专业课程实验报告

课程名称： 机器学习

开课学期： 2021 至 2022 学年 第 1 学期

专　　业： 智能科学与技术类

年级班级： 20 级 3 班

学生姓名： 严中圣

学生学号： 222020335220177

小 组 号： 1

实验教师： 褚金

人工智能学院

《机器学习》实验课报告书

实验编号：	1	**实验名称：**	BP 神经网络用于分类
姓　　名：	严中圣	**学　　号：**	222020335220177
日　　期：	2021 年 11 月 6 日	**教师打分：**	

1 实验摘要

本次实验要求利用 BP 神经网络实现多分类任务。数据集选用Statlog (Vehicle Silhouettes) Data Set，共计 18 个特征，4 个类别属性，846 条数据。我们建立了四层神经网络进行分类任务，其中包含两个隐藏层，输出层采用 sigmoid 函数，在原始数据集上达到了 72.352941% 的准确率。为了进一步提高准确率，我们首先对数据集进行了预处理，由于发现特征间存在很强的复共线性，故利用因子分析提取出 5 个公因子，再利用 Z-score 进行数据标准化；同时引入了学习率逐步衰减机制，使训练收敛更为精准，最后经训练测试集可达 90% 的准确率，良好的完成了分类任务。

2 程序流程框图

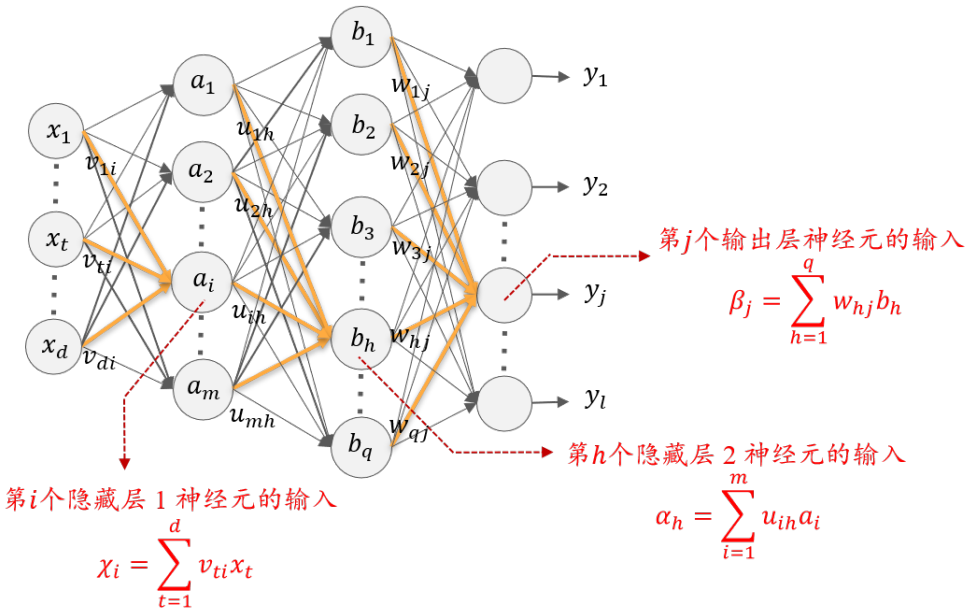

图 1: 网络结构设计图

$$
\begin{aligned}
a_m &= f(\chi_i - e_i) \\
b_h &= f(\alpha_h - \gamma_h) \\
y_j &= f(\beta_j - \theta_j)
\end{aligned} \tag{1}
$$

如图 1 所示，网络结构设计为四层网络层，包括一个输入层，一个输出层和两个隐藏层。数据输入后，通过误差逆传播 (BP) 算法对其中的权值及偏置进行不断更新。

3 实验原理

3.1 神经元模型

神经网络 (Neural Network) 是由具有适应性的简单单元组成的广泛并行互连的网络，它的组织能够模拟生物神经系统对真实世界物体所作出的交互反应。神经网络中最基本的成分是神经元模型，即上述定义中的" 简单单元" 在生物神经网络中每个神经元与其他神经元相连，当它" 兴奋" 时，就会向相连的神经元发送化学物质，从而改变这些神经元内的电位; 如果某神经元的电位超过了一个" 阈值" (threshold) 那么它就会被激活, 即" 兴奋" 起来，向其他神经元发送化学物质.

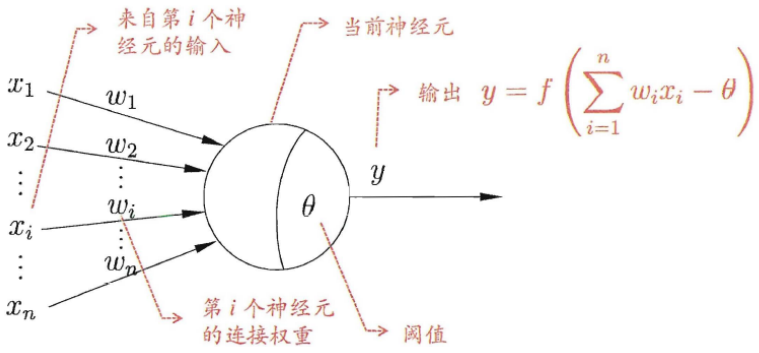

图 2: M-P 神经元模型

3.2 误差逆传播算法

对于我们所设计的如图一所示的网络结构，我们设置隐层神经元为 ReLU 函数，输出层采用 Sigmoid 函数

$$f_1 = ReLU(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{2}$$

$$f_2 = Sigmoid(x) = \frac{1}{1+e^{-x}}$$

此外设置损失函数为

$$E_k = \frac{1}{2}\sum_{j=1}^{l}(y_j^k \hat{-} y_j^k)^2 \tag{3}$$

BP 是一个迭代学习算法，在迭代的每一轮中采用广义的感知机学习规则对参数进行更新估计，则任意参数 v 的更新估计式为

$$v \leftarrow v + \Delta v \tag{4}$$

于是基于梯度下降策略，以目标的负梯度方向对参数进行调整，对式 (3) 的误差 E_k，给定学习率 η，有：

$$\Delta w_{hj} = -\eta \frac{\partial E_k}{\partial w_{hj}} \tag{5}$$

根据链式法则：

$$\frac{\partial E_k}{\partial w_{hj}} = \frac{\partial E_k}{\partial \hat{y_j}^k}\frac{\partial \hat{y_j}^k}{\partial \beta_j}\frac{\partial \beta_j}{\partial w_{hj}} \tag{6}$$

根据 β_j 的定义，显然有

$$\frac{\partial \beta_j}{\partial w_{hj}} = b_h \tag{7}$$

此外 Sigmoid 函数有一个很好的性质

$$f'(x) = f(x)(1 - f(x)) \tag{8}$$

于是设

$$g_j = -\frac{\partial E_k}{\partial \hat{y_j}^k}\frac{\partial \hat{y_j}^k}{\partial \beta_j} = \hat{y_j}^k(1 - \hat{y_j}^k)(y_j^k - \hat{y_j}^k) \tag{9}$$

所以 w_{hj} 的更新公式为

$$\Delta w_{hj} = \eta g_j b_h \tag{10}$$

同理可得：输出层偏置 θ_j 的更新公式为：

$$\Delta \theta_j = -\eta g_j \tag{11}$$

隐藏层 2 的权重及偏置更新公式为：

$$\begin{aligned}
\Delta u_{ih} &= -\eta \frac{\partial E_k}{\partial u_{ih}} \\
&= -\eta \sum_{j=1}^{l} -g_j w_{hj} b_h (1 - b_h) a_i \\
&= \sum_{j=1}^{l} \eta g_j w_{hj} b_h (1 - b_h) a_i \\
\Delta \gamma_h &= -\eta \frac{\partial E_k}{\partial \gamma_h} \\
&= -\sum_{j=1}^{l} \eta g_j w_{hj} b_h (1 - b_h)
\end{aligned} \tag{12}$$

隐藏层 1 的权重及偏置更新公式为：

$$\begin{aligned}
\Delta v_{ti} &= -\eta \frac{\partial E_k}{\partial v_{ti}} \\
&= -\sum_{j=1}^{l}\sum_{h=1}^{q} \eta g_j w_{hj} b_h (1 - b_h) u_{ih} a_i (1 - a_i) x_t \\
\Delta e_i &= -\eta \frac{\partial E_k}{\partial e_i} \\
&= \sum_{j=1}^{l}\sum_{h=1}^{q} \eta g_j w_{hj} b_h (1 - b_h) u_{ih} a_i (1 - a_i)
\end{aligned} \tag{13}$$

4 数据预处理及实验结果

4.1 数据预处理

针对所给数据集，我们首先分析了数据之间的相关性，得出相关系数热力图如附录 1 所示，可以发现许多特征之间仅仅是线性相关性便已经达到接近于 1 的相关性，这说明数据间存在很强的复共线性，极容易发生过拟合，故我们对数据进行因子分析提取主要特征。

首先对数据进行 KMO 检验和 Bartlett 球形检验，结果 KMO 测度为 0.8070，$p_{value} = 0.00336$，这进一步说明了变量间的相关性很强，适合作因子分析。对数据进行因子分析后，共提取 5 个公因子，方差解释率到达 91%，具体数据见附件 Factor_data.xlsx，代码见附录 2。

其次又发现数据间由于量纲等原因造成了巨大差异，故采用 Z-score 标准化对数据进行处理

$$x^* = \frac{x - \mu}{\sigma} \tag{14}$$

4.2 网络训练过程及实验结果

起初我们固定学习率进行训练时，若学习率过大尽管可以快速趋于极大值，但是最后会出现极大的波动，导致无法收敛，若学习率过小又会导致收敛过慢，训练时间太长。故为了更好的达到收敛效果，我们引入了学习率衰减机制，利用指数减缓的方式进行调整

$$\eta = 0.95^{epoch\_num} * \eta_0 \tag{15}$$

下面对网络进行训练，训练结果在训练集上准确率可达 90.107379%，在测试集数据上准确率达到了 88.584475%，可见如期比较良好地完成了分类任务。

5 实验总结

本次实验手写了一个含两个隐藏层的 BP 神经网络实现了多分类任务，在编码的过程中加深了对算法的理解，同时也掌握了许多算法优化的方法，也很好地锻炼的数学推导和编写程序的能力，也进一步地感受到了神经网络的强大之处。神经网络算法在当前深度学习火热的大环境下，广泛运用于图像处理、自然语言处理、控制算法等多个领域，一再验证了其效果的强大。这也更激励着我对神经网络和深度学习领域进一步的探索，期待未来能够为此领域做出自己的贡献。

6 参考文献

[1] 周志华. 机器学习 [M]. 清华大学出版社, 2016.

[2] Smith L N. Cyclical learning rates for training neural networks[C]//2017 IEEE winter conference on applications of computer vision (WACV). IEEE, 2017: 464-472.

[3] https://zhuanlan.zhihu.com/p/285601835

[4] https://blog.csdn.net/han_xiaoyang/article/details/50521064

==========

三线表:

表 1: “添加好友”页界面各控件属性设置

名称	类型	属性设置
newGroupComboBox	QComboBox	默认
newIDLineEdit	QLineEdit	默认
FriendsTableView	QTableView	horizontalHeaderVisibe: 勾选 horizontalHeaderDefaultSectionSize:120 horizontalHeaderMinimumSectionSize:25 horizionHeaderStretchLastSection: 勾选 verticalHeaderVisible: 取消勾选

有序列表:

(1) 能够实现 QQ 登录系统并具有独立的登录界面;

(2) 能够实现用户通过口令登录，且密码采用 MD5 加密算法封装验证;

(3) 能够实现 QQ 好友管理系统并具有独立的系统界面;

(4) 能够实现通过鼠标触发事件进行软件操作;

代码块:

```
QString LoginDialog::strToMd5(QString str)
{
    QString strMd5;
    QByteArray qba;
    qba=QCryptographicHash::hash(str.toLatin1(),QCryptographicHash::Md5);
   //调用QCryptographicHash类中生成密码散列的方法，生成二进制或文本数据
   的加密散列值
    strMd5.append(qba.toHex());
    return strMd5;
}
```

附录

A 附录 1 – 原始数据相关系数热力图

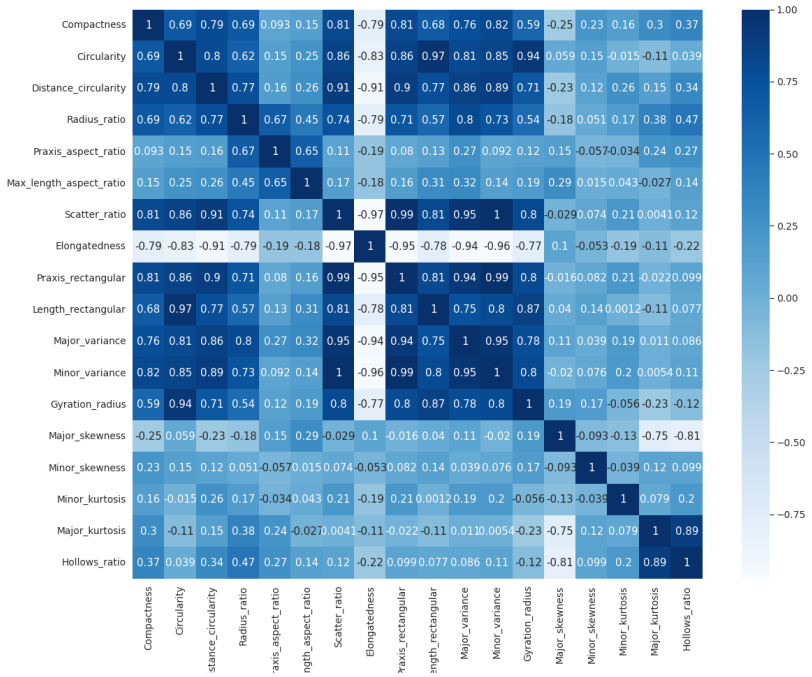

图 3: 相关系数热力图

B 附录 2 – 因子分析代码

```
! pip install factor_analyzer
import pandas as pd
import numpy as np
from pandas import DataFrame,Series
from factor_analyzer import FactorAnalyzer
import numpy.linalg as nlg
from math import sqrt
from numpy import eye, asarray, dot, sum, diag #导入eye,asarray,dot,sum,
    diag 函数
from numpy.linalg import svd #导入奇异值分解函数
import warnings
warnings.filterwarnings("ignore")

dat=pd.read_csv('C:\\Users\\14576\\Desktop\\data.csv')
data=dat.iloc[:,1:19]
C=data.corr()
def kmo(dataset_corr):
    corr_inv = np.linalg.inv(dataset_corr)
    nrow_inv_corr, ncol_inv_corr = dataset_corr.shape
```

```
    A = np.ones((nrow_inv_corr, ncol_inv_corr))
    for i in range(0, nrow_inv_corr, 1):
        for j in range(i, ncol_inv_corr, 1):
            A[i, j] = -(corr_inv[i, j]) / (math.sqrt(corr_inv[i, i] *
   corr_inv[j, j]))
            A[j, i] = A[i, j]
    dataset_corr = np.asarray(dataset_corr)
    kmo_num = np.sum(np.square(dataset_corr)) - np.sum(np.square(np.
   diagonal(A)))
    kmo_denom = kmo_num + np.sum(np.square(A)) - np.sum(np.square(np.
   diagonal(A)))
    kmo_value = kmo_num / kmo_denom
    return kmo_value
print("\nKMO测度:", kmo(C))
# 巴特利特球形检验
df2_corr1 = C.values
print(df2_corr1.shape)
print("\n巴特利特球形检验:", bartlett(df2_corr1[0], df2_corr1[1],
   df2_corr1[2], df2_corr1[3], df2_corr1[4],
                              df2_corr1[5], df2_corr1[6], df2_corr1[7],
   df2_corr1[8], df2_corr1[9],
                              df2_corr1[10], df2_corr1[11], df2_corr1
   [12],df2_corr1[13],df2_corr1[14],
                              df2_corr1[15],df2_corr1[16],df2_corr1[17]))
eig_value,eig_vector=nlg.eig(C) #计算特征值和特征向量
eig=pd.DataFrame() #利用变量名和特征值建立一个数据框
eig['names']=data.columns#列名
eig['eig_value']=eig_value#特征值

for k in range(1,17): #确定公共因子个数
    if eig['eig_value'][:k].sum()/eig['eig_value'].sum()>=0.9: #如果解释
   度达到80%, 结束循环
        print(k)
        break
print(eig['eig_value'][:5].sum()/eig['eig_value'].sum())#方差解释率

col0=list(sqrt(eig_value[0])*eig_vector[:,0]) #因子载荷矩阵第1列
col1=list(sqrt(eig_value[1])*eig_vector[:,1]) #因子载荷矩阵第2列
col2=list(sqrt(eig_value[2])*eig_vector[:,2]) #因子载荷矩阵第3列
col3=list(sqrt(eig_value[3])*eig_vector[:,3]) #因子载荷矩阵第4列
col4=list(sqrt(eig_value[4])*eig_vector[:,4]) #因子载荷矩阵第5列
A=pd.DataFrame([col0,col1,col2,col3,col4]).T #构造因子载荷矩阵A
A.columns=['factor1','factor2','factor3','factor4','factor5'] #因子载荷矩
   阵A的公共因子
h=np.zeros(18) #变量共同度, 反映变量对共同因子的依赖程度, 越接近1, 说明公
   共因子解释程度越高, 因子分析效果越好
D=np.mat(np.eye(18))#特殊因子方差, 因子的方差贡献度 , 反映公共因子对变量
```

```
    的贡献，衡量公共因子的相对重要性
A=np.mat(A) #将因子载荷阵A矩阵化
for i in range(18):
    a=A[i,:]*A[i,:].T #行平方和
    h[i]=a[0,0]  #计算变量X共同度,描述全部公共因子F对变量X_i的总方差所做
    的贡献，及变量X_i方差中能够被全体因子解释的部分
    D[i,i]=1-a[0,0] #因为自变量矩阵已经标准化后的方差为1，即Var(X_i)=第i
    个共同度h_i + 第i个特殊因子方差
def varimax(Phi, gamma = 1.0, q =18, tol = 1e-6): #定义方差最大旋转函数
    p,k = Phi.shape #给出矩阵Phi的总行数，总列数
    R = eye(k) #给定一个k*k的单位矩阵
    d=0
    for i in range(q):
        d_old = d
        Lambda = dot(Phi, R)#矩阵乘法
        u,s,vh = svd(dot(Phi.T,asarray(Lambda)**3 - (gamma/p) * dot(
   Lambda, diag(diag(dot(Lambda.T,Lambda))))))) #奇异值分解svd
        R = dot(u,vh)#构造正交矩阵R
        d = sum(s)#奇异值求和

    if d_old!=0 and d/d_old:
        return dot(Phi, R)#返回旋转矩阵Phi*R

rotation_mat=varimax(A)#调用方差最大旋转函数
rotation_mat=pd.DataFrame(rotation_mat)#数据框化
data=np.mat(data) #矩阵化处理

factor_score=(data).dot(A) #计算因子得分
factor_score=pd.DataFrame(factor_score)#数据框化
factor_score.columns=['factorA','factorB','factorC','factorD','factorE']
   #对因子变量进行命名
```

C 附录 3 – BP 神经网络训练代码

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler
import numpy.linalg as nlg
from matplotlib import cm
import math
from math import sqrt

# 读取数据
f = open(r'vehicle.dat', encoding='utf-8')
```

```
sentimentlist = []
count = 0
for line in f:
    # print(line)
    if count <= 21:
        count += 1
        continue
    s = line.strip().split(', ')
    sentimentlist.append(s)
    count += 1
f.close()
vehicle_data = pd.DataFrame(sentimentlist,columns=['Compactness', '
   Circularity', 'Distance_circularity', 'Radius_ratio',
                                    'Praxis_aspect_ratio', '
   Max_length_aspect_ratio','Scatter_ratio','Elongatedness',
                                    'Praxis_rectangular', '
   Length_rectangular', 'Major_variance',
                                    'Minor_variance','Gyration_radius', '
   Major_skewness', 'Minor_skewness',
                                    'Minor_kurtosis','Major_kurtosis', '
   Hollows_ratio', 'Class'])
Class = vehicle_data[['Class']]
del vehicle_data['Class']

Class_num = Class.copy(deep=True)  # 标签数值化
uniq_list = list(np.unique(Class))
for i in range(len(Class)):
    for j in range(7):
        if Class_num.iloc[i].item() == uniq_list[j]:
            Class_num.loc[i] = j
# 对dataframe里的数据类型进行修改
Class_num = np.array(Class_num, dtype=np.intc)
Class_num = pd.DataFrame(Class_num, columns=['Class'])
vehicle_data=StandardScaler().fit(vehicle_data).transform(vehicle_data)
#vehicle_data = np.array(vehicle_data, dtype=np.float)
vehicle_data = pd.DataFrame(vehicle_data, columns=['Compactness', '
   Circularity', 'Distance_circularity', 'Radius_ratio',
                                                      'Praxis_aspect_ratio'
   , 'Max_length_aspect_ratio', 'Scatter_ratio',
                                                      'Elongatedness', '
   Praxis_rectangular', 'Length_rectangular',
                                                      'Major_variance', '
   Minor_variance',
                                                      'Gyration_radius', '
   Major_skewness', 'Minor_skewness',
                                                      'Minor_kurtosis','
   Major_kurtosis','Hollows_ratio'])
```

```
class Layer:
    def __init__(self, n_input, n_output, activation=None, weights=None,
    bias=None):
        self.activation = activation
        self.weights = weights if weights is not None else np.random.
    randn(n_input, n_output) * np.sqrt(1 / n_output)
        self.bias = bias if bias is not None else np.random.rand(n_output
    ) * 0.1
        self.activation_output = None

    def forward(self, x_input):
        r = np.dot(x_input, self.weights) - self.bias  # 向量点积，结果为
    output维数
        self.activation_output = self.apply_activation(r)
        return self.activation_output

    def apply_activation(self, r):
        if self.activation is None:
            return r
        elif self.activation == 'relu':
            return np.maximum(r, 0)
        elif self.activation == 'sigmoid':
            x_ravel = r.ravel()  # 将numpy数组展平
            length = len(x_ravel)
            y = []
            for index in range(length):
                if x_ravel[index] >= 0:
                    y.append(1.0 / (1 + np.exp(-x_ravel[index])))
                else:
                    y.append(np.exp(x_ravel[index]) / (np.exp(x_ravel[
    index]) + 1))
            return np.array(y).reshape(r.shape)
            # return 1 / 1 + np.exp(-r)

    def apply_activation_derivative(self, r):
        if self.activation is None:
            return np.ones_like(r)
        elif self.activation == 'relu':
            grad = np.array(r, copy=True)
            grad[r > 0] = 1.
            grad[r <= 0] = 0.
            return grad
        elif self.activation == 'sigmoid':
            return r * (1 - r)
        return r
```

```
class Network:
    def __init__(self):
        self.layers = []
        self.train_loss = []
        self.test_loss = []
        self.train_accuracy = []
        self.test_accuracy = []

    def add_layer(self, layer):
        self.layers.append(layer)

    def feed_forward(self, x_input):
        # 前向传播
        for layer in self.layers:
            x_input = layer.forward(x_input)
        return x_input

    def backward(self, X, y, learning_rate):
        # 反向传播
        output = self.feed_forward(X)
        g = output * (1 - output) * (y - output)  # g.size=[n_output,1]
        for i in reversed(range(len(self.layers))):
            layer = self.layers[i]
            if layer == self.layers[-1]:  # 输出层
                last_layer = self.layers[i - 1]
                # print(len(last_layer.activation_output))
                delta_weight = [[] for q in range(len(last_layer.
activation_output))]
                for h in range(len(last_layer.activation_output)):
                    for j in range(len(layer.activation_output)):
                        delta_weight[h].append(learning_rate * g[j] *
last_layer.activation_output[h])
                delta_weight = np.array(delta_weight)*learning_rate
                # delta_weight=learning_rate*g*last_layer.
activation_output
                layer.weights = layer.weights + delta_weight
                delta_bias = -learning_rate * g
                layer.bias = layer.bias + delta_bias
            else:
                next_layer = self.layers[i + 1]
                if i + 1 == len(self.layers) - 1:  # 输出层前一隐藏层
                    last_layer = self.layers[i - 1]
                    delta_weight = [[] for m in range(len(last_layer.
activation_output))]  #u_ih
                    for ai in range(len(last_layer.activation_output)):
                        for h in range(len(layer.activation_output)):
```

```
                            sum = 0
                            for j in range(len(next_layer.
activation_output)):
                                sum += next_layer.weights[h][j] * g[j] *
layer.activation_output[h] * (
                                        1 - layer.activation_output[h]) *
 last_layer.activation_output[ai]
                            delta_weight[ai].append(sum)
                    delta_weight = np.array(delta_weight) *learning_rate
                    layer.weights = layer.weights + delta_weight
                    delta_bias = []
                    for h in range(len(layer.activation_output)):
                        sum = 0
                        for j in range(len(next_layer.activation_output))
:
                            sum += next_layer.weights[h][j] * g[j] *
layer.activation_output[h] * (
                                    1 - layer.activation_output[h]) *
(-1)
                        delta_bias.append(sum)
                    delta_bias = np.array(delta_bias) * learning_rate
                    layer.bias += delta_bias
                else:
                    output_layer = self.layers[-1]
                    delta_weight = [[] for i in range(len(X))]
                    for t in range(len(X)):
                        for ai in range(len(layer.activation_output)):
                            sum = 0
                            for h in range(len(next_layer.
activation_output)):
                                for j in range(len(y)):
                                    sum += output_layer.weights[h][j] * g
[j] * next_layer.activation_output[h] * (
                                            1 - next_layer.
activation_output[h]) * (next_layer.weights[ai][h]) * \
                                        layer.activation_output[ai] *
(1 - layer.activation_output[ai]) * X[t]
                            delta_weight[t].append(sum)
                    delta_weight = np.array(delta_weight) * learning_rate
                    layer.weights += delta_weight
                    delta_bias = []
                    for ai in range(len(layer.activation_output)):
                        sum = 0
                        for h in range(len(next_layer.activation_output))
:
                            for j in range(len(y)):
                                sum += output_layer.weights[h][j] * g[j]
```

```
    * next_layer.activation_output[h] * (
                                          1 - next_layer.activation_output[
    h]) * (next_layer.weights[ai][h]) * \
                                         layer.activation_output[ai] * (1 -
     layer.activation_output[ai]) * (-1)
                        delta_bias.append(sum)
                    delta_bias = np.array(delta_bias)*learning_rate
                    layer.bias = layer.bias + delta_bias

     def train(self, X_train, X_test, y_train, y_test, learning_rate,
    max_epochs):
         # 对标签进行one-hot编码
         y_onehot = np.zeros((y_train.shape[0], 4))
         y_onehot[np.arange(y_train.shape[0]), np.array(y_train).flatten()
    ] = 1
         y_test_onehot = np.zeros((y_test.shape[0], 4))
         y_test_onehot[np.arange(y_test.shape[0]), np.array(y_test).
    flatten()] = 1
         mses = []  #train_loss
         mses_test = [] #test_loss
         rate = learning_rate
         for epoch in range(max_epochs):
             # if epoch % 10 == 0:
             learning_rate = rate * math.pow(0.95,epoch)
             # learning_rate = learning_rate /2
             # print("current learning rate is %.2f"%learning_rate)
           for x in range(len(X_train)):
               self.backward(np.array(X_train.iloc[x]), y_onehot[x],
    learning_rate)
           mse = np.mean(np.square(y_onehot - self.feed_forward(X_train)))
           mses.append(mse)
           self.train_loss.append(mse)
           for x in range(len(X_test)):
             self.backward(np.array(X_test.iloc[x]), y_test_onehot[x],
    learning_rate)
           mse_test = np.mean(np.square(y_test_onehot - self.feed_forward(
    X_test)))
           mses_test.append(mse_test)
           self.test_loss.append(mse_test)
           train_ac = self.accuracy(self.predict(X_train),np.array(y_train
    ).flatten())*100
           self.train_accuracy.append(train_ac)
           print('Epoch: #%s, train loss: %f, test_loss: %f,
    train_Accuracy: %f%%, test_Accuracy: %f%%' %
                 (epoch + 1, float(mses[epoch]), float(mses_test[epoch]),
    train_ac,
                     self.accuracy(self.predict(X_test), np.array(y_test).
```

```
    flatten()) * 100))
            print("")

    def accuracy(self, y_predict, y_test):  # 计算准确度
        ac = np.sum(y_predict == y_test) / len(y_test)
        self.test_accuracy.append(ac)
        return ac

    def predict(self, X_predict):
        y_predict = self.feed_forward(X_predict)
        y_predict = np.argmax(y_predict, axis=1)
        return y_predict

#训练网络
X_train, X_test, y_train, y_test = train_test_split(data, Class_num,
    test_size=0.2, random_state=19)
nn = Network()  # 实例化网络类
nn.add_layer(Layer(5, 4, 'relu'))  # 隐藏层 1
nn.add_layer(Layer(4, 4, 'relu'))  # 隐藏层 2
nn.add_layer(Layer(4, 4, 'sigmoid'))  # 输出层
nn.train(X_train, X_test, y_train, y_test, learning_rate=0.1, max_epochs
    =200)
```