Java Graphics Report

Ben Crabbe

University of Bristol, UK

May 27, 2015

1 Introduction

For my research project I am working with neural networks using a high level framework. To get some experience before I begin properly I have decided to use this assignment to explore some of the issues with training neural networks. I hope it will also provide some good OO design practise.

2 Forward Propagation

A neural network consists of layers of one or more units or 'neurons' as shown in figure **??**. Each neuron, i, recieves inputs, x_i, from each of the neurons in the layer immediate below it. It computes a weighted sum of these inputs plus a bias term, b_i, and applies some non linear 'activation' function, $f(.)$, producing that neurons output, a_i. The weights in these sums can be thought of as the connection strength between two neurons, these and the bias terms are the adjustable parameters.

$$a_1^{(2)} = f(W_{11}^{(2)}x_1 + W_{12}^{(2)}x_2 + W_{13}^{(2)}x_3 + b_1^{(2)}) \tag{1}$$
$$a_2^{(2)} = f(W_{21}^{(2)}x_1 + W_{22}^{(2)}x_2 + W_{23}^{(2)}x_3 + b_2^{(2)}) \tag{2}$$
$$a_3^{(2)} = f(W_{31}^{(2)}x_1 + W_{32}^{(2)}x_2 + W_{33}^{(2)}x_3 + b_3^{(2)}) \tag{3}$$
$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(3)}a_1^{(2)} + W_{12}^{(3)}a_2^{(2)} + W_{13}^{(3)}a_3^{(2)} + b_1^{(3)}) \tag{4}$$

in general, and in vector form:

$$\boldsymbol{a}^{(l)} = \begin{bmatrix} f(\boldsymbol{W}_1^{(l)} \cdot \boldsymbol{a}^{(l-1)} + b_1^{(l)}) \\ f(\boldsymbol{W}_2^{(l)} \cdot \boldsymbol{a}^{(l-1)} + b_2^{(l)}) \\ \vdots \\ f(\boldsymbol{W}_n^{(l)} \cdot \boldsymbol{a}^{(l-1)} + b_n^{(l)}) \end{bmatrix} \tag{5}$$

To implement these I have made these classes:

- Driver: a simple initialiser class with some static testing methods. It also contains a random number generator, I thought it was better to have 1 of these in a static variable rather than creating one each time it was needed.

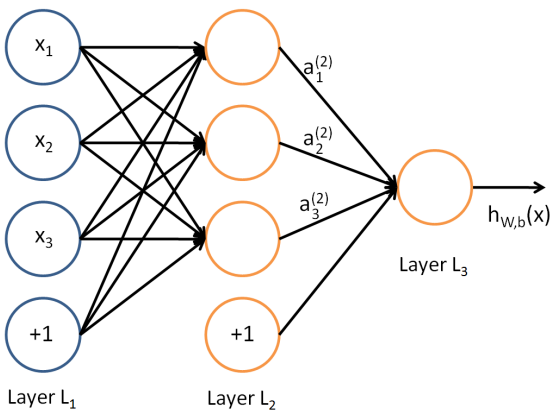

Figure 1: The illustration of equations 1,2,3 and 4.

- Neuron: contains a vector of connection weights
- HiddenLayer: contains a list of neurons. Has a method "FloatMatrix getOutput(FloatMatrix input)" which computes equations 1, 2, 3, 4 etc.
- InputLayer: I have made this class to handle producing inputs, to start with I want to examine the networks ability to produce a sine function, so there is a function generateRandomSinSample which produces a random number between $-\pi/2$ and $\pi/2$. I thought
- Network: contains the an InputLayer and a list of HiddenLayers. Has a function "FloatMatrix computeFoward-Pass(FloatMatrix input)" which computes the whole forward propagation taking input through the network returning the output.

FloatMatrix is a class from the jblas[1] library. I have used these for every vector since they provide optimised computations which should speed up the network.

One tricky issue is how best to deal with the bias' in each neuron, it is essentially the same as a weight but its input value does not come from the layer below, instead it is a constant. One way that is illustrated in figure **??** is to add a constant term to the input of each layer, then we can just treat the bias as an extra weight. Therefore each layer has numberOfNeurons in the previous layer + 1 weights. When calculating the output of each layer with the getOutput method:

```
FloatMatrix getOutput(FloatMatrix input)
{
    if(input.length!=numberOfInputs) {
        throw new Error("ERROR: input vector to layer" + layerNumber +
         " is not correct size. expected: " +
        numberOfInputs + " got: " + input.length);
    }
    FloatMatrix inputPlusBiasConstant = addConstantTermToInputVector(input);

    FloatMatrix outputVector = new FloatMatrix(neurons.size());
    int i=0;
    for(Neuron n: neurons) {
        outputVector.put(i, 0,
        activationFunction(n.getWeightVector().dot(inputPlusBiasConstant)));
        ++i;
    }
    return outputVector;
}
```

we add an additional element to input vector with "private FloatMatrix addConstantTermToInputVector(FloatMatrix input)" and everything else is taken care of.

Issues were ironed out by writing this test:

```
void testsNetworkDefinition(int... definition)
{
    System.out.println("Network initialised with layers: " + Arrays.toString(layerWidths) );

    System.out.println("\n\nLAYER 0");
    Driver.is(inputLayer.getWidth(), definition[0],
    "checking the sizes of the layers and the number of inputs they should recieve correct");
    int i=1;
    for(HiddenLayer l: hiddenLayers) {
        System.out.println("\n\nLAYER " + i);

        Driver.is(l.getWidth(),definition[i],"has correct number of neurons");
        Driver.is(l.getNumberOfInputs(),definition[i-1],"has the correct number of connections");
        ++i;
    }
    System.out.println("\n\nLAYER 0");
    Driver.is(inputLayer.getInput().length, definition[0],
```

[1] http://jblas.org/

```
        "is the size of the InputLayer.getInput() returned vector correct");
        Driver.is(hiddenLayers.get(0).getNumberOfInputs(), inputLayer.getInput().length,
        "is the size of the input vector the size expected by the first layer.");

        System.out.println("\n\nLAYER 1");
        FloatMatrix input = inputLayer.getInput();
        FloatMatrix firstLayerOutput = hiddenLayers.get(0).getOutput(input);
        Driver.is(firstLayerOutput.length, definition[1],
        "is the output of the first hidden layer the correct size");

        i=2;
        FloatMatrix input2forTest;
        input.copy(firstLayerOutput);
        while(i<definition.length) {
            System.out.println("\n\nLAYER " + hiddenLayers.get(i-1).getLayerNumber());
            Driver.is(input.length, hiddenLayers.get(i-1).getNumberOfInputs(),
            "is the output of the previous layer correctly sized");
            input = hiddenLayers.get(i-1).getOutput(input);
            ++i;
        }
    }
```

Here is the full code at this stage:

2.1 Driver

```

import java.util.*;
/**
Driver

Initialisation class

*/
class Driver
{
    static Random randomNumberGen;
    static int numberOfFails;

    Driver()
    {
        this.randomNumberGen = new Random();
        numberOfFails=0;
    }

    static void is(Object x, Object y)
    {
        System.out.print("testing: " + x.toString() + " = " + y.toString() );

        if (x==y || (x != null && x.equals(y)) ) {
            System.out.println("...pass");
            return;
        }
        System.out.println("...fail");
    }

    static void is(Object x, Object y, String description)
    {
        System.out.println("test description: " + description );

        System.out.print("testing: " + x.toString() + " = " + y.toString() );

        if (x==y || (x != null && x.equals(y)) ) {
```

```
            System.out.println("...pass");
            return;
        }
        System.out.println("...fail");
        ++numberOfFails;
    }

    static void finishTesting(String suiteName)
    {
        System.out.println("\n\n" + suiteName + " finished with " + numberOfFails + " fails.");
        System.out.println("*************************************************************\n\n");

        numberOfFails=0;
    }

    static void tests()
    {
        float randomX = Driver.randomNumberGen.nextFloat()*(float)Math.PI;
        System.out.println("x = " + randomX);
        System.out.println("y = " + (float)Math.sin(randomX));

    }

    public static void main(String[] args)
    {
        Driver program = new Driver();
        Network net = new Network(1,5,4,1);
        //Trainer trainer = new Trainer(Network);
    }
}
```

2.2 HiddenLayer

```
import org.jblas.*;
import java.util.*;
/**
Layers


*/

class HiddenLayer
{
    private int numberOfInputs;
    private List<Neuron> neurons;
    private int layerNumber;

    HiddenLayer(int layerNumber, int numberOfNeurons, int numberOfInputs)
    {
        this.layerNumber = layerNumber;
        this.numberOfInputs = numberOfInputs;
        neurons = new ArrayList<Neuron>();
        for(int i=1; i<=numberOfNeurons; ++i) {
            neurons.add(new Neuron(numberOfInputs));
        }
    }

    FloatMatrix getOutput(FloatMatrix input)
    {
        if(input.length!=numberOfInputs) {
            System.out.println("ERROR: input vector to layer" + layerNumber +
             " is not correct size. expected: " +
```

```
            numberOfInputs + " got: " + input.length);
31          throw new Error();
        }
33      FloatMatrix inputPlusBiasConstant = addConstantTermToInputVector(input);

35      FloatMatrix outputVector = new FloatMatrix(neurons.size());
        int i=0;
37      for(Neuron n: neurons) {
            outputVector.put(i, 0,
39          activationFunction(n.getWeightVector().dot(inputPlusBiasConstant)));
            ++i;
41      }
        return outputVector;
43  }

45  //rectified linear unit (ReLU) non linear activation applied on each neuron
    private float activationFunction(float z)
47  {
        if(z>0) return z;
49      else return 0;
    }
51
    private FloatMatrix addConstantTermToInputVector(FloatMatrix input)
53  {
        FloatMatrix inputPlusBiasConstant = new FloatMatrix(numberOfInputs+1);
55      inputPlusBiasConstant.put(0,0,-1);
        for(int i=1; i<=numberOfInputs; ++i) {
57          inputPlusBiasConstant.put(i,0,input.get(i-1,0));
        }
59      return inputPlusBiasConstant;
    }
61
    int getWidth()
63  {
        return neurons.size();
65  }

67  int getNumberOfInputs()
    {
69      return numberOfInputs;
    }
71
    int getLayerNumber()
73  {
        return layerNumber;
75  }

77  static void tests()
    {
79
    }
81
    public static void main(String[] args)
83  {
      HiddenLayer test = new HiddenLayer(1,3,2);
85  }
}
```

2.3 Neuron

```
import org.jblas.*;
2 import java.util.*;
```

```
/** Each neuron, i, in a layer, l, with l-1 having n neurons has a
    connection vector
        W_i^l = [ b, W_1i, W_2i, ..., W_ni]
    where W_ni is the connection to unit n in l-1
    and b is a bias term.
*/
class Neuron
{
    private int numberOfConnections;
    private FloatMatrix weights;//numberOfConnections+1 for bias

    Neuron(int numberOfInputs)
    {
        //there are n+1 connections. to n units in prev layer and a bias
        numberOfConnections=numberOfInputs+1;
        weights = FloatMatrix.rand(numberOfConnections);
    }

    FloatMatrix getWeightVector()
    {
        return weights.dup();
    }

    void tests()
    {
        System.out.println(weights.toString());

    }

    public static void main(String[] args)
    {
        Neuron n = new Neuron(3);
        n.tests();
    }
}
```

2.4 InputLayer

```
import org.jblas.*;
import java.util.*;


class InputLayer
{
    private int width;

    InputLayer(int width)
    {
        this.width = width;
    }

    FloatMatrix getInput()
    {
        return generateRandomSinSample();
    }

    private FloatMatrix generateRandomSinSample()
    {
        float randomX = Driver.randomNumberGen.nextFloat()*(float)Math.PI-((float)Math.PI/2);
        //float randomY = (float)Math.sin(randomX);
        /*FloatMatrix(int newRows, int newColumns, float... newData)
```

```
            Create a new matrix with newRows rows, newColumns columns using newData> as the data.
        */
        return new FloatMatrix(1,1,randomX);
    }

    int getWidth()
    {
        return width;
    }

    private void tests()
    {
        float randomX = Driver.randomNumberGen.nextFloat()*(float)Math.PI-((float)Math.PI/2);
        System.out.println("x = " + randomX);
        System.out.println("y = " + (float)Math.sin(randomX));

    }

    public static void main(String[] args)
    {
        InputLayer il = new InputLayer(3);
        il.tests();

    }
}
```

2.5 Network

```
import org.jblas.*;
import java.util.*;

class Network
{
    private InputLayer inputLayer;
    private List<HiddenLayer> hiddenLayers;
    private int[] layerWidths;

    Network(int... definition)
    {
        layerWidths = definition.clone();
        inputLayer = new InputLayer(layerWidths[0]);
        hiddenLayers = new ArrayList<HiddenLayer>();
        for(int i=1; i<definition.length; ++i) {
            hiddenLayers.add(new HiddenLayer(i, layerWidths[i], layerWidths[i-1]));
        }
    }

    FloatMatrix computeFowardPass(FloatMatrix input)
    {
        //FloatMatrix input = inputLayer.getInput();
        //System.out.println("input: " + input.toString());
        int i=1;
        for(HiddenLayer l: hiddenLayers) {
            input = l.getOutput(input);
            //System.out.println("layer " + i + ": " + input.toString());
            ++i;
        }
        return input;
    }

    void testsNetworkDefinition(int... definition)
    {
```

```
        System.out.println("Network initialised with layers: " + Arrays.toString(layerWidths) );

        System.out.println("\n\nLAYER 0");
        Driver.is(inputLayer.getWidth(), definition[0],
        "checking the sizes of the layers and the number of inputs they should recieve correct");
        int i=1;
        for(HiddenLayer l: hiddenLayers) {
            System.out.println("\n\nLAYER " + i);

            Driver.is(l.getWidth(),definition[i],"has correct number of neurons");
            Driver.is(l.getNumberOfInputs(),definition[i-1],"has the correct number of connections");
            ++i;
        }
        System.out.println("\n\nLAYER 0");
        Driver.is(inputLayer.getInput().length, definition[0],
        "is the size of the InputLayer.getInput() returned vector correct");
        Driver.is(hiddenLayers.get(0).getNumberOfInputs(), inputLayer.getInput().length,
        "is the size of the input vector the size expected by the first layer.");

        System.out.println("\n\nLAYER 1");
        FloatMatrix input = inputLayer.getInput();
        FloatMatrix firstLayerOutput = hiddenLayers.get(0).getOutput(input);
        Driver.is(firstLayerOutput.length, definition[1],
        "is the output of the first hidden layer the correct size");

        i=2;
        FloatMatrix input2forTest;
        input.copy(firstLayerOutput);
        while(i<definition.length) {
            System.out.println("\n\nLAYER " + hiddenLayers.get(i-1).getLayerNumber());
            Driver.is(input.length, hiddenLayers.get(i-1).getNumberOfInputs(),
            "is the output of the previous layer correctly sized");
            input = hiddenLayers.get(i-1).getOutput(input);
            ++i;
        }
      //  System.out.println("forward pass: " + computeFowardPass().toString());
    }

    public static void main(String[] args)
    {
        Driver d = new Driver();
        Network net =  new Network(1,5,1);
        net.testsNetworkDefinition(1,5,1);

        Network net2 =  new Network(1,9,8,7,3,7);
        net2.testsNetworkDefinition(1,9,8,7,3,7);
        Driver.finishTesting("testsNetworkDefinition");
    }
}
```

3 Training

I am interested in training the network using backpropagation and (stochastic) gradient descent. To handle the training I have made a Trainer class. The Trainer should be able to train any number of networks therefore the network should a member of it rather than the other way around.

The network is presented with an example consisting of input, x, and expected output, y. We compute the actual output of the network, $h_{W,b}(x)$ (the network is initialised with small random weights) and calculate the error as

$$J(W,b;x,y) = \frac{1}{2}\left\|h_{W,b}(x) - y\right\|^2. \tag{6}$$

we then update each weight and bias by

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) \tag{7}$$

$$b_{i}^{(l)} = b_{i}^{(l)} - \alpha \frac{\partial}{\partial b_{i}^{(l)}} J(W,b) \tag{8}$$

to compute the values of $\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b)$ and $\frac{\partial}{\partial b_{i}^{(l)}} J(W,b)$ we use backpropagation. To quote its description from `http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/`

> The intuition behind the backpropagation algorithm is as follows. Given a training example (x, y), we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $h_{W,b}(x)$. Then, for each node(neuron) i in layer l, we would like to compute an "error term" $\delta_i^{(l)}$ that measures how much that node was "responsible" for any errors in our output. For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(n_l)}$ (where layer n_l is the output layer). For the hidden units we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as an input.

This is the vectorised algorithm:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}, using the equations defining the forward propagation steps

2. For the output layer (layer n_l), set
$$\boldsymbol{\delta}^{(n_l)} = -(\boldsymbol{y} - \boldsymbol{a}^{(n_l)}) \bullet f'(\boldsymbol{z}^{(n_l)}) \tag{9}$$
where $\boldsymbol{a}^{(n_l)}$ is the output vector, $\boldsymbol{y}$ is the expected output, and $\boldsymbol{z}^{(n_l)} = [\boldsymbol{W}_1^{(n_l)} \cdot \boldsymbol{a}^{(n_l-1)} + b_1^{(n_l)}, \ldots, \boldsymbol{W}_n^{(n_l)} \cdot \boldsymbol{a}^{(n_l-1)} + b_n^{(n_l)}]$ is the vector of the weighted sums of inputs for each neuron in the final layer. The function $f'(.)$ should be applied element wise to z and depends on the form of the activation function, I will use $f(x) = max(0, x)$ therefore $f'(z_i^{(n_l)}) = z_i^{(n_l)}$ if $z_i^{(n_l)} > 0$ and $= 0$ if $z_i^{(n_l)} \leq 0$. The $\bullet$ denotes element wise multiplication.

3. And for the hidden layers
$$\boldsymbol{\delta}^{(l)} = \boldsymbol{W}^{(l+1)} \boldsymbol{\delta}^{(l+1)} \bullet f'(\boldsymbol{z}^{(l)}) \tag{10}$$
$$= \begin{bmatrix} W_{11} & \ldots & W_{1n} \\ \vdots & \ddots & \vdots \\ W_{m1} & \ldots & W_{mn} \end{bmatrix} \boldsymbol{\delta}^{(l+1)} \bullet f'(\boldsymbol{z}^{(l)}) \tag{11}$$
where $W_{ij}^{(l+1)}$ is the weight in the connection between neuron i in layer l and neuron j in layer $l+1$, or the jth element of the ith neurons weight vector in the layer $l+1$.

4. Then compute the matrix of partial derivatives $\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b)$ and $\frac{\partial}{\partial b_{i}^{(l)}} J(W,b)$ for each layer as
$$\nabla_{W^{(l)}} J(W,b) = \delta^{(l)} (a^{(l-1)})^T \tag{12}$$
$$\nabla_{b^{(l)}} J(W,b) = \delta^{(l)} \tag{13}$$

which is implemented with this method in the Network class:

```
List<List<FloatMatrix>> backpropagate(FloatMatrix networkOutput, FloatMatrix expectedOutput)
{
    //lists of matricies for each layer. These will hold the dE/dW^(l)_ij dE/db^(l)_i terms
    List<FloatMatrix> gradLoss_wrtW = new ArrayList<FloatMatrix>(hiddenLayers.size());
    List<FloatMatrix> gradLoss_wrtB = new ArrayList<FloatMatrix>(hiddenLayers.size());
```

```
        //get each layer's position in hiddenLayers list also postion in gradLoss_wrtW/b
        int lastLayer = hiddenLayers.size()-1;

        //add each layer to 0 position in the list, then they all shift down each time
        //gradLoss_wrtB_l = that layers delta vector
        FloatMatrix deltasLplus1 = hiddenLayers.get(lastLayer).computeDeltas(networkOutput,
        expectedOutput);
        gradLoss_wrtB.add(0, deltasLplus1);
        gradLoss_wrtW.add(0, deltasLplus1.mmul( hiddenLayers.get(lastLayer).getInputs().transpose()))
    ;

        FloatMatrix deltasL;
        for(int layer=lastLayer-1; layer>=0; --layer) {
            deltasL = hiddenLayers.get(layer).computeDeltas( hiddenLayers.get(layer+1).
                                                                getWeightMatrix(),
                                                        deltasLplus1);
            gradLoss_wrtB.add(0, deltasL);
            gradLoss_wrtW.add(0, deltasL.mmul( hiddenLayers.get(layer).getInputs().transpose()));
            deltasLplus1.copy(deltasL);
        }
        List<List<FloatMatrix>> gradWandB = new ArrayList<List<FloatMatrix>>();
        gradWandB.add(gradLoss_wrtW);
        gradWandB.add(gradLoss_wrtB);
        return gradWandB;
    }
```

Because the deltas computation was different for the output layer I created an OutputLayer subclass of Hiddenlayer

```
import org.jblas.*;
import java.util.*;
/**
Layers


*/

class OutputLayer extends HiddenLayer
{
    OutputLayer(int layerNumber, int numberOfNeurons, int numberOfInputs)
    {
        super(layerNumber, numberOfNeurons, numberOfInputs);
    }

    FloatMatrix computeDeltas(FloatMatrix networkOutput, FloatMatrix expectedOutput)
    {
        //d = (y-a):
        deltas = expectedOutput.sub(networkOutput);
        //d = -d bullet f'(z)
        deltas.muli(-1).muli(fDashOfActivationZ);
        return deltas.dup();
    }

    void testComputeDeltasFinalLayer()
    {
        fDashOfActivationZ = FloatMatrix.rand(numberOfNeurons);
        FloatMatrix networkOutput = FloatMatrix.rand(numberOfNeurons, 1);
        FloatMatrix expectedOutput = FloatMatrix.rand(numberOfNeurons, 1);

        FloatMatrix deltas = computeDeltas(networkOutput, expectedOutput);
        Driver.printMatrixDetails("deltas", deltas);
        Driver.is(deltas.rows, numberOfNeurons, "does delta Matrix have correct rows (number of
    neurons)");
        Driver.is(deltas.columns, 1, "does delta Matrix have correct columns");
    }

```

```
    void tests()
    {
        testComputeDeltasFinalLayer();
    }

    public static void main(String[] args)
    {
        OutputLayer l = new OutputLayer(1, 2, 3);
        l.tests();
        OutputLayer l2 = new OutputLayer(1, 19, 21);
        l2.tests();
    }
}
```

regular hidden layers compute deltas with

```
    FloatMatrix computeDeltas(FloatMatrix weightMatrixLplus1, FloatMatrix deltasLplus1)
    {
        if(weightMatrixLplus1.columns!=numberOfNeurons) {
            System.out.println("computeDeltas in layer" + layerNumber + " recieved: ");
            Driver.printMatrixDetails("weightMatrixLplus1", weightMatrixLplus1);
            Driver.printMatrixDetails("deltasLplus1", deltasLplus1);
            throw new Error("weightMatrixLplus1 should have " + numberOfNeurons +
                            "columns." );
        }
        //deltas = W^(l+1) * d^(l+1)
        deltas = weightMatrixLplus1.transpose().mmul(deltasLplus1);
        //d = d bullet f'(z)
        deltas.muli(fDashOfActivationZ);
        return deltas.dup();
    }
```

I have changed the forwards pass methods to store all the information needed in the backwardsPass methods, I also had to change a number of the access modifiers to protected to allow the Output layer to set them.

```
    private int numberOfInputs;
    private List<Neuron> neurons;
    private int layerNumber;
    protected int numberOfNeurons;
    private FloatMatrix activationZ;
    protected FloatMatrix fDashOfActivationZ;
    protected FloatMatrix deltas;
    private FloatMatrix inputs;
    private FloatMatrix outputs;

    FloatMatrix getOutput(FloatMatrix input)
    {
        if(input.length!=numberOfInputs) {
            throw new Error("ERROR: input vector to layer" + layerNumber +
             " is not correct size. expected: " +
            numberOfInputs + " got: " + input.length);
        }
        //save the input for backwards pass
        this.inputs = input.dup();
        FloatMatrix inputPlusBiasConstant = addConstantTermToInputVector(input);
        FloatMatrix outputVector = new FloatMatrix(numberOfNeurons);
        int i=0;
        for(Neuron n: neurons) {
            float neuronActivation = n.getWeightVector().dot(inputPlusBiasConstant);
            activationZ.put(i, 0, neuronActivation);
            fDashOfActivationZ.put(i, 0, activationFunctionDash(neuronActivation));
            outputVector.put(i, 0, activationFunction(neuronActivation));
            ++i;
```

```
29          }
            outputs = outputVector.dup();
31          return outputVector;
        }
```

I had a number of bugs in the backPropagate method which took some time to iron out. Matricies wern't the right size to be multiplied etc.. I wasn't entirely confident that I had the correct equations in the first place since the tutorial I was following (the stanford link above) used a different definition of the network where the weights belong to the layer producing the input rather than recieving it. The backPropagate/compute deltas methods were difficult to test since they were using values computed in the forwards pass.

Eventually I found that I was multiplying the deltas by the transpose of the outputs rather than the inputs by mistake.

I started out checking the my weight matrix extraction method in the HiddenLayers was working correctly

```
        FloatMatrix getWeightMatrix()
2       {
            FloatMatrix weightMatrix = new FloatMatrix(numberOfNeurons, numberOfInputs);
4           int i=0;
            for(Neuron n: neurons) {
6               weightMatrix.putRow(i, n.getWeightVectorNoBias());
                ++i;
8           }
            return weightMatrix;
10      }

12      void testGetWeightMatrix()
        {
14          FloatMatrix weightMatrix = getWeightMatrix();
            int unit=0;
16          for(Neuron n: neurons) {
                FloatMatrix weightsN = n.getWeightVectorNoBias();
18              for(int i=0; i<numberOfInputs; ++i) {
                    Driver.is( weightsN.get(i,0), weightMatrix.get(unit, i),
20                  "checking that weight matrix equals each element of the weights");
                }
22              ++unit;
            }
24      }


26
```

then I checked that compute deltas were working as expected:

```
1       void testComputeDeltas()
        {
3           fDashOfActivationZ = FloatMatrix.rand(numberOfNeurons);
            FloatMatrix weightMatrixLplus1 = FloatMatrix.rand(3, numberOfNeurons);
5           FloatMatrix deltasLplus1 = FloatMatrix.rand(3);

7           FloatMatrix deltas = computeDeltas(weightMatrixLplus1,deltasLplus1);
            Driver.printMatrixDetails("deltas", deltas);
9           Driver.is(deltas.rows, numberOfNeurons, "does delta Matrix have correct rows (number of
        neurons)");
            Driver.is(deltas.columns, 1, "does delta Matrix have correct columns");
11
            weightMatrixLplus1 = FloatMatrix.rand(20, numberOfNeurons);
13          deltasLplus1 = FloatMatrix.rand(20);

15          deltas = computeDeltas(weightMatrixLplus1,deltasLplus1);
            Driver.printMatrixDetails("deltas", deltas);
17          Driver.is(deltas.rows, numberOfNeurons, "does delta Matrix have correct rows (number of
        neurons)");
            Driver.is(deltas.columns, 1, "does delta Matrix have correct columns");
```

```
    }
```

and then I took a copy of backPropate and filled it with calls to new method in HiddenLayer class printLayerDetails() and a static helper method in Driver printMatrixDetails :

```
    void printLayerDetails()
    {
        System.out.println("\nLAYER " + layerNumber);
        System.out.println("number of inputs: " + numberOfInputs);
        System.out.println("number of neurons: " + numberOfNeurons);
        Driver.printMatrixDetails("activationZ", activationZ);
        Driver.printMatrixDetails("fDashOfActivationZ", fDashOfActivationZ);
        Driver.printMatrixDetails("deltas", deltas);
        Driver.printMatrixDetails("inputs", inputs);
        Driver.printMatrixDetails("outputs", outputs);
        FloatMatrix weightMatrix = getWeightMatrix();
        Driver.printMatrixDetails("weightMatrix", weightMatrix);
        FloatMatrix biasVector = getBiasVector();
        Driver.printMatrixDetails("biasVector", biasVector);
        System.out.println("\n");
    }

    static void printMatrixDetails(String name, FloatMatrix m)
    {
        System.out.println("FloatMatrix " + name + " has " + m.rows + " rows and " + m.columns + "
    columns. Contains: ");
        System.out.println(m.toString()+ "\n");
    }
```

which I found very helpful for checking the state of the layers in the backPropogate method:

```
    List<List<FloatMatrix>> backpropagateWithChecks(FloatMatrix networkOutput, FloatMatrix
    expectedOutput)
    {
        Driver.is(networkOutput.length, layerWidths[layerWidths.length-1], "are the netOutputs right
    size");
        Driver.is(expectedOutput.length, layerWidths[layerWidths.length-1]);

        //lists of matricies for each layer. These will hold the dE/dW^(l)_ij dE/db^(l)_i terms
        List<FloatMatrix> gradLoss_wrtW = new ArrayList<FloatMatrix>(hiddenLayers.size());
        List<FloatMatrix> gradLoss_wrtB = new ArrayList<FloatMatrix>(hiddenLayers.size());

        //get each layer's position in hiddenLayers list also postion in gradLoss_wrtW/b
        int lastLayer = hiddenLayers.size()-1;

        //add each layer to 0 position in the list, then they all shift down each time
        //gradLoss_wrtB_l = that layers delta vector
        FloatMatrix deltasLplus1 = hiddenLayers.get(lastLayer).computeDeltas(networkOutput,
    expectedOutput);
        gradLoss_wrtB.add(0, deltasLplus1);
        gradLoss_wrtW.add(0, deltasLplus1.mmul( hiddenLayers.get(lastLayer).getInputs().transpose()))
    ;
        hiddenLayers.get(lastLayer).printLayerDetails();

        Driver.is(hiddenLayers.get(lastLayer).getNumberOfNeurons(), deltasLplus1.rows,
             " are output layer deltas the sma number of rows as number of neurons");
        Driver.is(1, deltasLplus1.columns, " are output layer deltas have 1 col");
        Driver.printMatrixDetails("gradLoss_wrtW.get(0)", gradLoss_wrtW.get(0));

        FloatMatrix deltasL;
        for(int layer=lastLayer-1; layer>=0; --layer) {
            hiddenLayers.get(layer).printLayerDetails();
```

```
            deltasL = hiddenLayers.get(layer).computeDeltas( hiddenLayers.get(layer+1).
    getWeightMatrix(),
                                                             deltasLplus1);
            Driver.is(hiddenLayers.get(layer).getNumberOfNeurons(), deltasL.rows,
             " are layer deltas the sma number of rows as number of neurons");
            Driver.is(1, deltasL.columns,  " do layer deltas have 1 col");

            gradLoss_wrtB.add(0, deltasL);

            Driver.is(gradLoss_wrtB.get(0).rows,
             hiddenLayers.get(layer).getNumberOfNeurons(),
             " are each hiddenLayers gradLoss_wrtB the sma number of rows as its number of neurons");
            Driver.is(gradLoss_wrtB.get(0).columns,
             1,
             " are each hiddenLayers gradLoss_wrtB has 1 col");

            gradLoss_wrtW.add(0, deltasL.mmul( hiddenLayers.get(layer).getInputs().transpose()));

            Driver.printMatrixDetails("gradLoss_wrtW.get(0)", gradLoss_wrtW.get(0));

            Driver.is(gradLoss_wrtW.get(0).rows,
             hiddenLayers.get(layer).getWeightMatrix().rows,
             " are each hiddenLayers gradLoss_wrtW the sma number of rows as its weight matrix");
              Driver.is(gradLoss_wrtW.get(0).columns,
             hiddenLayers.get(layer).getWeightMatrix().columns,
             " are each hiddenLayers gradLoss_wrtW the sma number of cols as its weight matrix");
             deltasLplus1.copy(deltasL);
        }
        List<List<FloatMatrix>> gradWandB = new ArrayList<List<FloatMatrix>>();
        gradWandB.add(gradLoss_wrtW);
        gradWandB.add(gradLoss_wrtB);
        Driver.finishTesting("backpropagateWithChecks");
        return gradWandB;
    }

    void testBackPropagate()
    {
        System.out.println("testBackPropagate");
        testForwardPass();
        List<List<FloatMatrix>> gradWandB = backpropagateWithChecks(
            FloatMatrix.rand(layerWidths[layerWidths.length-1]),
            FloatMatrix.rand(layerWidths[layerWidths.length-1])     );
        List<FloatMatrix> gradW = gradWandB.get(0);
        List<FloatMatrix> gradB = gradWandB.get(1);

        for(int layer=0; layer<hiddenLayers.size(); ++layer) {
            hiddenLayers.get(layer).printLayerDetails();
            Driver.printMatrixDetails("gradW.get(layer)", gradW.get(layer));
            Driver.is(gradW.get(layer).rows, hiddenLayers.get(layer).getNumberOfNeurons(),
            "does GradW rows equal number of nuerons");
            Driver.is(gradW.get(layer).columns, hiddenLayers.get(layer).getWeightMatrix().columns,
            "does GradW columns equal weightMatrix columns");
        }
        Driver.finishTesting("testBackPropagate");
    }
```

The Trainer class which handles all of this accumulating weight/bias updates over a batch of examples seemed a better place for creating the examples, rather than the InputLayer class, which is now redundant.

```
import org.jblas.*;
import java.util.*;

class Trainer
{
```

```
    private Network net;
    private int inputWidth;
    private int trainingBatchSize=10000;
    private int validationSetSize=20;
    private float weightDecay=0;
    private float momentum=(float)0.9;
    private float learningRate=(float)0.001;

    Trainer(Network net, int inputWidth)
    {
        this.net = net;
        this.inputWidth = inputWidth;
    }

    void trainNetwork()
    {
        int batchNumber=1;
        while(batchNumber<10000) {
            presentTrainingBatch();
            float validationError = measureValidationError();
            System.out.println("Batch number = " + batchNumber +
                               ". Validation error = " + validationError);
            ++batchNumber;
        }
    }

    private float measureValidationError()
    {
        float validationError = 0;
        for(int i=1; i<=validationSetSize; ++i) {
            List<FloatMatrix> example = getExample();
            FloatMatrix netOutput = net.computeFowardPass(example.get(0));
            validationError += computeExampleLoss(netOutput, example.get(1));
        }
        return validationError/(float)validationSetSize;
    }

    private void presentTrainingBatch()
    {
        List<FloatMatrix> deltaW_l = new ArrayList<FloatMatrix>();
        List<FloatMatrix> deltaB_l = new ArrayList<FloatMatrix>();
        for(int i=1; i<=trainingBatchSize; ++i) {
            List<FloatMatrix> example = getExample();
            List<List<FloatMatrix>> gradLoss_wrtWandB = presentTrainingExample(example.get(0),
    example.get(1));
            List<FloatMatrix> gradLoss_wrtW = gradLoss_wrtWandB.get(0);
            List<FloatMatrix> gradLoss_wrtB = gradLoss_wrtWandB.get(1);
            if(i==1) {
                deltaW_l = Driver.copySizingInMatrixList(gradLoss_wrtW);
                deltaB_l = Driver.copySizingInMatrixList(gradLoss_wrtB);
                net.initialisePreviousUpdates(gradLoss_wrtW,gradLoss_wrtB);
            }
            //accumulate each update gradLoss_wrtW/B in deltaW_l
            Driver.addFloatMatrixListsi(deltaW_l, gradLoss_wrtW);
            Driver.addFloatMatrixListsi(deltaB_l, gradLoss_wrtB);
        }
        Driver.scalarMultiplyFloatMatrixListsi(deltaW_l, (1/(float)trainingBatchSize));
        Driver.scalarMultiplyFloatMatrixListsi(deltaB_l, (1/(float)trainingBatchSize));
        net.updateParameters(deltaW_l, deltaB_l, weightDecay, momentum, learningRate);
    }

    private List<List<FloatMatrix>> presentTrainingExample(FloatMatrix input, FloatMatrix label)
    {
        FloatMatrix netOutput = net.computeFowardPass(input);
        return net.backpropagate(netOutput, label);
    }
```

```
72      private float computeExampleLoss(FloatMatrix netOutput, FloatMatrix expectedOutput)
        {
74          netOutput.subi(expectedOutput);
            return (float)0.5*netOutput.dot(netOutput);
76      }

78      //returns a 2 floatMatricies, first is the input, 2nd is the expected output
        private List<FloatMatrix> getExample()
80      {
            return generateRandomSinSample();
82      }

84      private List<FloatMatrix> generateRandomSinSample()
        {
86          List<FloatMatrix> example = new ArrayList<FloatMatrix>();

88          float randomX = Driver.randomNumberGen.nextFloat()*(float)Math.PI-((float)Math.PI/2);
            float randomY = (float)Math.sin(randomX);
90          /*FloatMatrix(int newRows, int newColumns, float... newData)
              Create a new matrix with newRows rows, newColumns columns using newData> as the data.
92          */
            example.add(new FloatMatrix(1,1,randomX));
94          example.add(new FloatMatrix(1,1,randomY));
            return example;
96      }


98


100 }
```

At the end of each batch of examples the accumulated weight updates are passed to the Network's updateParameters method which implements weight decay regularisation

$$W^{(l)} = W^{(l)} - \alpha\left[\left(\frac{1}{m}\Delta W^{(l)}\right) + \lambda W^{(l)}\right] \tag{14}$$

$$b^{(l)} = b^{(l)} - \alpha\left[\frac{1}{m}\Delta b^{(l)}\right] \tag{15}$$

where α is the learning rate, λ is the weight decay strength. And uses a simple momentum which speeds up the gradient descent:

$$\boldsymbol{v}_{t+1} = \mu\boldsymbol{v}_t - \alpha\nabla E(\boldsymbol{W}) \tag{16}$$

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t + \boldsymbol{v}_{t+1} \tag{17}$$

where μ is the momentum.

```
    void updateParameters(List<FloatMatrix> deltaW_l, List<FloatMatrix> deltaB_l,
2                         float weightDecay, float momementum, float learningRate)
    {
4       int layerNumber=0;
        for(HiddenLayer layer: hiddenLayers) {
6           FloatMatrix currentWeights = layer.getWeightMatrix();
            FloatMatrix currentBiases = layer.getBiasVector();
8           FloatMatrix weightDecayTerm = currentWeights.mul(weightDecay);
            FloatMatrix weightMomentumTerm = previousWeightUpdate.get(layerNumber).mul(momementum);
10          FloatMatrix biasMomentumTerm = previousBiasUpdate.get(layerNumber).mul(momementum);
            FloatMatrix weightUpdates = weightMomentumTerm.sub(
12              deltaW_l.get(layerNumber).add(weightDecayTerm).mul(learningRate)
            );
14          FloatMatrix biasUpdates = biasMomentumTerm.sub(
                deltaB_l.get(layerNumber).mul(learningRate)
```

```
            );
            layer.setWeightMatrix(currentWeights.addi(weightUpdates));
            layer.setBiasVector(currentBiases.addi(biasUpdates));
            previousWeightUpdate.get(layerNumber).copy(weightUpdates);
            previousBiasUpdate.get(layerNumber).copy(biasUpdates);
            ++layerNumber;
        }
    }
```

The trainer runs a training batch, then checks the average error $J(W, b)$ over a validation set. Ideally the validation error should generally decrease after each batch. However at this stage we dont see that:

```
Batch number = 1. Validation error = 0.1880772
Batch number = 2. Validation error = 0.23654647
Batch number = 3. Validation error = 0.23251526
Batch number = 4. Validation error = 0.25341678
Batch number = 5. Validation error = 0.21497807
Batch number = 6. Validation error = 0.24840948
Batch number = 7. Validation error = 0.30167556
Batch number = 8. Validation error = 0.21359357
Batch number = 9. Validation error = 0.33649924
Batch number = 10. Validation error = 0.21925502
Batch number = 11. Validation error = 0.2683097
Batch number = 12. Validation error = 0.24874222
Batch number = 13. Validation error = 0.19440642
Batch number = 14. Validation error = 0.18211803
Batch number = 15. Validation error = 0.20515862
Batch number = 16. Validation error = 0.24847682
Batch number = 17. Validation error = 0.23287967
Batch number = 18. Validation error = 0.26681724
Batch number = 19. Validation error = 0.28868058
Batch number = 20. Validation error = 0.28547257
Batch number = 21. Validation error = 0.26883692
Batch number = 22. Validation error = 0.17752711
Batch number = 23. Validation error = 0.2787137
Batch number = 24. Validation error = 0.201895
Batch number = 25. Validation error = 0.26796734
Batch number = 26. Validation error = 0.22178277
```

Hopefully this is because of the settings of the hyperparameters (learning rate, momentum, number of layers, number of neurons, initial weight settings etc) rather than a problem with the code/algorithm.

Through observing the sizes of the weight updates, the network/layer details I found that often the network output would get stuck at 0 for every input. Thinking about the ReLU activation function $max(0, z)$, if the activation, z is negative then the output of the network is 0, so how can we compute sin of a -ve number? The output can only ever be positive. ReLU has been the activation function of choice in most recent papers on neural networks/convolutional neural networks, usually applied to image recognition where the output is a 1000 odd dimensional vector of class scores where you want a value 0-1 indicating the probability of the image belonging to that class. Perhaps it performs well in this context, but it wont work here, and probably not in my project, which is also a regression task. I replaced this with with a logistic function, which also didn't really work, and then after reading over **[?]** I used what he advises which is $f(z) = 1.75912 \times tanh(\frac{2}{3}z)$

```
    // non linear activation applied on each neuron
    private float activationFunction(float z)
    {
        //return z>0 ? z : 0; //ReLU
        //return 1/(float)(1+Math.exp(-z));//logistic function
        return (float)1.7159*(float)Math.tanh(0.666*z);//tanh
    }

    //f'(z) for back propagate
    private float activationFunctionDash(float z)
    {
```

```
        // return z>0 ? 1 : 0;
        //return activationFunction(z)*(1-activationFunction(z));//logistic
        return 1-(float)Math.pow(activationFunction(z),2);//tanh
    }
```

I also adopted his recomendation for weight initialisation which is to draw the weights from a uniform distribution with mean = 0 and range = $m^{-1/2}$ where m is the number of inputs to that neuron.

```
    Neuron(int numberOfInputs)
    {
        //there are n+1 connections. to n units in prev layer and a bias
        numberOfConnections=numberOfInputs+1;
        weights = FloatMatrix.rand(numberOfConnections);
        weights.muli(1/(float)Math.sqrt(numberOfInputs));
        weights.subi(1/((float)Math.sqrt(numberOfInputs)*2));
    }
```

with these changes and through fideling the other parameters I found that with values

```
    private float weightDecay=(float)0;
    private float momentum=(float)0.9;
    private float learningRate=(float)0.005;
    //layer widths:
    Network net = new Network(1,3,7,11,16,1);
```

I was able to achieve a result. I saw that the validation error decreased, eventually reaching a minimum of 0.048531346 After 1600000 examples before then exploding 200000 examples later . Here is output of the network $h(x)$ against y=sin(x) at that minimum.

```
x: [-3.141593] h(x): [-0.754851] y: [0.000000]
x: [-2.810899] h(x): [-0.753412] y: [-0.324699]
x: [-2.480205] h(x): [-0.750735] y: [-0.614213]
x: [-2.149511] h(x): [-0.745695] y: [-0.837166]
x: [-1.818817] h(x): [-0.736080] y: [-0.969400]
x: [-1.488123] h(x): [-0.717419] y: [-0.996584]
x: [-1.157429] h(x): [-0.680335] y: [-0.915773]
x: [-0.826735] h(x): [-0.604660] y: [-0.735724]
x: [-0.496041] h(x): [-0.450187] y: [-0.475947]
x: [-0.165347] h(x): [-0.167762] y: [-0.164595]
x: [0.165347]  h(x): [0.198991]  y: [0.164594]
x: [0.496041]  h(x): [0.478950]  y: [0.475947]
x: [0.826735]  h(x): [0.619430]  y: [0.735724]
x: [1.157429]  h(x): [0.681451]  y: [0.915773]
x: [1.488123]  h(x): [0.710252]  y: [0.996584]
x: [1.818816]  h(x): [0.724581]  y: [0.969400]
x: [2.149511]  h(x): [0.732008]  y: [0.837167]
x: [2.480205]  h(x): [0.735967]  y: [0.614213]
x: [2.810899]  h(x): [0.738135]  y: [0.324700]
x: [3.141592]  h(x): [0.739353]  y: [0.000000]
After 1600000 examples. Validation error = 0.048531346
```

As you can see it tracks the function well around 0 but it doesn't manage to follow it back down to 0 near $\pm\pi$. Still this is conformation that the algorithm at least working!

to better examine the results, and to practise graphics I have