Program Structures & Algorithms

Rahul Deepak Zore – NUID 001821307 | Sanchit Chavan - NUID 001824408

Final Project Report

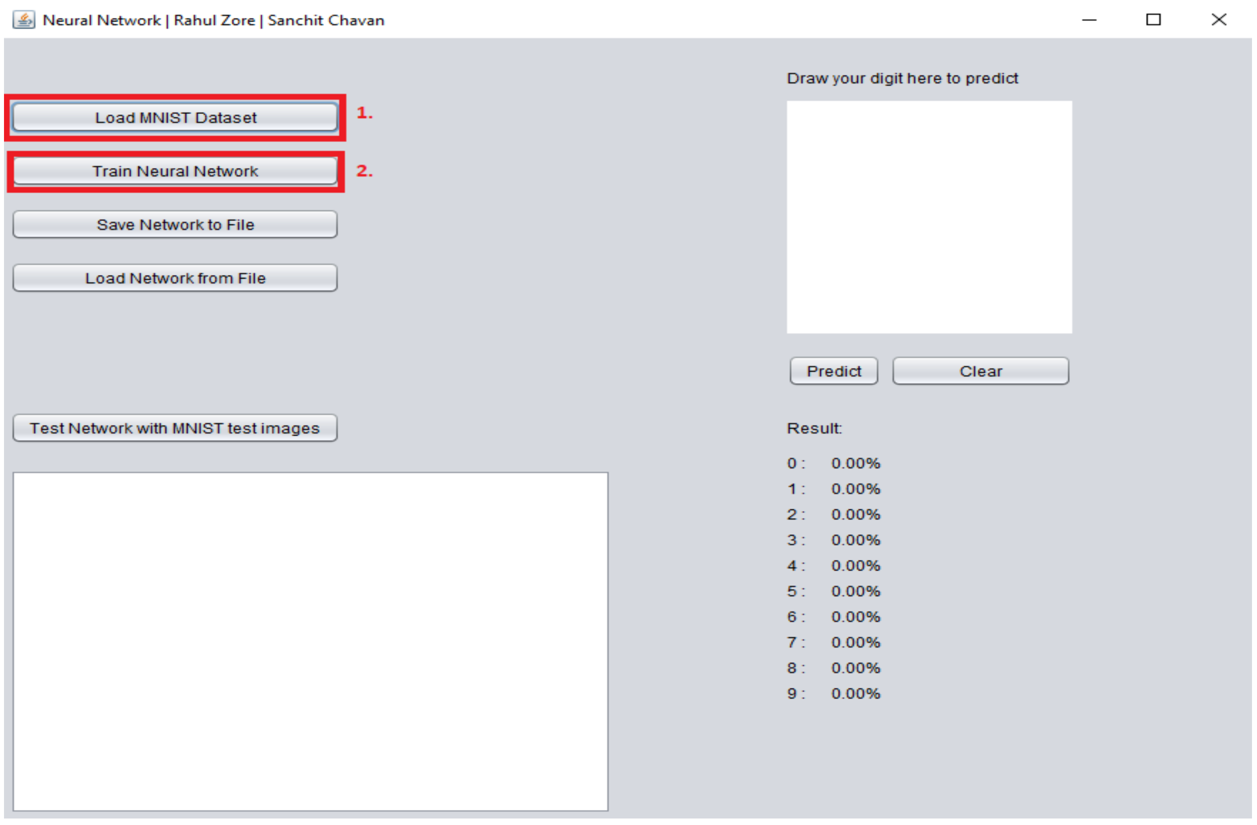
Handwritten Number Recognition Neural Network

Description:

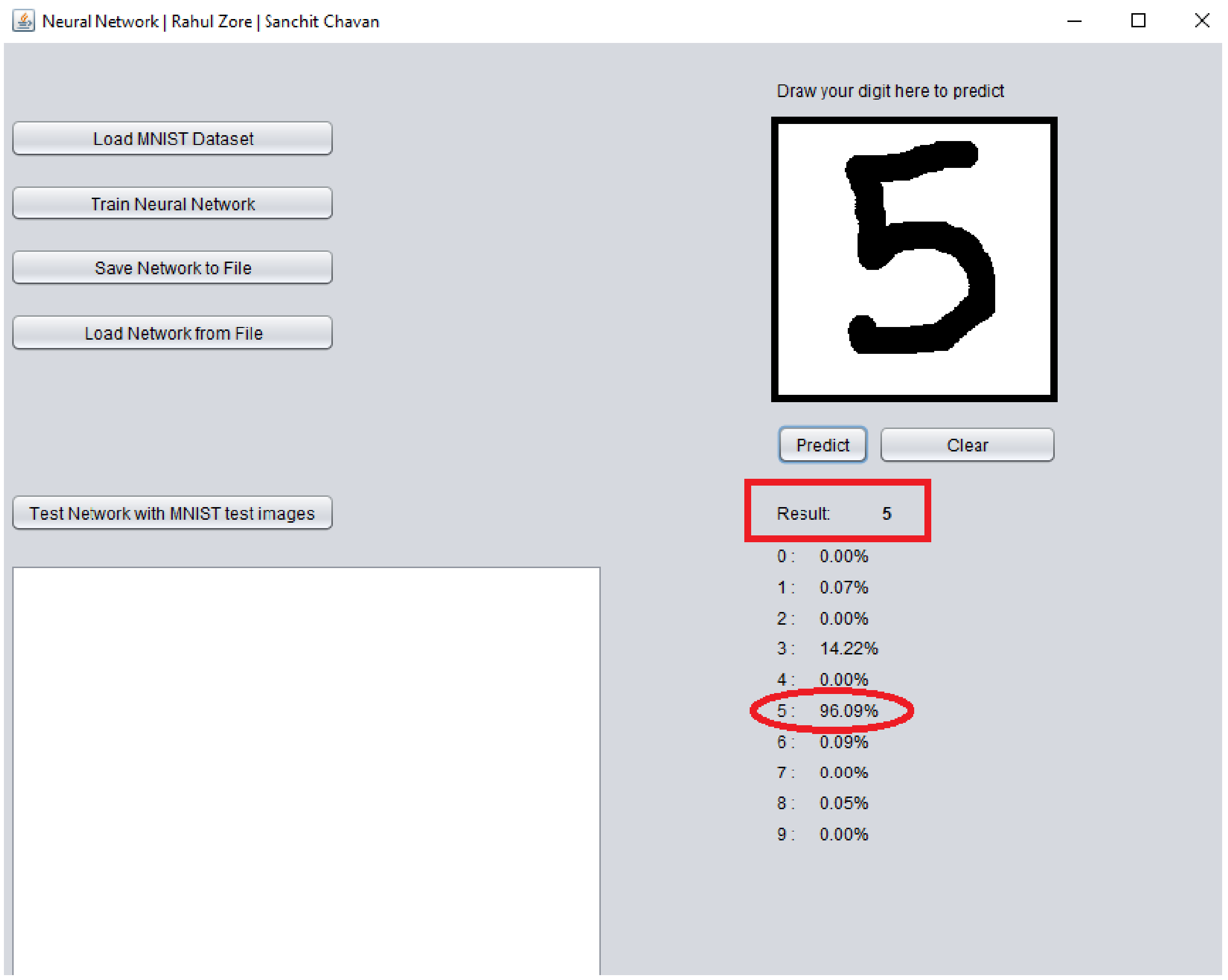
An artificial Neural Network built from scratch that includes a Perceptron that can model any number of neurons and layers to create a Network of Neurons. The network is trained using the MNIST dataset images of handwritten numbers and the designed Neural Network can be used to predict the handwritten number by the user. The network can also be saved after training and it can be loaded again to directly test the handwritten numbers. The network is tested with the MNIST testing dataset as well. Backpropagation method is used to adjust the weights according to the Target result and Sigmoid function is used as the Activation function.

Functioning of the Program:

1. Creating a Network providing Layer object to it and loading the MNIST dataset
2. Training the Network using MNIST training dataset (train-images-idx3-ubyte)



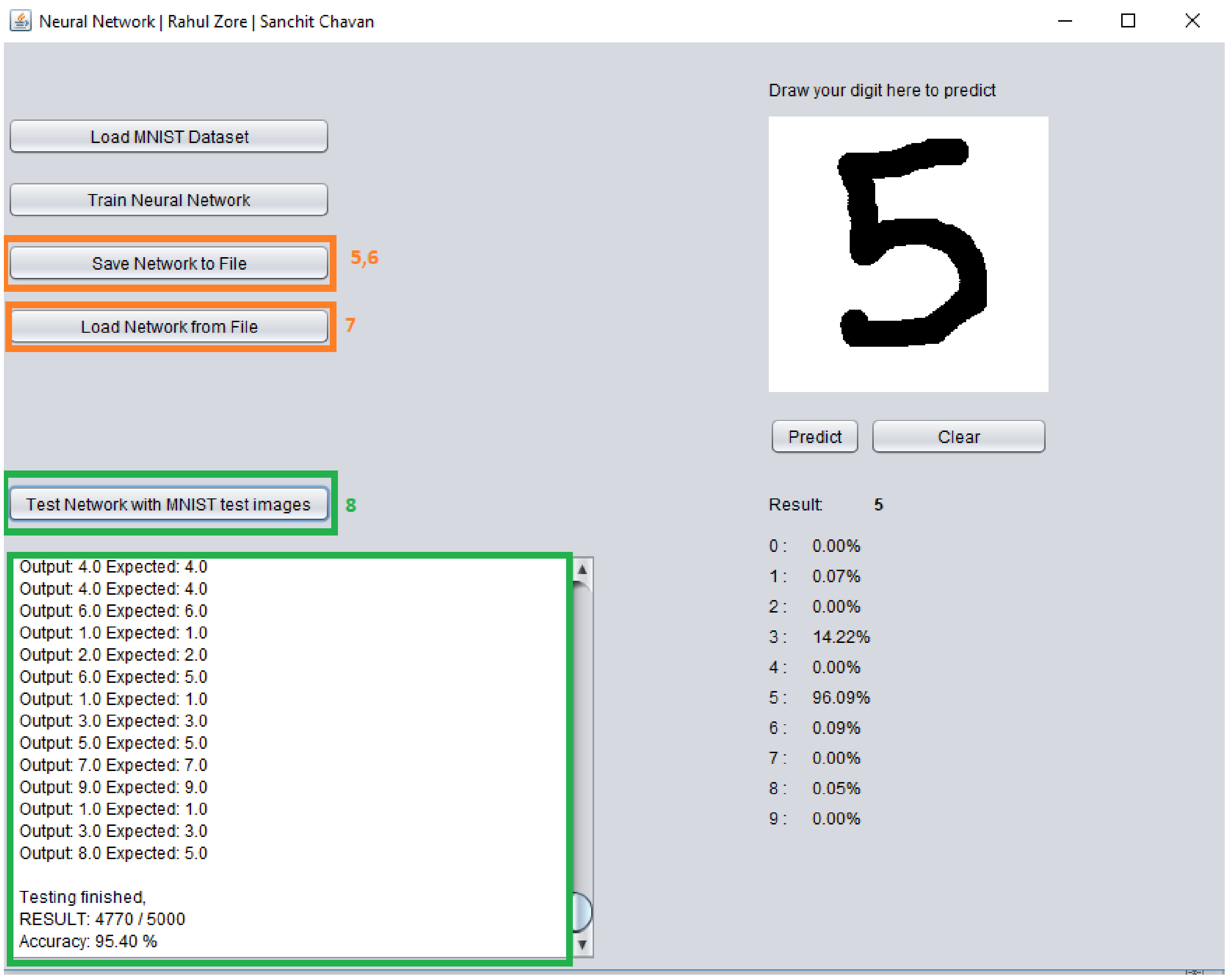
3. Creating binary data array of the images by reading the Handwritten Numbers from the Drawing Panel and comparing them with the MNIST training dataset to calculate the closest match and returning it as the output.
4. Displaying the accuracy in percentage of the handwritten number with respect to all digits from 0-9, comparing it with all the digits. The number with the highest accuracy match is predicted as the result.


Neural Network | Rahul Zore | Sanchit Chavan
Load MNIST Dataset
Train Neural Network
Save Network to File
Load Network from File
Test Network with MNIST test images
Draw your digit here to predict
Predict
Clear

Result: 5

```
0 :   0.00%
1 :   0.07%
2 :   0.00%
3 :   14.22%
4 :   0.00%
5 :   96.09%
6 :   0.09%
7 :   0.00%
8 :   0.05%
9 :   0.00%
```

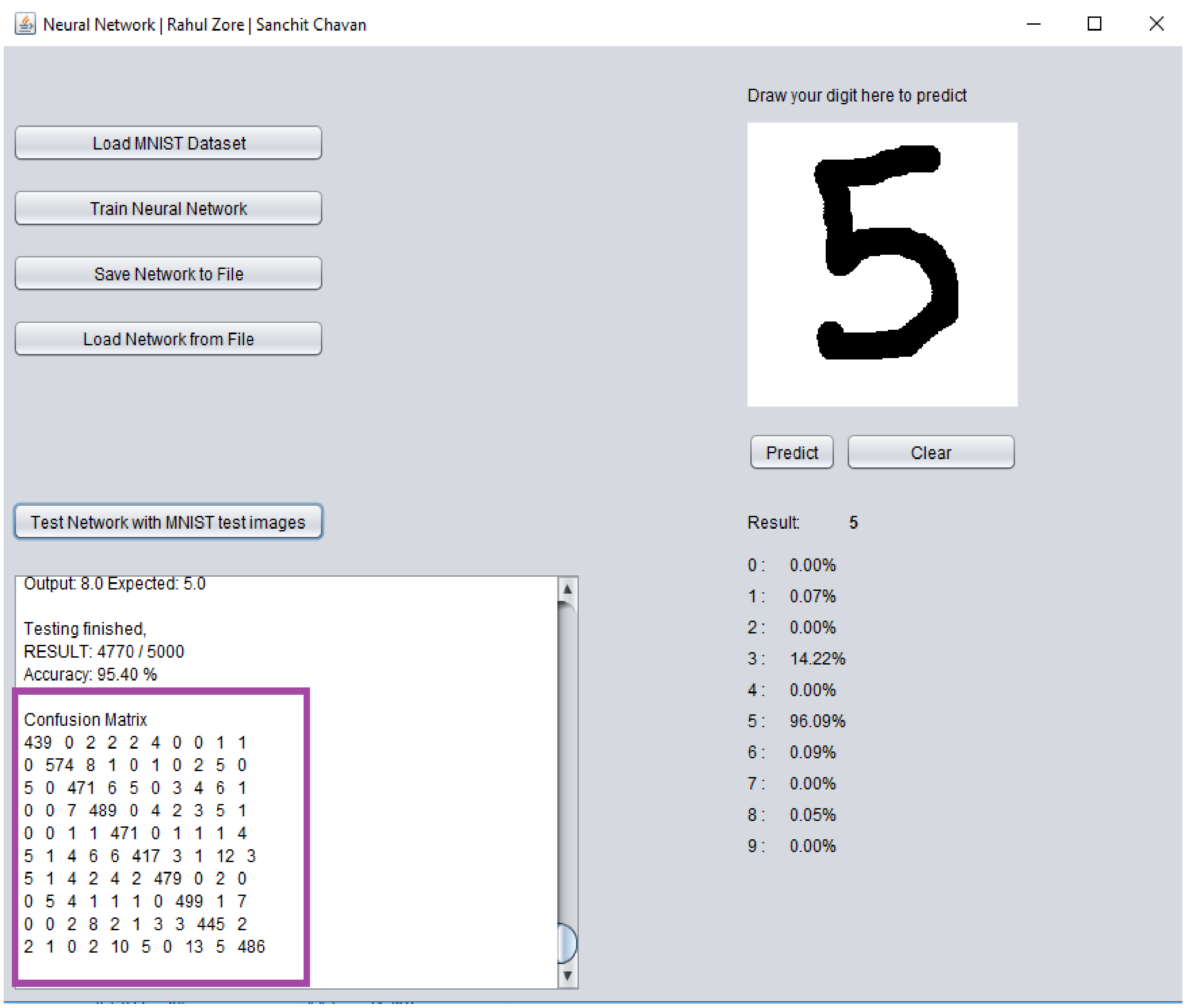
5. Saving the neural network that is created and trained to a text file (perceptron.txt)
6. Saving the training images from the MNIST dataset to trainingSet.txt and saving the test images from the MNIST dataset to testSet.txt
7. Loading the created Network after training it so that it can be loaded again for future testing.
8. Testing the neural network with the MNIST test data and displaying its accuracy with respect to that data.


Neural Network | Rahul Zore | Sanchit Chavan
Load MNIST Dataset
Train Neural Network
Save Network to File
5,6
Load Network from File
7
Test Network with MNIST test images
8
Draw your digit here to predict
Predict
Clear

```
Output: 4.0 Expected: 4.0
Output: 4.0 Expected: 4.0
Output: 6.0 Expected: 6.0
Output: 1.0 Expected: 1.0
Output: 2.0 Expected: 2.0
Output: 6.0 Expected: 5.0
Output: 1.0 Expected: 1.0
Output: 3.0 Expected: 3.0
Output: 5.0 Expected: 5.0
Output: 7.0 Expected: 7.0
Output: 9.0 Expected: 9.0
Output: 1.0 Expected: 1.0
Output: 3.0 Expected: 3.0
Output: 8.0 Expected: 5.0

Testing finished,
RESULT: 4770 / 5000
Accuracy: 95.40 %
```

```
Result    5
0 :   0.00%
1 :   0.07%
2 :   0.00%
3 :   14.22%
4 :   0.00%
5 :   96.09%
6 :   0.09%
7 :   0.00%
8 :   0.05%
9 :   0.00%
```

Confusion Matrix (Expected Output vs Target Output):


Neural Network | Rahul Zore | Sanchit Chavan
Load MNIST Dataset
Train Neural Network
Save Network to File
Load Network from File
Test Network with MNIST test images
Draw your digit here to predict
Predict
Clear

```
Output: 8.0 Expected: 5.0

Testing finished,
RESULT: 4770 / 5000
Accuracy: 95.40 %

Confusion Matrix
439  0  2  2  2  4  0  0  1  1
0  574  8  1  0  1  0  2  5  0
5  0  471  6  5  0  3  4  6  1
0  0  7  489  0  4  2  3  5  1
0  0  1  1  471  0  1  1  1  4
5  1  4  6  6  417  3  1  12  3
5  1  4  2  4  2  479  0  2  0
0  5  4  1  1  1  0  499  1  7
0  0  2  8  2  1  3  3  445  2
2  1  0  2  10  5  0  13  5  486
```

```
Result:     5
0 :   0.00%
1 :   0.07%
2 :   0.00%
3 :   14.22%
4 :   0.00%
5 :   96.09%
6 :   0.09%
7 :   0.00%
8 :   0.05%
9 :   0.00%
```

Neural Network Explanation along with code snippets:

1. Activation Function (Sigmoid Function):

$$\varepsilon(x) = \frac{1}{1 + e^{-x}}$$

2. Calculating the output:
   a. Calculate the sum:

$$x_n^l = b_n^l + \sum_k w_{kn}^l \, O_k^{l-1}$$

k = amount of previous neurons

$l = layer$

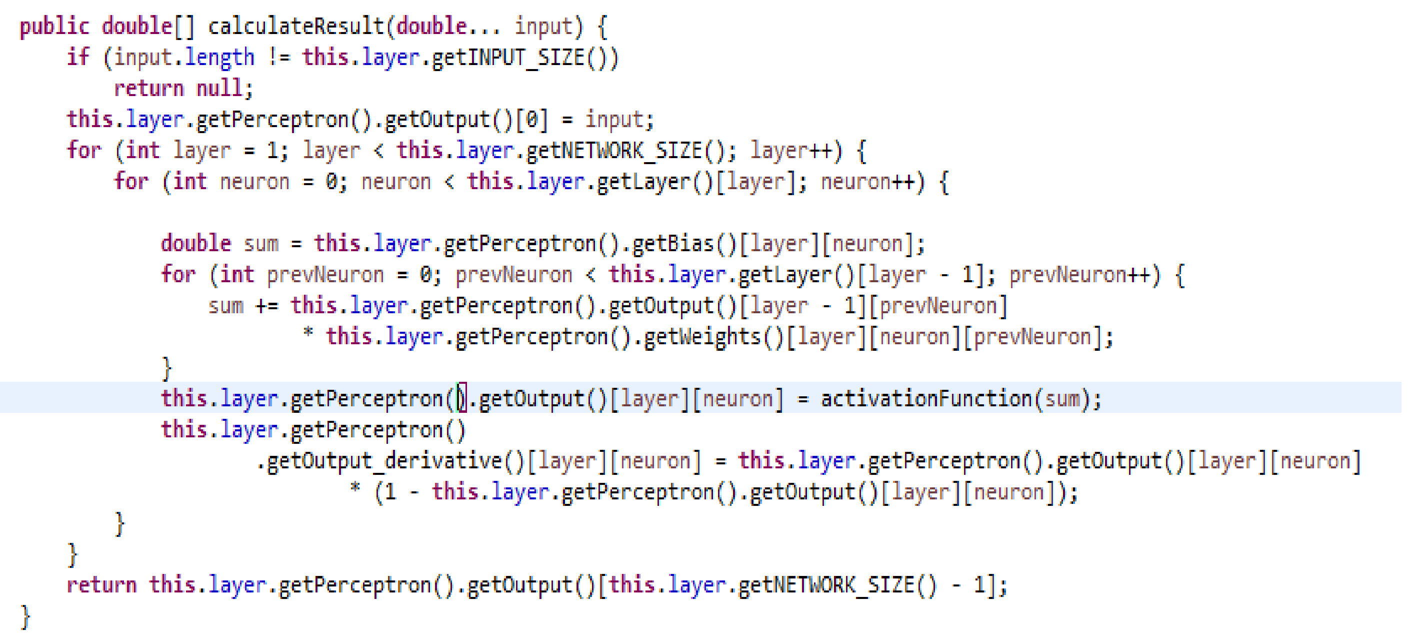
$p = previous\ neuron$

$n = neuron\ in\ layer\ l$

   b. The desired output is the Sigmoid Value of the sum:

$$O_n^l = \varepsilon\left(x_n^l\right)$$

Code Snippet:



```
public double[] calculateResult(double... input) {
    if (input.length != this.layer.getINPUT_SIZE())
        return null;
    this.layer.getPerceptron().getOutput()[0] = input;
    for (int layer = 1; layer < this.layer.getNETWORK_SIZE(); layer++) {
        for (int neuron = 0; neuron < this.layer.getLayer()[layer]; neuron++) {

            double sum = this.layer.getPerceptron().getBias()[layer][neuron];
            for (int prevNeuron = 0; prevNeuron < this.layer.getLayer()[layer - 1]; prevNeuron++) {
                sum += this.layer.getPerceptron().getOutput()[layer - 1][prevNeuron]
                       * this.layer.getPerceptron().getWeights()[layer][neuron][prevNeuron];
            }
            this.layer.getPerceptron().getOutput()[layer][neuron] = activationFunction(sum);
            this.layer.getPerceptron()
                    .getOutput_derivative()[layer][neuron] = this.layer.getPerceptron().getOutput()[layer][neuron]
                            * (1 - this.layer.getPerceptron().getOutput()[layer][neuron]);
        }
    }
    return this.layer.getPerceptron().getOutput()[this.layer.getNETWORK_SIZE() - 1];
}
```

Calculating the error to update the weights:

$$E = \tfrac{1}{2}(t - y)^2,$$

where

E is the squared error,

t is the target output for a training sample, and

y is the actual output of the output neuron.

Calculation delta weight (i.e. change in weight) to update the weights to train the network:

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \mathrm{net}_j} = \begin{cases} (o_j - t_j)o_j(1 - o_j) & \text{if } j \text{ is an output neuron,} \\ (\sum_{\ell \in L} \delta_\ell w_{j\ell})o_j(1 - o_j) & \text{if } j \text{ is an inner neuron.} \end{cases}$$

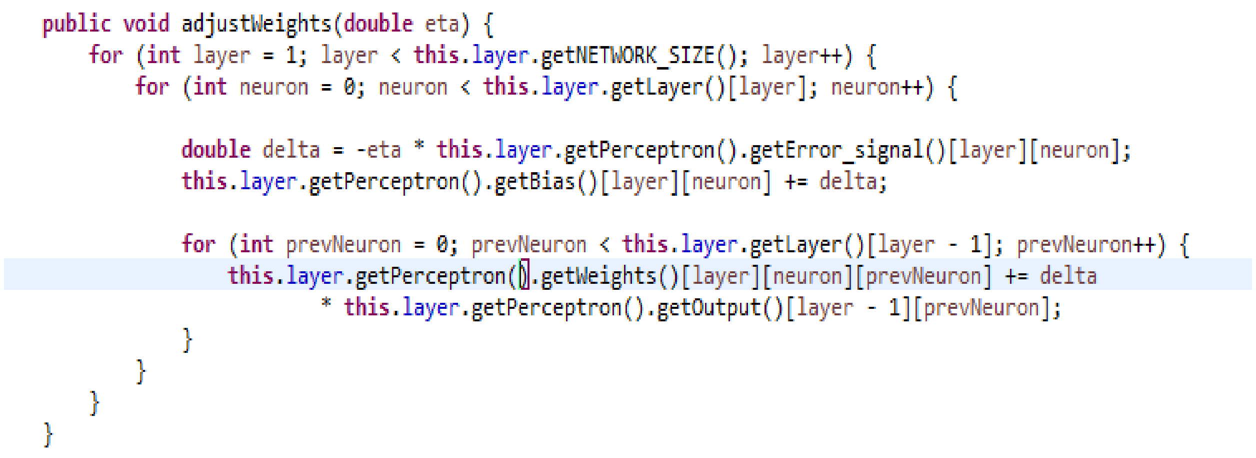
$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = \begin{cases} -\eta o_i (o_j - t_j)o_j(1 - o_j) & \text{if } j \text{ is an output neuron,} \\ -\eta o_i \left(\sum_{\ell \in L} \delta_\ell w_{j\ell}\right) o_j(1 - o_j) & \text{if } j \text{ is an inner neuron.} \end{cases}$$

Where η = learning rate which should be greater than 0.

Back Propagation Code Snippets:

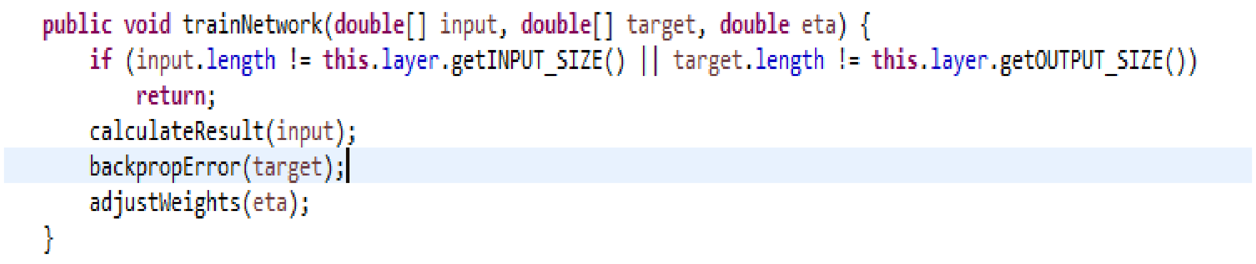
```
public void backpropError(double[] target) {
    for (int neuron = 0; neuron < this.layer.getLayer()[this.layer.getNETWORK_SIZE() - 1]; neuron++) {
        this.layer.getPerceptron().getError_signal()[this.layer.getNETWORK_SIZE()
                - 1][neuron] = (this.layer.getPerceptron().getOutput()[this.layer.getNETWORK_SIZE() - 1][neuron]
                        - target[neuron])
                        * this.layer.getPerceptron().getOutput_derivative()[this.layer.getNETWORK_SIZE()
                                - 1][neuron];
    }
    for (int layer = this.layer.getNETWORK_SIZE() - 2; layer > 0; layer--) {
        for (int neuron = 0; neuron < this.layer.getLayer()[layer]; neuron++) {
            double sum = 0;
            for (int nextNeuron = 0; nextNeuron < this.layer.getLayer()[layer + 1]; nextNeuron++) {
                sum += this.layer.getPerceptron().getWeights()[layer + 1][nextNeuron][neuron]
                        * this.layer.getPerceptron().getError_signal()[layer + 1][nextNeuron];
            }
            this.layer.getPerceptron().getError_signal()[layer][neuron] = sum
                    * this.layer.getPerceptron().getOutput_derivative()[layer][neuron];
        }
    }
}
```

Weight Adjustment Code Snippet:



```java
public void adjustWeights(double eta) {
    for (int layer = 1; layer < this.layer.getNETWORK_SIZE(); layer++) {
        for (int neuron = 0; neuron < this.layer.getLayer()[layer]; neuron++) {

            double delta = -eta * this.layer.getPerceptron().getError_signal()[layer][neuron];
            this.layer.getPerceptron().getBias()[layer][neuron] += delta;

            for (int prevNeuron = 0; prevNeuron < this.layer.getLayer()[layer - 1]; prevNeuron++) {
                this.layer.getPerceptron().getWeights()[layer][neuron][prevNeuron] += delta
                        * this.layer.getPerceptron().getOutput()[layer - 1][prevNeuron];
            }
        }
    }
}
```

Training Network Code Snippet:



```java
public void trainNetwork(double[] input, double[] target, double eta) {
    if (input.length != this.layer.getINPUT_SIZE() || target.length != this.layer.getOUTPUT_SIZE())
        return;
    calculateResult(input);
    backpropError(target);
    adjustWeights(eta);
}
```

Important Classes:

1. Perceptron.java

This class contains the initial weights, bias and learning rate which is the core requirement of Neural Network.

2. Layer.java

This class contains an array which will be used to hold the hidden layers. There can be any number of hidden layers defined in the Neural Network through this array.

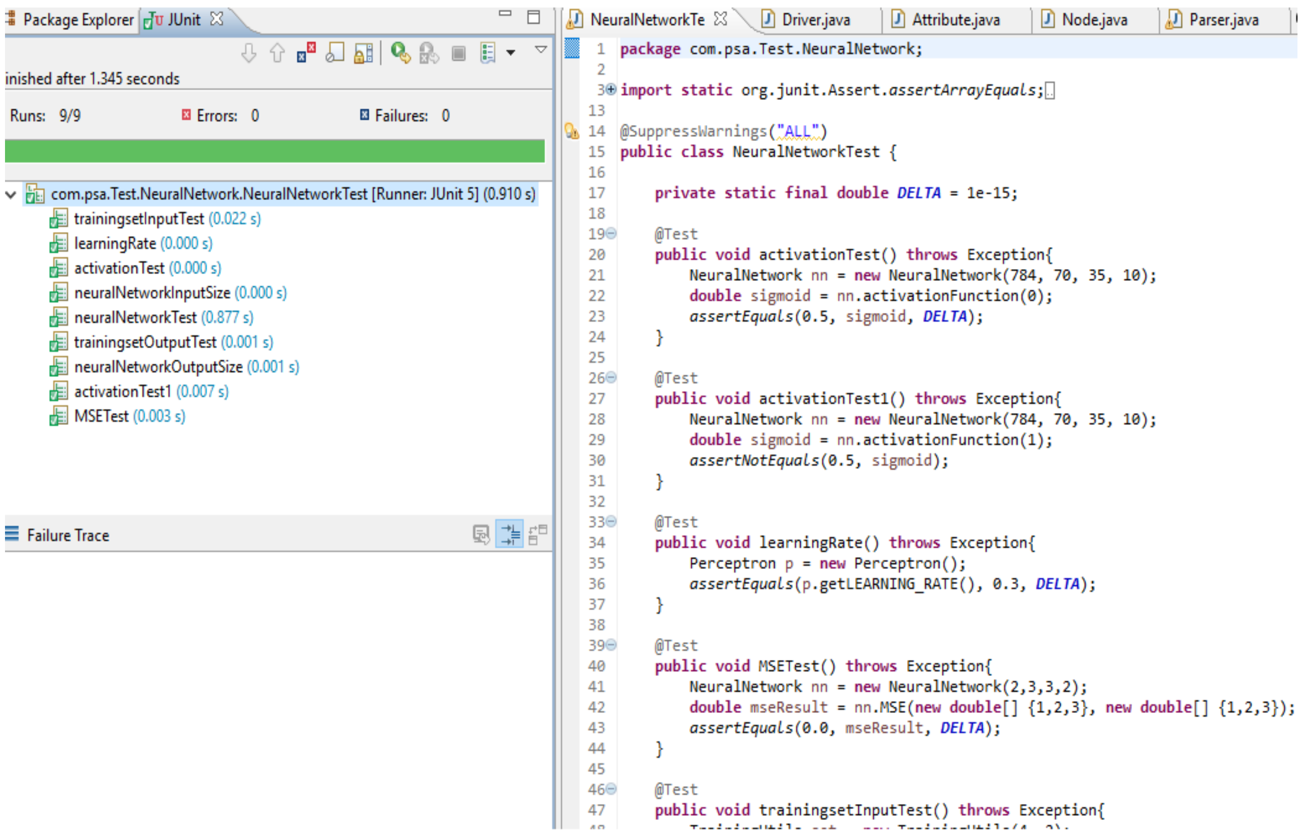
3. NeuralNetwork.java

This is our multi-layered Perceptron aka Neural Network which contains all the neural network functionalities such as Adjusting Weights, Calculating Back Propagation, Training the Neural Network, Calculating the final output.

4. Utils.java

This is a helper class which contains functions to get random values that are used to create random arrays for weights and calculating highest index value of the output array.

Screenshot of Test Cases Results:



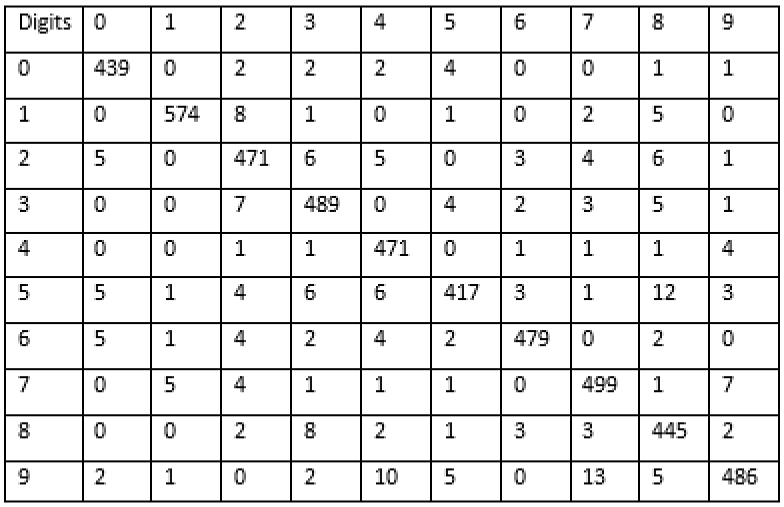
```
inished after 1.345 seconds
Runs: 9/9        Errors: 0        Failures: 0

com.psa.Test.NeuralNetwork.NeuralNetworkTest [Runner: JUnit 5] (0.910 s)
    trainingsetInputTest (0.022 s)
    learningRate (0.000 s)
    activationTest (0.000 s)
    neuralNetworkInputSize (0.000 s)
    neuralNetworkTest (0.877 s)
    trainingsetOutputTest (0.001 s)
    neuralNetworkOutputSize (0.001 s)
    activationTest1 (0.007 s)
    MSETest (0.003 s)
```

```java
package com.psa.Test.NeuralNetwork;

import static org.junit.Assert.assertArrayEquals;

@SuppressWarnings("ALL")
public class NeuralNetworkTest {

    private static final double DELTA = 1e-15;

    @Test
    public void activationTest() throws Exception{
        NeuralNetwork nn = new NeuralNetwork(784, 70, 35, 10);
        double sigmoid = nn.activationFunction(0);
        assertEquals(0.5, sigmoid, DELTA);
    }

    @Test
    public void activationTest1() throws Exception{
        NeuralNetwork nn = new NeuralNetwork(784, 70, 35, 10);
        double sigmoid = nn.activationFunction(1);
        assertNotEquals(0.5, sigmoid);
    }

    @Test
    public void learningRate() throws Exception{
        Perceptron p = new Perceptron();
        assertEquals(p.getLEARNING_RATE(), 0.3, DELTA);
    }

    @Test
    public void MSETest() throws Exception{
        NeuralNetwork nn = new NeuralNetwork(2,3,3,2);
        double mseResult = nn.MSE(new double[] {1,2,3}, new double[] {1,2,3});
        assertEquals(0.0, mseResult, DELTA);
    }

    @Test
    public void trainingsetInputTest() throws Exception{
```

Conclusion:

Testing the Neural Network using MNIST test dataset gives the following results:

*Confusion Matrix:



Digits	0	1	2	3	4	5	6	7	8	9
0	439	0	2	2	2	4	0	0	1	1
1	0	574	8	1	0	1	0	2	5	0
2	5	0	471	6	5	0	3	4	6	1
3	0	0	7	489	0	4	2	3	5	1
4	0	0	1	1	471	0	1	1	1	4
5	5	1	4	6	6	417	3	1	12	3
6	5	1	4	2	4	2	479	0	2	0
7	0	5	4	1	1	1	0	499	1	7
8	0	0	2	8	2	1	3	3	445	2
9	2	1	0	2	10	5	0	13	5	486

* RESULT: 4770 / 5000

* Accuracy: 95.40 %

References:

1. **http://neuralnetworksanddeeplearning.com/**
2. **https://en.wikipedia.org/wiki/Backpropagation**
3. **https://www.youtube.com/playlist?list=PLRqwX-V7Uu6Y7MdSCaIfsxc561QI0U0Tb**