Analysis Of Self-Organizing Maps

Using A Java Based Implementation

Matthew A. Letter

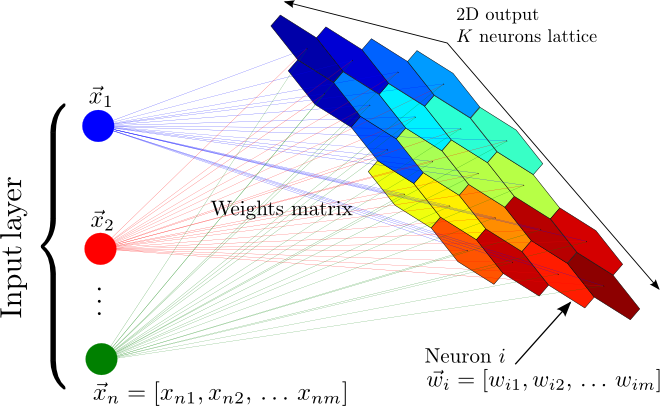
November 13th 2014

Neural Networks 547

**Self-Organizing Map**

Self Organizing Maps (SOM) are based on competitive learning; where output neurons of the network compete among themselves to be activated or fired.1 The result of which is that only one neuron per group is on at any one time.1 In this study, the Self-Organizing map consists of a 2 dimensional matrix with a varying number of neurons. The neurons become selectively tuned to various input patterns during the course of the competitive learning process.1

Java was used to code the SOM network. A Graphical user interface was used to create a real-time updating graph of the weight changes laid out on the inputs. This was done in order to see “weight grouping” and to gauge weather the SOM fit the data by the end of a run. There were 2 given inputs to the SOM with 1600 data points that were pulled at random. 1600 data point pulls was considered one iteration. Below is a pictorial schema for the self organizing map system.2



In the following tables convergence is defined by when the RMS magnitude of the weight change, after one iteration, is less than 0.00001. The following variables and values are set unless otherwise noted based on hayken.1 M=5,tao = 1000/log(sigma0), sigma0=M/2, eta=0.01, and all initial weight values are randomly chosen between -0.5 and 0.5.

Tables 1: Investigates different learning rates on SOM convergence. It appears that the lower the learning rate the faster the convergence.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number of SOM neurons** | **# Of iterations** | **Learning rate** | **Was Shuffled** | **Converged?** |
| 5X5 | 1642 | 0.2 | yes | yes |
| 5X5 | 1014 | 0.1 | yes | yes |
| 5X5 | 796 | 0.05 | yes | yes |
| 5X5 | 646 | 0.01 | yes | yes |
| 5X5 | 302 | 0.001 | yes | yes |

Table 2: Investigates varying the number of SOM neurons effect on convergence. It appears that the larger M is the longer it takes to converge. Notice the anomaly for 50X50.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number of SOM neurons** | **# Of iterations** | **Learning rate** | **Was Shuffled** | **Converged?** |
| 5X5 | 646 | 0.01 | yes | yes |
| 10X10 | 1686 | 0.01 | yes | yes |
| 20X20 | 2508 | 0.01 | yes | yes |
| 50X50 | 286 | 0.01 | yes | yes |
| 100X100 | 10000 | 0.01 | yes | no |

Table 3: Investigates varying tao and how it affects convergence. It appears that the smaller tao gets the faster the system converges (this doesn’t mean the system is a good fit for the data jus that it converge to our criterion).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Number of SOM neurons** | **# Of iterations** | **Learning rate** | **Was Shuffled** | **Tao** | **Converged?** |
| 5X5 | 128 | 0.01 | yes | 10000/log2.5 | yes |
| 5X5 | 560 | 0.01 | yes | 1000/log2.5 | yes |
| 5X5 | 96 | 0.01 | yes | 100/log2.5 | yes |
| 5X5 | 20 | 0.01 | yes | 10/log2.5 | yes |
| 5X5 | 4 | 0.01 | yes | 1/log2.5 | no |

Table 4: Investigates varying sigma0 and how it affects convergence. It appears that the smaller sigma0 gets the faster the system converges (this doesn’t mean the system is a good fit for the data jus that it converge to our criterion).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Number of SOM neurons** | **# Of iterations** | **Learning rate** | **Was Shuffled** | **Sigma0** | **Converged?** |
| 5X5 | 652 | 0.01 | yes | M | yes |
| 5X5 | 542 | 0.01 | yes | M/2 | yes |
| 5X5 | 92 | 0.01 | yes | M/5 | yes |
| 5X5 | 1 | 0.01 | yes | M/10 | yes |
| 5X5 | 1 | 0.01 | yes | M/20 | yes |

Based on the tables above a few interesting cases have been chosen for graphic displays and RMS magnitude change as well as an “ideal case”. First the ideal case will be presented when eta=0.01, tao = 1000/logsima0, sigma0 = M/2, and M = 30X30.

Figure 1: Iteration 0 graph of an “ideal” setup based off the previous tables. As you can see the weights are randomly spread over the initial random range of 0.5 to -0.5.

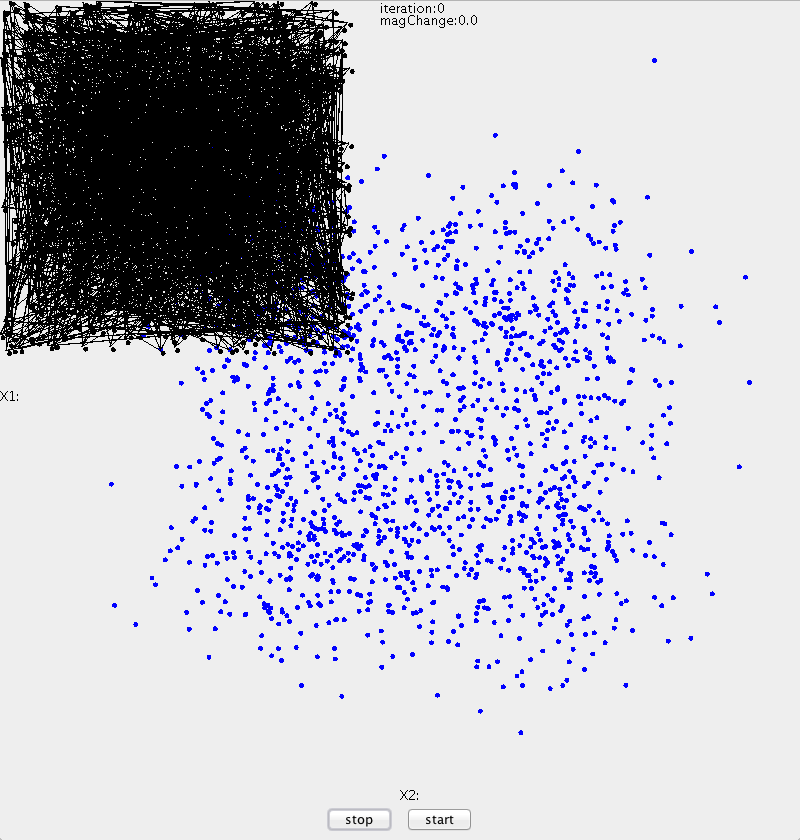


Figure 2: Iteration 1 graph of an “ideal” setup based off the previous tables. The weights immediately snap to the center of the data field.

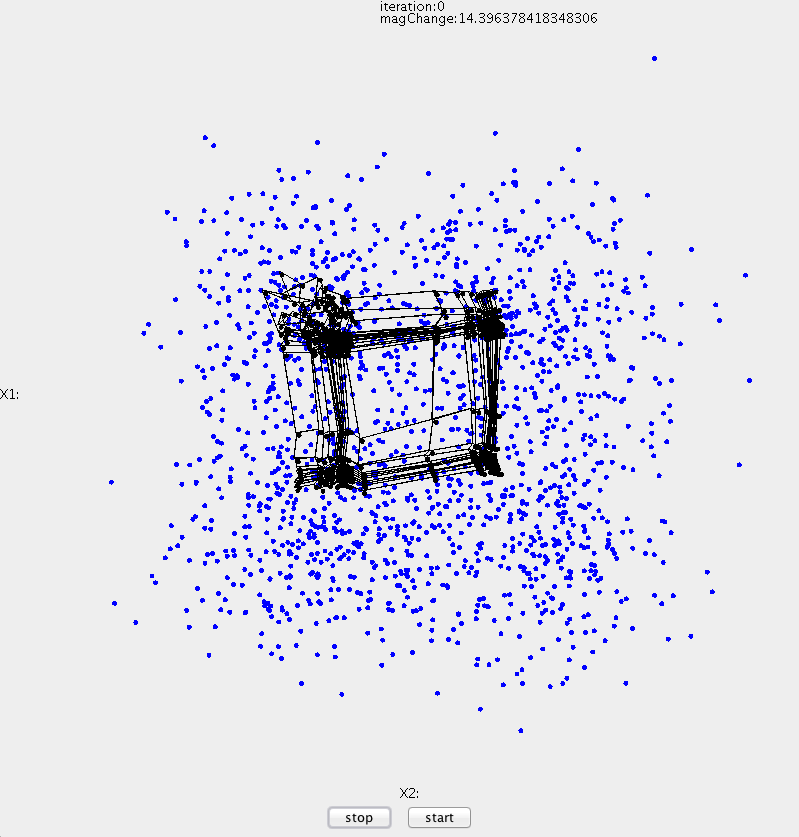


Figure 3: Iteration 216 graph of an “ideal” setup based off the previous tables. Based on the figure, it appears as if the weight map matches the data fairly well.

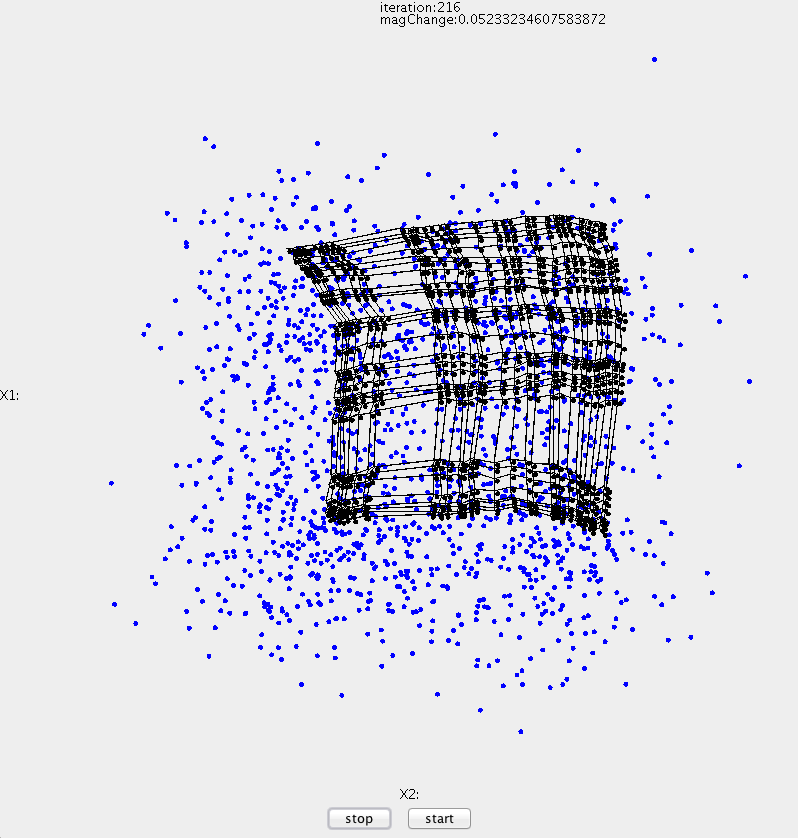


Figure 4: Iteration 972 graph of an “ideal” setup based off the previous tables. Letting the SOM lattice keep running past the point of apparent convergence yielded this interesting graph.

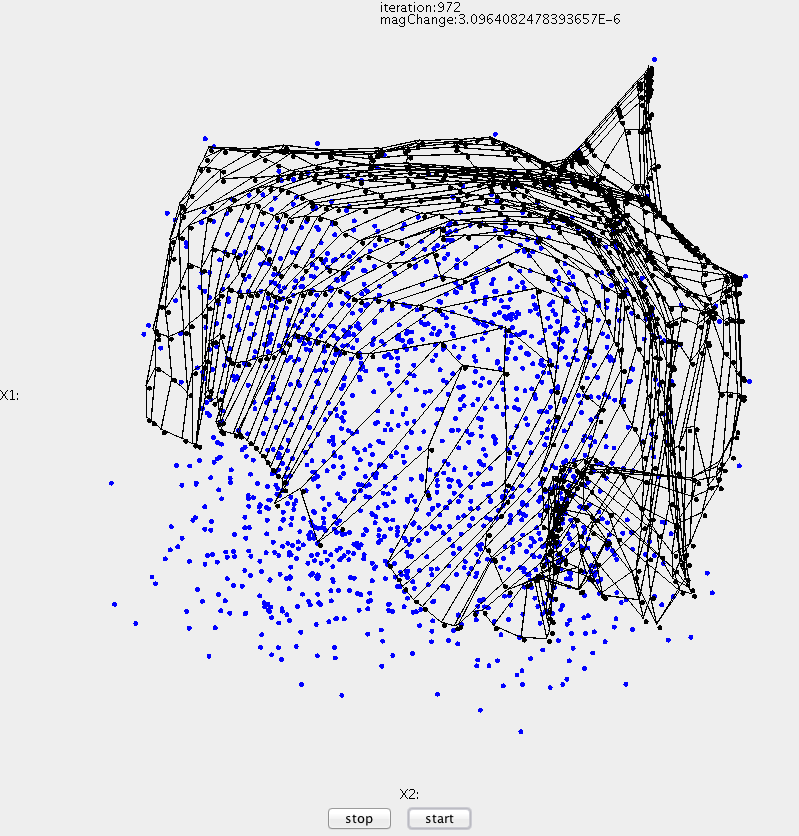


Figure 5: RMS magnitude change graph for figure 3 for the first 50 iterations. It appears as though there is a large initial jump over to the data and then the rest of the iterations are small jumps to better match the data

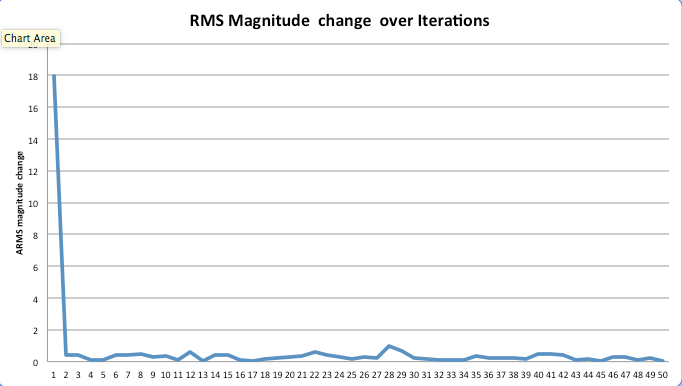


Figure 6: From table 1 a learning rate of 0.001 was chosen for visualization and an M of 10.

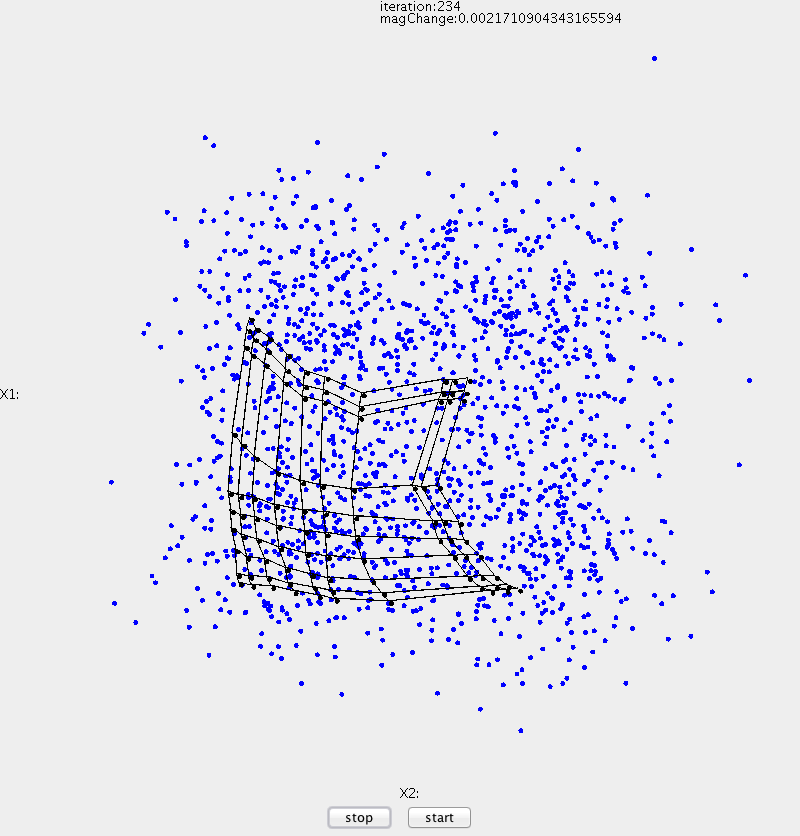


Figure 7: From table 2 an M of 50 was chosen. It seems the map fits the data fairly well.

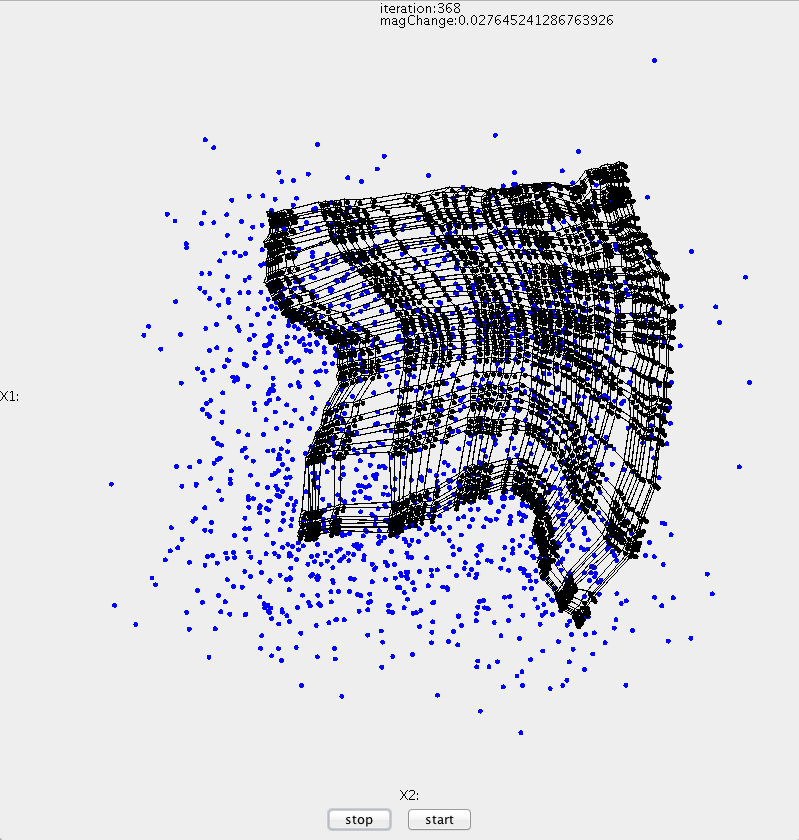


Figure 8: RMS magnitude weight change per iteration for figure 7. Notice again the large weight change during the first iteration.

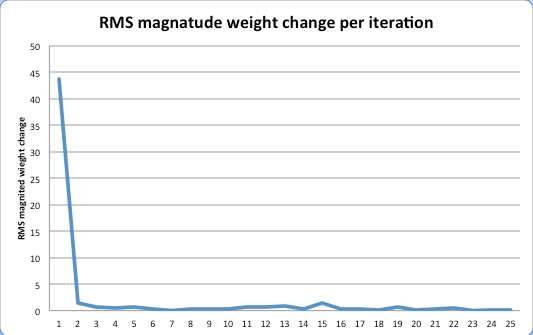


Figure 10: an un-shuffled presentation of the data. After 4 runs this graph was constantly produced, where the SOM would gravitate towards the initial data it was presented which was the bottom right quadrant.

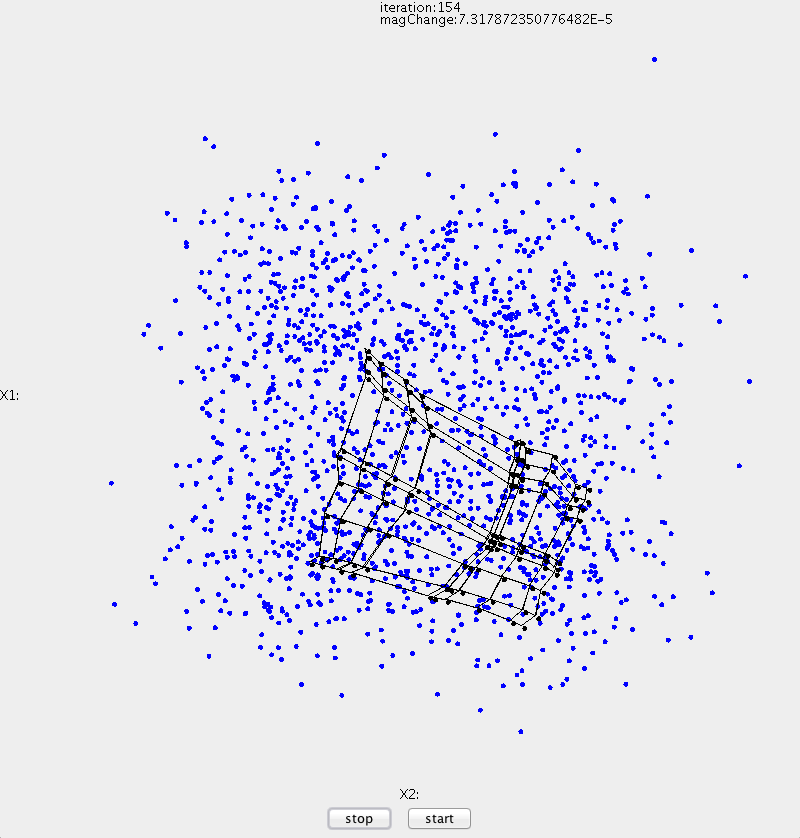


Figure 10: Tao is set to 1/log25 and M is set to 10. It does not appear that the SOM fits the data properly even though it met the convergence criterion.

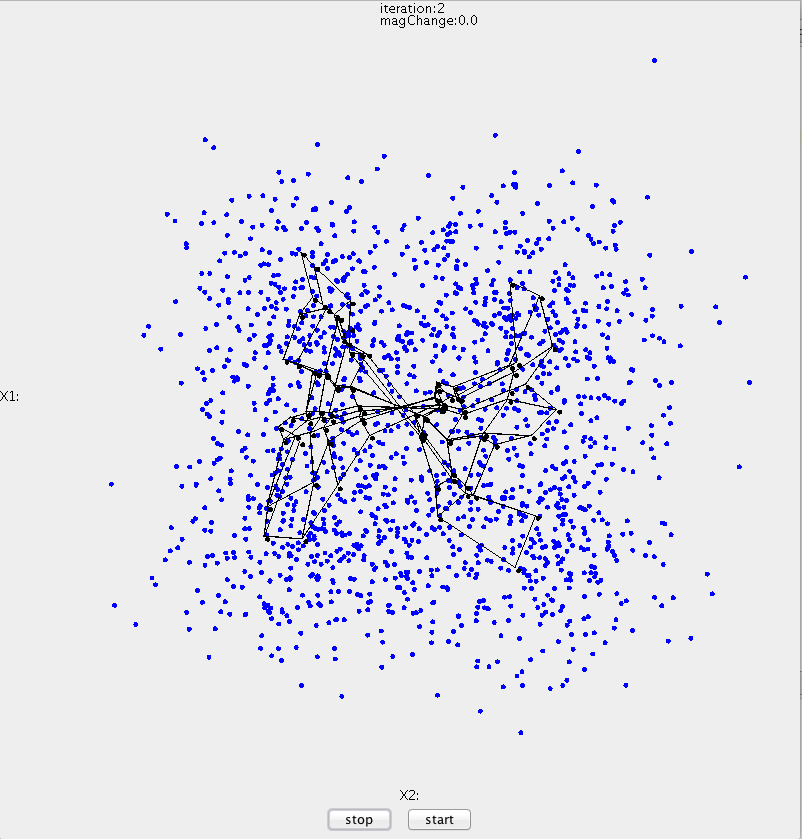


Figure 11: sigma0 is set to M/20 and M is set to 10. It does not appear that the SOM fits the data properly even though it met the convergence criterion.

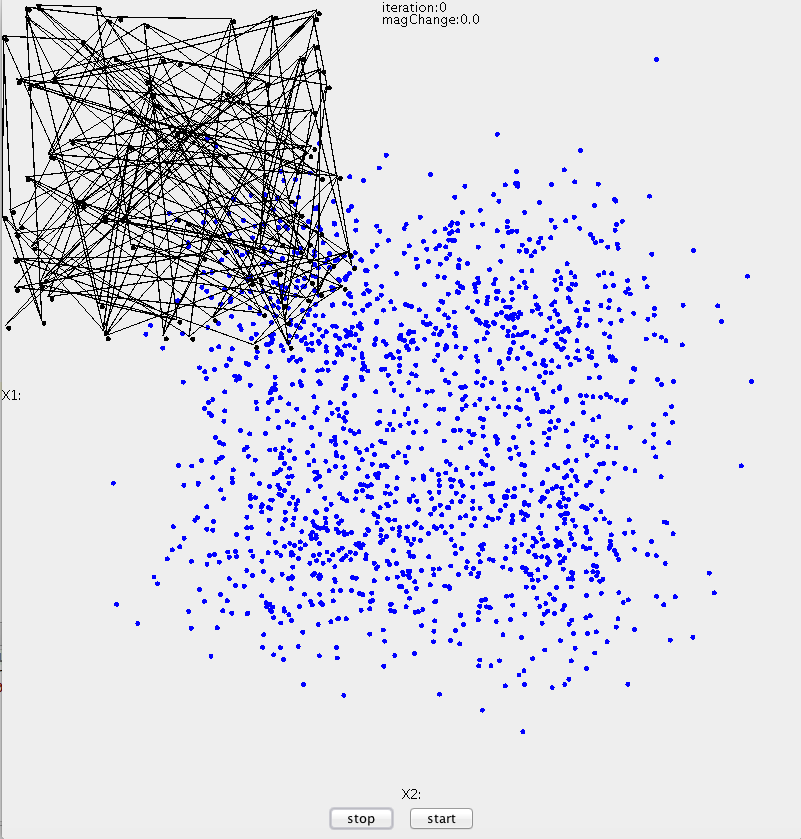
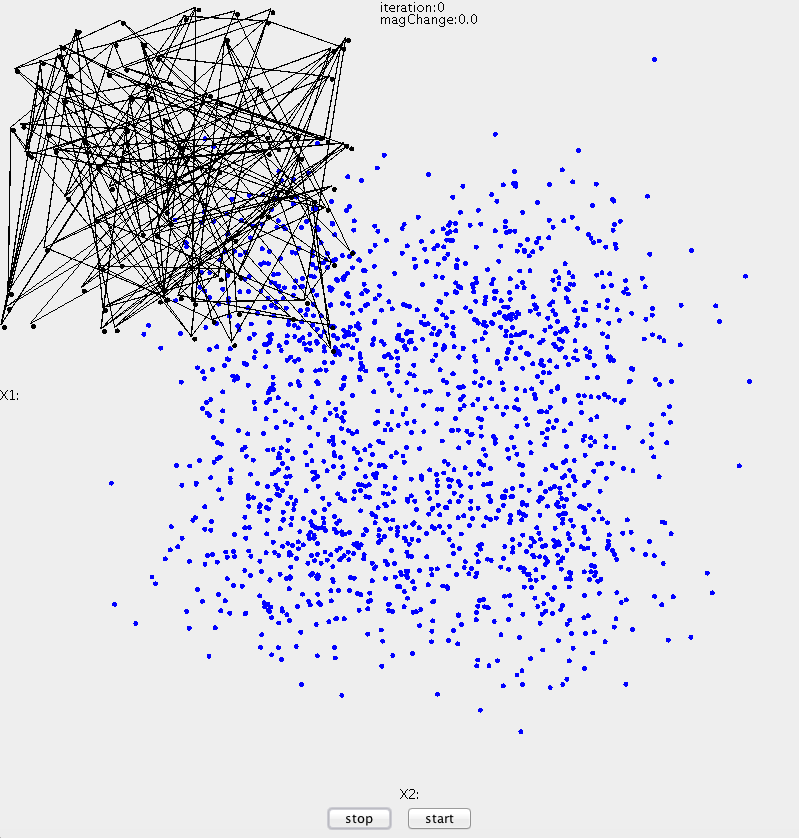


Figure 12: Setting all values to bad values, again the graph looks like figure 11 suggesting that sigma0 plays a large roll in the SOM algorithm



**Results and Conclusions**

In conclusion I would agree with Haykin that the learning rate should be set around 0.01 and tao and sigma0 should be set such that they initially encompasses the expected data inputs. The downside to this is that we need to know something about the data to begin with in order to set these values properly.

It would seem that having enough SOM neurons to cover the data is also important, where not having enough can lead to improper representations of the data by the SOM lattice. Based on the data, the fields of tao, M, sigma0, and eta all play a non-trivial roll in the SOM algorithm.

Citations

1. Haykin, Simon S. *Neural Networks: A Comprehensive Foundation*. Upper Saddle River, NJ: Prentice Hall, 1999. Print
2. "Self Organizing Maps." *Self Organizing Maps*. N.p., n.d. Web. <http%3A%2F%2Fwww.stuartreid.co.za%2Fartificial-intelligence-and-statistics-principal-component-analysis-and-self-organizing-maps%2F>.

Source Code

package app;

import Core.EVector;

import Core.LinesComponent;

import Core.NeuronMatrix;

import Core.TrainNetwork;

import util.ImportData;

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.util.Collections;

import java.util.Vector;

public class App{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* vary the next 4 variables for study

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private final int M = 10;

private final int NUM\_INPUT = 1600;

private static double ETA = 0.01;

private static int MAX\_ITERATIONS = 1000;

private TrainNetwork trainNetwork;

//gui

JFrame testFrame = new JFrame();

LinesComponent comp =null;

JPanel buttonsPanel = new JPanel();

JButton newLineButton = new JButton("stop");

JButton clearButton = new JButton("start");

//end gui

NeuronMatrix matrix = null;

Vector<EVector> dataVectors;

Vector<EVector> inputVectors = new Vector<EVector>();

/\*\*

\* everything we need to build and run an som network

\*/

public App() {

matrix = new NeuronMatrix(M);

trainNetwork = new TrainNetwork(ETA,MAX\_ITERATIONS);

//getData

ImportData data = new ImportData("/Users/matthewletter/Documents/Self\_Organizing\_Map/data/Data");

System.err.println(data.toString());

dataVectors = data.getData();

outPut(matrix);

Collections.shuffle(dataVectors);

//get some data

for (int i = 0; i < NUM\_INPUT; i++) {

inputVectors.add(dataVectors.get(i));

}

trainNetwork.setTraining(matrix, inputVectors);

trainNetwork.startTraining(this);

}

/\*\*

\* visualize the data

\* @param m

\*/

public void outPut(NeuronMatrix m){

comp = new LinesComponent(m,M,dataVectors);

testFrame.setDefaultCloseOperation(JFrame.DISPOSE\_ON\_CLOSE);

comp.setPreferredSize(new Dimension(800, 800));

testFrame.getContentPane().add(comp, BorderLayout.CENTER);

buttonsPanel.add(newLineButton);

buttonsPanel.add(clearButton);

testFrame.getContentPane().add(buttonsPanel, BorderLayout.SOUTH);

newLineButton.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

comp.stopStart(false);

}

});

clearButton.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

comp.stopStart(true);

}

});

testFrame.pack();

testFrame.setVisible(true);

}

/\*\*

\* update the next frame

\* @param m

\* @param iteration

\* @param magChange

\*/

public void updateView(NeuronMatrix m, int iteration, double magChange){

comp.updateMatrix(m,iteration,magChange);

}

/\*\*

\* builds the app for the som network

\* @param args

\*/

public static void main(String args[]) {

App app = new App();

}

}

package util;

/\*\*

\* Created by matthewletter on 11/8/14.

\*/

import Core.EVector;

import com.sun.tools.internal.ws.wsdl.document.Import;

import java.io.File;

import java.util.Scanner;

import java.util.Vector;

/\*\*

\* Created by matthewletter on 10/30/14.

\* used to import data from files

\*/

public class ImportData {

Vector<EVector> Data;

/\*\*

\* takes a file and parses it into a Vector of data classes

\* can get data back by calling getData() as a Vector

\* @param filename String name of file path to parse

\* @throws Exception

\*/

public ImportData(String filename){

this.Data = new Vector<EVector>();

System.out.println(filename);

String str;

String strLine[];

Scanner scanner;

try {

scanner = new Scanner(new File(filename));

int count = 0;

while (scanner.hasNext()) {

str = scanner.nextLine();

//clean data

str = str.replaceAll("\\s+", " ");

strLine = str.split(" ");

//build VectorDataFor1SetOfInputs into a vector

if (strLine.length == 4) {

double tempvals[] = new double[strLine.length];

tempvals[0] = Double.parseDouble(strLine[0]);

tempvals[1] = Double.parseDouble(strLine[1]);

tempvals[2] = Double.parseDouble(strLine[2]);

tempvals[3] = Double.parseDouble(strLine[3]);

EVector tempData = new EVector();

tempData.add(tempvals[2]);

tempData.add(tempvals[3]);

Data.add(tempData);

}

count++;

}

// System.out.println(count);

scanner.close();

}

catch(Exception e){

}

}

/\*\*

\* get the data vector

\* @return

\*/

public Vector<EVector> getData()

{

return this.Data;

}

@Override

public String toString() {

return "Data{" +

"Data=" + Data.get(0).get(0) + Data.get(0).get(1)+

'}'+"\\n Size="+ Data.size();

}

}

package Core;

import app.App;

import java.util.Random;

import java.util.Vector;

/\*\*

\* Created by matthewletter on 11/8/14.

\*/

public class TrainNetwork{

private double ETA = 0;

private int taoDependancyFactor = 0;

private double sigma0 = 0;

private double tao = 0;

private NeuronMatrix neuronMatrix = null;

private Vector<EVector> inputs = null;

/\*\*

\* provide the train function with some basic data

\* @param eta

\* @param t

\*/

public TrainNetwork(double eta, int t) {

this.ETA = eta;

this.taoDependancyFactor =t;

}

/\*\*

\* return the rad of the neighborhood

\* @param iteration

\* @return

\*/

private double getNeighborhoodRadius(double iteration) {

return sigma0 \* Math.exp(-iteration/ tao);

}

private double neighborhoodFunction(double distSq, double radius) {

double radiusSq = radius \* radius;

return Math.exp(-(distSq)/(2 \* radiusSq));

}

/\*\*

\* set some data points of the trainning

\* @param latToTrain

\* @param in

\*/

public void setTraining(NeuronMatrix latToTrain, Vector<EVector> in)

{

neuronMatrix = latToTrain;

inputs = in;

sigma0 = neuronMatrix.getM()/2;

tao = taoDependancyFactor / Math.log(sigma0);

}

/\*\*

\* start trainning the network

\* @param app

\*/

public void startTraining(App app) {

double dist;

double rad = 0;

double sumErr=0;

double magChange=0;

int iteration = 0;

double r=0;

Neuron winningNeuron = null;

Neuron currentNeuron = null;

int xi=0;

int yi=0;

int xf=0;

int yf=0;

EVector curInput = null;

double learningRate = ETA;

double time = System.currentTimeMillis();

Random myRandom = new Random(System.nanoTime());

boolean go = true;

while (iteration < 10000&&go) {

sumErr=0;

r = getNeighborhoodRadius(iteration);

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

//get an input data point

curInput = inputs.elementAt(i);

//startTraining winner take all on the input and get a neuron back

winningNeuron = neuronMatrix.getWinnerTakeAll(curInput);

/\*get our indexs of all the points within our radius\*/

xi = (int) (winningNeuron.getX1() - r - 1);

yi = (int)(winningNeuron.getX2() - r - 1);

//find the end of the radius

xf = (int)(xi + (r \* 2) + 1);

yf = (int)(yi + (r \* 2) + 1);

//make sure we are in bounds

if (xf > neuronMatrix.getM()){

xf = neuronMatrix.getM();

}

//make sure we didn't go negative

if (xi < 0) xi = 0;

//make sure we are in bounds

if (yf > neuronMatrix.getM()){

yf = neuronMatrix.getM();

}

//make sure we didn't go negative

if (yi < 0) yi = 0;

//loop through all the neurons within range

for (int x=xi; x<xf; x++) {

for (int y=yi; y<yf; y++) {

currentNeuron = neuronMatrix.getNeuron(x, y);

//get the dist between the neurons

dist = winningNeuron.distance(currentNeuron);

if (dist <= (r \* r)) {

rad = neighborhoodFunction(dist, r);

//adjust the weights and get a average mag change in return

sumErr += currentNeuron.adjustWeights(curInput, learningRate, rad);

}

}

}

}

magChange=Math.sqrt((sumErr\*sumErr)/inputs.size());

System.err.println(magChange);

if(magChange<0.000001){

go = false;

}

if(iteration%1==0){

//System.err.println(magChange);

app.updateView(neuronMatrix, iteration, magChange);

}

iteration++;

learningRate = ETA \* Math.exp(-(double)iteration/ taoDependancyFactor);

}

System.out.println("ran in :" + ((System.currentTimeMillis() - time) / 1000) + " seconds");

System.err.println("done"+iteration);

}

}

package Core;

/\*\*

\* Created by matthewletter on 11/8/14.

\*/

public class NeuronMatrix {

private int m;

private Neuron[][] matrix;

public NeuronMatrix(int m) {

this.m = m;

matrix = new Neuron[m][m];

for (int x=0; x<m; x++) {

for (int y=0; y<m; y++) {

matrix[x][y] = new Neuron(x,y,2);

}

}

}

/\*\*

\* return a 2d array of the lattice of SOM neurons

\* @param x

\* @param y

\* @return

\*/

public Neuron getNeuron(int x, int y) {

return matrix[x][y];

}

/\*\*

\* return M of the MxM lattice

\* @return

\*/

public int getM(){

return m;

}

/\*\*

\* find out who is the winner of the winner take all

\* @param in

\* @return

\*/

public Neuron getWinnerTakeAll(EVector in) {

Neuron winningNeuron = matrix[0][0];

double bestDist = in.eDistance(winningNeuron.getW());

double curDist=0;

for(Neuron[] nL: matrix){

for(Neuron n : nL){

curDist = in.eDistance(n.getW());

if (curDist < bestDist) {

winningNeuron = n;

bestDist = curDist;

}

}

}

return winningNeuron;

}

}

package Core;

/\*\*

\* Created by matthewletter on 11/11/14.

\*/

import javax.swing.\*;

import java.awt.\*;

import java.awt.geom.Ellipse2D;

import java.awt.geom.Line2D;

import java.util.Vector;

public class LinesComponent extends JComponent{

private NeuronMatrix matrix;

private int M;

private int iteration = 0;

private double magChange = 0;

private Vector<EVector> dataVectors;

private boolean stopFlag = true;

public LinesComponent(NeuronMatrix matrix, int M, Vector<EVector> dataVectors) {

this.matrix=matrix;

this.M = M;

this.dataVectors = dataVectors;

}

public void updateMatrix(NeuronMatrix matrix, int iteration, double magChange) {

if(stopFlag) {

this.matrix = matrix;

this.iteration = iteration;

this.magChange = magChange;

repaint();

}

}

/\*\*

\* stop/start the visual

\* @param s

\*/

public void stopStart(boolean s){

stopFlag = s;

}

@Override

protected void paintComponent(Graphics g) {

super.paintComponent(g);

double x;

double y;

double offset = 0.5;

double mul = 350;

double x1 = -1;

double y1 = -1;

double x2 = -1;

double y2 = -1;

double x3 = -1;

double y3 = -1;

double x4 = -1;

double y4 = -1;

g.drawString("iteration:" + Integer.toString(iteration), 380, 12);

g.drawString("magChange:" + Double.toString(magChange), 380, 24);

g.drawString("X1:", 0, 400);

g.drawString("X2:", 400, 799);

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

g.setColor(Color.BLUE);

((Graphics2D) g).fill(new Ellipse2D.Double(((Double) dataVectors.get(i).elementAt(0) + offset) \* mul, ((Double) dataVectors.get(i).elementAt(1) + offset) \* mul, 5, 5));

}

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

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

g.setColor(Color.BLACK);

x = (((Double) matrix.getNeuron(i, j).getW().elementAt(0)).doubleValue() + offset) \* mul;

y = (((Double) matrix.getNeuron(i, j).getW().elementAt(1)).doubleValue() + offset) \* mul;

if (i > 0 && i < M - 1 && j > 0 && j < M - 1) {

x1 = (((Double) matrix.getNeuron(i - 1, j).getW().elementAt(0)).doubleValue() + offset) \* mul;

y1 = (((Double) matrix.getNeuron(i - 1, j).getW().elementAt(1)).doubleValue() + offset) \* mul;

x2 = (((Double) matrix.getNeuron(i + 1, j).getW().elementAt(0)).doubleValue() + offset) \* mul;

y2 = (((Double) matrix.getNeuron(i + 1, j).getW().elementAt(1)).doubleValue() + offset) \* mul;

x3 = (((Double) matrix.getNeuron(i, j - 1).getW().elementAt(0)).doubleValue() + offset) \* mul;

y3 = (((Double) matrix.getNeuron(i, j - 1).getW().elementAt(1)).doubleValue() + offset) \* mul;

x4 = (((Double) matrix.getNeuron(i, j + 1).getW().elementAt(0)).doubleValue() + offset) \* mul;

y4 = (((Double) matrix.getNeuron(i, j + 1).getW().elementAt(1)).doubleValue() + offset) \* mul;

((Graphics2D) g).draw(new Line2D.Double(x, y, x1, y1));

((Graphics2D) g).draw(new Line2D.Double(x, y, x2, y2));

((Graphics2D) g).draw(new Line2D.Double(x, y, x3, y3));

((Graphics2D) g).draw(new Line2D.Double(x, y, x4, y4));

} else if (i == 0 && j > 0 && j < M - 1) {

x2 = (((Double) matrix.getNeuron(i + 1, j).getW().elementAt(0)).doubleValue() + offset) \* mul;

y2 = (((Double) matrix.getNeuron(i + 1, j).getW().elementAt(1)).doubleValue() + offset) \* mul;

x3 = (((Double) matrix.getNeuron(i, j - 1).getW().elementAt(0)).doubleValue() + offset) \* mul;

y3 = (((Double) matrix.getNeuron(i, j - 1).getW().elementAt(1)).doubleValue() + offset) \* mul;

x4 = (((Double) matrix.getNeuron(i, j + 1).getW().elementAt(0)).doubleValue() + offset) \* mul;

y4 = (((Double) matrix.getNeuron(i, j + 1).getW().elementAt(1)).doubleValue() + offset) \* mul;

((Graphics2D) g).draw(new Line2D.Double(x, y, x2, y2));

((Graphics2D) g).draw(new Line2D.Double(x, y, x3, y3));

((Graphics2D) g).draw(new Line2D.Double(x, y, x4, y4));

} else if (i == M - 1 && j > 0 && j < M - 1) {

x1 = (((Double) matrix.getNeuron(i - 1, j).getW().elementAt(0)).doubleValue() + offset) \* mul;

y1 = (((Double) matrix.getNeuron(i - 1, j).getW().elementAt(1)).doubleValue() + offset) \* mul;

x3 = (((Double) matrix.getNeuron(i, j - 1).getW().elementAt(0)).doubleValue() + offset) \* mul;

y3 = (((Double) matrix.getNeuron(i, j - 1).getW().elementAt(1)).doubleValue() + offset) \* mul;

x4 = (((Double) matrix.getNeuron(i, j + 1).getW().elementAt(0)).doubleValue() + offset) \* mul;

y4 = (((Double) matrix.getNeuron(i, j + 1).getW().elementAt(1)).doubleValue() + offset) \* mul;

((Graphics2D) g).draw(new Line2D.Double(x, y, x3, y3));

((Graphics2D) g).draw(new Line2D.Double(x, y, x4, y4));

((Graphics2D) g).draw(new Line2D.Double(x, y, x1, y1));

} else if (j == 0 && i > 0 && i < M - 1) {

x1 = (((Double) matrix.getNeuron(i - 1, j).getW().elementAt(0)).doubleValue() + offset) \* mul;

y1 = (((Double) matrix.getNeuron(i - 1, j).getW().elementAt(1)).doubleValue() + offset) \* mul;

x2 = (((Double) matrix.getNeuron(i + 1, j).getW().elementAt(0)).doubleValue() + offset) \* mul;

y2 = (((Double) matrix.getNeuron(i + 1, j).getW().elementAt(1)).doubleValue() + offset) \* mul;

x4 = (((Double) matrix.getNeuron(i, j + 1).getW().elementAt(0)).doubleValue() + offset) \* mul;

y4 = (((Double) matrix.getNeuron(i, j + 1).getW().elementAt(1)).doubleValue() + offset) \* mul;

((Graphics2D) g).draw(new Line2D.Double(x, y, x1, y1));

((Graphics2D) g).draw(new Line2D.Double(x, y, x2, y2));

((Graphics2D) g).draw(new Line2D.Double(x, y, x4, y4));

} else if (j == M - 1 && i > 0 && i < M - 1) {

x1 = (((Double) matrix.getNeuron(i - 1, j).getW().elementAt(0)).doubleValue() + offset) \* mul;

y1 = (((Double) matrix.getNeuron(i - 1, j).getW().elementAt(1)).doubleValue() + offset) \* mul;

x2 = (((Double) matrix.getNeuron(i + 1, j).getW().elementAt(0)).doubleValue() + offset) \* mul;

y2 = (((Double) matrix.getNeuron(i + 1, j).getW().elementAt(1)).doubleValue() + offset) \* mul;

x3 = (((Double) matrix.getNeuron(i, j - 1).getW().elementAt(0)).doubleValue() + offset) \* mul;

y3 = (((Double) matrix.getNeuron(i, j - 1).getW().elementAt(1)).doubleValue() + offset) \* mul;

((Graphics2D) g).draw(new Line2D.Double(x, y, x1, y1));

((Graphics2D) g).draw(new Line2D.Double(x, y, x2, y2));

((Graphics2D) g).draw(new Line2D.Double(x, y, x3, y3));

}

g.setColor(Color.BLACK);

((Graphics2D) g).fill(new Ellipse2D.Double(x, y, 5, 5));

}

}

}

}

package Core;

import java.util.Vector;

/\*\*

\* Created by matthewletter on 11/8/14.

\* modified vector to simplify input output

\*/

public class EVector extends Vector {

public double eDistance(EVector v2){

double totalDis = 0;

double dis;

for (int x=0; x<size(); x++) {

dis = (Double) elementAt(x) - (Double) v2.elementAt(x);

dis = dis\*dis;

totalDis += dis;

}

return totalDis;

}

}

package Core;

/\*\*

\* Created by matthewletter on 11/8/14.

\*/

public class Neuron {

private EVector weights;

private int x1;

private int x2;

/\*\*

\* everything we need to make a neuron

\* @param x1

\* @param x2

\* @param numWeights

\*/

public Neuron(int x1, int x2, int numWeights) {

this.x1 =x1;

this.x2 =x2;

weights = new EVector();

for (int i=0; i<numWeights; i++) weights.addElement(Math.random()-0.5);//random weight between -.5 and .5

}

/\*\*

\* get X1

\* @return

\*/

public int getX1() {

return x1;

}

/\*\*

\* get X2

\* @return

\*/

public int getX2() {

return x2;

}

/\*\*

\* get weight vector

\* @return

\*/

public EVector getW() {

return weights;

}

/\*\*

\* get the distance between two neurons

\* @param n2

\* @return

\*/

public double distance(Neuron n2) {

int x1D;

int x2D;

x1D = (getX1() - n2.getX1())\*(getX1() - n2.getX1());

x2D = (getX2() - n2.getX2())\*(getX2() - n2.getX2());

return Math.sqrt(x1D + x2D);

}

/\*\*

\* adjust neuron weight based on a given input

\* @param input

\* @param learningRate

\* @param rad

\* @return

\*/

public double adjustWeights(EVector input, double learningRate,

double rad){

double t = 0;

double v = 0;

double change = 0;

for (int w=0; w<weights.size(); w++) {

t = (Double)weights.elementAt(w);

v = (Double)input.elementAt(w);

change += rad \* learningRate \* (v - t);

t += rad \* learningRate \* (v - t);

weights.setElementAt(t, w);

}

return change/weights.size();

}

}