Analysis Of The Backpropagation Algorithm

Using A Java Based Implementation

Matthew A. Letter

October 23th 2014

Neural Networks 547

**Backpropagation Algorithm**

Backpropagation, an abbreviation for "backward propagation of errors", is a common method for training artificial neural networks used in conjunction with an optimization method such as gradient descent. The method calculates the gradient of a loss function with respects to all the weights in the network. The gradient is fed to the optimization method, which in turn uses it to update the weights, in an attempt to minimize the loss function.

Backpropagation requires a known, desired output for each input value in order to calculate the loss function gradient. It is therefore usually considered to be a supervised learning method. It is a generalization of the [delta rule](http://en.wikipedia.org/wiki/Delta_rule) to multi-layered feedforward networks, made possible by using the chain rule to iteratively compute gradients for each layer. Backpropagation requires that the activation function used by the artificial neurons (or "nodes") be differentiable.1

Java was used to code the Backpropagation implementation (network) and the jmathplot java API was used to plot the data into a presentable format automatically at the end of each test. The network was given two input nodes (plus bias node), four outputs, and has the ability to handle multiple hidden layer network topologies, including appropriate bias nodes addition with all. The network was given a set of learning data from four classes non-linearly. The testing data was made up of four non-linearly separable classes. The stop criterion for learning was when, over the last 20 runs, the RMS error for the last 10 runs minus the rms error for the last 20 runs were less than .0001. Furthermore runs were limited to 10000 as an upper bound. In general, for the test classes, the Backpropagation network could correctly classify 86% of the given test points (found using the average of 100 runs with a learning rate of 0.01 and random start weights with 1 hidden layer of 10 neurons). Below are four plots: the plotted training data (Figure 1), the plotted testing data (Figure 2), before learning class boundaries (Figure 3), after learning class boundaries with one hidden layer (Figure 4), and after learning class boundaries with two hidden layer (Figure 5).

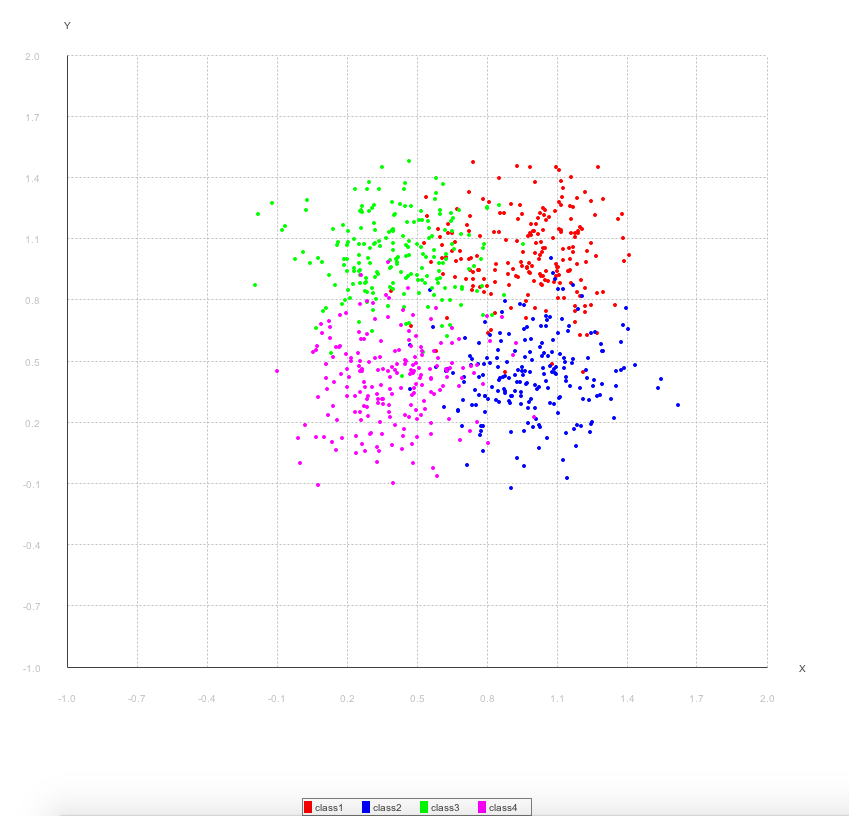


Figure 1: Training data colored by class(notice the data could potentially be separated by an cross).

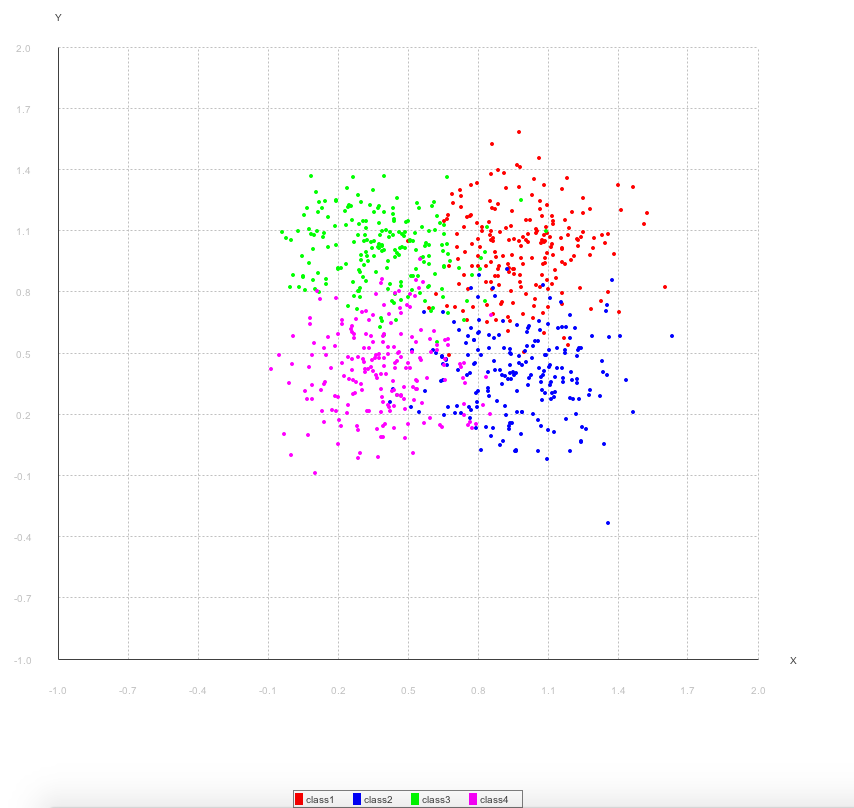


Figure 2: Testing Data colored by class.

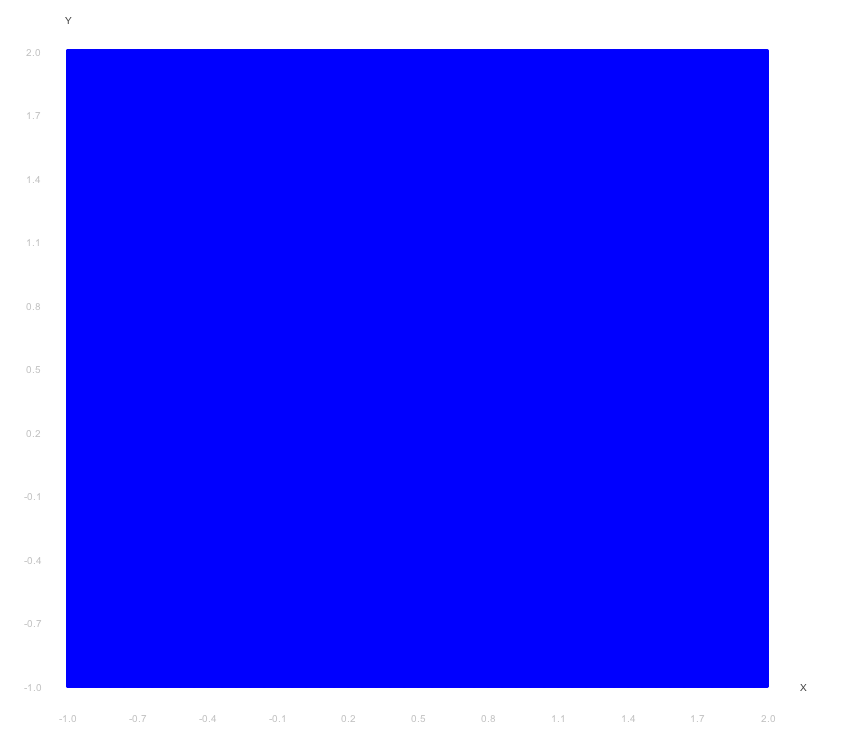


Figure 3: decision boundaries before training.

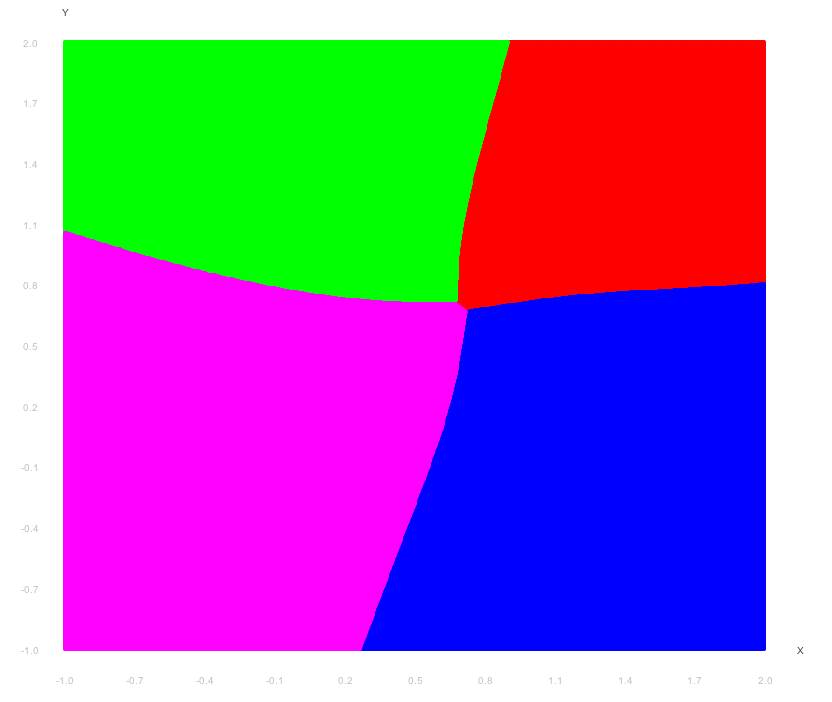


Figure 4: decision boundaries after training with one hidden layer with 10 neurons making correct decisions %86.02 of the time.

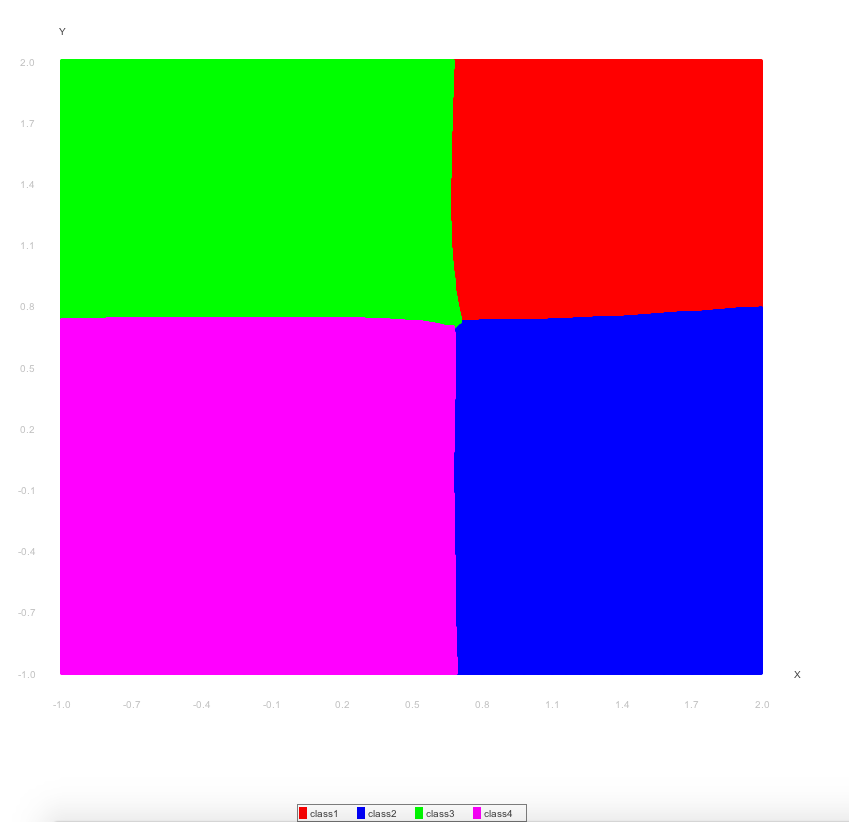


Figure 5: decision boundaries after training with two hidden layers with 10 neurons each, making correct decisions %85.68 of the time.

It can be gleaned from figures 4 and 5 that our convergence criterion can be attained for both a single layer hidden layer and 2 hidden layers. Although the two hidden layer run took nearly twice as long to reach the convergence criterion; this will be looked at with comparison tables. The below tables are using 1 hidden layer with 10 neurons unless specified.

Tables 6-10 have varying stating weights (Table 4), learning rate (Table 5), and shuffling of the data (Table 6).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **All**  **Weights** | **# Of Epochs** | **Learning rate** | **Was Shuffled** | **Converged?** | **Correct Decision rate** |
| -1 | 1000 | 0.01 | yes | no | %25 |
| -.5 | 3880 | 0.01 | yes | yes | %57.75 |
| 0 | 3783 | 0.01 | yes | yes | %60.37 |
| .5 | 3612 | 0.01 | yes | yes | %61.12 |
| 1 | 1000 | 0.01 | yes | no | %25 |
| Random +-0.5 | 128 | 0.01 | yes | yes | %85.75 |

Table 6: The above shows that setting all weights to the same value does not yield good results. A random weighted set is added to emphasize this point.

|  |  |  |
| --- | --- | --- |
| **Weights** | **# Of Epochs** | **Learning rate** |
| Random +-0.5 | No convergence | 1.0 |
| Random +-0.5 | 278 | 0.1 |
| Random +-0.5 | 128 | 0.01 |

Table 7: The weights are set to Random +-0.5 (based on Table 6). As the learning rate gets large the weights no longer converge by the convergence criterion.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Weights** | **# Of Epochs** | **Learning rate** | **Correct Decision rate** | **Was Shuffled** |
| Random +-0.5 | 248 | 0.01 | %81.62 | False |
| Random +-0.5 | 128 | 0.01 | %85.75 | True |

Table 8: The weights are set to Random +-0.5 (based on Table 6). Shuffling appears to have an effect on convergence rate and on correct decision rate after learning.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Weights** | **# Of Epochs** | **Learning rate** | **Correct Decision rate** | **# Of Neurons in the Hidden Layer** | **Was Shuffled** |
| Random +-0.5 | 10 | 0.01 | %86.12 | 50 | False |
| Random +-0.5 | 76 | 0.01 | %81.62 | 10 | False |
| Random +-0.5 | 128 | 0.01 | %85.75 | 5 | True |

Table 9: The more neurons in the hidden layer the faster the convergence. Let it be noted that although convergence was done in a smaller number of epochs the amount of actual time to reach it was much larger than that of the smaller set of neurons.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Weights** | **# Of Epochs** | **Learning rate** | **Correct Decision rate** | **# Of hidden layers** |
| Random +-0.5 | 128 | 0.01 | %85.75 | 1 |
| Random +-0.5 | 188 | 0.01 | %85.15 | 2 |

Table 10: Increasing the amount of hidden layers, in this case, appears to increase the amount of time to convergence and yields roughly the same Correct Decision rate.

In Figures 7 and 8, a generalization graph was produced for 100 neurons.

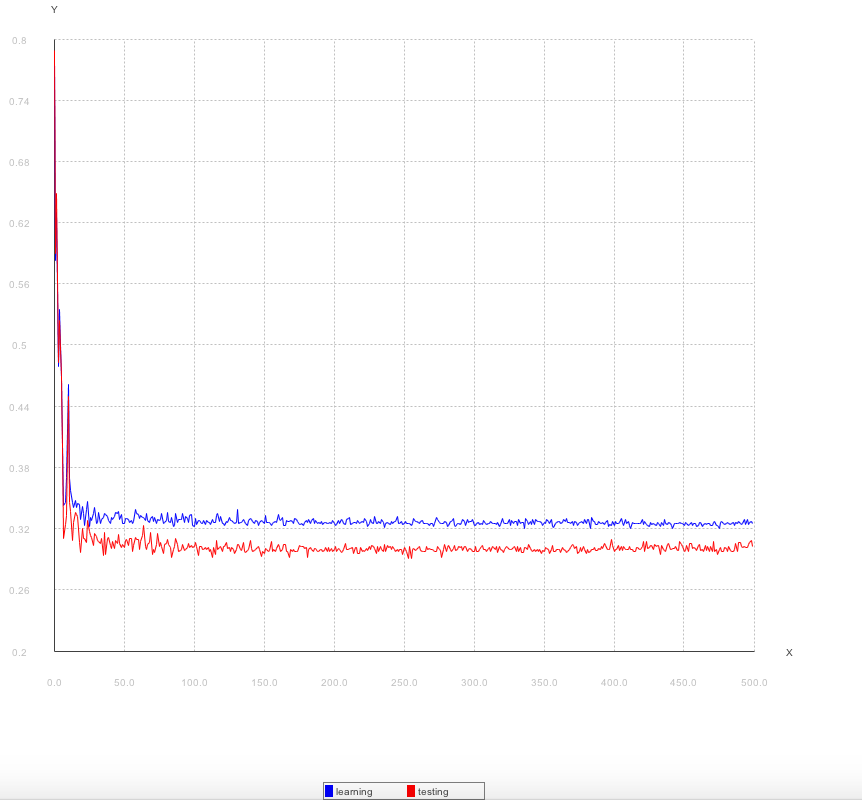


Figure 7: based on the previous results, an ideal situation of 50 neurons with 1 hidden layer, a learning rate of 0.01, shuffling, and random rates was used to build a graph of the RMS error rate change.

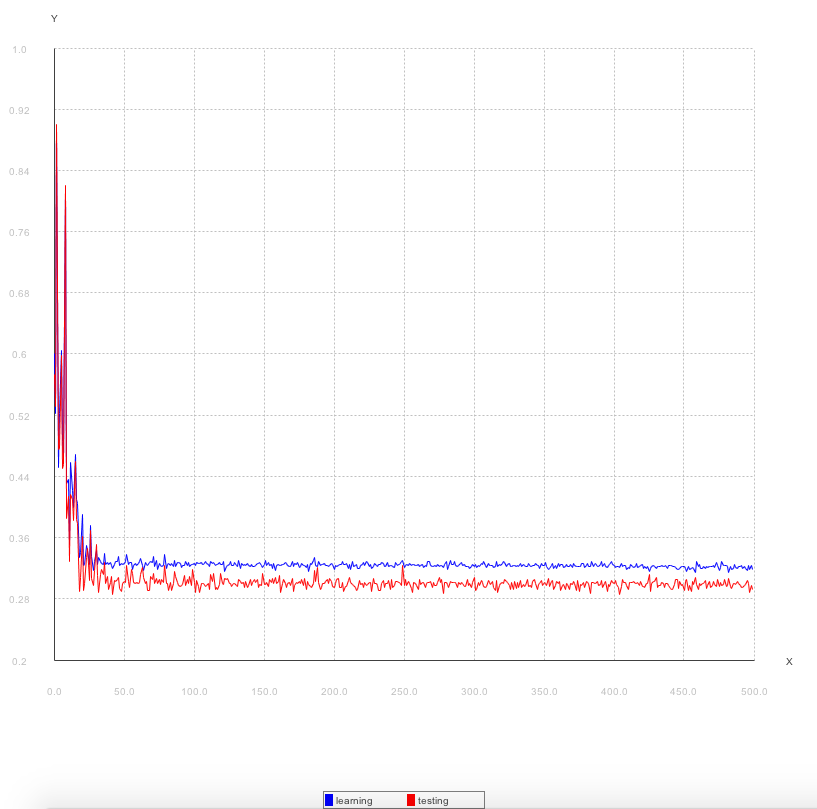


Figure 8: Same parameters as figure 7 but with two hidden layers instead of one.

A few of the more “interesting” parameter variations were chosen for graphing in the next section.

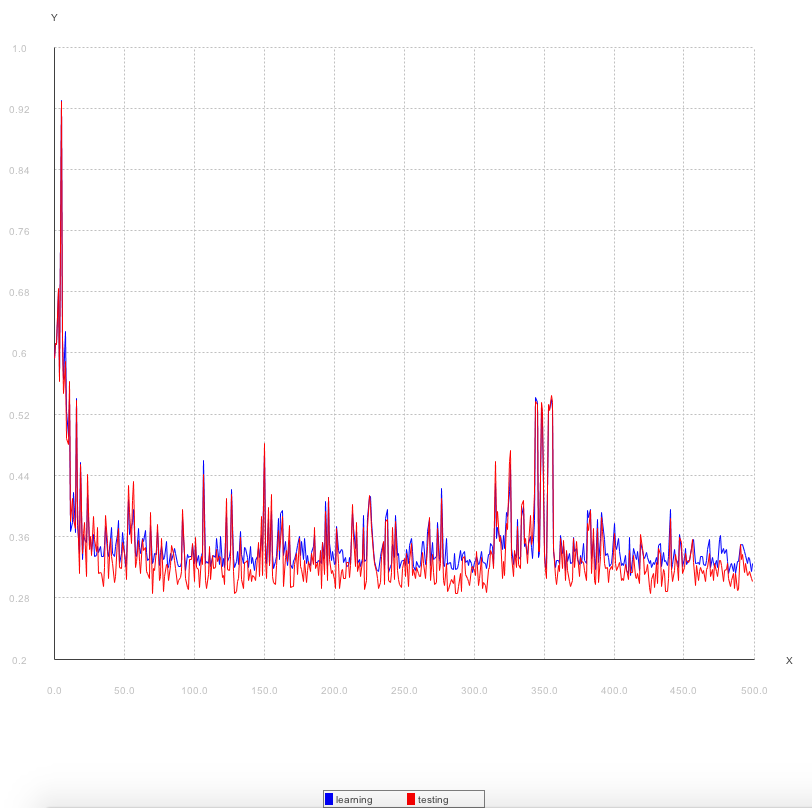


Figure 9: A high learning rate (based off table 7) was chosen to show how a high learning rate creates a vary “noisy” asymptote. Making the convergence criterion impossible to reach.

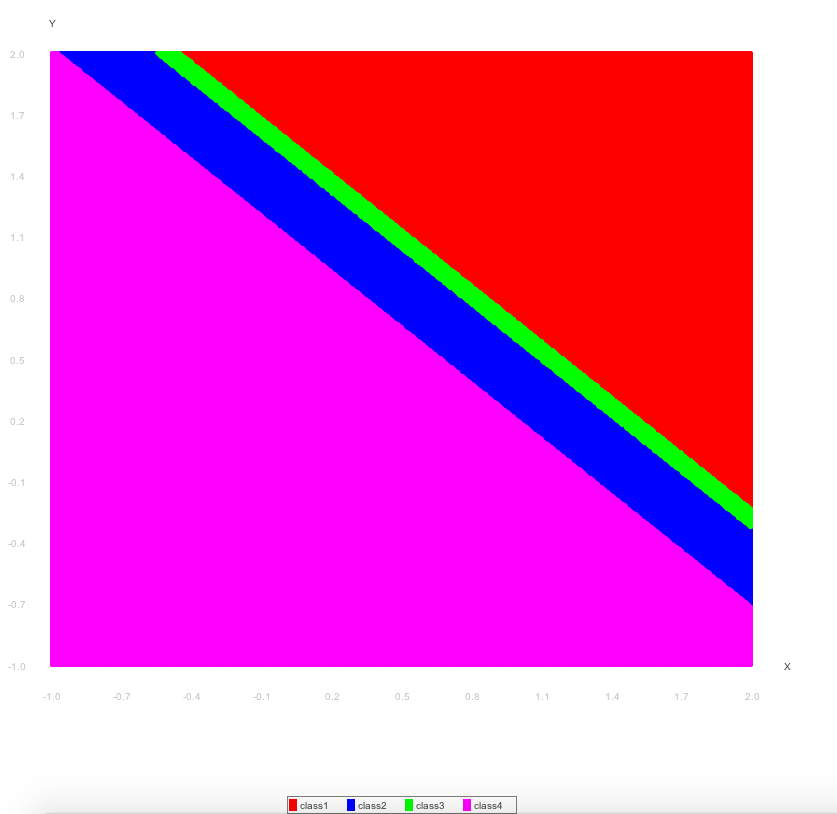


Figure 10: all weights are set to 1 (based off table 6). This was chosen to show how a setting all the weights to the same number makes for rather interesting (though wrong) decision boundaries.

**Results and Conclusions**

There are many things to be gleaned from the above data. From the Backpropagation results presented above it is observed that learning rate decrease positively correlates with the convergence rate and increases the likeliness that it will converge. Starting the weights randomly between +-0.5 makes the error rate converge and randomly shuffling the input learning data appears to have a large positive effects on the convergence rate.

Based on the results, the correct way to make the Backpropagation network rate reach converge the fastest is to choose random weight starting parameters, a learning rate less than or equal to 0.01, and two hidden layers with greater than or equal to 50 neurons each.

Citations

1. "Backpropagation." *Wikipedia*. Wikimedia Foundation, 17 Oct. 2014. Web. 21 Oct. 2014. <http://en.wikipedia.org/wiki/Backpropagation>.
2. "Delta Rule." *Wikipedia*. Wikimedia Foundation, 24 Aug. 2014. Web. 05 Oct. 2014. <http://en.wikipedia.org/wiki/Delta\_rule>.
3. "Backpropagation." *Backpropagation*. N.p., n.d. Web. 23 Oct. 2014. <http://galaxy.agh.edu.pl/~vlsi/AI/backp\_t\_en/backprop.html>.

Source code Perceptron

/\*\*

\* Created by matthewletter on 10/7/14.

\*/

import java.io.File;

import java.io.FileNotFoundException;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Scanner;

public class NeuralNet

{

private final int INPUT\_NEURONS = 2;

private final int NUMBER\_OF\_HIDDEN\_LAYERS = 1;

private final int HIDDEN\_NEURONS = 10;

private final int OUTPUT\_NEURONS = 4;

private final double LEARNING\_RATE = 0.01; // Rho.

private final int EPOCHES = 2000;

private final int SAMPLES = 800;

public ArrayList<Sample> matrix;

public ArrayList<Layer> Layers = new ArrayList<Layer>();

private double inputSampleValues[] = new double[INPUT\_NEURONS];

private double expected[] = new double[OUTPUT\_NEURONS];

private int classOutput[][] = new int[][]

{{1, 0, 0, 0},

{0, 1, 0, 0},

{0, 0, 1, 0},

{0, 0, 0, 1}};

/\*\*

\* entrypoint for network tests, sets up the network and starts running EPOCHES;

\*/

private void nNet()

{

buildMatrix();

int sample = 0;

//class 1

File f1 = new File("/Users/matthewletter/Documents/BackPropagation/data/TrainingData.txt");

ArrayList<Sample> samples = parseFile(f1);

printMinMaxOfData(samples);

//plotFile(f1);

File f2 = new File("/Users/matthewletter/Documents/BackPropagation/data/TestingData.txt");

ArrayList<Sample> test = parseFile(f2);

printMinMaxOfData(test);

buildLayers();

//plotFile(f2);

//buildColors(matrix);

System.out.println("\nbefore training");

System.out.println("Network test is " + getRunStats(test) + "% correct.");

double[] sumold = new double[10];

double[] sumnew = new double[10];

double time = System.currentTimeMillis();

int count =0;

double oldError = 10.0;

double olderError = 5.0;

double change = 0.0;

int num = 1;

// Train the network.

double[] x1 = new double[EPOCHES];

double[] x2 = new double[EPOCHES];

double[] y = new double[EPOCHES];

int index = 0;

for(int epoch = 0; epoch < EPOCHES; epoch++) {

Collections.shuffle(samples);

for (Sample s : samples) {

inputSampleValues[0] = s.X1;

inputSampleValues[1] = s.X2;

sample = s.expectedClass;

for (int i = 0; i < OUTPUT\_NEURONS; i++) {

//System.out.println(sample + " : " + i);

expected[i] = classOutput[sample][i];

}

feedForward();

backPropagation();

}

//System.out.println("error:" + getErrorStats(test) + "");

x1[count] = getErrorStats(samples);

x2[count] = getErrorStats(test);

sumnew[index]= x2[count];

if(num%10==9){

index = 0;

}

if(num%10==9 && count > 60){

olderError = 0;

oldError = 0;

for (int i = 0; i < sumnew.length; i++) {

oldError += sumnew[i];

olderError += sumold[i];

}

change = (oldError-olderError)\*(oldError-olderError);

System.out.println("old"+oldError+" older:" + olderError);

if (change < .0000001) {

break;

}

}

if(num%10==9 && count > 20){

for (int i = 0; i < sumnew.length; i++) {

sumold[i]=sumnew[i];

}

}

num++;

count++;

} // epoch

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

y[i] = i;

}

System.out.println("ended at epoche: "+count);

Plotter.generalize(x1,x2,y);

System.out.println("\nfinished testing "+ EPOCHES + " epochs in "+((System.currentTimeMillis() - time)

/1000)+" seconds");

System.out.println("\nafter training");

System.out.println("Network test is " + getRunStats(test) + "% correct.");

buildColors(matrix);

//System.out.println("\nTest network against original input:");

//testNetworkTraining(samples);

//System.out.println("\nTest network against noisy input:");

//testNetworkWithNoise1();

}

/\*\*

\* produce stats for a training epoche

\* @param samples input samples

\* @return

\*/

private double getErrorStats(ArrayList<Sample> samples)

{

double sum = 0.0;

double act = 0.0;

double expc = 0.0;

for(Sample s : samples)

{

inputSampleValues[0] = s.X1;

inputSampleValues[1] = s.X2;

int sample = s.expectedClass;

for(int j = 0; j < OUTPUT\_NEURONS; j++)

{

expected[j] = classOutput[sample][j];

}

feedForward();

if(max(Layers.get(Layers.size() - 1).nodeOutputs) != max(expected)){

expc = max(expected) + 1;

act = max(Layers.get(Layers.size() - 1).nodeOutputs) + 1;

sum += (expc-act)\*(expc-act);

}

}

return Math.sqrt((sum / (SAMPLES\*4)));

}

/\*\*

\* produce stats for a training epoche

\* @param samples input samples

\* @return

\*/

private void buildColors(ArrayList<Sample> samples)

{ Plotter p = new Plotter();

double[] redx;

double[] bluex;

double[] greenx;

double[] magentax;

double[] redy;

double[] bluey;

double[] greeny;

double[] magentay;

int countr0=0;

int countb0=0;

int countg0=0;

int countm0=0;

int color = 0;

double sum = 0.0;

double act = 0.0;

double expc = 0.0;

for(Sample s : samples)

{

inputSampleValues[0] = s.X1;

inputSampleValues[1] = s.X2;

feedForward();

color = max(Layers.get(Layers.size() - 1).nodeOutputs);

try {

switch (color) {

case 0: countr0++;

break;

case 1: countb0++;

break;

case 2: countg0++;

break;

case 3: countm0++;

break;

default:

throw new Exception("wtf");

}

}

catch(Exception e){

e.printStackTrace();

}

}

redx = new double[countr0];

bluex = new double[countb0];

greenx = new double[countg0];

magentax = new double[countm0];

redy = new double[countr0];

bluey = new double[countb0];

greeny = new double[countg0];

magentay = new double[countm0];

countr0=0;

countb0=0;

countg0=0;

countm0=0;

for(Sample s : samples)

{

inputSampleValues[0] = s.X1;

inputSampleValues[1] = s.X2;

feedForward();

color = max(Layers.get(Layers.size() - 1).nodeOutputs);

try {

switch (color) {

case 0:

redx[countr0] = s.X1;

redy[countr0] = s.X2;

countr0++;

break;

case 1:

bluex[countb0] = s.X1;

bluey[countb0] = s.X2;

countb0++;

break;

case 2:

greenx[countg0] = s.X1;

greeny[countg0] = s.X2;

countg0++;

break;

case 3:

magentax[countm0] = s.X1;

magentay[countm0] = s.X2;

countm0++;

break;

default:

throw new Exception("wtf");

}

}

catch(Exception e){

e.printStackTrace();

}

}

p.regions(redx,bluex,greenx,magentax,redy,bluey,greeny,magentay);

}

public void buildMatrix(){

//-.3 - 1.65 -.02

int x=0;

int y=0;

matrix = new ArrayList<Sample>();

for (double i = -1; i <2 ; i+=.004) {

y=0;

for (double j = -1; j <2 ; j+=.004) {

y++;

matrix.add(new Sample(i,j));

}

x++;

}

System.out.println();

}

/\*\*

\* build the layers of the network

\*/

private void buildLayers() {

int previousLayerSize = INPUT\_NEURONS;

for (int i = 0; i < NUMBER\_OF\_HIDDEN\_LAYERS; i++) {//add nodeOutputs layers

Layers.add(new Layer(previousLayerSize,HIDDEN\_NEURONS));

previousLayerSize = HIDDEN\_NEURONS;

}

Layers.add(new Layer(previousLayerSize,4));

}

/\*\*

\* feedforward implementation of the network

\*/

private void feedForward()

{

double sum = 0.0;

// Calculate input to nodeOutputs layer.

for(int hid = 0; hid < Layers.get(0).NUMBER\_OF\_NEURONS; hid++)

{

sum = 0.0;

for(int inp = 0; inp < INPUT\_NEURONS; inp++)

{

sum += inputSampleValues[inp] \* Layers.get(0).inputToSelf[inp][hid];

} // inp

sum += Layers.get(0).inputToSelf[INPUT\_NEURONS][hid];

Layers.get(0).nodeOutputs[hid] = sigmoid(sum);

} // hid

//for each layer after the inputSampleValues

for (int i = 1; i <= NUMBER\_OF\_HIDDEN\_LAYERS; i++) {//<= to get the output layer

// Calculate previousOutput to nodeOutput layer.

for(int out = 0; out < Layers.get(i).NUMBER\_OF\_NEURONS; out++)

{

sum = 0.0;

//iter over previous neurons outputs

for(int nodeNum = 0; nodeNum < Layers.get(i-1).NUMBER\_OF\_NEURONS; nodeNum++)

{

sum += Layers.get(i-1).nodeOutputs[nodeNum] \* Layers.get(i)

.inputToSelf[nodeNum][out];

}

// Add in bias.

sum += Layers.get(i).inputToSelf[INPUT\_NEURONS][out];

//set actual output to that of the sigmoid function

Layers.get(i).nodeOutputs[out] = sigmoid(sum);

}

}

}

/\*\*

\* runs the back propagation algorithm for neural nets

\*/

private void backPropagation()

{

// Calculate the output layer error

for(int out = 0; out < Layers.get(Layers.size()-1).NUMBER\_OF\_NEURONS; out++)

{

Layers.get(Layers.size()-1).err[out] = (expected[out] - Layers.get(Layers.size()-1).nodeOutputs[out]) \*

sigmoidDerivative(Layers.get(Layers.size()-1).nodeOutputs[out]);

}

// Update the weights for the output layer.

for(int out = 0; out < Layers.get(Layers.size()-1).NUMBER\_OF\_NEURONS; out++)

{

for(int hid = 0; hid < Layers.get(Layers.size()-2).NUMBER\_OF\_NEURONS; hid++)

{

Layers.get(Layers.size()-1).inputToSelf[hid][out] += (LEARNING\_RATE \* Layers.get(Layers.size()-1).err[out] \*

Layers.get(Layers.size()-2).nodeOutputs[hid]);

} // 1st hid

Layers.get(Layers.size()-1).inputToSelf[Layers.get(Layers.size()-1).NUMBER\_OF\_NEURONS][out] +=

(LEARNING\_RATE \* Layers.get(Layers.size()-1).err[out]); // Update the bias.

} // output

// Calculate the nodeOutputs layer error

for (int i = Layers.size()-1; i >0 ; i--) {

// Calculate the nodeOutputs layer error (step 3 for nodeOutputs cell).

for(int hid = 0; hid < Layers.get(i-1).NUMBER\_OF\_NEURONS; hid++)

{

Layers.get(i-1).err[hid] = 0.0;

for(int out = 0; out < Layers.get(i).NUMBER\_OF\_NEURONS; out++)

{

Layers.get(i-1).err[hid] += Layers.get(i).err[out] \* Layers.get(i).inputToSelf[hid][out];

}

Layers.get(i-1).err[hid] \*= sigmoidDerivative(Layers.get(i-1).nodeOutputs[hid]);

}

// Update the weights for the nodeOutputs layer (step 4).

if(i-1 > 0) {

for (int hid = 0; hid < Layers.get(i - 1).NUMBER\_OF\_NEURONS; hid++) {

for (int inp = 0; inp < Layers.get(i - 2).NUMBER\_OF\_NEURONS; inp++) {

Layers.get(i - 1).inputToSelf[inp][hid] += (LEARNING\_RATE \* Layers.get(i - 1).err[hid] \* Layers.get(i

- 2).nodeOutputs[inp]);

}

Layers.get(i - 1).inputToSelf[Layers.get(i - 2).NUMBER\_OF\_NEURONS][hid] += (LEARNING\_RATE \* Layers.get(i

- 1).err[hid]); // Update the bias.

}

}

}

// Update the weights for the nodeOutputs layer (step 4).

for(int hid = 0; hid < Layers.get(0).NUMBER\_OF\_NEURONS; hid++)

{

for(int inp = 0; inp < INPUT\_NEURONS; inp++)

{

Layers.get(0).inputToSelf[inp][hid] += (LEARNING\_RATE \* Layers.get(0).err[hid] \* inputSampleValues[inp]);

} // inp

Layers.get(0).inputToSelf[INPUT\_NEURONS][hid] += (LEARNING\_RATE \* Layers.get(0).err[hid]); // Update the bias.

} // hid

return;

}

/\*\*

\* used to parse the provided text files

\* @param f file

\* @return ArrayList of Sample

\*/

public ArrayList<Sample> parseFile(File f){

Scanner scanner;

String[] sA;

String s;

ArrayList<Sample> samples = new ArrayList<Sample>();

try {

scanner = new Scanner(f);

s = scanner.nextLine();

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

//System.out.println(s);

while(scanner.hasNext()){

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

sA = s.split(" ");

if(sA.length==4) {

samples.add(new Sample(Integer.parseInt(sA[0])-1,

Integer.parseInt(sA[0]), Double.parseDouble(sA[2]),

Double.parseDouble(sA[3])));

}

s = scanner.nextLine();

}

scanner.close();

}catch(FileNotFoundException e){

e.printStackTrace();

}

return samples;

}

/\*\*

\* used to parse the provided text files

\* @param f file

\* @return ArrayList of Sample

\*/

public void plotFile(File f){

Scanner scanner;

String[] sA;

String s;

Plotter p = new Plotter();

int count = 0;

int index=0;

double[] redx = new double[200];

double[] bluex = new double[200];

double[] greenx = new double[200];

double[] magentax = new double[200];

double[] redy = new double[200];

double[] bluey = new double[200];

double[] greeny = new double[200];

double[] magentay = new double[200];

ArrayList<Sample> samples = new ArrayList<Sample>();

try {

scanner = new Scanner(f);

s = scanner.nextLine();

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

//System.out.println(s);

while(scanner.hasNext()){

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

sA = s.split(" ");

if(index==200){

index=0;

}

if(sA.length==4) {

if(count<200) {

redx[index] = Double.parseDouble(sA[2]);

redy[index] = Double.parseDouble(sA[3]);

}

else if(count<400) {

bluex[index] = Double.parseDouble(sA[2]);

bluey[index] = Double.parseDouble(sA[3]);

}

else if(count<600) {

greenx[index] = Double.parseDouble(sA[2]);

greeny[index] = Double.parseDouble(sA[3]);

}

else if(count<800) {

magentax[index] = Double.parseDouble(sA[2]);

magentay[index] = Double.parseDouble(sA[3]);

}

}

s = scanner.nextLine();

count++;

index++;

}

scanner.close();

}catch(FileNotFoundException e){

e.printStackTrace();

}

p.regions(redx,bluex,greenx,magentax,redy,bluey,greeny,magentay);

}

/\*\*

\* print the min and max samples as a double

\* @param samples list of x1 x2 values

\*/

private void printMinMaxOfData(ArrayList<Sample> samples){

double max = 0;

double min = 0;

for(Sample s : samples){

if(s.X1>max){

max = s.X1;

}

if(s.X1<min){

min = s.X1;

}

if(s.X2>max){

max = s.X2;

}

if(s.X2<min){

min = s.X2;

}

}

System.out.println("min:"+min+" max:"+max);

}

/\*\*

\* produce stats for a training epoche

\* @param samples input samples

\* @return

\*/

private double getRunStats(ArrayList<Sample> samples)

{

double sum = 0.0;

for(Sample s : samples)

{

inputSampleValues[0] = s.X1;

inputSampleValues[1] = s.X2;

int sample = s.expectedClass;

for(int j = 0; j < OUTPUT\_NEURONS; j++)

{

expected[j] = classOutput[sample][j];

}

feedForward();

if(max(Layers.get(Layers.size() - 1).nodeOutputs) == max(expected)){

sum += 1;

}

}

return (sum / SAMPLES \* 100.0);

}

private void testNetworkTraining(ArrayList<Sample> samples)

{

// This function simply tests the training vectors against network.

for(Sample s : samples)

{

inputSampleValues[0] = s.X1;

inputSampleValues[1] = s.X2;

feedForward();

for(int j = 0; j < INPUT\_NEURONS; j++)

{

System.out.print(inputSampleValues[j] + "\t");

} // j

//System.out.print("Output: " + max(actual) + "\n");

} // i

return;

}

/\*\*

\* as decribed in class, take the max output

\* @param outputVector

\* @return

\*/

private int max(double[] outputVector)

{

// This function returns the maxIndex of the max of outputVector().

int maxIndex = 0;

double max = outputVector[maxIndex];

for(int i = 0; i < OUTPUT\_NEURONS; i++)

{

if(outputVector[i] > max){

max = outputVector[i];

maxIndex = i;

}

}

return maxIndex;

}

private double sigmoid(double outputSum)

{

return (1.0 / (1.0 + Math.exp(-outputSum)));

}

private double sigmoidDerivative(double val)

{

return (val \* (1.0 - val));

}

public static void main(String[] args)

{

new NeuralNet().nNet();

}

}

/\*\*

\* Created by matthewletter on 9/30/14.

\*/

public class Sample {

public double X1 = 0;

public double X2 = 0;

public int expectedClass = 0;

public int index = 0;

Sample(int expectedClass, int index, double X1, double X2 ){

this.X1=X1;

this.X2=X2;

this.expectedClass=expectedClass;

this.index=index;

}

Sample(double X1, double X2 ){

this.X1=X1;

this.X2=X2;

}

}

import java.util.Random;

/\*\*

\* Created by matthewletter on 10/16/14.

\*/

public class Layer {

//whats coming in

public int INPUT\_NEURONS = 2;

//how many nodes do I have

public int NUMBER\_OF\_NEURONS = 4;

//my weights

public double [][] inputToSelf = new double[INPUT\_NEURONS + 1][NUMBER\_OF\_NEURONS];

//my outputs

public double nodeOutputs[] = new double[NUMBER\_OF\_NEURONS];

//error

public double err[] = new double[NUMBER\_OF\_NEURONS];

/\*\*

\* builds a layer of neurons

\* @param INPUT\_NEURONS number of neurons coming into the layer

\* @param NUMBER\_OF\_NEURONS number of neurons in the layer

\*/

public Layer(int INPUT\_NEURONS, int NUMBER\_OF\_NEURONS) {

this.INPUT\_NEURONS = INPUT\_NEURONS;

this.NUMBER\_OF\_NEURONS = NUMBER\_OF\_NEURONS;

inputToSelf = new double[INPUT\_NEURONS + 1][NUMBER\_OF\_NEURONS];

nodeOutputs = new double[NUMBER\_OF\_NEURONS];

err = new double[NUMBER\_OF\_NEURONS];

assignRandomWeights();

}

/\*\*

\* assign random weights to each neuron in the layer

\*/

private void assignRandomWeights() {

//each intput gets 1 weight for every neuron

for (int inp = 0; inp <= INPUT\_NEURONS; inp++) // Do not subtract 1 here.

{

for (int nodeNum = 0; nodeNum < NUMBER\_OF\_NEURONS; nodeNum++) {

// Assign a random weight value between -0.5 and 0.5

//inputToSelf[inp][nodeNum] = new Random().nextDouble() \* 1 - .5;

inputToSelf[inp][nodeNum] = 1;

}

}

}

}

import org.math.plot.Plot2DPanel;

import javax.swing.\*;

import java.awt.\*;

import java.util.ArrayList;

/\*\*

\* Created by matthewletter on 10/15/14.

\*/

public class Plotter {

public static void generalize(double[] x1,double[] x2,double[] y) {

// create your PlotPanel (you can use it as a JPanel)

Plot2DPanel plot = new Plot2DPanel();

// define the legend position

plot.addLegend("SOUTH");

plot.addLinePlot("learning", Color.BLUE, y, x1);

plot.addLinePlot("testing", Color.RED, y, x2);

// put the PlotPanel in a JFrame like a JPanel

JFrame frame = new JFrame("class1 vs class2");

frame.setSize(1000, 1000);

frame.setContentPane(plot);

frame.setVisible(true);

frame.setDefaultCloseOperation(frame.EXIT\_ON\_CLOSE);

}

public void regions(double[] x1,double[] x2,double[] x3,double[] x4,double[] y1,double[] y2,

double[] y3,double[] y4) {

// create your PlotPanel (you can use it as a JPanel)

Plot2DPanel plot = new Plot2DPanel();

// define the legend position

plot.addLegend("SOUTH");

System.out.println(x1.length+" : "+x2.length);

if(x1.length>0&&y1.length>0) {

plot.addScatterPlot("class1", Color.RED, x1, y1);

}

if(x2.length>0&&y2.length>0) {

plot.addScatterPlot("class2", Color.BLUE, x2, y2);

}

if(x3.length>0&&y3.length>0) {

plot.addScatterPlot("class3", Color.GREEN, x3, y3);

}

if(x4.length>0&&y4.length>0) {

plot.addScatterPlot("class4", Color.MAGENTA, x4, y4);

}

// put the PlotPanel in a JFrame like a JPanel

JFrame frame = new JFrame("class1 vs class2");

frame.setSize(1000, 1000);

frame.setContentPane(plot);

frame.setVisible(true);

frame.setDefaultCloseOperation(frame.EXIT\_ON\_CLOSE);

}

}