总结：本文提出了一种新的算法框架，前提：用支持向量机SVM对live数据库的5种失真图像进行数据统计，发现其中的规律。第一步，用SVM和RBF对输入图像进行分类，判定他们失真所属的类别（分别为ff,blur,wn,jpeg,jp2k），这里需要注意，我们对输入图像的估计结果不是完全的属于某一种，而是基于概率(权重)的估计，用p(i) (i=1…5)来表示,第二步根据每种失真图像的类别进行相应的打分，用q(i)(i=1…5)表示，最后图像的分数可以表示为

径向基函数（RBF）:

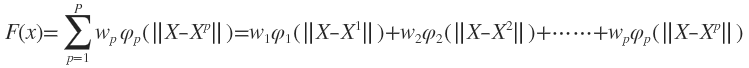
RBF网络能够逼近任意的非线性函数，可以处理系统内的难以解析的规律性，具有良好的泛化能力，并有很快的学习收敛速度，已成功应用于非线性函数逼近、时间序列分析、数据分类、模式识别、信息处理、图像处理、系统建模、控制和故障诊断等。

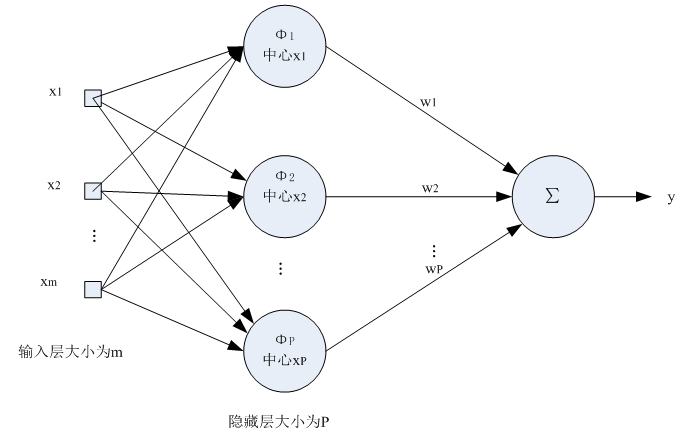
**径向基函数解决插值问题**

完全内插法要求插值函数经过每个样本点，即http://pic002.cnblogs.com/images/2012/103496/2012071417362382.png。样本点总共有P个。

RBF的方法是要选择P个基函数，每个基函数对应一个训练数据，各基函数形式为http://pic002.cnblogs.com/images/2012/103496/2012071417383089.png，由于距离是径向同性的，因此称为径向基函数。||X-Xp||表示差向量的模，或者叫2范数。

基于为径向基函数的插值函数为：

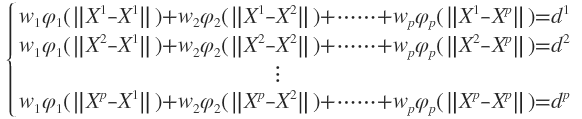




输入X是个m维的向量，样本容量为P，P>m。可以看到输入数据点Xp是径向基函数φp的中心。

隐藏层的作用是把向量从低维m映射到高维P，低维线性不可分的情况到高维就线性可分了。

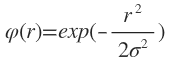
将插值条件代入：



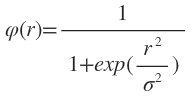
写成向量的形式为http://pic002.cnblogs.com/images/2012/103496/2012071418490996.png，显然Φ是个规模这P对称矩阵，且与X的维度无关，当Φ可逆时，有http://pic002.cnblogs.com/images/2012/103496/2012071418505872.png。

对于一大类函数，当输入的X各不相同时，Φ就是可逆的。下面的几个函数就属于这“一大类”函数：

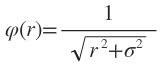
1）Gauss（高斯）函数



2）Reflected Sigmoidal（反常S型）函数



3）Inverse multiquadrics（拟多二次）函数

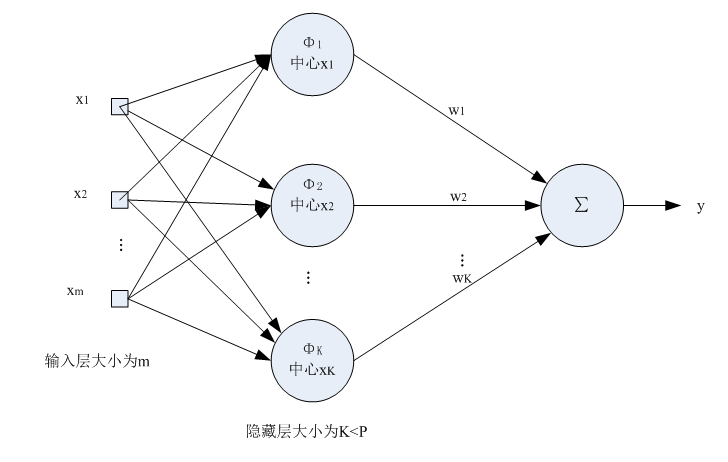


σ称为径向基函数的扩展常数，它反应了函数图像的宽度，σ越小，宽度越窄，函数越具有选择性。

完全内插存在一些问题：

1）插值曲面必须经过所有样本点，当样本中包含噪声时，神经网络将拟合出一个错误的曲面，从而使泛化能力下降。

由于输入样本中包含噪声，所以我们可以设计隐藏层大小为K，K<P，从样本中选取K个（假设不包含噪声）作为Φ函数的中心。



2）基函数个数等于训练样本数目，当训练样本数远远大于物理过程中固有的自由度时，问题就称为超定的，插值矩阵求逆时可能导致不稳定。

拟合函数F的重建问题满足以下3个条件时，称问题为适定的：

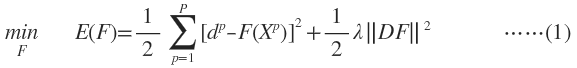
1. 解的存在性
2. 解的唯一性
3. 解的连续性

不适定问题大量存在，为解决这个问题，就引入了正则化理论。

### 正则化理论

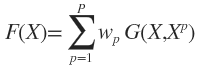
正则化的基本思想是通过加入一个含有解的先验知识的约束来控制映射函数的光滑性，这样相似的输入就对应着相似的输出。

寻找逼近函数F(x)通过最小化下面的目标函数来实现：



加式的第一项好理解，这是均方误差，寻找最优的逼近函数，自然要使均方误差最小。第二项是用来控制逼近函数光滑程度的，称为正则化项，λ是正则化参数，D是一个线性微分算子，代表了对F(x)的先验知识。曲率过大（光滑度过低）的F(x)通常具有较大的||DF||值，因此将受到较大的惩罚。

直接给出(1)式的解：



权向量http://pic002.cnblogs.com/images/2012/103496/2012071419282252.png\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(2)

G(X,Xp)称为Green函数，G称为Green矩阵。Green函数与算子D的形式有关，当D具有旋转不变性和平移不变性时，http://pic002.cnblogs.com/images/2012/103496/2012071419305611.png。这类Green函数的一个重要例子是多元Gauss函数：

http://pic002.cnblogs.com/images/2012/103496/2012071419332343.png。

### 正则化RBF网络

输入样本有P个时，隐藏层神经元数目为P，且第p个神经元采用的变换函数为G(X,Xp)，它们相同的扩展常数σ。输出层神经元直接把净输入作为输出。输入层到隐藏层的权值全设为1,隐藏层到输出层的权值是需要训练得到的：逐一输入所有的样本，计算隐藏层上所有的Green函数，根据(2)式计算权值。

### 广义RBF网络

Cover定理指出：将复杂的模式分类问题非线性地映射到高维空间将比投影到低维空间更可能线性可分。

广义RBF网络：从输入层到隐藏层相当于是把低维空间的数据映射到高维空间，输入层细胞个数为样本的维度，所以隐藏层细胞个数一定要比输入层细胞个数多。从隐藏层到输出层是对高维空间的数据进行线性分类的过程，可以采用单层感知器常用的那些学习规则，参见[神经网络基础和感知器](http://www.cnblogs.com/zhangchaoyang/articles/2588728.html)。

注意广义RBF网络只要求隐藏层神经元个数大于输入层神经元个数，并没有要求等于输入样本个数，实际上它比样本数目要少得多。因为在标准RBF网络中，当样本数目很大时，就需要很多基函数，权值矩阵就会很大，计算复杂且容易产生病态问题。另外广RBF网与传统RBF网相比，还有以下不同：

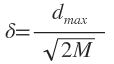
1. 径向基函数的中心不再限制在输入数据点上，而由训练算法确定。
2. 各径向基函数的扩展常数不再统一，而由训练算法确定。
3. 输出函数的线性变换中包含阈值参数，用于补偿基函数在样本集上的平均值与目标值之间的差别。

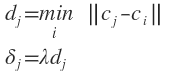
因此广义RBF网络的设计包括：

结构设计--隐藏层含有几个节点合适

参数设计--各基函数的数据中心及扩展常数、输出节点的权值。

下面给出计算数据中心的两种方法：

1. 数据中心从样本中选取。样本密集的地方多采集一些。各基函数采用统一的偏扩展常数：  
     
     
   dmax是所选数据中心之间的最大距离，M是数据中心的个数。扩展常数这么计算是为了避免径向基函数太尖或太平。
2. 自组织选择法，比如对样本进行聚类、梯度训练法、资源分配网络等。各聚类中心确定以后，根据各中心之间的距离确定对应径向基函数的扩展常数。

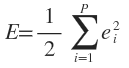
  
  
λ是重叠系数。

接下来求权值W时就不能再用http://pic002.cnblogs.com/images/2012/103496/2012071421022791.png了，因为对于广义RBF网络，其行数大于列数，此时可以求Φ伪逆。

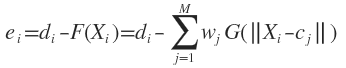
http://pic002.cnblogs.com/images/2012/103496/2012071421054823.png

 数据中心的监督学习算法

最一般的情况，RBF函数中心、扩展常数、输出权值都应该采用监督学习算法进行训练，经历一个误差修正学习的过程，与BP网络的学习原理一样。同样采用梯度下降法，定义目标函数为

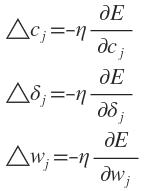


ei为输入第i个样本时的误差信号。

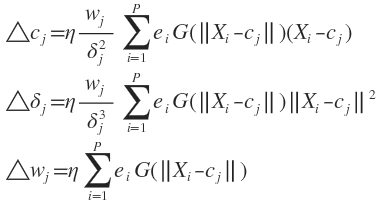


上式的输出函数中忽略了阈值。

为使目标函数最小化，各参数的修正量应与其负梯度成正比，即



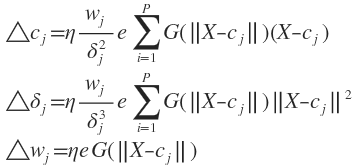
具体计算式为



上述目标函数是所有训练样本引起的误差总和，导出的参数修正公式是一种批处理式调整，即所有样本输入一轮后调整一次。目标函数也可以为瞬时值形式，即当前输入引起的误差

http://pic002.cnblogs.com/images/2012/103496/2012071423003161.png

此时参数的修正值为：



下面我们就分别用本文最后提到的聚类的方法和数据中心的监督学习方法做一道练习题。

考虑Hermit多项式的逼近问题

http://pic002.cnblogs.com/images/2012/103496/2012071423151741.png

训练样本这样产生：样本数P=100，xi且服从[-4,4]上的均匀分布，样本输出为F(xi)+ei，ei为添加的噪声，服从均值为0，标准差为0.1的正态分布。

1. 用聚类方法求数据中心和扩展常数，输出权值和阈值用伪逆法求解。隐藏节点数M=10，隐藏节点重叠系数λ=1，初始聚类中心取前10个训练样本。

#include<iostream>

#include<algorithm>

#include<limits>

#include<cassert>

#include<cmath>

#include<ctime>

#include<cstdlib>

#include<vector>

#include<iomanip>

#include"matrix.h"

**using** **namespace** std**;**

const int P**=**100**;** //输入样本的数量

vector**<**double**>** X**(**P**);** //输入样本

Matrix**<**double**>** Y**(**P**,**1**);** //输入样本对应的期望输出

const int M**=**10**;** //隐藏层节点数目

vector**<**double**>** center**(**M**);** //M个Green函数的数据中心

vector**<**double**>** delta**(**M**);** //M个Green函数的扩展常数

Matrix**<**double**>** Green**(**P**,**M**);** //Green矩阵

Matrix**<**double**>** Weight**(**M**,**1**);** //权值矩阵

/\*Hermit多项式函数\*/

inline double Hermit**(**double x**){**

**return** 1.1**\*(**1**-**x**+**2**\***x**\***x**)\***exp**(-**1**\***x**\***x**/**2**);**

**}**

/\*产生指定区间上均匀分布的随机数\*/

inline double uniform**(**double floor**,**double ceil**){**

**return** floor**+**1.0**\***rand**()/**RAND\_MAX**\*(**ceil**-**floor**);**

**}**

/\*产生区间[floor,ceil]上服从正态分布N[mu,sigma]的随机数\*/

inline double RandomNorm**(**double mu**,**double sigma**,**double floor**,**double ceil**){**

double x**,**prob**,**y**;**

**do{**

x**=**uniform**(**floor**,**ceil**);**

prob**=**1**/**sqrt**(**2**\***M\_PI**\***sigma**)\***exp**(-**1**\*(**x**-**mu**)\*(**x**-**mu**)/(**2**\***sigma**\***sigma**));**

y**=**1.0**\***rand**()/**RAND\_MAX**;**

**}while(**y**>**prob**);**

**return** x**;**

**}**

/\*产生输入样本\*/

void generateSample**(){**

**for(**int i**=**0**;**i**<**P**;++**i**){**

double in**=**uniform**(-**4**,**4**);**

X**[**i**]=**in**;**

Y**.**put**(**i**,**0**,**Hermit**(**in**)+**RandomNorm**(**0**,**0.1**,-**0.3**,**0.3**));**

**}**

**}**

/\*寻找样本离哪个中心最近\*/

int nearest**(**const vector**<**double**>&** center**,**double sample**){**

int rect**=-**1**;**

double dist**=**numeric\_limits**<**double**>::**max**();**

**for(**int i**=**0**;**i**<**center**.**size**();++**i**){**

**if(**fabs**(**sample**-**center**[**i**])<**dist**){**

dist**=**fabs**(**sample**-**center**[**i**]);**

rect**=**i**;**

**}**

**}**

**return** rect**;**

**}**

/\*计算簇的质心\*/

double calCenter**(**const vector**<**double**>** **&**g**){**

int len**=**g**.**size**();**

double sum**=**0.0**;**

**for(**int i**=**0**;**i**<**len**;++**i**)**

sum**+=**g**[**i**];**

**return** sum**/**len**;**

**}**

/\*KMeans聚类法产生数据中心\*/

void KMeans**(){**

assert**(**P**%**M**==**0**);**

vector**<**vector**<**double**>** **>** group**(**M**);** //记录各个聚类中包含哪些样本

double gap**=**0.001**;** //聚类中心的改变量小于为个值时，迭代终止

**for(**int i**=**0**;**i**<**M**;++**i**){** //从P个输入样本中随机选P个作为初始聚类中心

center**[**i**]=**X**[**10**\***i**+**3**];** //输入是均匀分布的，所以我们均匀地选取

**}**

**while(**1**){**

**for(**int i**=**0**;**i**<**M**;++**i**)**

group**[**i**].**clear**();** //先清空聚类信息

**for(**int i**=**0**;**i**<**P**;++**i**){** //把所有输入样本归到对应的簇

int c**=**nearest**(**center**,**X**[**i**]);**

group**[**c**].**push\_back**(**X**[**i**]);**

**}**

vector**<**double**>** new\_center**(**M**);** //存储新的簇心

**for(**int i**=**0**;**i**<**M**;++**i**){**

vector**<**double**>** g**=**group**[**i**];**

new\_center**[**i**]=**calCenter**(**g**);**

**}**

bool flag**=false;**

**for(**int i**=**0**;**i**<**M**;++**i**){** //检查前后两次质心的改变量是否都小于gap

**if(**fabs**(**new\_center**[**i**]-**center**[**i**])>**gap**){**

flag**=true;**

**break;**

**}**

**}**

center**=**new\_center**;**

**if(!**flag**)**

**break;**

**}**

**}**

/\*生成Green矩阵\*/

void calGreen**(){**

**for(**int i**=**0**;**i**<**P**;++**i**){**

**for(**int j**=**0**;**j**<**M**;++**j**){**

Green**.**put**(**i**,**j**,**exp**(-**1.0**\*(**X**[**i**]-**center**[**j**])\*(**X**[**i**]-**center**[**j**])/(**2**\***delta**[**j**]\***delta**[**j**])));**

**}**

**}**

**}**

/\*求一个矩阵的伪逆\*/

Matrix**<**double**>** getGereralizedInverse**(**const Matrix**<**double**>** **&**matrix**){**

**return** **(**matrix**.**getTranspose**()\***matrix**).**getInverse**()\*(**matrix**.**getTranspose**());**

**}**

/\*利用已训练好的神经网络，由输入得到输出\*/

double getOutput**(**double x**){**

double y**=**0.0**;**

**for(**int i**=**0**;**i**<**M**;++**i**)**

y**+=**Weight**.**get**(**i**,**0**)\***exp**(-**1.0**\*(**x**-**center**[**i**])\*(**x**-**center**[**i**])/(**2**\***delta**[**i**]\***delta**[**i**]));**

**return** y**;**

**}**

int main**(**int argc**,**char **\***argv**[]){<**br**>**　　 srand**(**time**(**0**));**

generateSample**();** //产生输入和对应的期望输出样本

KMeans**();** //对输入进行聚类，产生聚类中心

sort**(**center**.**begin**(),**center**.**end**());** //对聚类中心（一维数据）进行排序

//根据聚类中心间的距离，计算各扩展常数

delta**[**0**]=**center**[**1**]-**center**[**0**];**

delta**[**M**-**1**]=**center**[**M**-**1**]-**center**[**M**-**2**];**

**for(**int i**=**1**;**i**<**M**-**1**;++**i**){**

double d1**=**center**[**i**]-**center**[**i**-**1**];**

double d2**=**center**[**i**+**1**]-**center**[**i**];**

delta**[**i**]=**d1**<**d2**?**d1**:**d2**;**

**}**

calGreen**();** //计算Green矩阵

Weight**=**getGereralizedInverse**(**Green**)\***Y**;** //计算权值矩阵

//根据已训练好的神经网络作几组测试

**for(**int x**=-**4**;**x**<**5**;++**x**){**

cout**<<**x**<<**"\t"**;**

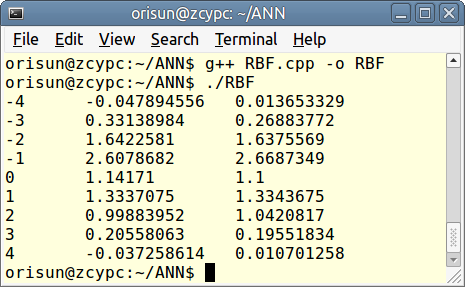
cout**<<**setprecision**(**8**)<<**setiosflags**(**ios**::**left**)<<**setw**(**15**);**

cout**<<**getOutput**(**x**)<<**Hermit**(**x**)<<**endl**;** //先输出我们预测的值，再输出真实值

**}**

**return** 0**;**

**}**



1. 用梯度下降法训练RBF网络，设η=0.001，M=10，初始权值为[-0.1,0.1]内的随机数，初始数据中心为[-4,4]内的随机数，初始扩展常数取[0.1,0.3]内的随机数，目标误差为0.9，最大训练次数为5000。

#include<iostream>

#include<cassert>

#include<cmath>

#include<ctime>

#include<cstdlib>

#include<vector>

#include<iomanip>

**using** **namespace** std**;**

const int P**=**100**;** //输入样本的数量

vector**<**double**>** X**(**P**);** //输入样本

vector**<**double**>** Y**(**P**);** //输入样本对应的期望输出

const int M**=**10**;** //隐藏层节点数目

vector**<**double**>** center**(**M**);** //M个Green函数的数据中心

vector**<**double**>** delta**(**M**);** //M个Green函数的扩展常数

double Green**[**P**][**M**];** //Green矩阵

vector**<**double**>** Weight**(**M**);** //权值矩阵

const double eta**=**0.001**;** //学习率

const double ERR**=**0.9**;** //目标误差

const int ITERATION\_CEIL**=**5000**;** //最大训练次数

vector**<**double**>** error**(**P**);** //单个样本引起的误差

/\*Hermit多项式函数\*/

inline double Hermit**(**double x**){**

**return** 1.1**\*(**1**-**x**+**2**\***x**\***x**)\***exp**(-**1**\***x**\***x**/**2**);**

**}**

/\*产生指定区间上均匀分布的随机数\*/

inline double uniform**(**double floor**,**double ceil**){**

**return** floor**+**1.0**\***rand**()/**RAND\_MAX**\*(**ceil**-**floor**);**

**}**

/\*产生区间[floor,ceil]上服从正态分布N[mu,sigma]的随机数\*/

inline double RandomNorm**(**double mu**,**double sigma**,**double floor**,**double ceil**){**

double x**,**prob**,**y**;**

**do{**

x**=**uniform**(**floor**,**ceil**);**

prob**=**1**/**sqrt**(**2**\***M\_PI**\***sigma**)\***exp**(-**1**\*(**x**-**mu**)\*(**x**-**mu**)/(**2**\***sigma**\***sigma**));**

y**=**1.0**\***rand**()/**RAND\_MAX**;**

**}while(**y**>**prob**);**

**return** x**;**

**}**

/\*产生输入样本\*/

void generateSample**(){**

**for(**int i**=**0**;**i**<**P**;++**i**){**

double in**=**uniform**(-**4**,**4**);**

X**[**i**]=**in**;**

Y**[**i**]=**Hermit**(**in**)+**RandomNorm**(**0**,**0.1**,-**0.3**,**0.3**);**

**}**

**}**

/\*给向量赋予[floor,ceil]上的随机值\*/

void initVector**(**vector**<**double**>** **&**vec**,**double floor**,**double ceil**){**

**for(**int i**=**0**;**i**<**vec**.**size**();++**i**)**

vec**[**i**]=**uniform**(**floor**,**ceil**);**

**}**

/\*根据网络，由输入得到输出\*/

double getOutput**(**double x**){**

double y**=**0.0**;**

**for(**int i**=**0**;**i**<**M**;++**i**)**

y**+=**Weight**[**i**]\***exp**(-**1.0**\*(**x**-**center**[**i**])\*(**x**-**center**[**i**])/(**2**\***delta**[**i**]\***delta**[**i**]));**

**return** y**;**

**}**

/\*计算单个样本引起的误差\*/

double calSingleError**(**int index**){**

double output**=**getOutput**(**X**[**index**]);**

**return** Y**[**index**]-**output**;**

**}**

/\*计算所有训练样本引起的总误差\*/

double calTotalError**(){**

double rect**=**0.0**;**

**for(**int i**=**0**;**i**<**P**;++**i**){**

error**[**i**]=**calSingleError**(**i**);**

rect**+=**error**[**i**]\***error**[**i**];**

**}**

**return** rect**/**2**;**

**}**

/\*更新网络参数\*/

void updateParam**(){**

**for(**int j**=**0**;**j**<**M**;++**j**){**

double delta\_center**=**0.0**,**delta\_delta**=**0.0**,**delta\_weight**=**0.0**;**

double sum1**=**0.0**,**sum2**=**0.0**,**sum3**=**0.0**;**

**for(**int i**=**0**;**i**<**P**;++**i**){**

sum1**+=**error**[**i**]\***exp**(-**1.0**\*(**X**[**i**]-**center**[**j**])\*(**X**[**i**]-**center**[**j**])/(**2**\***delta**[**j**]\***delta**[**j**]))\*(**X**[**i**]-**center**[**j**]);**

sum2**+=**error**[**i**]\***exp**(-**1.0**\*(**X**[**i**]-**center**[**j**])\*(**X**[**i**]-**center**[**j**])/(**2**\***delta**[**j**]\***delta**[**j**]))\*(**X**[**i**]-**center**[**j**])\*(**X**[**i**]-**center**[**j**]);**

sum3**+=**error**[**i**]\***exp**(-**1.0**\*(**X**[**i**]-**center**[**j**])\*(**X**[**i**]-**center**[**j**])/(**2**\***delta**[**j**]\***delta**[**j**]));**

**}**

delta\_center**=**eta**\***Weight**[**j**]/(**delta**[**j**]\***delta**[**j**])\***sum1**;**

delta\_delta**=**eta**\***Weight**[**j**]/**pow**(**delta**[**j**],**3**)\***sum2**;**

delta\_weight**=**eta**\***sum3**;**

center**[**j**]+=**delta\_center**;**

delta**[**j**]+=**delta\_delta**;**

Weight**[**j**]+=**delta\_weight**;**

**}**

**}**

int main**(**int argc**,**char **\***argv**[]){**

srand**(**time**(**0**));**

/\*初始化网络参数\*/

initVector**(**Weight**,-**0.1**,**0.1**);**

initVector**(**center**,-**4.0**,**4.0**);**

initVector**(**delta**,**0.1**,**0.3**);**

/\*产生输入样本\*/

generateSample**();**

/\*开始迭代\*/

int iteration**=**ITERATION\_CEIL**;**

**while(**iteration**-->**0**){**

**if(**calTotalError**()<**ERR**)** //误差已达到要求，可以退出迭代

**break;**

updateParam**();** //更新网络参数

**}**

cout**<<**"迭代次数:"**<<**ITERATION\_CEIL**-**iteration**-**1**<<**endl**;**

//根据已训练好的神经网络作几组测试

**for(**int x**=-**4**;**x**<**5**;++**x**){**

cout**<<**x**<<**"\t"**;**

cout**<<**setprecision**(**8**)<<**setiosflags**(**ios**::**left**)<<**setw**(**15**);**

cout**<<**getOutput**(**x**)<<**Hermit**(**x**)<<**endl**;** //先输出我们预测的值，再输出真实值

**}**

**return** 0**;**

**}**

