

**INFO 6205 Program Structures and Algorithms**

**IMPLEMENTATION OF A NEURAL NETWORK:  
IMAGE CLASSIFIER**

**A Project By:**

**Venkateshkumar Sivakumar**

**Ashwin Lakshman**

**ABBREVIATIONS:**

ANN	Artificial Neural Network
BIST	Built-in Self-Test
LR	Learning Rate
MSE	Mean Squared Error

**ABSTRACT:**

Artificial Neural Networks (ANN) are non-linear statistical data modeling tools, often used to model complex relationships between inputs and outputs or to find patterns in data.

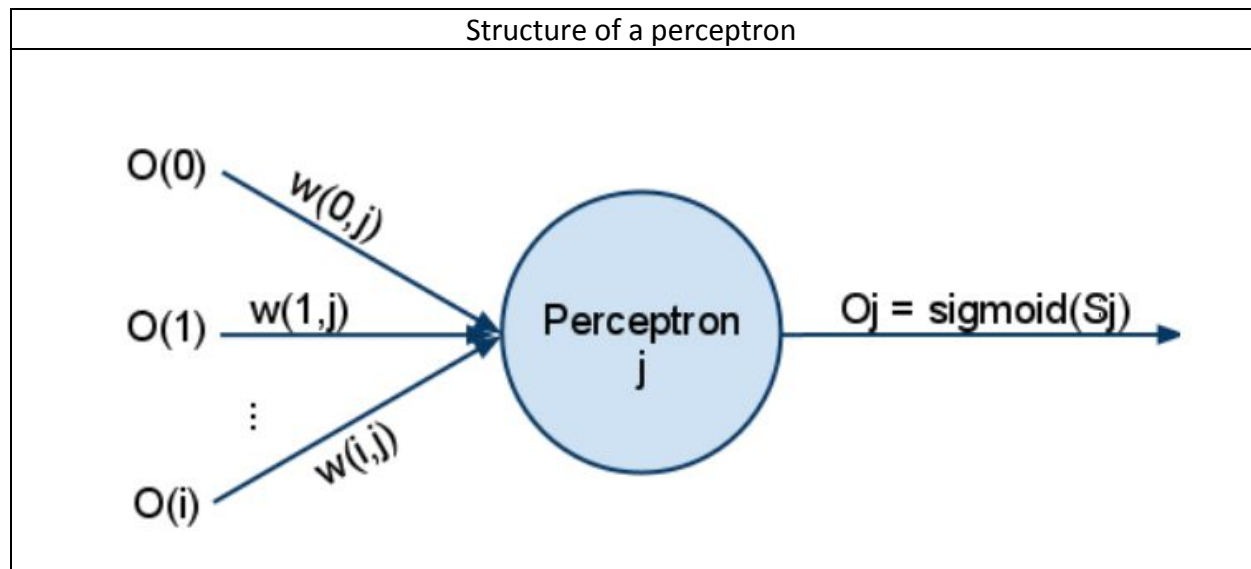
In this project, a three-layer ANN is implemented entirely with 32-bit single precision floating point arithmetic to guarantee flexibility and accuracy for its wide range of applications. This ANN allows the reconfiguration of number of perceptrons per layer as well as supervised learning through back propagation. The Built-in Self-Test(BIST) measures the quality of learning from the square of the mean error.

The ANN we developed is a fully functional pattern recognizer which is used to identify the presence of a person in any provided input image. This classifier is used to locate the silhouette of a person in an image and the most probable case is calculated.

**ALGORITHM:**

An ANN consists of Perceptrons( or nodes) organized into three layers: Input, Hidden and Output Layer. A perceptron in a layer will be connected to each of the perceptrons in the previous layer, with a weightage for each of the connections.

Since each of the perceptrons is a nonlinear function which acts on the outputs of the weighted sums from other perceptrons, and by adjusting the weights on each connection, a smooth function can be estimated.



### FEED FORWARD NETWORK:

The feed forward network of a perceptron works as follows

- a. The weighted sum of all the inputs of that perceptron are calculated using the formula:  

$$\text{sum} = \sum \text{weight}_{ij} * \text{input}_i$$
, where  $i$  is 0 to  $N$ .
- b. The weighted sum is then passed through an Activation function. In this case, a Sigmoid function was implemented as follows:  

$$O_j = 1 / (1 + e^{-S_j})$$
- c. The output from the sigmoid function is then fed to the next layer where it will be used to calculate the weighted sum for each of the perceptrons. The perceptron output for the output layer is then regarded as the neural network's output.

## BACK PROPAGATION:

The back propagation process for the perceptron is as follows:

- a. First, the error output of the perceptron is calculated. In case the perceptron is located in the output layer, the error is the difference between the output and the learning target ( $t_{ij}$ ).

$$e_j = t_j - o_j$$

If the perceptron is not in the output layer, the error is calculated from a weighted sum of forward layer's errors ( $e_o$ ) and the derivative of the sigmoid function.

$$e_j = o_j (1 - o_j) * \sum w_{jo} e_o$$

- b. The change in weight of each perceptron is then calculated from the error of the perceptron and the previous values of the connections.

A learning rate is applied so that the overall error can be slowly but eventually minimized. The learning rate (LR) is currently assigned the value of 0.1.

$$\Delta w_{ij} = -LR o_j e_j$$

- c. The new weights are then saved for each of the inputs of the perceptron.

## DESIGN:

The generic ANN is designed such that the number of perceptrons per layer can be easily reconfigured to meet the application requirements.

While performing training and testing, the training images are randomly picked from the two categories of inputs.

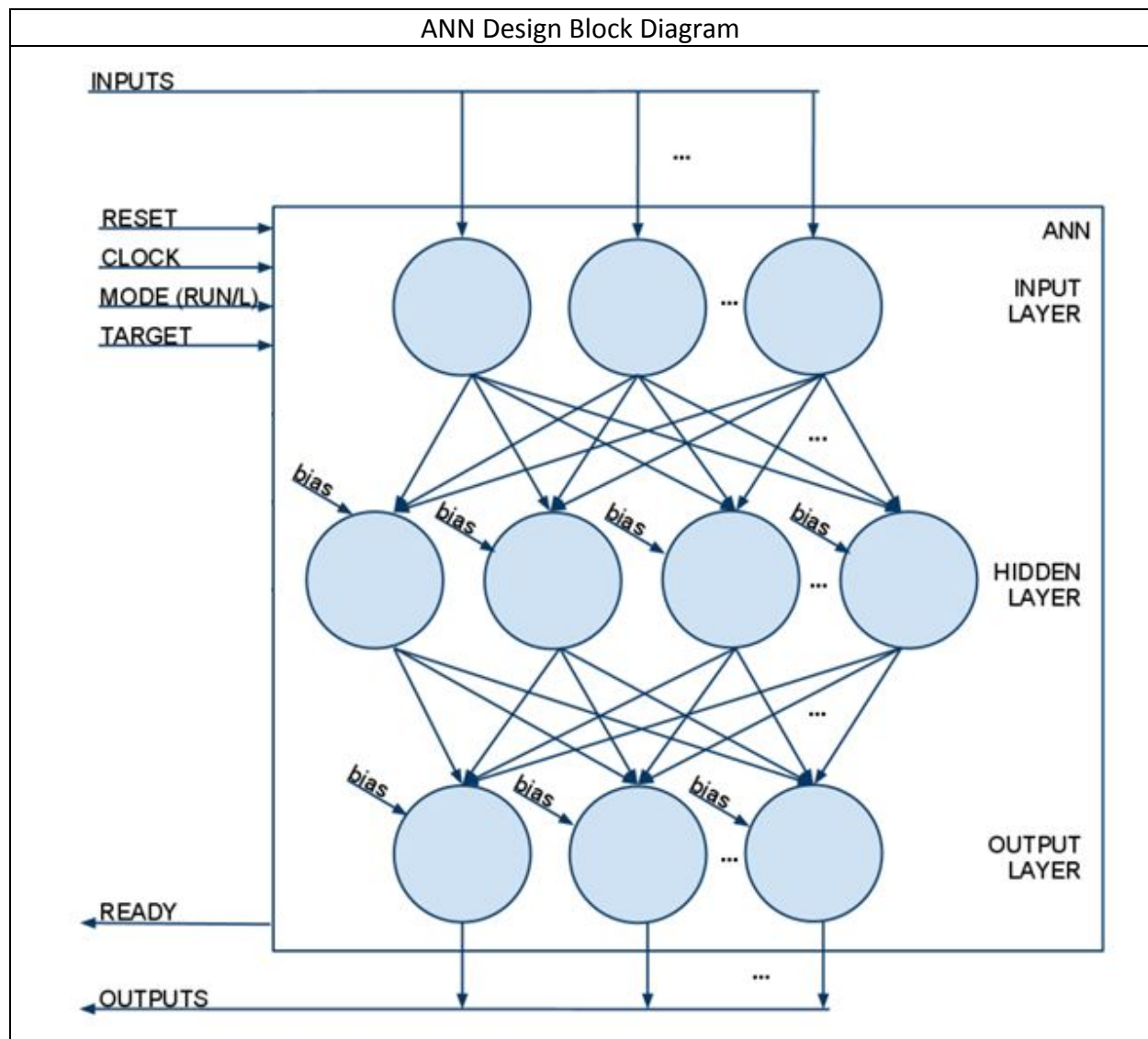
A size of  $(N_I + 1) * N_H + (N_H + 1) * N_O$  array is needed for the storage of all weights, since each individual weights are assigned to each connection, where,

$N_I$  - the number of perceptrons in the input layer,

$N_H$  - the number of perceptrons in the hidden layer,

$N_O$  - the number of perceptrons in the output layer.

In this ANN, each perceptron also receives a Bias Input which holds the constant value 1. The weight of this input is then updated during back propagation and thus it enhances the accuracy and effectiveness of the ANN.



When the MODE is set at RUN, the ANN leaves idle state and enters the RUN state where it executes the following sequential steps:

- a. The weighted sum and output from the sigmoid function for each perceptron in the Hidden layer is calculated.
- b. The weighted sum and output from the sigmoid function for each perceptron in the Output layer is calculated

When the Mode is set at LEARN, the ANN leaves idle state and enters the LEARN state where it executes the following sequential steps:

- a. The weighted sum and output from the sigmoid function for each perceptron in the Hidden layer is calculated.
- b. The weighted sum and output from the sigmoid function for each perceptron in the Output layer is calculated.
- c. The error for each perceptron is calculated in the output layer by subtracting the output of that perceptron from the Target value.
- d. The Mean Squared Error (MSE) of the output layer is calculated from the errors of each perceptron.
- e. The derivative of the sigmoid function multiplied by the error for each perceptron in the output layer is calculated and assigned as Delta.
- f. Delta Weight for each inputs of each perceptrons in the output layer are calculated and is used to update the weight of each connection.
- g. The error for each perceptron in the hidden layer is calculated using the Delta values for each perceptron in the output layer.
- h. The derivative of the sigmoid function multiplied by the error of each perceptron in the hidden layer is calculated as Delta.
- i. Delta Weight for each input of each perceptrons in the hidden layer are calculated and is used to update the weight of each connection.

#### PSEUDO CODE:

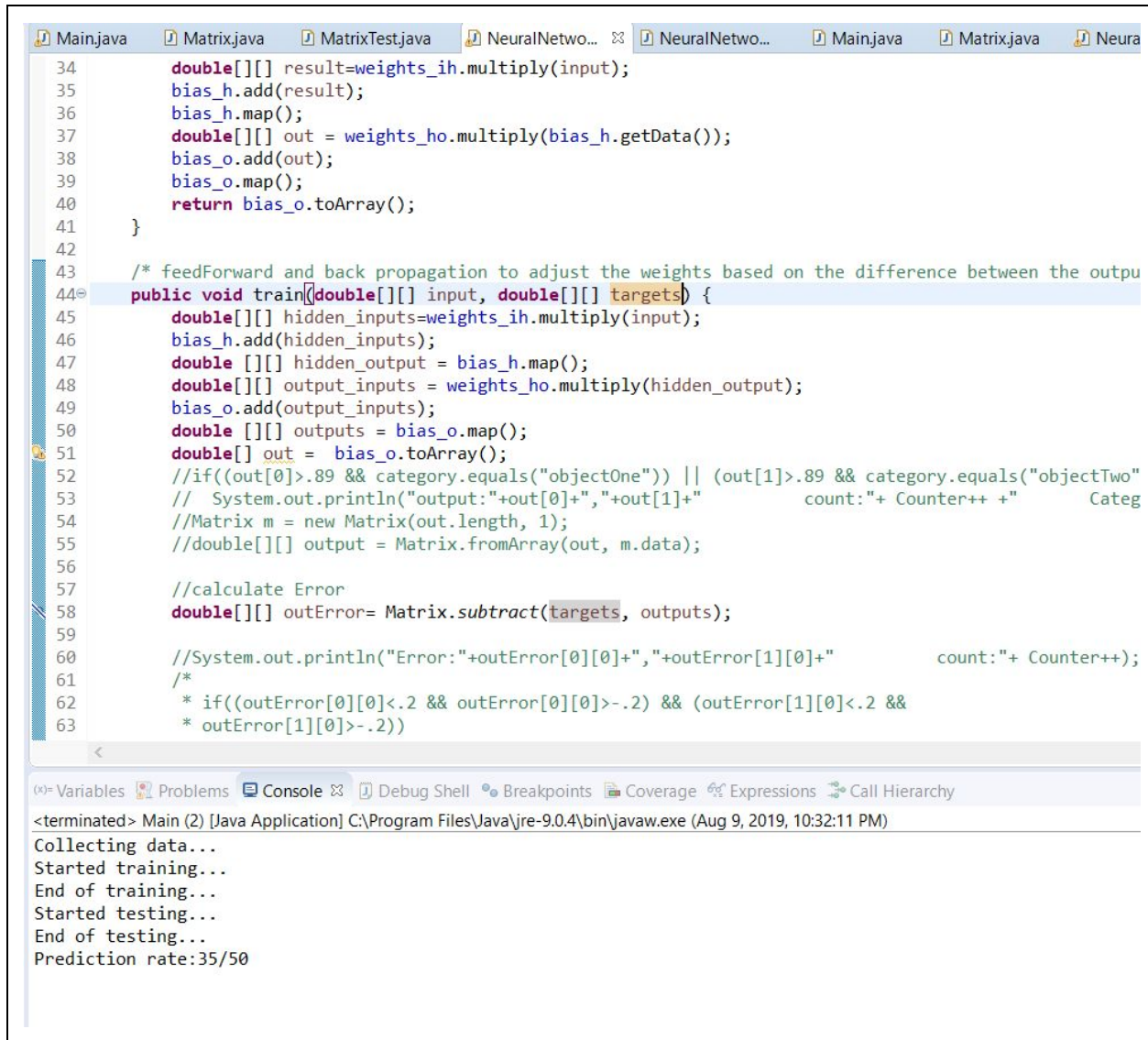
```
function train(inputs, target)
{
    hidden_inputs = ( weights_input_to_hidden * inputs ) + bias_hidden
    hidden_output = calculate sigmoid function for hidden_inputs (  $O_j = 1 / (1 + e^{-s_j})$  )
    output_inputs = ( weights_hidden_to_output * hidden_inputs ) + bias_output
    calculate sigmoid function for output_inputs (  $O_j = 1 / (1 + e^{-s_j})$  )
    output = calculate sigmoid value of output_inputs

    //Back Propagation
    output_error = target - output
    hidden_error = transpose weights of hidden_outputs * output_error
    gradients = desigmoid value of output * output_error * learning_rate
}
```

```
hidden_gradients = hidden_output * hidden_error * learning_rate  
delta = ( gradients * transpose of hidden_output ) + weights of hidden_output  
bias_output + gradients  
hidden_delta = ( hidden_gradients * transpose of inputs ) + weights of hidden_inputs  
bias_hidden + hidden_gradients  
}
```

```
function feed_forward( inputs )  
{  
    hidden_inputs = ( weights_input_to_hidden * inputs ) + bias_hidden  
    hidden_output = calculate sigmoid function for hidden_inputs (  $O_j = 1 / (1 + e^{-s_j})$  )  
    output_inputs = ( weights_hidden_to_output * hidden_inputs ) + bias_output  
    calculate sigmoid function for output_inputs (  $O_j = 1 / (1 + e^{-s_j})$  )  
    output = calculate sigmoid value of output_inputs  
}
```

## OUTPUT:



```
34     double[][] result=weights_ih.multiply(input);
35     bias_h.add(result);
36     bias_h.map();
37     double[][] out = weights_ho.multiply(bias_h.getData());
38     bias_o.add(out);
39     bias_o.map();
40     return bias_o.toArray();
41 }
42
43 /* feedForward and back propagation to adjust the weights based on the difference between the output
44 public void train(double[][] input, double[][] targets) {
45     double[][] hidden_inputs=weights_ih.multiply(input);
46     bias_h.add(hidden_inputs);
47     double [][] hidden_output = bias_h.map();
48     double[][] output_inputs = weights_ho.multiply(hidden_output);
49     bias_o.add(output_inputs);
50     double [][] outputs = bias_o.map();
51     double[] out = bias_o.toArray();
52     //if((out[0]>.89 && category.equals("objectOne")) || (out[1]>.89 && category.equals("objectTwo")
53     // System.out.println("output:"+out[0]+","+out[1]+ "count:"+ Counter++ +" Categ
54     //Matrix m = new Matrix(out.length, 1);
55     //double[][] output = Matrix.fromArray(out, m.data);
56
57     //calculate Error
58     double[][] outError= Matrix.subtract(targets, outputs);
59
60     //System.out.println("Error:"+outError[0][0]+","+outError[1][0]+ "count:"+ Counter++);
61     /*
62     * if((outError[0][0]<.2 && outError[0][0]>-.2) && (outError[1][0]<.2 &&
63     * outError[1][0]>-.2))
```

<

Variables Problems Console Debug Shell Breakpoints Coverage Expressions Call Hierarchy

<terminated> Main (2) [Java Application] C:\Program Files\Java\jre-9.0.4\bin\javaw.exe (Aug 9, 2019, 10:32:11 PM)

Collecting data...  
Started training...  
End of training...  
Started testing...  
End of testing...  
Prediction rate:35/50



## TEST CASES:

### MatrixTest

EclipseWork - Artificial/src/test/java/Artificial/Artificial/MatrixTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Debug Project Explorer JUnit

Finished after 0.022 seconds

Runs: 7/7 Errors: 0 Failures: 0

Artificial.Artificial.MatrixTest [Runner: JUnit 4] (0.000 s)

- testDSigmoid (0.000 s)
- testDiv (0.000 s)
- testSubtract (0.000 s)
- testFromArray (0.000 s)
- testDotMultiply (0.000 s)
- testMultiply (0.000 s)
- testTranspose (0.000 s)

Failure Trace

```
103
104 @Test
105 public void testFromArray() {
106     double[] in1 = new double[] {1.0,2.0} ;
107
108     double [][] result = Matrix.fromArray(in1,new double[2][2]);
109     assertEquals(1.0, result[0][0],0);
110     assertEquals(2.0, result[1][0],0);
111 }
112
113
114 @Test
115 public void testDiv() {
116     double[][] in1 = new double[2][2] ;
117     for(int i=0;i<2;i++) {
118         for(int j=0;j<2;j++) {
119             in1[i][j]=255;
120         }
121     }
122
123     double [][] result = Matrix.divMap(in1);
124     assertEquals(1.0, result[0][0],0);
125     assertEquals(1.0, result[1][0],0);
126     assertEquals(1.0, result[0][1],0);
127     assertEquals(1.0, result[1][1],0);
128 }
129
130
131 }
132
```

## NeuralNetwork

EclipseWork - Artificial/src/test/java/Artificial/Artificial/NeuralNetworkTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Debug Project Explorer JUnit

Finished after 0.016 seconds

Runs: 2/2 Errors: 0 Failures: 0

Artificial.Artificial.NeuralNetworkTest [Runner: JUnit 4] (0.000 s)

- backProbTest (0.000 s)
- feedForwardTest (0.000 s)

Failure Trace

```
93 }
94 }
95 weights_ho.setData(who);
96 double[][] bh = new double[2][1];
97 for(int i=0;i<2;i++) {
98     for(int j=0;j<1;j++) {
99         bh[i][j] = .1;
100     }
101 }
102 bias_h.setData(bh);
103 double[][] bo = new double[1][1];
104 for(int i=0;i<1;i++) {
105     for(int j=0;j<1;j++) {
106         bo[i][j] = 1;
107     }
108 }
109 bias_o.setData(bo);
110 nn.setBias_h(bias_h);
111 nn.setBias_o(bias_o);
112 nn.setWeights_ho(weights_ho);
113 nn.setWeights_ih(weights_ih);
114 nn.train(val, target);
115 double result [][]= nn.getWeights_ih().getData();
116 assertEquals(0.5001, result[0][0],.1);
117 assertEquals(0.5001, result[0][0],.1);
118 assertEquals(0.5001, result[0][0],.1);
119 assertEquals(0.5001, result[0][0],.1);
120
121 }
122 }
```