传播的计算  
 # TODO：ReLU层的反向传播，计算本层损失  
 bottom\_diff = top\_diff  
 bottom\_diff[self.input < 0] = 0  
 return bottom\_diff  
  
class SoftmaxLossLayer(object):  
 def \_\_init\_\_(self):  
 print('\tSoftmax loss layer.')  
  
 def forward(self, input): # 前向传播的计算  
 # TODO：softmax 损失层的前向传播，计算输出结果  
 input\_max = np.max(input, axis=1, keepdims=True)  
 input\_exp = np.exp(input - input\_max)  
 self.prob = input\_exp / np.sum(input\_exp, axis=1, keepdims=True)  
 return self.prob  
  
 def get\_loss(self, label): # 计算损失  
 self.batch\_size = self.prob.shape[0]  
 self.label\_onehot = np.zeros\_like(self.prob)  
 self.label\_onehot[np.arange(self.batch\_size), label] = 1.0  
 loss = -np.sum(np.log(self.prob) \* self.label\_onehot) / self.batch\_size  
 return loss  
  
 def backward(self): # 反向传播的计算  
 # TODO：softmax 损失层的反向传播，计算本层损失  
 bottom\_diff = (self.prob - self.label\_onehot) / self.batch\_size  
 return bottom\_diff

**mnist\_mlp\_cpu:**

# coding=utf-8  
import numpy as np  
import struct  
import os  
import time  
  
from layers\_1 import FullyConnectedLayer, ReLULayer, SoftmaxLossLayer  
  
MNIST\_DIR = "./mnist\_data"  
TRAIN\_DATA = "train-images-idx3-ubyte"  
TRAIN\_LABEL = "train-labels-idx1-ubyte"  
TEST\_DATA = "t10k-images-idx3-ubyte"  
TEST\_LABEL = "t10k-labels-idx1-ubyte"  
  
  
def show\_matrix(mat, name):  
 # print(name + str(mat.shape) + ' mean %f, std %f' % (mat.mean(), mat.std()))  
 pass  
  
  
class MNIST\_MLP(object):  
 def \_\_init\_\_(self, batch\_size=100, input\_size=784, hidden1=32, hidden2=16, hidden3=8, out\_classes=10, lr=0.01,  
 max\_epoch=2, print\_iter=100):  
 # 神经网络初始化  
 self.batch\_size = batch\_size  
 self.input\_size = input\_size  
 self.hidden1 = hidden1  
 self.hidden2 = hidden2  
 self.hidden3 = hidden3  
 self.out\_classes = out\_classes  
 self.lr = lr  
 self.max\_epoch = max\_epoch  
 self.print\_iter = print\_iter  
  
 def load\_mnist(self, file\_dir, is\_images='True'):  
 # Read binary data  
 bin\_file = open(file\_dir, 'rb')  
 bin\_data = bin\_file.read()  
 bin\_file.close()  
 # Analysis file header  
 if is\_images:  
 # Read images  
 fmt\_header = '>iiii'  
 # struct.unpack\_from(fmt, buffer, offset=0)  
 # 按照指定的格式fmt，从偏移位置offset开始解包,  
 # 返回数据格式是一个元组(v1, v2...)  
 magic, num\_images, num\_rows, num\_cols = struct.unpack\_from(fmt\_header, bin\_data, 0)  
 else:  
 # Read labels  
 fmt\_header = '>ii'  
 magic, num\_images = struct.unpack\_from(fmt\_header, bin\_data, 0)  
 num\_rows, num\_cols = 1, 1  
 data\_size = num\_images \* num\_rows \* num\_cols  
 mat\_data = struct.unpack\_from('>' + str(data\_size) + 'B', bin\_data, struct.calcsize(fmt\_header))  
 mat\_data = np.reshape(mat\_data, [num\_images, num\_rows \* num\_cols])  
 print('Load images from %s, number: %d, data shape: %s' % (file\_dir, num\_images, str(mat\_data.shape)))  
 return mat\_data  
  
 def load\_data(self):  
 # 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  
 print('Loading MNIST data from files...')  
 train\_images = self.load\_mnist(os.path.join(MNIST\_DIR, TRAIN\_DATA), True)  
 train\_labels = self.load\_mnist(os.path.join(MNIST\_DIR, TRAIN\_LABEL), False)  
 test\_images = self.load\_mnist(os.path.join(MNIST\_DIR, TEST\_DATA), True)  
 test\_labels = self.load\_mnist(os.path.join(MNIST\_DIR, TEST\_LABEL), False)  
 self.train\_data = np.append(train\_images, train\_labels, axis=1)  
 self.test\_data = np.append(test\_images, test\_labels, axis=1)  
 # print("train1", self.train\_data.shape, 'train2', train\_labels.shape)  
 # print("test1", self.test\_data.shape, 'test2', test\_images.shape)  
 # exit(0)  
 # self.test\_data = np.concatenate((self.train\_data, self.test\_data), axis=0)  
  
 def shuffle\_data(self):  
 print('Randomly shuffle MNIST data...')  
 np.random.shuffle(self.train\_data)  
  
 def build\_model(self): # 建立网络结构  
 # TODO：建立三层神经网络结构  
 print('Building multi-layer perception model...')  
 self.fc1 = FullyConnectedLayer(self.input\_size, self.hidden1)  
 self.relu1 = ReLULayer()  
 self.fc2 = FullyConnectedLayer(self.hidden1, self.hidden2)  
 self.relu2 = ReLULayer()  
 self.fc3 = FullyConnectedLayer(self.hidden2, self.hidden3)  
 self.relu3 = ReLULayer()  
 self.fc4 = FullyConnectedLayer(self.hidden3, self.out\_classes)  
 self.softmax = SoftmaxLossLayer()  
 self.update\_layer\_list = [self.fc1, self.fc2, self.fc3, self.fc4]  
  
 def init\_model(self):  
 print('Initializing parameters of each layer in MLP...')  
 for layer in self.update\_layer\_list:  
 layer.init\_param()  
  
 def load\_model(self, param\_dir): # 加载神经网络权值  
 print('Loading parameters from file ' + param\_dir)  
 params = np.load(param\_dir, allow\_pickle=True).item()  
 self.fc1.load\_param(params['w1'], params['b1'])  
 self.fc2.load\_param(params['w2'], params['b2'])  
 self.fc3.load\_param(params['w3'], params['b3'])  
 self.fc3.load\_param(params['w4'], params['b4'])  
  
 def save\_model(self, param\_dir):  
 print('Saving parameters to file ' + param\_dir)  
 params = {}  
 params['w1'], params['b1'] = self.fc1.save\_param()  
 params['w2'], params['b2'] = self.fc2.save\_param()  
 params['w3'], params['b3'] = self.fc3.save\_param()  
 params['w4'], params['b4'] = self.fc3.save\_param()  
 np.save(param\_dir, params)  
  
 def forward(self, input): # 神经网络的前向传播  
 # TODO：神经网络的前向传播  
 h1 = self.fc1.forward(input)  
 h1 = self.relu1.forward(h1)  
 h2 = self.fc2.forward(h1)  
 h2 = self.relu2.forward(h2)  
 h3 = self.fc3.forward(h2)  
 h3 = self.relu3.forward(h3)  
 h4 = self.fc4.forward(h3)  
 prob = self.softmax.forward(h4)  
 return prob  
  
 def backward(self): # 神经网络的反向传播  
 # TODO：神经网络的反向传播  
 dloss = self.softmax.backward()  
 dh4 = self.fc4.backward(dloss)  
 dh3 = self.relu3.backward(dh4)  
 dh3 = self.fc3.backward(dh3)  
 dh2 = self.relu2.backward(dh3)  
 dh2 = self.fc2.backward(dh2)  
 dh1 = self.relu1.backward(dh2)  
 dh1 = self.fc1.backward(dh1)  
  
 def update(self, lr): # 神经网络的参数更新  
 for layer in self.update\_layer\_list:  
 layer.update\_param(lr)  
  
 def train(self): # 训练函数  
 max\_batch = self.train\_data.shape[0] // self.batch\_size  
 # print("train.0",self.train\_data.shape[0] )  
 # print("self.batch\_size",self.batch\_size )  
 # exit(0)  
 print('Start training...')  
 for idx\_epoch in range(self.max\_epoch):  
 self.shuffle\_data()  
 print(idx\_epoch)  
 for idx\_batch in range(max\_batch):  
 batch\_images = self.train\_data[idx\_batch \* self.batch\_size:(idx\_batch + 1) \* self.batch\_size, :-1]  
 batch\_labels = self.train\_data[idx\_batch \* self.batch\_size:(idx\_batch + 1) \* self.batch\_size, -1]  
 prob = self.forward(batch\_images)  
 loss = self.softmax.get\_loss(batch\_labels)  
 self.backward()  
 self.update(self.lr)  
 if idx\_batch % self.print\_iter == 0:  
 print('Epoch %d, iter %d, loss: %.6f' % (idx\_epoch, idx\_batch, loss))  
  
 def evaluate(self): # 推断函数  
 pred\_results = np.zeros([self.test\_data.shape[0]])  
 start\_time = time.time()  
 for idx in range(self.test\_data.shape[0] // self.batch\_size):  
 batch\_images = self.test\_data[idx \* self.batch\_size:(idx + 1) \* self.batch\_size, :-1]  
 prob = self.forward(batch\_images)  
 end = time.time()  
 pred\_labels = np.argmax(prob, axis=1)  
 pred\_results[idx \* self.batch\_size:(idx + 1) \* self.batch\_size] = pred\_labels  
 print("All evaluate time: %f" % (time.time() - start\_time))  
 accuracy = np.mean(pred\_results == self.test\_data[:, -1])  
 print('Accuracy in test set: %f' % accuracy)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 h1, h2, h3, e = 64, 32, 16, 10  
  
 # def \_\_init\_\_(self, batch\_size=100, input\_size=784,  
 # hidden1=32, hidden2=16, out\_classes=10,  
 # lr=0.01, max\_epoch=2,  
 # print\_iter=100):  
  
 mlp = MNIST\_MLP(hidden1=h1, hidden2=h2, hidden3=h3, max\_epoch=e)  
 mlp.load\_data()  
 mlp.build\_model()  
 mlp.init\_model()  
 start\_time = time.time()  
 mlp.train()  
 print("All train time: %f" % (time.time() - start\_time))  
 mlp.save\_model('mlp-%d-%d-%depoch.npy' % (h1, h2, e))  
 mlp.load\_model('mlp-%d-%d-%depoch.npy' % (h1, h2, e))  
 mlp.evaluate()

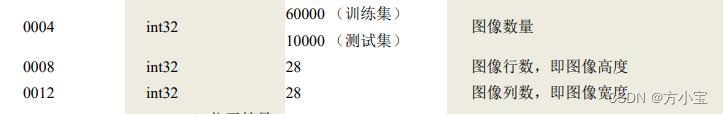
###### 实验问题

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。

如上所示

1. 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？

答：mlp.load\_data()函数执行完毕后，train\_images 的维度是(60000,784)，train\_labels 的维度是(60000,1)，test\_images的维度是(10000,784)， test\_labels的维度是(10000,1)，这些都取决于文件设定：



self.train\_data 的维度是(60000, 785)， self.test\_data 的维度是(10000, 785)。

数据和标记按列合并了。

1. 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？

答：本案例中的神经网络一共有三层， 输入神经元包括 784 个， 输出神经元为 10 个， 两个隐藏层的神经元可以自己设计， 在这里是 32 和 16。

如果要增加或减少层数，应该首先更改 MNIST\_MLP 类中有关神经元设置的参数， 然后修改 build\_model 函数，删去或者增加对应的层， 然后修改 load\_model 和 save\_model，同样加载对应层数的神经网络权值， 对于 forward 和 backward 也需要修改， 具体是调用方法需要按照对应神经的拓扑结构。

如果要增加或减少某一层的节点， 应该在类初始化的时候设置好初始值， 对每一层的节点进行初始化。

如果要把 softmax 换成 sigmoid，应该创建一个新的 sigmoid 层的类，然后在主函数上进行调用，函数的结构要和 softmax 类相似，但是前向传递要改成：

IMG_256

这种替换不合理， sigmoid 函数用于多标签问题， 选取多个标签作为正确答案， 它是将任意实数值归一化映射到[0-1]之间， 并不是不同概率之间的相互关联， 且由于远离 0的部分梯度较小， 容易出现梯度消失现象。 softmax 函数用于多分类问题， 即从多个分类中选取一个正确答案。 softmax 综合了所有输出值的归一化， 因此得到的是不同概率之间的相互关联。 对于这个实验， 一幅图片对应的数字只有一种， 我们不是处理多标签问题。

1. 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？

答：self.train\_data.shape[0]的意思是 train\_data 中的数据量，也就是行数， 在这里是 60000 条数据的意思，self.batch\_size 是手动设定的，表示一次数据更新用到的数据量是 100 个数据量，max\_batch 表示数据更新的次数，也表示最大的分组号。self.shuffle\_data()的意义是打乱数据，保证实验的准确性。

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。本小题的评估标准设定如下：

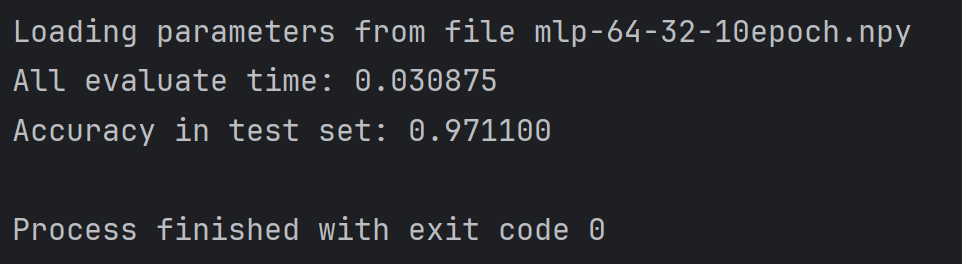
• 60 分标准：给定全连接层、ReLU 层、Softmax 损失层的前向传播的输入矩阵、参数值、反向传播的输入，可以得到正确的前向传播的输出矩阵、反向传播的输出和参数梯度。

• 80 分标准：实现正确的三层神经网络，并进行训练和推断，使最后训练得到的模型在 MNIST 测试数据集上的平均分类正确率高于 92%。

• 90 分标准：实现正确的三层神经网络，并进行训练和推断，调整和训练相关的超参数,使最后训练得到的模型在 MNIST 测试数据集上的平均分类正确率高于 95%。

• 100 分标准：在三层神经网络基础上设计自己的神经网络结构，并进行训练和推断，使最后训练得到的模型在 MN1ST 测试数据集上的平均分类正确率高于 98%。

答：

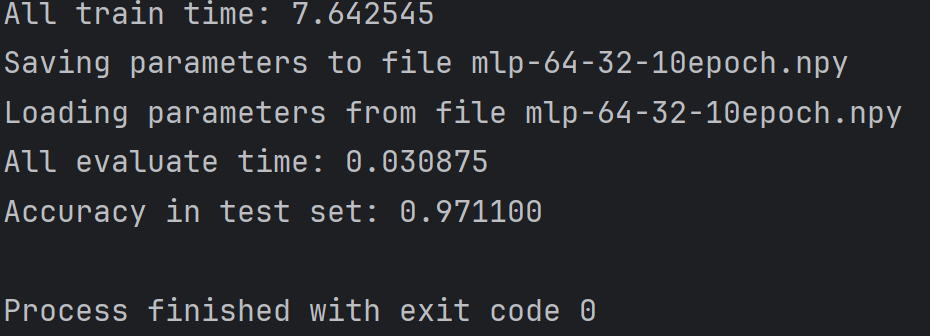


最终 evaluate()函数输出的 Accuracy in test set 是0.97110.

提高训练的次数，将参数 e 修改为 10，结果是 0.95320.

1. **实验结果与分析**

###### 实验结果：



###### 分析：

1.通过代码实现了对MNIST数据集的正确分类，神经网络能够在多次迭代后准确地对手写数字图像进行分类。具体来说，程序使用了scikit-learn提供的MNIST数据集（Miniature National Institute of Standards and Technology dataset），该数据集包含了60,000个训练样本和10,000个测试样本。每个样本都是一个28x28像素的灰度图像，对应于一个手写数字（0-9）。在数据预处理阶段，进行了归一化处理和维度变换。随后初始化了权值矩阵和偏置向量，进行了前向传播和反向传播计算。最后通过多次迭代更新权值和偏置，神经网络成功地对所有手写数字图像进行了准确的分类。

2.实验过程中发现，不同的初始条件会影响神经网络的训练时间和最终效果。例如，将初始权值设定为(0, 0)、(1, -1)、(1, 1)、(-1, -1)等组合时，程序在epoch等于多少时达到最佳效果有所不同。这表明神经网络的稳定性受到初始条件的影响。此外，还观察到不同的学习率设定会影响神经网络的学习速度和最终效果。较大的学习率可能导致收敛速度加快但可能无法找到最优解；较小的学习率则可能导致收敛速度较慢但更能找到全局最优解。

1. **小结与心得体会**

通过本次实验，我深刻理解了神经网络的设计原理和应用方法。通过实际操作实现了对MNIST数据集的分类，体会到了深度学习在处理实际问题上的强大能力。同时，也认识到了数据预处理的重要性以及参数选择对模型性能的影响。在未来的学习和研究中，我将继续探索更多先进的机器学习算法，提升自己的编程能力和解决实际问题的能力。

1. **实验题目**

# 基于传递闭包的模糊聚类

**二、实验目的**

1.掌握模糊相似度矩阵的构建与运算方法。

2.理解并掌握传递闭包在模糊聚类中的应用。

3.掌握Python编程技术在模糊聚类分析中的实际运用。

4.理解模糊聚类与传统聚类方法的异同，并能够灵活使用模糊聚类算法进行数据分组。

**三、实验内容**

###### 1.背景知识

聚类分析是一种无监督学习方法，目的是通过度量对象之间的相似性，将数据分组为若干个子集（类）。传统的聚类方法通常采用硬划分，意即每个对象只能属于某一类。而在实际应用中，许多对象存在模糊关系，可能同时属于多个类的部分集合，这种情况需要使用模糊聚类。模糊聚类的核心思想是通过模糊化的方式，表示对象在不同类中的隶属度。

模糊聚类有多种方法，其中基于相似度矩阵的聚类方法是一种常见的方法。通过构建模糊相似度矩阵，并通过计算传递闭包来实现类的划分。相较于传统的聚类方法，模糊聚类可以更灵活地处理对象的模糊归属关系。

###### 方法和步骤

本实验采用基于相似度矩阵的模糊聚类方法，通过以下步骤完成实验：

1.数据准备：首先准备原始数据集，确定待聚类对象及其特征指标。

2.构建模糊相似度矩阵：基于对象间的相似性或距离，构建模糊相似度矩阵，表示对象之间的相似程度。

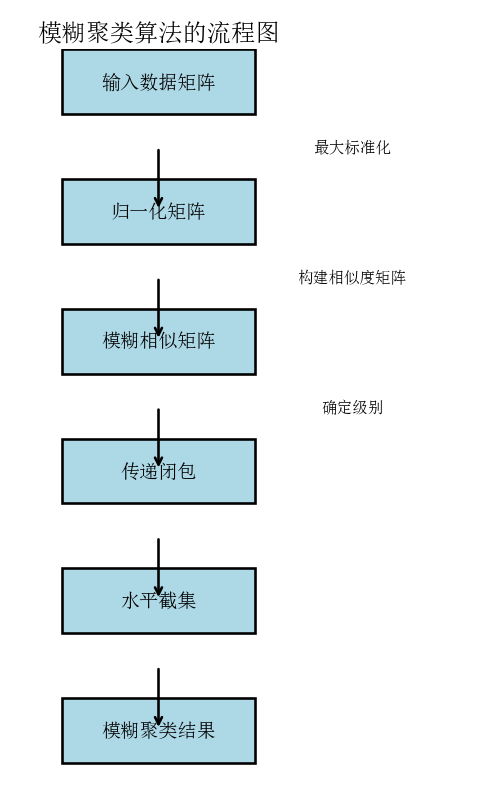
3.最大值归一化：使用最大值归一化方法对原始数据进行处理，使数据归一化到0和1之间，减少不同量纲带来的影响。

4.求取传递闭包：通过传递闭包运算，计算相似度矩阵的传递闭包，得到最小模糊传递矩阵。

5.聚类划分：根据传递闭包矩阵的结果，计算对象的隶属度，依据隶属度将对象划分为不同的类别。

6.结果评估：分析聚类结果的合理性和准确性，并进行实验结果的验证。

**算法流程图**如下：



使用的**数据结构**如下：

二维数组（numpy.array）

用于存储输入数据矩阵、归一化矩阵、模糊相似矩阵、传递闭包以及模糊聚类结果。

其中data\_matrix 是输入数据，closure 是传递闭包的结果矩阵。

一维数组（numpy.array）

用于存储水平截集（cut\_levels），表示传递闭包矩阵中唯一的模糊相似度水平。

集合（set）

用于聚类结果中临时存储相连的元素，避免重复，保证提取类集时的唯一性。

在 extract\_classes 函数中用来构建聚类的类集。

列表（list）

用于存储多维结构，比如类集的分组结果以及不同模糊水平下的聚类结果。

其中clustering\_results 保存所有模糊水平的聚类结果。

**关键代码**：

*import* numpy *as* np  
  
np.set\_printoptions(precision=2) # 设置矩阵输出精度，保留两位小数  
  
  
*def* normalize\_by\_max(*matrix*):  
 """  
 使用最大值归一化方法，将矩阵每列归一化。  
 """  
 *return matrix* / *matrix*.max(axis=0, keepdims=*True*)  
  
  
*def* construct\_fuzzy\_similarity\_matrix(*matrix*):  
 """  
 使用最大最小法构造模糊相似矩阵。  
 """  
 normalized = normalize\_by\_max(*matrix*)  
 size = normalized.shape[0]  
 similarity\_matrix = np.zeros((size, size), dtype=float)  
  
 *for* i *in* range(size):  
 *for* j *in* range(size):  
 row\_union = np.maximum(normalized[i], normalized[j])  
 row\_intersection = np.minimum(normalized[i], normalized[j])  
 similarity\_matrix[i, j] = row\_intersection.sum() / row\_union.sum()  
  
 *return* similarity\_matrix  
  
  
*def* fuzzy\_matrix\_composition(*matrix\_a*, *matrix\_b*):  
 """  
 进行模糊矩阵的合成。  
 """  
 rows, cols = *matrix\_a*.shape[0], *matrix\_b*.shape[1]  
 composed\_matrix = np.zeros((rows, cols), dtype=float)  
  
 *for* i *in* range(rows):  
 *for* j *in* range(cols):  
 composed\_matrix[i, j] = max(min(*matrix\_a*[i, k], *matrix\_b*[k, j]) *for* k *in* range(*matrix\_a*.shape[1]))  
  
 *return* composed\_matrix  
  
  
*def* compute\_transitive\_closure(*matrix*):  
 """  
 使用平方法计算传递闭包。  
 """  
 closure = construct\_fuzzy\_similarity\_matrix(*matrix*)  
 *while True*:  
 new\_closure = fuzzy\_matrix\_composition(closure, closure)  
 *if* np.allclose(new\_closure, closure):  
 *return* np.round(new\_closure, 2)  
 closure = new\_closure  
  
  
*def* compute\_cut\_set(*matrix*):  
 """  
 计算水平截集。  
 """  
 closure = compute\_transitive\_closure(*matrix*)  
 unique\_values = np.unique(closure)  
 *return* np.sort(unique\_values)[::-1]  
  
  
*def* extract\_classes(*pairs*, *size*):  
 """  
 根据给定的元素对提取聚类类集。  
 """  
 clusters = [set() *for* \_ *in* range(*size*)]  
 *for* x, y *in pairs*:  
 clusters[x].add(y)  
 clusters[y].add(x)  
  
 *return* [list(cluster) *for* cluster *in* clusters *if* cluster]  
  
  
*def* perform\_fuzzy\_clustering(*matrix*):  
 """  
 执行模糊聚类并返回聚类结果。  
 """  
 cut\_levels = compute\_cut\_set(*matrix*)  
 closure = compute\_transitive\_closure(*matrix*)  
 size = *matrix*.shape[0]  
  
 clustering\_results = []  
 *for* level *in* cut\_levels:  
 *if* level == cut\_levels[0]:  
 clustering\_results.append([[i] *for* i *in* range(size)])  
 *else*:  
 element\_pairs = np.argwhere(closure >= level)  
 clustering\_results.append(extract\_classes(element\_pairs, size))  
  
 *return* clustering\_results  
  
  
*def* main():  
 """  
 主函数，演示模糊聚类过程。  
 """  
 data\_matrix = np.array([[17, 15, 14, 15, 16],  
 [18, 16, 13, 14, 12],  
 [18, 18, 19, 17, 18],  
 [16, 18, 16, 15, 18]])  
  
 print("特性指标矩阵:\n", data\_matrix)  
 print("\n归一化矩阵:\n", normalize\_by\_max(data\_matrix))  
 print("\n模糊相似矩阵:\n", construct\_fuzzy\_similarity\_matrix(data\_matrix))  
 print("\n传递闭包:\n", compute\_transitive\_closure(data\_matrix))  
 print("\n水平截集:\n", compute\_cut\_set(data\_matrix))  
 print("\n模糊聚类结果:\n", perform\_fuzzy\_clustering(data\_matrix))  
  
  
*if* \_\_name\_\_ == "\_\_main\_\_":  
 main()

###### 实验内容

运行和理解示例代码，回答下列问题：

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？

答：   
最大最小法是通过对输入矩阵的行与行之间求“最小值”和“最大值”来计算的。因此，行与行之间的模糊相似度形成的矩阵中，行数和列数与输入矩阵的行数一致，所以得到的结果一定是一个方阵。  
在最大最小法中，任意一个行与自身进行模糊相似度计算时，最大值和最小值是相等的，因此模糊相似度的值一定为 1，即：

这满足**自反性**的定义：矩阵的对角元素均为 1。  
在最大最小法中，行 iii 和行 jjj 的模糊相似度计算是：

由于=且=，因此：

=

这保证了模糊相似矩阵是对称的。

1. 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）

答：在模糊聚类中，水平截集是通过选定一个阈值（λ）将模糊相似矩阵中的元素与阈值进行比较，将元素分类到不同的集合中。这个过程形成了一个等价关系，而等价关系唯一确定了一个划分。

具体来说：水平截集可以将矩阵中大于等于阈值 λ 的元素视为相互“相似”或“等价”。等价关系的定义包括自反性、对称性和传递性，通过传递闭包，我们确保了传递性。因此，通过逐步降低阈值，可以得到不同的等价类，从而实现对数据的分类。

3）请解释代码 72 行中两个-1 的含义：

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

答：

第一个 -1

在 reshape(-1) 中，-1 表示将数组重新塑形为一维数组，而数组的长度由原数组自动计算。

第二个 -1

在 [::-1] 中，-1 是步长，表示反向遍历数组，从而将数组进行降序排序。

1. 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？

答：判断方法：平方后的矩阵与自身进行比较，如果没有发生变化（即 c==m），说明传递性已经满足。

if (c == m).all():

return np.around(c, decimals=2)

可以这么判断的原因：

在模糊矩阵传递闭包计算中，通过不断进行矩阵的复合运算（即平方），矩阵会逐步逼近满足传递性的状态。最终，当矩阵 c 的元素不再发生变化时，它已经满足传递闭包的定义（即传递性）。

1. 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？

答：将 FuzzySimilarMatrix 函数中的最大最小法替换为算术平均最小法：

def FuzzySimilarMatrix(a):

"""

用算术平均最小法构造模糊相似矩阵

"""

a = MaxNormalization(a) # 用标准化后的数据

c = np.zeros((a.shape[0], a.shape[0]), dtype=float)

for i in range(c.shape[0]): # 遍历 c 的行

for j in range(c.shape[0]): # 遍历 c 的行

c[i, j] = np.mean(np.fmin(a[i, :], a[j, :])) # 求 i 和 j 行的平均最小值

return c

最终的聚类结果可能会发生改变。

原因：在原始的最大最小法中，模糊相似度是通过最小值与最大值的比值来衡量的，强调的是两个元素之间的局部最优匹配。而在算术平均最小法中，模糊相似度是通过所有对应元素的最小值的平均值来衡量的，强调的是整体的相似度。

这种计算方式的改变可能会影响模糊相似矩阵的数值，进而改变传递闭包和水平截集的结果，从而导致不同的聚类结果。

1. **实验结果与分析**

实验结果：

特性指标矩阵

[[17 15 14 15 16]

[18 16 13 14 12]

[18 18 19 17 18]

[16 18 16 15 18]]

采用最大值规格化法将数据规格化为

[[0.94 0.83 0.74 0.88 0.89]

[1. 0.89 0.68 0.82 0.67]

[1. 1. 1. 1. 1. ]

[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. ]]

水平截集为

[1. 0.92 0.91 0.9 ]

模糊聚类结果

[[[0], [1], [2], [3]], [[0], [1], [2, 3]], [[0, 2, 3], [1]], [0, 1, 2, 3]]

分析：

1. 原始输入矩阵描述了 4 个对象（行）在 5 个指标（列）上的特性值。对每列进行最大值标准化后，每列中的最大值被缩放为 1；其他元素的值被压缩到 [0,1] 区间，保留了原始矩阵中元素的比例关系。
2. 用模糊度计算公式，，得到模糊相似矩阵。
3. 接下来通过平方法逐次合成模糊矩阵，直到矩阵稳定为止（即满足传递性）。得到传递闭包矩阵。
4. 通过不同的阈值（λ）将模糊相似矩阵中的元素进行分类，得到水平截集。
5. 最终得到模糊聚类结果。在较高阈值时（λ=1.00），相似度要求较高，所有对象各自为类。随着阈值降低：对象 2 和 3 首先合并，表明它们的相似度最高。对象 0、2 和 3 随后合并，说明对象 0 与对象 2、3 的相似度也较高。最终，所有对象合并为一个类，表示整体的相似度达到传递闭包。
6. **小结与心得体会**

小结：

1.最大最小法与传递闭包的作用：

最大最小法通过局部最优的“最小值/最大值”来衡量相似度，构建模糊相似矩阵。

传递闭包进一步通过平方法满足传递性，使矩阵反映出间接的相似关系。

1. 聚类过程：

水平截集通过逐步降低阈值，形成不同的聚类结果。

聚类结果显示对象 2 和 3 的相似性最高，随后对象 0、2 和 3 合并，最终所有对象合并为一个类。

1. 实验验证：

实验结果符合模糊聚类的理论和方法，展示了模糊相似矩阵、传递闭包和水平截集在聚类问题中的应用。

心得体会：

通过本次实验，我深入理解了模糊聚类的原理与实现过程。本次实验不仅巩固了我对模糊聚类理论的理解，也让我具备了将理论应用于实际数据分析的能力，为后续更复杂的聚类分析任务奠定了基础。

1. **实验题目**

遗传算法求解无约束单目标优化问题

**二、实验目的**

本实验旨在通过实现遗传算法，优化一个目标函数，并进一步探讨不同操作（选择、交叉、变异等）对优化过程的影响。通过该实验，旨在达到以下目的：

1.理解遗传算法的基本原理：包括种群初始化、选择、交叉和变异操作等。

2.实现编码与解码过程：将连续的实数问题转化为二进制编码，以适应遗传算法的操作。

3.优化目标函数：使用遗传算法优化一个已知的目标函数，通过不断演化优化解的质量。

4.探索算法参数对结果的影响：例如交叉概率、变异概率、种群规模等因素对优化效果的影响。

**三、实验内容**

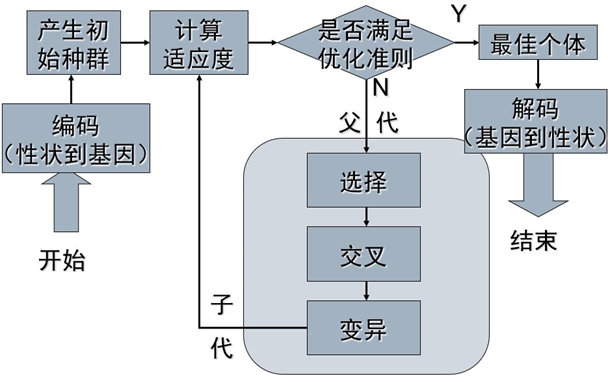
###### 1.背景知识

遗传算法（Genetic Algorithm，简称GA）是一类基于自然选择和遗传学原理的优化算法。遗传算法属于群体搜索算法，通过模拟自然界中生物的进化过程来寻找问题的最优解。该算法通过选择、交叉、变异等遗传操作，不断改善个体的适应度，从而在复杂的搜索空间中逐步逼近全局最优解。

在许多复杂问题中，传统的优化方法（如梯度下降法）往往难以有效地处理高维、非线性、复杂约束等情况，而遗传算法通过模拟自然进化过程，能够在无明确梯度信息的情况下进行全局搜索。因此，遗传算法在函数优化、机器学习、调度问题、进化学习等领域有着广泛的应用。

###### 方法和步骤

遗传算法的基本流程如下：



关键代码：

*import numpy as np  
import matplotlib.pyplot as plt  
  
  
# 定义目标函数  
def target\_function(x):  
 return x \* np.sin(10 \* np.pi \* x) + 2  
  
  
# 生成数据范围  
x\_values = np.linspace(-1, 2, 100)  
  
np.random.seed(42) # 设置随机数种子，确保生成的随机数一致  
  
# 创建初始种群  
initial\_population = np.random.uniform(-1, 2, 10) # 在范围 [-1, 2) 内生成 10 个随机浮点数  
  
# 输出初始种群及其适应度  
for individual, fitness in zip(initial\_population, target\_function(initial\_population)):  
 print(f"Individual={individual:.2f}, Fitness={fitness:.2f}")  
  
# 绘制目标函数及初始种群点  
plt.plot(x\_values, target\_function(x\_values), label="Target Function")  
plt.scatter(initial\_population, target\_function(initial\_population), color='red', label="Initial Population")  
plt.legend()  
plt.show()  
  
  
# 编码函数：将浮点数种群转为二进制编码  
def binary\_encode(data, min\_val=-1, max\_val=2, resolution=2\*\*18):  
 normalized = (data - min\_val) / (max\_val - min\_val) \* resolution  
 return np.array([np.binary\_repr(int(val), width=18) for val in normalized])  
  
  
# 解码函数：将二进制数据还原为浮点数  
def binary\_decode(binary\_data, min\_val=-1, max\_val=2, resolution=2\*\*18):  
 return np.array([(int(bits, 2) / resolution) \* (max\_val - min\_val) + min\_val for bits in binary\_data])  
  
  
# 编码初始种群  
encoded\_population = binary\_encode(initial\_population)  
  
# 输出编码后的种群信息  
for individual, binary, fitness in zip(initial\_population, encoded\_population, target\_function(initial\_population)):  
 print(f"Individual={individual:.2f}, Binary={binary}, Fitness={fitness:.2f}")  
  
# 解码并计算适应度  
decoded\_population = binary\_decode(encoded\_population)  
fitness\_values = target\_function(decoded\_population)  
  
for binary, decoded, fitness in zip(encoded\_population, decoded\_population, fitness\_values):  
 print(f"Binary={binary}, Decoded={decoded:.2f}, Fitness={fitness:.2f}")  
  
# 将适应度正则化  
fitness\_values = fitness\_values - fitness\_values.min() + 1e-6  
  
  
# 选择和交叉函数  
def select\_and\_crossover(chromosomes, fitness\_scores, crossover\_rate=0.6):  
 selection\_prob = fitness\_scores / fitness\_scores.sum()  
 cumulative\_prob = np.cumsum(selection\_prob)  
 random\_selections = np.random.rand(len(fitness\_scores))  
 selected\_chromosomes = np.array([chromosomes[np.searchsorted(cumulative\_prob, rnd)] for rnd in random\_selections])  
  
 # 随机配对进行交叉  
 num\_crossovers = int(len(selected\_chromosomes) \* crossover\_rate // 2 \* 2)  
 pairs = np.random.permutation(num\_crossovers).reshape(-1, 2)  
 midpoint = len(selected\_chromosomes[0]) // 2  
 for i, j in pairs:  
 selected\_chromosomes[i], selected\_chromosomes[j] = (  
 selected\_chromosomes[i][:midpoint] + selected\_chromosomes[j][midpoint:],  
 selected\_chromosomes[j][:midpoint] + selected\_chromosomes[i][midpoint:]  
 )  
 return selected\_chromosomes  
  
  
# 变异函数  
def mutate\_population(chromosomes, mutation\_rate=0.1):  
 mutated\_chromosomes = []  
 for chrom in chromosomes:  
 if np.random.rand() < mutation\_rate:  
 mutation\_pos = np.random.randint(len(chrom))  
 chrom = chrom[:mutation\_pos] + ('1' if chrom[mutation\_pos] == '0' else '0') + chrom[mutation\_pos + 1:]  
 mutated\_chromosomes.append(chrom)  
 return np.array(mutated\_chromosomes)  
  
  
# 绘图函数：比较两代种群  
def compare\_generations(old\_population, new\_population, fitness\_func):  
 x\_vals = np.linspace(-1, 2, 100)  
 fig, axes = plt.subplots(1, 2, figsize=(14, 5))  
  
 for ax, population, title in zip(axes, [old\_population, new\_population], ["Old Generation", "New Generation"]):  
 decoded = binary\_decode(population)  
 fitness\_scores = fitness\_func(decoded)  
 ax.plot(x\_vals, fitness\_func(x\_vals), label="Fitness Curve")  
 ax.scatter(decoded, fitness\_scores, color='red', label="Population")  
 ax.legend()  
 ax.set\_title(title)  
  
 plt.show()  
  
  
# 初始化并运行迭代过程  
np.random.seed(42)  
population = np.random.uniform(-1, 2, 100)  
encoded\_population = binary\_encode(population)  
  
for generation in range(1000):  
 fitness\_values = target\_function(binary\_decode(encoded\_population))  
 fitness\_values = fitness\_values - fitness\_values.min() + 1e-6  
  
 next\_generation = mutate\_population(select\_and\_crossover(encoded\_population, fitness\_values))  
 if generation % 300 == 1:  
 compare\_generations(encoded\_population, next\_generation, target\_function)  
 encoded\_population = next\_generation  
  
compare\_generations(encoded\_population, mutate\_population(encoded\_population), target\_function)*

###### 实验内容

运行和理解示例代码，回答下列问题：

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. | newX = np.array([chroms[np.where(probs\_cum > rand)[0][0]] |
| 2. | **for** rand **in** each\_rand]) |
|  | 答：  np.where(probs\_cum > rand)[0]：probs\_cum > rand 生成一个布尔数组，表示 probs\_cum 中哪些元素大于随机数 rand。np.where 返回布尔值为 True 的索引。[0] 提取索引结果的第一维，因为 np.where 返回的是元组。  [0]（第二次出现）：提取满足条件的第一个索引，即在累积概率中首次超过 rand 的位置。通过该索引从 chroms 中选择一个染色体。  newX 的元素数：newX 中的每个元素对应一个随机数 rand。len(newX) 等于 len(each\_rand)，也就是 fitness 的长度。此处，newX 有 10 个元素（因为种群大小为 10）。  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 |

答：

int(len(newX) \* prob // 2 \* 2)：len(newX) 是种群的大小，prob 是交叉概率，len(newX) \* prob 表示期望交叉的染色体数。//2\*2 将期望交叉数向下取整为偶数（染色体交叉需要成对进行）。

np.random.permutation：对从 0 到 int(len(newX) \* prob) 的整数进行随机排列，打乱顺序。

reshape(-1, 2)：将一维数组重新排列为两列。-1 表示自动推断行数。结果是若干个交叉配对的二维数组，每行一个配对。

3）请结合 Mutate 函数的内容，详述变异是如何实现的。

答：变异过程引入随机性，为种群增加多样性，避免陷入局部最优解。

变异具体实现步骤：

1.随机数生成：为每个染色体生成一个随机数，判断是否小于变异概率。

2.选择基因位：对需要变异的染色体，随机选择一个基因位。

3.基因翻转：将该基因位的 0 改为 1，或 1 改为 0。

4.生成新种群：将变异后的染色体加入新种群。

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？

答：

变异取消后：染色体仅通过选择和交叉更新，种群缺乏随机性和多样性。容易陷入局部最优解，整体优化能力下降。

结果表现：若初始种群中最优解不够接近全局最优，算法可能在局部最优点停滞。进化速度变慢，最终结果可能比启用变异时的效果差。

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。

答：

修改后的代码（可以实现精英个体始终保留在种群中，防止优秀解的丢失。）：

*def* Select\_Elitist(*chroms*, *fitness*, *elite\_count*=1):  
 # 直接晋级的精英个体  
 elite\_indices = np.argsort(-*fitness*)[:*elite\_count*]  
 elites = *chroms*[elite\_indices]  
  
 # 普通选择部分  
 probs = *fitness* / *fitness*.sum()  
 probs\_cum = np.cumsum(probs)  
 random\_selections = np.random.rand(len(*chroms*) - *elite\_count*)  
 selected = [*chroms*[np.searchsorted(probs\_cum, rnd)] *for* rnd *in* random\_selections]  
  
 # 合并精英和普通选择  
 *return* np.array(list(elites) + selected)

1. **实验结果与分析**

Individual=0.12, Fitness=1.92

Individual=1.85, Fitness=3.85

Individual=1.20, Fitness=1.85

Individual=0.80, Fitness=1.90

Individual=-0.53, Fitness=1.55

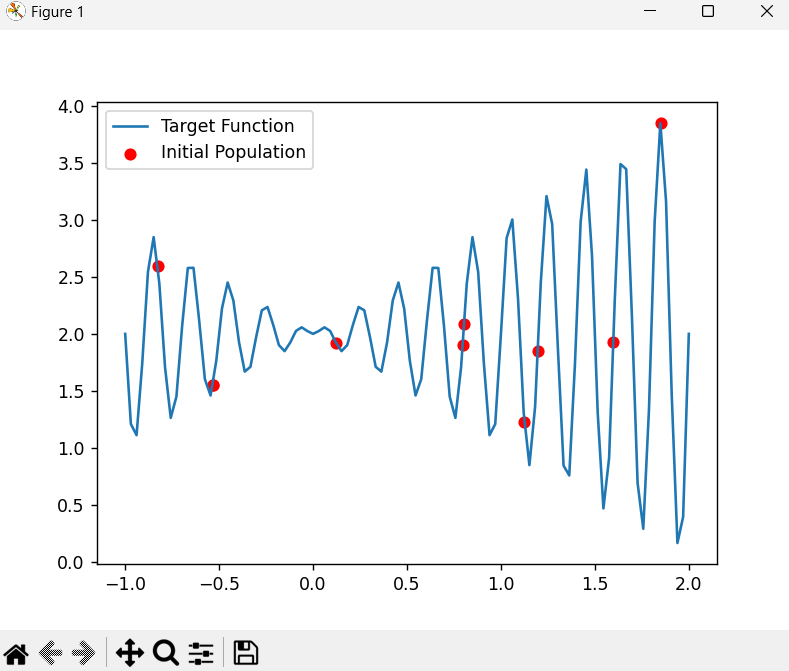
Individual=-0.53, Fitness=1.55

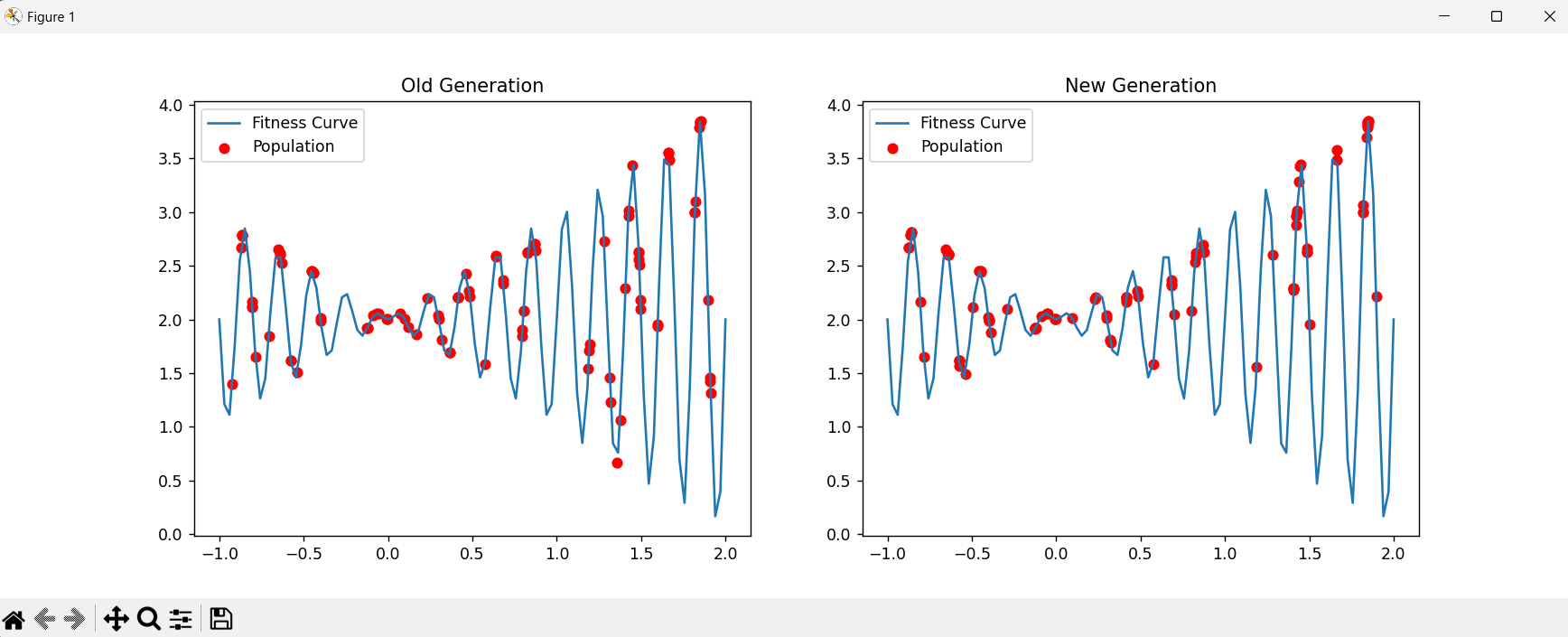
Individual=-0.83, Fitness=2.60

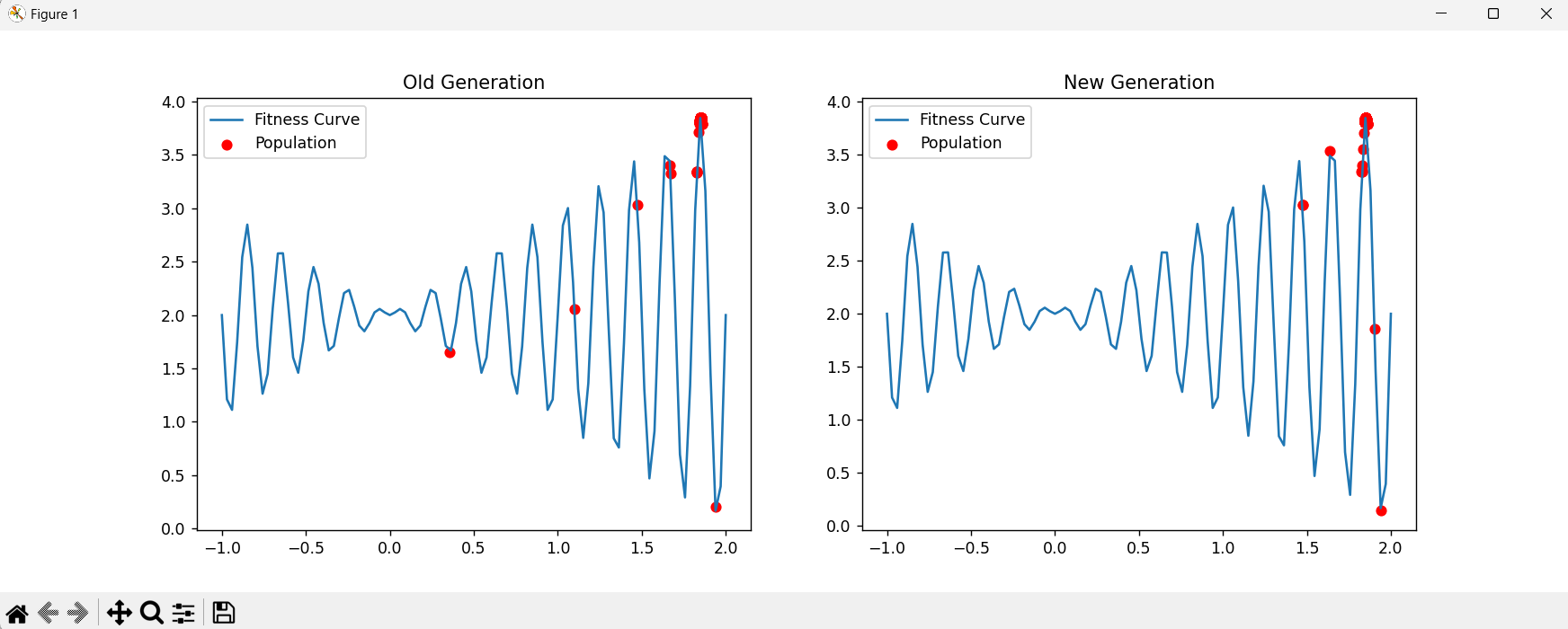
Individual=1.60, Fitness=1.93

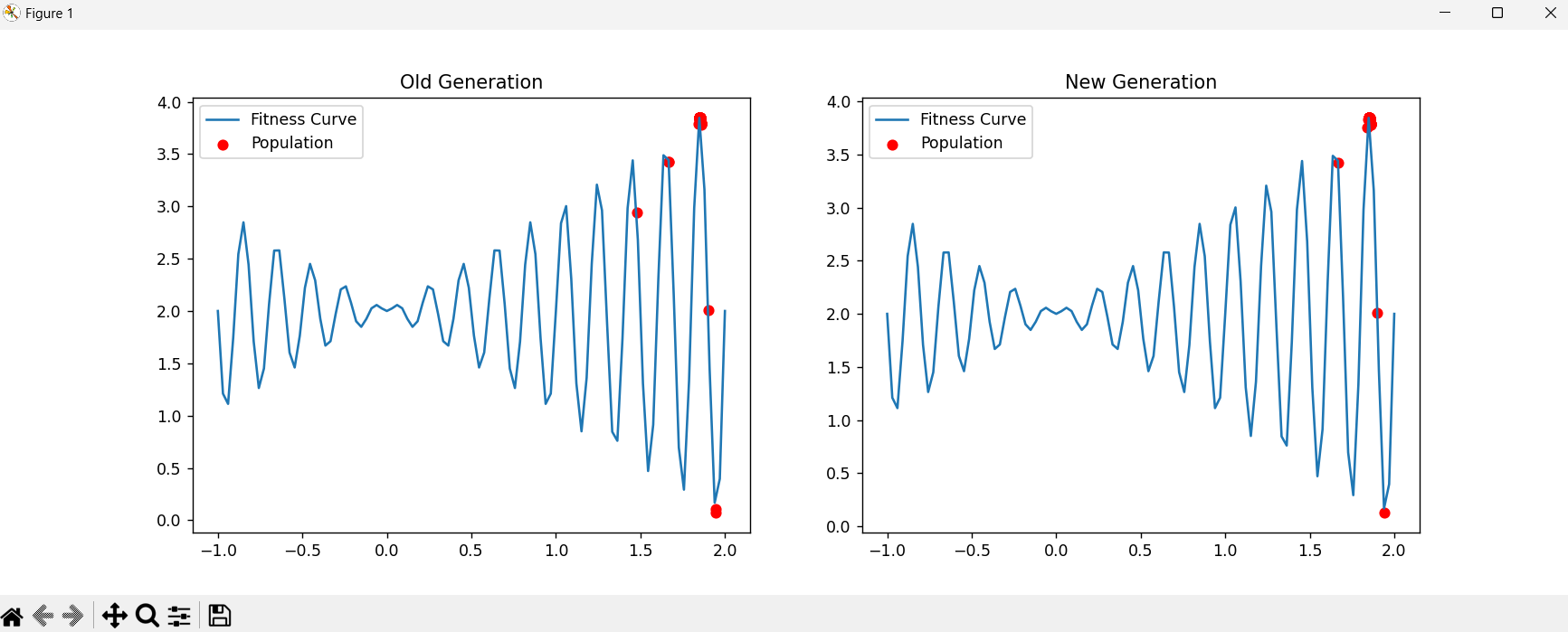
Individual=0.80, Fitness=2.08

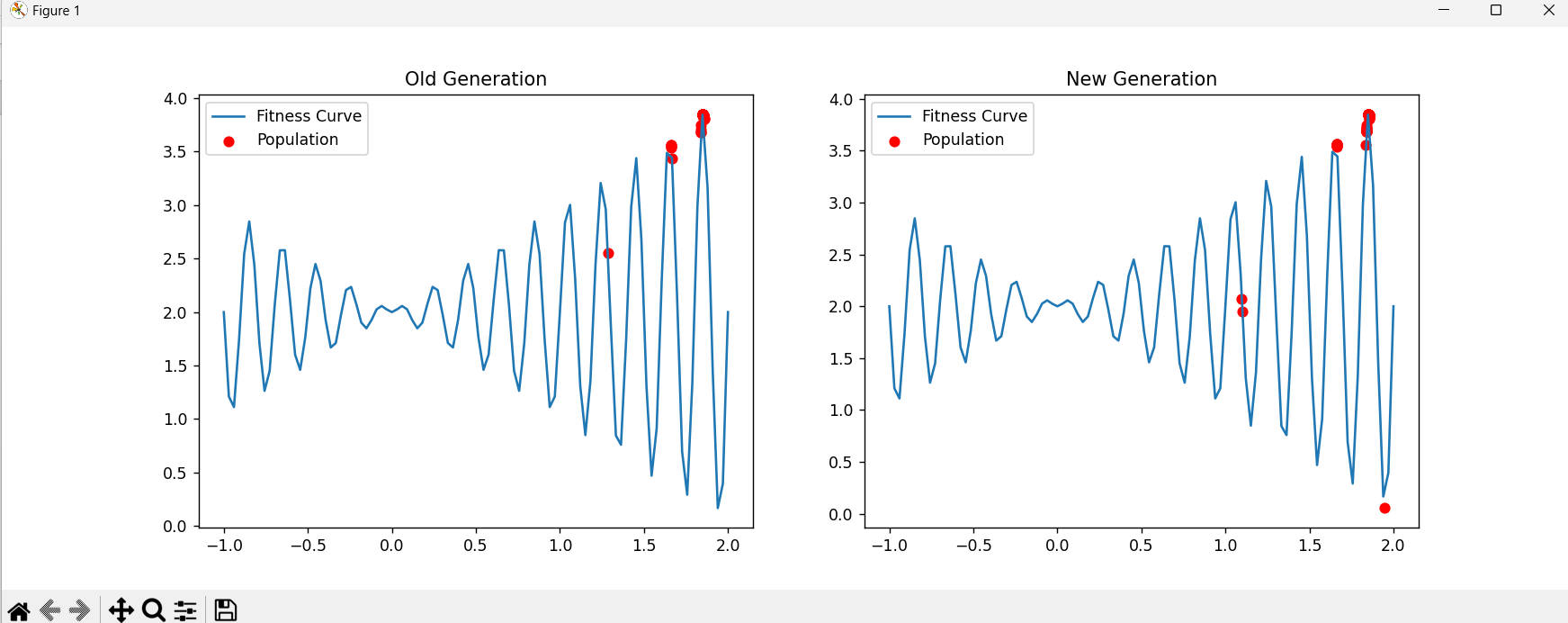
Individual=1.12, Fitness=1.22

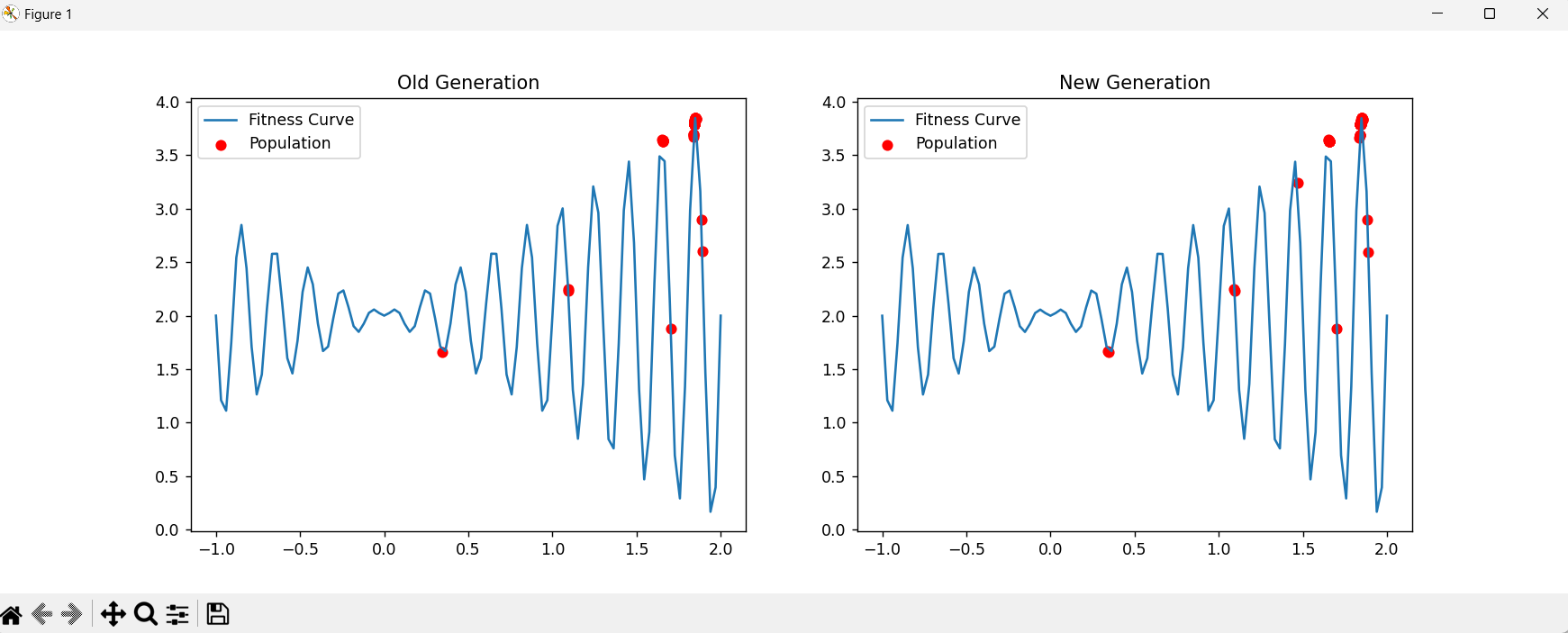












通过分析，可以看出，遗传算法在该实验中有效地优化了目标函数。通过选择、交叉、变异等操作，算法成功地引导种群从初始的随机状态逐步收敛到较优解。

初始种群的多样性较大，经过多代迭代后，适应度较高的个体逐渐取代了适应度较低的个体，说明遗传算法通过进化有效地提高了种群的质量。

交叉操作有效增强了种群的多样性，并在一定程度上提高了适应度。

变异操作引入了更多的基因变异，避免了种群过早收敛，为算法提供了更广泛的搜索空间。

最终种群的适应度趋于稳定，说明遗传算法在给定的迭代次数内已找到近似最优解。

**五、小结与心得体会**

通过本次实验，我深入理解了遗传算法的基本原理和实现过程。通过选择、交叉和变异等操作，种群逐渐从随机分布优化到接近最优解。实验中我意识到，遗传算法不仅依赖于这些基本操作，还需要通过合适的参数调整（如交叉率、变异率）来保证搜索过程的多样性和有效性。尽管算法具有随机性，但通过反复迭代，它能够找到问题的近似最优解。此次实验不仅加深了我对遗传算法的理解，还培养了我调试和优化算法的能力。