**神经网络在自适应光学中的应用**

**项目总结报告**

**学生姓名：吴光宇**

**指导老师：李新阳**

目录

[1. 课题研究意义 3](#_Toc491180379)

[2. 国内外发展现状 3](#_Toc491180380)

[3. 研究内容 3](#_Toc491180381)

[3.1. 工具及环境 3](#_Toc491180382)

[3.1.1. 硬件配置 3](#_Toc491180383)

[3.1.2. 软件环境 3](#_Toc491180384)

[3.1.3. 开发环境介绍 4](#_Toc491180385)

[3.2. 算法简介 4](#_Toc491180386)

[3.2.1. 算法选择 4](#_Toc491180387)

[3.2.2. 基于BP网络的多层感知器算法原理介绍 5](#_Toc491180388)

[3.2.3. 基于BP网络的多层感知器回归算法及其缺陷 7](#_Toc491180389)

[3.2.4. K-Means聚类算法 8](#_Toc491180390)

[3.3. 代码实现 9](#_Toc491180391)

[3.3.1. 数据处理 9](#_Toc491180392)

[3.3.2. 基于BP网络的多层感知器回归 14](#_Toc491180393)

[3.3.3. K-Means算法 17](#_Toc491180394)

[4. 不足之处及展望 18](#_Toc491180395)

# 课题研究意义

通过实验数据探索事物潜在的物理规律是传统科学研究的一种重要方法，但是基于过去存在的算法和IT架构无法有效处理科学实验中产生的庞大数据，而近年来基于分布式计算体系提供的大量计算资源及衍生出的内存计算框架，为处理科学实验产生的庞大数据提供了可能。

作为物理规律的直接承载体——数据，我们是否可以在还未得出物理规律的前提下，通过数据预测一些可能出现的现象，甚至通过海量计算能力以及相关的算法分析出物理规律。

神经网络在自适应光学中的应用，该课题希望通过基于分布式的神经网络算法对实验数据进行处理，能够预测一些可能出现的现象或者得出一些可能存在的物理规律。

# 国内外发展现状

国内外关于神经网络在自适应光上的应用相关资料较少。

# 研究内容

工具及环境

### 硬件配置

物理机2台

### 软件环境

分布式环境组件：

1. hadoop2.7.3
2. scala2.11.8
3. spark2.0.2测试版本（非正式发行版本）
4. zeppelin0.7.0

IDE

IntellijIdea

机器学习库：

MLLib

使用的语言：

Java，python，scala

### 开发环境介绍

Hadoop是近年来比较流行的开源分布式计算框架，有着较为成熟的生态体系和比较完备的各种资源，spark是hadoop生态系统中比较重要的一个组件，他是hadoop中mapreduce计算框架的延伸和发展，与mapreduce不同的是它采用内存作为中间存储，极大得提高了整个分布式系统得IO上限，同时优化了任务分配机制，有效得提高了计算速度。

Zeppelin是一个展示层组件，可以提供一个集群与用户交互得平台。

本次基于hadoop集群，选择得机器学习库是spark自带得MLLib，但是该机器学习库并不是太完善，官方发行的版本中没有基于BP网络的多层感知器回归算法，所以选择了带有该算法的测试版本。

Spark原生支持java，python和scala所以本次使用的语言也主要是这三种。

算法简介

### 算法选择

基于本次课题的目的：

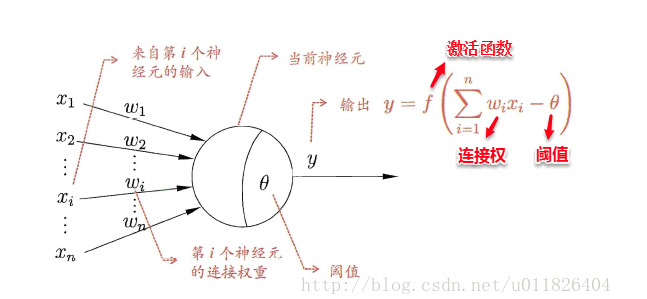
1.通过现有数据预测

2.通过现有数据进行探索

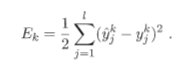
选取的算法为：基于BP网络的多层感知器回归算法和K-Means聚类算法。

对于预测问题，通常归类为回归问题，对于回归问题常使用的算法主要为广义线性回归，回归树以及神经网络。从数据结构来看，数据的分布呈现高度的非线性，如果使用广义线性回归需要拟合的参数数量太多，拟合难度会很大，所以选择了适合拟合非线性数据的神经网络（足够多的隐层神经元可以以任意精度拟合非线性函数），综合来看比较适合的算法有RBF神经网络和基于BP网络的多层感知器回归算法，这二者都是BP网络的衍生算法，不同的是RBF算法以距离中心点的欧式距离作为激活函数的自变量，并选择径向基函数作为激活函数，在样本拟合中主要体现为局部逼近，而多层感知器回归算法则接受所有的样本输入，以权值于样本的乘积作为激活函数的输入，选择Sigmoid函数作为激活函数，在样本拟合中主要体现为全局逼近。在后面的数据处理中由于将具有时间序列的离散值处理成了不具备时间序列的离散值，所以采用具有全局逼近特性的多层感知器回归算法可能效果更好。

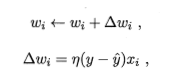
### 基于BP网络的多层感知器算法原理介绍



如上图所示，这是单个神经元的模型，当我们有一个输入向量[x1,x2,……..,xn]，以及期望的得到的正确输出向量[y1,y2,…….,ym],通过右边的式子进行一次计算，可以得到输出向量,这时输出向量和我们正确向量之间会存在差距，我们用平方误差来衡量这种差距：

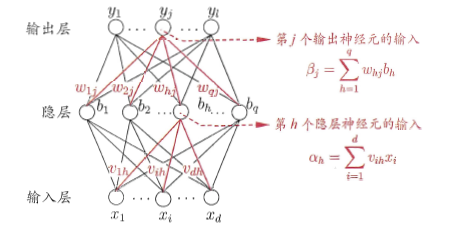


此时我们希望减小这种差距，而减小的方式就是通过调整[w1,w2,…….,wn]权值向量，可以选择按以下方式调整：



且阈值可以看成输入固定为-1的节点，然后将对阈值和权值的调整统一为权值的调整，为学习率，学习率较小可以使迭代更为平滑，但是会增加学习成本，学习率较大可以加快学习速度，但是容易造成系统震荡。

而BP网络则是以上神经元组成的一个网络：



此时训练样例记为，输出向量记为，即：



我们用均方误差来衡量整个系统的优劣：



此时我们的目标是尽可能减小Ek，即求解以权值向量为自变量，EK为因变量的最小值问题，在机器学习中，比较实用的求解最小值问题的方法有两种，第一种是梯度下降方式，第二种是拟牛顿下降方式（牛顿下降的改良版），这里以梯度下降方式来求解，首先求解梯度，并以梯度负方向，学习率为作为权值更新的值：



根据链式求导法则可得：



而根据定义可得：

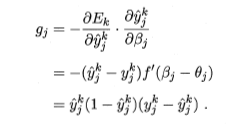


我们仍旧采用sigmoid函数作为激活函数，而该函数具有一个良好的性质：



即该函数的倒数可以完全由其本身表示而不依赖自变量。

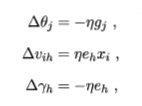
则根据上述条件可得



从而得出权值向量的更新值为



类似可以得到：



则可以递归训练整个系统直至达到最大迭代次数或者预先设定的精度要求。

### 基于BP网络的多层感知器回归算法及其缺陷

多层感知器回归算法的各层之间采用BP网络的方式进行权值更新，但是基础的多层感知器会读取一次所有的训练样本，然后计算均方误差然后进行更新，这在数据量十分庞大的时候会出现内存不足的情况，改进的BP算法可以解决这个问题。

另外一个问题是隐层的层数和每层神经元的个数，到目前为止还没有相关的理论可以直接求出上述两值的最优解，只能通过不断的参数调优，来寻求最优解，但是在相同的误差下，隐层数或神经元数较少的模型为优解。

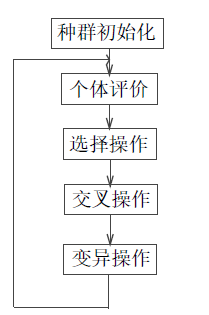
而更为关键的问题是在寻求最小值的过程中使用了梯度下降算法，而与这个方法伴生的问题就是极小值和最小值的问题，即求解的可能会是极小值而不是最小值，这会对最后得出的模型的精度有比较严重的影响，解决这个问题的方式主要有以下几种：

在调参的过程中使用随机的初始化权值向量，使梯度下降算法从不同的起点开始搜索，最后取精度最高的模型。

第二种方式是和遗传算法结合使用，遗传算法可以在整个解空间种搜索出比较靠近最优解的一个解空间，然后让梯度下降算法在该解空间种进行搜索，可以有效保障寻求到最小值而非极小值。

遗传算法介绍如下：

遗传算法就是通过不断地进化，将种群中我们想要的染色体保留下来，在进过多次进化之后，种群中大部分的染色体都会是比较优势的染色体，通常遗传算法的过程如下：



种群初始化：即确定染色体的形式，需要选择一种方式对解空间的数据进行编码（当然可以直接用原数据作为染色体，但是这样效果会很差）

迭代阶段：选择阶段，选择阶段分为适应性选择和随机选择，将预先设定的种群大小的绝大部分按比例分为适应性选择数量和随机选择数量，其中适应性数量将远大于随机选择数量，适应性选择通过适应函数对每个染色体进行评估，选取那些适应性高的进入下一代，适应性函数往往就是我们的目标函数，对于BP网络来说就是使EK函数，而存活的条件就是使EK尽可能的小，之后在未通过适应性选择的染色体中随机选取一定数量的染色体加入存活列表，这使一些具有潜在优秀基因的个体有一定几率存活下来。

将剩余的种群数量按比例分为交叉产生个体数量和变异产生个体数量，其中交叉产生个体数量远远大于变异产生个体数量（在自然界中通过变异产生提高生物适应性基因的概率很低），交叉操作就是随机选择两条染色体，然后随机选择染色体上的一段基因组成一个新的染色体，变异操作就是使染色体上的一个基因随机发送改变（通常表现为一个bit位反转）。

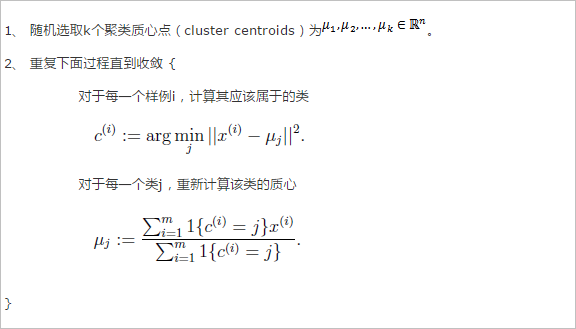
然后通过一定次数的迭代，可以有效的在解空间中搜索出多个较优解。

第三种方式是放弃梯度下降算法改用拟牛顿算法。

### K-Means聚类算法

K-Means聚类算法是聚类算法中比较简单的一种，聚类算法属于无监督学习算法，与有监督学习算法不同的是，在无监督学习算法中没有标签，即数据没有预先的分类或者预测值，而完全按照数据可能具有的规律来进行划分，所有无监督算法特别适合用来探索数据可能存在的一些规律。

K-Means算法可以将样本聚层K个cluster，具体算法描述如下：



需要注意的是，在该算法中我们需要确定的参数只有一个，那就是K，而K通常是通过确定为当集合内误差和稳定时的取值而不是当集合内误差最小时的取值，因为只要k的值等于样本的值，那么集合内误差的值就最小且为零。

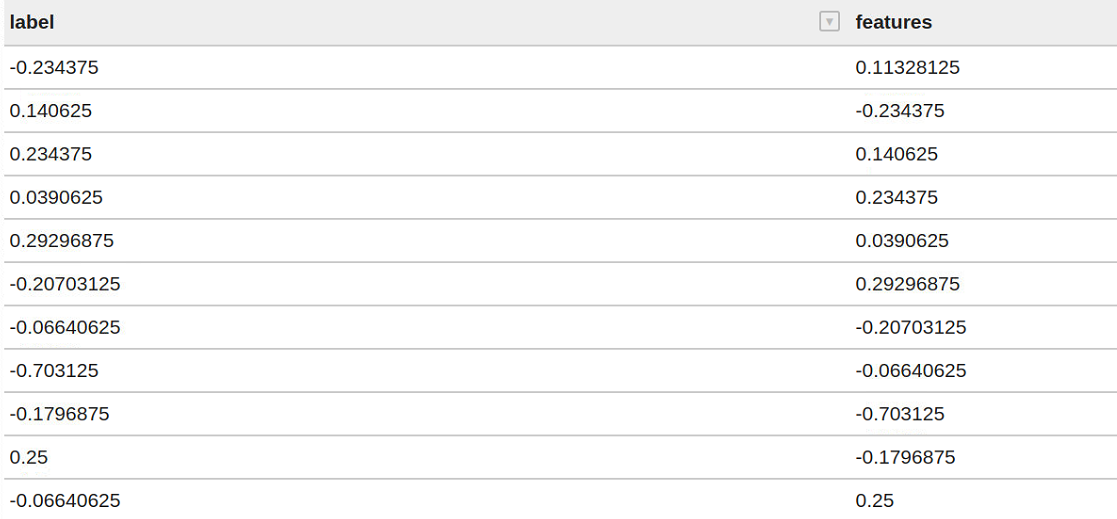
代码实现

### 数据处理

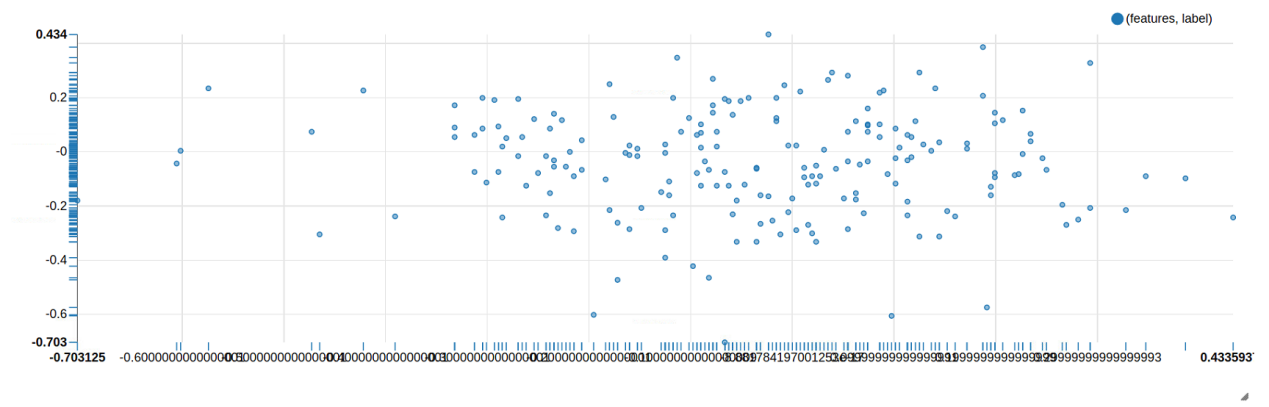
本次处理的数据明显是一组带有时间序列特征的数据，即每一帧的斜率值都有先后顺序，首先需要将这种特征去掉，去掉的方式是将将当前帧的下一帧或者下几帧作为当前帧的label，则可以将时间特性去掉，让数据变成适合传统机器学习算法的离散数据，数据读取及处理程序如下：（本次读取数据为200行（太多了是真的跑不动，这可能导致模型并未收敛））

*/\*\*  
 \* Created by kh on 2017/7/11.  
 \*/***import** java.io.\*;  
**import** java.util.ArrayList;  
  
**public class** readFile {  
 **private** String **filePath**;  
 **private** String **txtFilePath**;  
 **private** String **txtFilePathV**;  
  
 **public** readFile(String fPath){  
 **filePath**=fPath;  
 **txtFilePath**=**"E:\\testG.txt"**;  
 **txtFilePathV**=**"E:\\testV.txt"**;  
 }  
  
 **public void** readData(){  
 File dataFile=**new** File(**filePath**);  
 **if**(!dataFile.exists()&&!dataFile.canRead()){  
 System.***out***.println(**"数据文件不存在或者不可读"**);  
 System.*exit*(1);  
 }  
 File txtFile=**new** File(**txtFilePath**);  
 **if**(!txtFile.exists()){  
 **try**{  
 txtFile.createNewFile();  
 }**catch** (IOException e){  
 e.printStackTrace();  
 }  
 }  
  
 File txtFileV=**new** File(**txtFilePathV**);  
 **if**(!txtFileV.exists()){  
 **try**{  
 txtFileV.createNewFile();  
 }**catch** (IOException e){  
 e.printStackTrace();  
 }  
 }  
  
 **try**{  
 FileInputStream in=**new** FileInputStream(dataFile);  
 DataInputStream dis=**new** DataInputStream(in);  
 **byte**[] itemBuf=**new byte**[2];  
 FileOutputStream out=**new** FileOutputStream(txtFile,**true**);  
 PrintStream ps=**new** PrintStream(out);  
 FileOutputStream outV=**new** FileOutputStream(txtFileV,**true**);  
 PrintStream psV=**new** PrintStream(outV);  
  
 ArrayList panXAarryList=**new** ArrayList();  
 ArrayList panYArrayList=**new** ArrayList();  
 ArrayList panVArrayList=**new** ArrayList();  
  
 **while**(dis.read(itemBuf)!=-1){  
 **short**[] frameX=**new short**[256];  
 **short**[] frameY=**new short**[256];  
 **short**[] frameV=**new short**[512];  
  
 frameX[0]=*byteToShort*(itemBuf);  
  
 **int** xCount=1,yCount=0;  
 **for** (**int** i = 1; i < 512; i++) {  
 dis.read(itemBuf);  
 **if**(i%2!=0){  
 frameY[yCount]=*byteToShort*(itemBuf);  
 yCount++;  
 }  
 **else**{  
 frameX[xCount]=*byteToShort*(itemBuf);  
 xCount++;  
 }  
 }  
 panXAarryList.add(frameX);  
 panYArrayList.add(frameY);  
 **for** (**int** i = 0; i < 512; i++) {  
 dis.read(itemBuf);  
 frameV[i]=*byteToShort*(itemBuf);  
 }  
 panVArrayList.add(frameV);  
 }  
  
 **for** (**int** i = 0; i < 201; i++) {  
 **int** flag=0;  
 ps.println(((**short**[])panXAarryList.get(i+1))[0]/256.0+**" "**+**"1:"**+((**short**[])panXAarryList.get(i))[0]/256.0);  
 */\*for(int j=0;j<((short[])panXAarryList.get(i)).length;j++){  
 /f(flag<144){  
 String message=(j+1)+":"+(((short[])panXAarryList.get(i))[j]/256.0)\*100+" ";  
 ps.print(message);  
 flag++;  
 }  
 }\*/* }  
  
 */\*double dataX[][]=new double[144][panXAarryList.size()];  
 double dataY[][]=new double[144][panYArrayList.size()];  
 double dataV[][]=new double[153][panVArrayList.size()];  
  
 for (int i = 0; i < panXAarryList.size(); i++) {  
 int flag=0;  
 for(int j=0;j<((short[])panXAarryList.get(i)).length;j++){  
 if(flag<144){  
 dataX[j][i]=((short[])panXAarryList.get(i))[j]/256.0;  
 flag++;  
 }  
 }  
 }  
 for (int i = 0; i < panYArrayList.size(); i++) {  
 int flag=0;  
 for(int j=0;j<((short[])panYArrayList.get(i)).length;j++){  
 if(flag<144){  
 dataY[j][i]=((short[])panYArrayList.get(i))[j]/256.0;  
 flag++;  
 }  
 }  
 }  
  
 for (int i = 0; i < panVArrayList.size(); i++) {  
 int flag=0;  
 for(int j=0;j<((short[])panYArrayList.get(i)).length;j++){  
 if(flag<153){  
 dataV[j][i]=((short[])panVArrayList.get(i))[j]\*5/8191.0;  
 flag++;  
 }  
 }  
 }  
  
 for(int i=0;i<144;i++){  
 String message="第"+(i+1)+"子孔斜率：";  
 ps.println(message);  
 for (int j = 0; j < dataX[i].length; j++) {  
 ps.println(dataX[i][j]+" "+dataY[i][j]);  
 }  
 }  
  
 for(int i=0;i<153;i++){  
 String message="第"+(i+1)+"驱动器电压：";  
 psV.println(message);  
 for (int j = 0; j < dataV[i].length; j++) {  
 psV.println(dataV[i][j]);  
 }  
 }\*/* }**catch** (IOException e){  
 e.printStackTrace();  
 }**finally** {  
 }  
 }  
  
 **public static short** byteToShort(**byte**[] b) {  
 **short** s = 0;  
 **short** s0 = (**short**) (b[0] & 0xff);*// 最低位* **short** s1 = (**short**) (b[1] & 0xff);  
 s1 <<= 8;  
 s = (**short**) (s0 | s1);  
 **return** s;  
 }  
  
 **public static void** main(String[] args) {  
 readFile test=**new** readFile(**"E:\\16年07月18日\_00时29分23秒.ocamdq-gv"**);  
 test.readData();  
 }  
}

处理后数据如下：



数据分布散点图：



### 基于BP网络的多层感知器回归

（以下部分代码是MLLIB包中的源码，摘录于此处用于分析）

final val layers: IntArrayParam = new IntArrayParam(this, "layers",

"Sizes of layers including input and output from bottom to the top." +

" E.g., Array(780, 100, 10) means 780 inputs, " +

"hidden layer with 100 neurons and output layer of 10 neurons.",

(t: Array[Int]) => t.forall(ParamValidators.gt(0)) && t.length > 1

)

该部分代码定义神经网络结构，输入是一个Array[int],第一个维指明输入数据的特征值个数，比如本次采用单个子孔径的单一方向的单个斜率数据，所以特征值是一位，其实本来应该将所有子孔径的单方向斜率作为特征值带入，因为当前子孔径斜率其实会受到所有子孔径斜率的影响，即特征值是144位（单机尝试过，电脑跑崩溃了）。

final val blockSize: IntParam = new IntParam(this, "blockSize",

"Block size for stacking input data in matrices.",

ParamValidators.gt(0))

该部分代码主要定义了用于在矩阵中堆叠输入数据块的大小，可以来加快计算速度。

final val solver: Param[String] = new Param[String](this, "solver",

"The solver algorithm for optimization. Supported options: " +

s"${MultilayerPerceptronRegressor.supportedSolvers.mkString(", ")}. (Default l-bfgs)",

ParamValidators.inArray[String](MultilayerPerceptronRegressor.supportedSolvers))

该部分代码定义了用来寻找EK最小值的方式，提供了两种方式，梯度下降（gd）和拟牛顿方式（l-bfgs），默认是l-bfgs。

def setMaxIter(value: Int): this.type = set(maxIter, value)

def setTol(value: Double): this.type = set(tol, value)

该部分代码定义了最大迭代次数和模型停止误差，这是我们需要调整的比较重要的参数，当模型迭代次数等于该值或者小于给定误差的时候，模型停止训练。

final val initialWeights: Param[Vector] = new Param[Vector](this, "initialWeights",

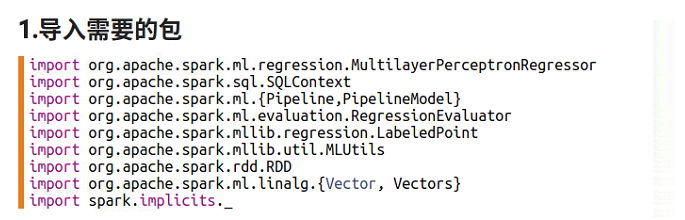
"The initial weights of the model")

该部分代码定义了初始权重，即模型训练的起点，其通过随机种子来初始化不同的初始权重。

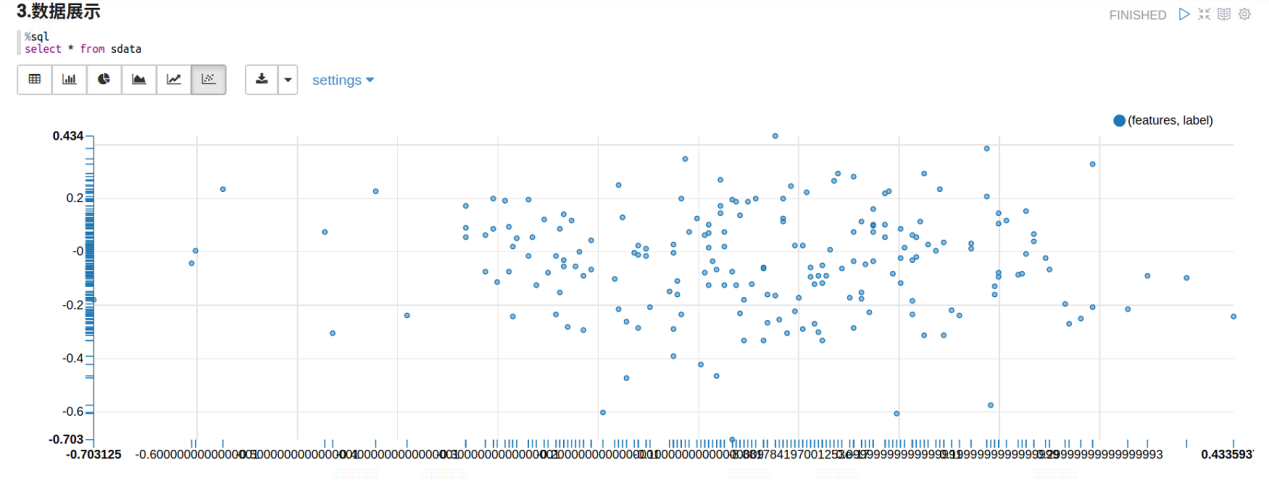
def setStandardizeLabels(value: Boolean): this.type = set(stdLabels, value)

指明是否将标签缩放到0到1之间。

以下是实现预测的代码（zeppelin平台，详细运行结果见附录，图片可以放大观看）：









import org.apache.spark.ml.regression.MultilayerPerceptronRegressor

import org.apache.spark.sql.SQLContext

import org.apache.spark.ml.{Pipeline,PipelineModel}

import org.apache.spark.ml.evaluation.RegressionEvaluator

import org.apache.spark.mllib.regression.LabeledPoint

import org.apache.spark.mllib.util.MLUtils

import org.apache.spark.rdd.RDD

import org.apache.spark.ml.linalg.{Vector, Vectors}

import spark.implicits.\_

val data = spark.read.format("libsvm").load("file:///usr/local/spark/data/TtestG.txt")

val test = spark.read.format("libsvm").load("file:///usr/local/spark/data/Rresult.txt")

case class showData(label: Double,features: Double)

val sData = spark.sparkContext.textFile("file:///usr/local/spark/data/testG.txt").map(\_.split(" ")).map(p => showData(p(0).toDouble,p(1).toDouble)).toDF()

data.show()

sData.registerTempTable("sdata")

val Array(train, test) = data.randomSplit(Array(0.8, 0.2))

var iter=100

var tol=0.0001

var pl=2

var rmse=10.0

while(rmse>0.01&&tol>0.00000001)

{

while(rmse>0.01&&iter<1050)

{

while(rmse>0.01&&pl<150)

{

val layers = Array[Int](1, pl, 1)

val trainer = new MultilayerPerceptronRegressor().setLayers(layers).setSolver("gd").setSeed(1234L).setMaxIter(iter).setTol(tol).setStepSize(0.02)

val model = trainer.fit(train)

val result = model.transform(test)

val predictionAndLabels = result.select("label", "prediction")

val evaluator = new RegressionEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("rmse")

rmse = evaluator.evaluate(predictionAndLabels)

println("当前数据集误差 = " + rmse)

pl=pl+2

println("当前神经元个数 = "+pl)

}

pl=2

iter=iter+50

println("当前迭代次数 = "+iter)

}

iter=100

tol=tol/5.0

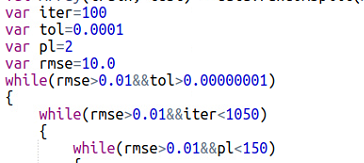
println("当前收敛误差 = "+tol)

}

（单机版BP网络（python实现）见附录test.py）

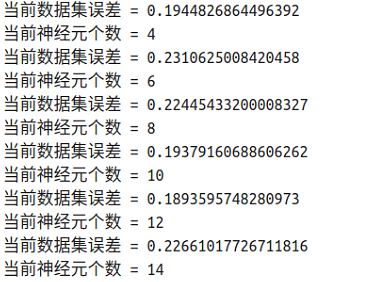
### 运行结果分析

本次实验采用了两组数据，一组数据是单子孔镜x斜率数据，数据样本量为200组，另一组数据为单子孔镜及其周边6子孔镜x数据，数据量为4095组，所有数据均采用70%训练数据和30%测试数据，数据未采用前期聚类处理。每组数据需要调整的参数依次为：隐层神经元个数，迭代次数，模型收敛误差，还可调参数为隐层层数，学习步长。



可以看到，收敛误差调节范围：0.0001-0.00000001，迭代次数为100-1050，神经元个数为2-150。

第一组数据：  
当收敛误差为0.0001：



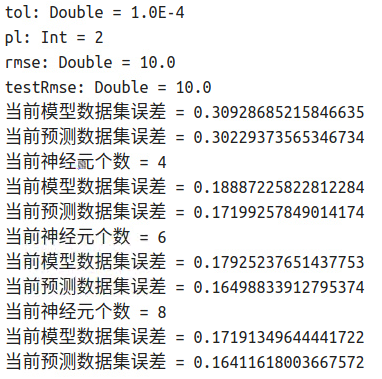
当收敛误差为0.00001时：  


当收敛误差进一步减小时：

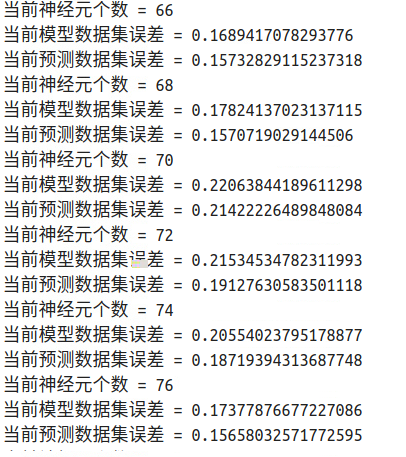


第二组数据：

当误差为0.0001时：



当误差进一步缩小后大体呈现依旧如下

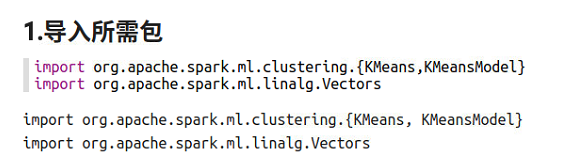


详细运行结果见附录，运行结果分析：

可以看到当采用单子孔镜数据集时普遍误差在0.19-0.21之间，但对比第二组数据可以发现，当增大数据集，增加特征个数后，误差集中在0.16-0.19之间，误差有所下降，但是误差相对来说依旧很大，这充分说明模型训练的效果及其不理想，可能原因如下：

1. 数据量不够，一般来说数据量只有达到一定程度之后，训练的模型才有意义，因为在数据量到达一定量级之后，在训练模型之前就可以对数据进行一些处理，而使其特征更加明显，使训练的模型更优。
2. 模型选取的特征值不够，单特征值是极不合理的，如前文所属，也许考虑实际情况，将子孔镜附近的子孔镜斜率也加入特征向量，拟合效果可能更好。
3. 参数优化选取的区间不合适。
4. 该数据集不适合该算法，这只能通过重新选取算法或者设计算法才能解决。

### K-Means算法





import org.apache.spark.ml.clustering.{KMeans,KMeansModel}

import org.apache.spark.ml.linalg.Vectors

import spark.implicits.\_

case class model\_instance (features: org.apache.spark.ml.linalg.Vector)

val rawData = sc.textFile("file:///usr/local/spark/data/kmeans.txt")

val df = rawData.map(line => { model\_instance( Vectors.dense(line.split(" ").map(\_.toDouble)) )}).toDF()

var k=3

while(k<100)

{

val kmeansmodel = new KMeans().setK(k).setFeaturesCol("features").setPredictionCol("prediction").fit(df)

val results = kmeansmodel.transform(df)

//results.collect().foreach(row => {println( row(0) + " is predicted as cluster " + row(1))})

println(kmeansmodel.computeCost(df))

println(k)

k=k+1

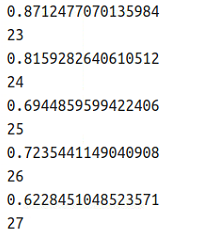
}

//通过KMeansModel类自带的clusterCenters属性获取到模型的所有聚类中心情况//

kmeansmodel.clusterCenters.foreach(center => {println("Clustering Center:"+center)})

//通过计算集合内误差平方和（Within Set Sum of Squared Error, WSSSE) 的方法来度量聚类的有效性

kmeansmodel.computeCost(df)



可以看到，在24-26的时候出现了误差增大的情况，而且在该区间附近误差稳定在0.7附近，所以该4096个数据划分位25类比较合适。（具体的运行过程及结果在附件中）

# 不足之处及展望

其实这些单独的算法应该是一个完整算法的一部分，在数据量足够大的情况下，先对数据进行预处理，可以使用一些特征提取方法，然后根据数据特征进行一些聚类处理，比如k-means聚类处理，可以有效降低数据离散度对后续拟合算法的影响，而且可以将数据本身的一些特征引用在后续拟合算法中，然后将处理过的数据作为BP网络的训练样本，采用遗传算法在解空间进行全局寻优，然后将遗传算法寻找到的解空间作为BP网络梯度寻优的初始量，进而求得最优化模型，但是需要完整完成该算法的过程，对数据的量和集群的计算能力有一定的要求，鉴于实际数据集和实际物理环境，对该算法进行了拆分，所以最后训练的模型不是太理想。

可以修改的部分其实很多，如果能过增大数据集将会使模型精度得到比较大的提升。增加参数调优时预设值的范围，以及减小参数寻优时的步长可能会使模型精度得到提升，但是这非常依赖集群的计算能力。其次在本次BP网络的选取上我们选择了多层感知机，也许使用RBF径向基网络可能会获得更好的模型，另外本次我们选择的是直接使用相关的机器学习库，也许可以自己编写相关算法代码，然后根据数据的物理意义做相应的调整，使算法的契合度更高，进而得出更为优秀的模型。

本次测试理论模型在实现的时候因为实际条件做了很多妥协，所以没有完全验证理论模型的可靠性与可行行。

后续任务可以参考上面的一些不足进行改进。