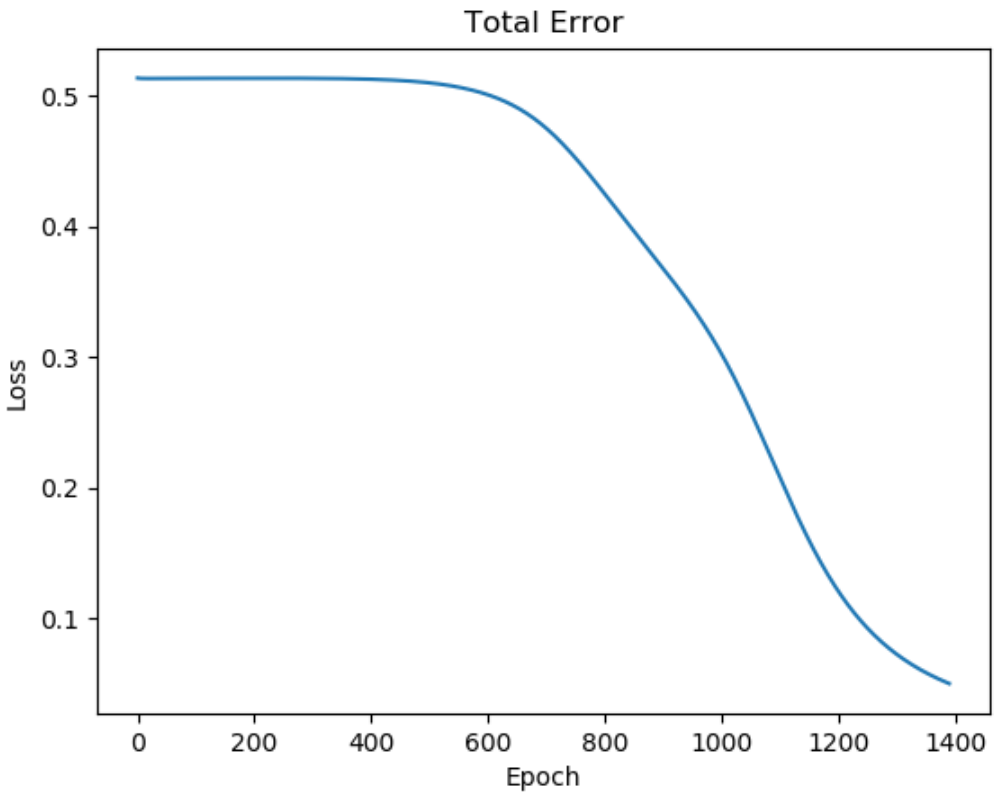
Assignment Part 1a – Backpropagation Learning

Runze Wang 99829855

1)

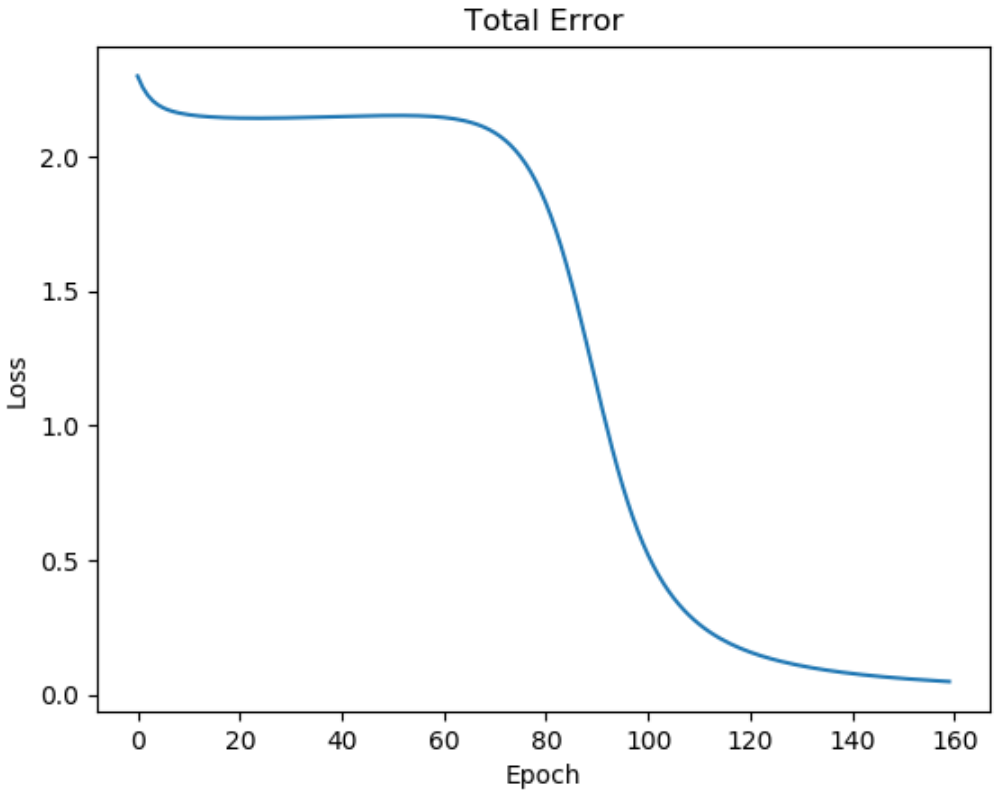
a)



The neural net is trained with XOR training set. On average of 10 trials, it takes 1495 epochs to reach a total error of less than 0.05. The graph above is plotted to reflect the change in total loss with epochs during the 10^{th} trial.

From this graph, we can infer that our neural net initially has a loss of 0.5, starts to converge at epoch 600, and eventually converges at epoch 1400 approximately, which is less than the average epoch (1495). Note that one trial in our experiment reaches 2069 epochs to lower the loss to the same level.

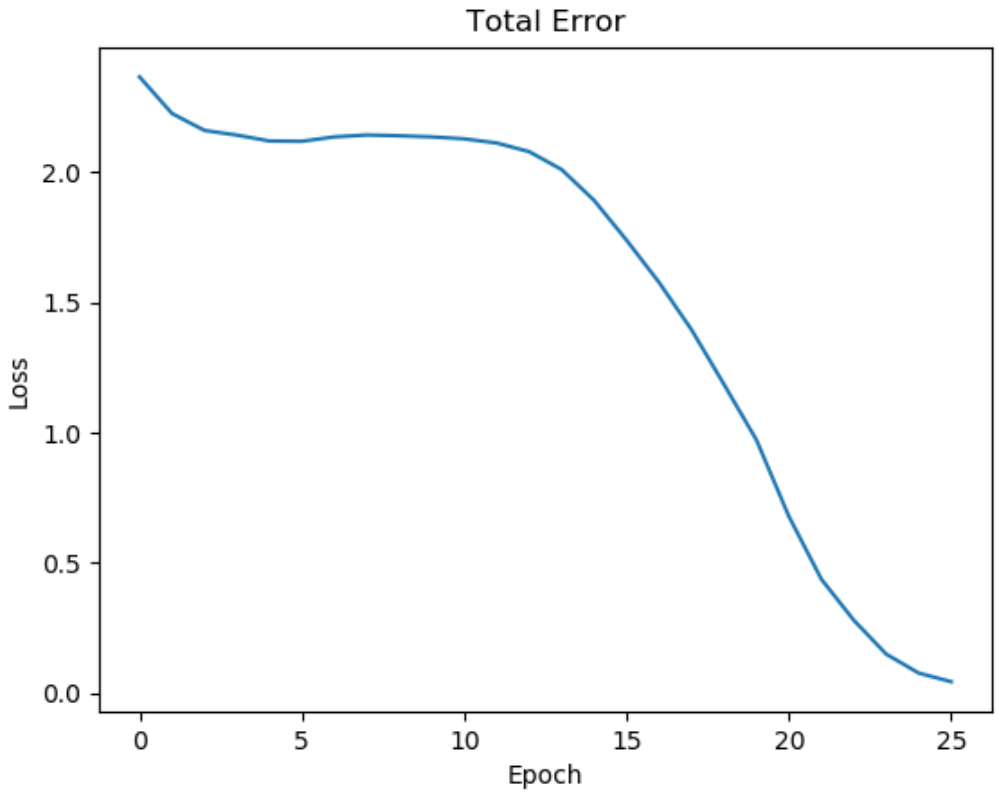
b)



In a bipolar representation, on average of 10 trials, it takes 169 epochs to reach a total error of less than 0.05. The graph above is plotted to reflect the change in total loss with epochs during the 10^{th} trial.

From this graph, we can infer that our neural net initially has a loss of 2, starts to converge at epoch 80, and eventually converges at epoch 160 approximately, which is less than the average epoch (169).

c)



When momentum is set to 0.9, on average of 10 trials, it takes 28 epochs to reach a total error of less than 0.05. The graph above is plotted to reflect the change in total loss with epochs during the 10^{th} trial.

From this graph, we can infer that our neural net initially has a loss of 2, starts to converge at epoch 12, and eventually converges at epoch 25 approximately, which is less than the average epoch (28).

Appendix

- NeuralNet.java

```
import com.google.gson.Gson;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class NeuralNet implements NeuralNetInterface{
    private int argNumInputs;
    private int argNumHidden;
    private static final int ARG_NUM_OUTPUTS = 1;
    private double argLearningRate;
    private double argMomentumTerm;
    private double argA;
    private double argB;
    private boolean bipolar;
    private Matrix weightIH;
    private Matrix weightHO;
    private Matrix biasH;
    private Matrix biasO;
    private Matrix d_weightIH;
    private Matrix d_weightHO;
    private Matrix d_biasH;
    private Matrix d_biasO;


    public NeuralNet(
        int argNumInputs,
        int argNumHidden,
        double argLearningRate,
        double argMomentumTerm,
        double argA,
        double argB,
        boolean bipolar
    ){
        this.argNumInputs = argNumInputs;
        this.argNumHidden = argNumHidden;
        this.argLearningRate = argLearningRate;
        this.argMomentumTerm = argMomentumTerm;
        this.argA = argA;
        this.argB = argB;
        this.bipolar = bipolar;
    }

    @Override
    public void zeroWeights() {
        this.weightIH = new Matrix(argNumHidden, argNumInputs);
        this.weightHO = new Matrix(ARG_NUM_OUTPUTS, argNumHidden);

        this.biasH = new Matrix(argNumHidden, 1);
        this.biasO = new Matrix(ARG_NUM_OUTPUTS, 1);
```

```
        this.d_weightIH = new Matrix(argNumHidden, argNumInputs);
        this.d_biasH = new Matrix(argNumHidden, 1);
        this.d_weightHO = new Matrix(ARG_NUM_OUTPUTS, argNumHidden);
        this.d_biasO = new Matrix(ARG_NUM_OUTPUTS, 1);
        return;
    }

    @Override
    public void initializeWeights() {
        this.weightIH = new Matrix(argNumHidden, argNumInputs, -0.5,
0.5);
        this.weightHO = new Matrix(ARG_NUM_OUTPUTS, argNumHidden, -0.5,
0.5);

        this.biasH = new Matrix(argNumHidden, 1, -0.5, 0.5);
        this.biasO = new Matrix(ARG_NUM_OUTPUTS, 1, -0.5, 0.5);

        this.d_weightIH = new Matrix(argNumHidden, argNumInputs);
        this.d_biasH = new Matrix(argNumHidden, 1);
        this.d_weightHO = new Matrix(ARG_NUM_OUTPUTS, argNumHidden);
        this.d_biasO = new Matrix(ARG_NUM_OUTPUTS, 1);

        return;
    }

    @Override
    public double outputFor(double[] X) {
        // Input to hidden
        Matrix dataI = Matrix.parseArray(X);
        Matrix dataH = Matrix.multiply(weightIH, dataI);
        dataH.add(biasH);

        // Activation function input -> hidden
        if (this.bipolar) {
            dataH.bipolarSigmoid();
        } else {
            dataH.msigmoid(this.argA, this.argB);
        }

        // Hidden to output
        Matrix dataO = Matrix.multiply(weightHO, dataH);
        dataO.add(biasO);
        if (this.bipolar) {
            dataO.bipolarSigmoid();
        } else {
            dataO.msigmoid(this.argA, this.argB);
        }

        double output = Matrix.toArray(dataO)[0];

        return output;
    }

    @Override
    public double train(double[] X, double argValue) {
        // Input to hidden
```

```
        Matrix dataI = Matrix.parseArray(X);
        Matrix dataH = Matrix.multiply(weightIH, dataI);
        dataH.add(biasH);

        // Activation function input -> hidden
        if (this.bipolar) {
            dataH.bipolarSigmoid();
        } else {
            dataH.msigmoid(this.argA, this.argB);
        }

        // Hidden to output
        Matrix dataO = Matrix.multiply(weightHO, dataH);
        dataO.add(biasO);

        if (this.bipolar) {
            dataO.bipolarSigmoid();
        } else {
            dataO.msigmoid(this.argA, this.argB);
        }

        double output = Matrix.toArray(dataO)[0];

        //loss computation
        double loss = argValue - output;
        Matrix lossM = new Matrix(1,1);
        lossM.add(loss);

        // hidden to output gradient
        Matrix gradient = this.getGradient(dataO);
        gradient.multiply(lossM);
        gradient.multiply(argLearningRate);

        //momentum
        Matrix hiddenDataTranpose = Matrix.transpose(dataH);
        Matrix deltaHiddenData = Matrix.multiply(gradient,
hiddenDataTranpose);

        //update hidden to output weight
        d_weightHO.multiply(argMomentumTerm);
        d_weightHO.add(deltaHiddenData);
        d_biasO.multiply(argMomentumTerm);
        d_biasO.add(gradient);

        weightHO.add(d_weightHO);
        biasO.add(d_biasO);

        Matrix h_o_weight_transpose = Matrix.transpose(weightHO);
        // h_o_w_t 4*1 lossM 1*1
        Matrix hidden_loss = Matrix.multiply(h_o_weight_transpose,
lossM);

        //gradient for input to hidden
        Matrix hidden_gradient = this.getGradient(dataH);
        // hidden_gradient 4*1 hidden_loss 4*1 -> 1*1
        hidden_gradient.multiply(hidden_loss);
        hidden_gradient.multiply(argLearningRate);
```

```
        //momentum
        Matrix inputDataTranspose = Matrix.transpose(dataI);
        Matrix deltaInputData = Matrix.multiply(hidden_gradient,
inputDataTranspose);

        //update input to hidden weight
        d_weightIH.multiply(argMomentumTerm);
        d_weightIH.add(deltaInputData);
        d_biasH.multiply(argMomentumTerm);
        d_biasH.add(hidden_gradient);
        weightIH.add(d_weightIH);
        biasH.add(d_biasH);

        return loss;
    }

    @Override
    public void load(String argFileName) throws IOException {
        File modelFile = new File(argFileName);
        Scanner modelReader = new Scanner(modelFile);
        String model = modelReader.nextLine();
        modelReader.close();

        Gson gson = new Gson();
        NeuralNet loadedNN = gson.fromJson(model, NeuralNet.class);
        if(loadedNN.getArgNumInputs() != this.argNumInputs ||
            loadedNN.getArgNumHidden() != this.argNumHidden ||
            loadedNN.getArgLearningRate() != this.argLearningRate ||
            loadedNN.getArgMomentumTerm() != this.argMomentumTerm ||
            loadedNN.getArgA() != this.argA ||
            loadedNN.getArgB() != this.argB ||
            loadedNN.isBipolar() != this.bipolar
        ){
            throw new IOException("Model does not match current
configuration");
        }

        this.weightIH = loadedNN.getWeightIH();
        this.weightHO = loadedNN.getWeightHO();
        this.biasH = loadedNN.getBiasH();
        this.biasO = loadedNN.getBiasO();
        this.d_weightIH = loadedNN.getD_weightIH();
        this.d_weightHO = loadedNN.getD_weightHO();
        this.d_biasO = loadedNN.getD_biasO();
        this.d_biasH = loadedNN.getD_biasH();
        return;
    }

    @Override
    public void save(File argFile) throws IOException {
        Gson gson = new Gson();
        String jsonModel = gson.toJson(this);
        if(!argFile.exists()){
            argFile.createNewFile();
        }
        FileWriter myWriter = new
```

```
FileWriter(argFile.getAbsolutePath());
        myWriter.write(jsonModel);
        myWriter.close();
        return;
    }

    private Matrix getGradient(Matrix m) {
        if (this.bipolar) {
            return m.dbipolarSigmoid();
        } else {
            return m.dsigmoid(0, 1 );
        }
    }

    public int getArgNumInputs() {
        return argNumInputs;
    }

    public int getArgNumHidden() {
        return argNumHidden;
    }

    public double getArgLearningRate() {
        return argLearningRate;
    }

    public double getArgMomentumTerm() {
        return argMomentumTerm;
    }

    public double getArgA() {
        return argA;
    }

    public double getArgB() {
        return argB;
    }

    public Matrix getWeightIH() {
        return weightIH;
    }

    public Matrix getWeightHO() {
        return weightHO;
    }

    public boolean isBipolar() {
        return bipolar;
    }

    public Matrix getBiasH() {
        return biasH;
    }

    public Matrix getBiasO() {
        return biasO;
    }
```

```
    public Matrix getD_biasH() {
        return d_biasH;
    }

    public Matrix getD_biasO() {
        return d_biasO;
    }

    public Matrix getD_weightHO() {
        return d_weightHO;
    }

    public Matrix getD_weightIH() {
        return d_weightIH;
    }

    public static int getArgNumOutputs() {
        return ARG_NUM_OUTPUTS;
    }
}
```

- NeuralNetRunner.java

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import com.github.sh0nk.matplotlib4j.Plot;
import com.github.sh0nk.matplotlib4j.PythonExecutionException;


public class NeuralNetRunner {
    public static final double LEARNING_RATE = 0.2;
    public static final double MOMENTUM = 0.9;
    public static final double LOSS = 0.05;
    public static final double TRIALS = 10;

    //public static final double [][] X = { {0, 0}, {1, 0}, {0, 1}, {1,
1} };
    public static final double [][] X = { {-1, -1}, {-1, 1}, {1, -1},
{1, 1} };

    public static void main(String[] args) throws IOException,
PythonExecutionException {
        if(args.length != 3){
            System.out.print("Three arguments required!");
            return;
        }

        int inputNum;
        int hiddenNum;
        boolean bipolar;
        try{
```

```
            inputNum = Integer.parseInt(args[0]);
            hiddenNum = Integer.parseInt(args[1]);
            bipolar = Boolean.parseBoolean(args[2]);
        }catch(Exception e){
            System.out.print("Arguments not in the right type!");
            return;
        }

        System.out.println("Input layer number of neurons:" +
inputNum);
        System.out.println("Hidden layer number of neurons:" +
hiddenNum);

        // Initialize a new neural net
        NeuralNet nn = new NeuralNet(inputNum, hiddenNum,
LEARNING_RATE, MOMENTUM, 0, 1, bipolar);

        // Train the neural net
        int trial = 0;
        int sumEpoch = 0;
        int trialToPlot = 9;
        while(trial < TRIALS){
            int epoch = 0;
            Map<Integer, Double> map = new HashMap<>();
            double totalLoss;
            // Initialize weights
            nn.initializeWeights();

System.out.println("================================================
==============");
            System.out.println("Trail: "+trial);
            do{
                totalLoss = 0;
                for(double [] data: X){
                    double singleLoss = nn.train(data, /*(int)data[0] ^
(int)data[1]*/ data[0] == data[1] ? -1 : 1);
                    totalLoss += 0.5 * Math.pow(singleLoss, 2);
                }
                map.put(epoch, totalLoss);
                epoch++;
            }while(totalLoss > LOSS);
            trial++;
            sumEpoch += epoch;
            System.out.println("Epochs: "+epoch);
            System.out.println("Final loss: "+totalLoss);
            if(trial == trialToPlot){
                Plot plt = Plot.create();
                List<Integer> epochs = new ArrayList<>();
                List<Double> losses = new ArrayList<>();
                for(int e : map.keySet()){
                    epochs.add(e);
                    losses.add(map.get(e));
                }
                plt.plot().add(epochs, losses);
                plt.xlabel("Epoch");
                plt.ylabel("Loss");
                plt.title("Total Error");
```

```
                plt.show();
            }
        }

System.out.println("================================================
==============");
        System.out.println("Training finished!");
        System.out.println("Average Epoch: " + (double)sumEpoch/trial);
    }
}
```

- Matrix.java

```
public class Matrix {
    double[][] m;
    int rows, cols;

    public Matrix(int rows,int cols) {
        m = new double[rows][cols];
        this.rows = rows;
        this.cols = cols;
        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                m[i][j] = 0;
            }
        }
    }

    public Matrix(int rows, int cols, double value) {
        m = new double[rows][cols];
        this.rows = rows;
        this.cols = cols;
        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                m[i][j] = value;
            }
        }
    }

    public Matrix(int rows, int cols, double min, double max) {
        m = new double[rows][cols];
        double range = max - min;
        this.rows = rows;
        this.cols = cols;
        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                m[i][j] = Math.random() * range + min;
            }
        }
    }

    public void add(double c) {
        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                this.m[i][j] += c;
            }
```

```
        }
    }

    public void add(Matrix mtx) {
        if(cols!=mtx.cols || rows!=mtx.rows) {
            return;
        }

        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                this.m[i][j] += mtx.m[i][j];
            }
        }
    }

    public static Matrix subtract(Matrix a, Matrix b) {
        Matrix res = new Matrix(a.rows, a.cols);
        for(int i = 0; i < a.rows; i++) {
            for(int j = 0; j < a.cols; j++) {
                res.m[i][j] = a.m[i][j] - b.m[i][j];
            }
        }
        return res;
    }

    public static Matrix multiply(Matrix a, Matrix b) {
        if(a.cols != b.rows) return null;
        Matrix res = new Matrix(a.rows,b.cols);
        for(int i = 0; i < res.rows; i++){
            for(int j = 0; j < res.cols; j++){
                double ele = 0;
                for(int k=0;k<a.cols;k++){
                    ele += a.m[i][k] * b.m[k][j];
                }
                res.m[i][j] = ele;
            }
        }
        return res;
    }

    public void multiply(double a) {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                m[i][j] *= a;
            }
        }
    }

    public void multiply(Matrix a) {
        if(a.cols != this.cols) return;
        if(a.rows != this.rows) return;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                m[i][j] *= a.m[i][j];
            }
        }
    }
```

```
    public void msigmoid(double a, double b) {
        for(int i=0;i<rows;i++)
        {
            for(int j=0;j<cols;j++)
                this.m[i][j] =
NeuralNetInterface.customSigmoid(this.m[i][j], a, b);
        }
    }

    public Matrix dsigmoid(double a, double b){
        Matrix res = new Matrix(rows, cols);
        for(int i=0;i<rows;i++)
        {
            for(int j=0;j<cols;j++)
                res.m[i][j] = m[i][j] * (1 - m[i][j]);
        }
        return res;
    }

    public double bipolarSigmoid(double x) {
        return 2 / (1 + Math.exp(-x)) - 1;
    }

    public void bipolarSigmoid() {
        for (int i = 0; i < this.rows; i++) {
            for (int j = 0; j < this.cols; j++) {
                this.m[i][j] = bipolarSigmoid(this.m[i][j]);
            }
        }
    }

    public Matrix dbipolarSigmoid(){
        Matrix res = new Matrix(rows, cols);
        for (int i = 0; i < this.rows; i++) {
            for (int j = 0; j < this.cols; j++) {
                res.m[i][j] = (m[i][j] + 1) * (1 - m[i][j]) / 2;
            }
        }
        return res;

    }

    public static Matrix transpose(Matrix a) {
        Matrix res = new Matrix(a.cols, a.rows);
        for (int i = 0; i < a.rows; i++) {
            for (int j = 0; j < a.cols; j++) {
                res.m[j][i] = a.m[i][j];
            }
        }
        return res;
    }

    public static Matrix parseArray(double[] array){
        Matrix res = new Matrix(array.length, 1);
        for (int i = 0; i < res.rows; i++) {
            res.m[i][0] = array[i];
```

```
        }
        return res;
    }

    public static double[] toArray(Matrix mtx){
        double[] res = new double[mtx.cols];
        for (int i = 0; i < mtx.rows; i++) {
            res[i] = mtx.m[i][0];
        }
        return res;
    }

    public static void print(Matrix p) {
        for(int i = 0; i < p.rows; i++){
            for(int j = 0; j < p.cols; j++){
                System.out.print(p.m[i][j]);
                System.out.print(" ");
            }
            System.out.println();
        }
    }
}
```

- MatrixTest.java

```
import org.junit.Assert;
import org.junit.Test;

public class MatrixTest {
    @Test
    public void testMultiply(){
        Matrix a = new Matrix(2,3, 2);
        Matrix b = new Matrix(3, 2, 3);
        Matrix c = new Matrix(2,3, 2);
        Matrix d = new Matrix(2,3,4);
        double[][] expected = new double[][]{{18,18}, {18,18}};
        double[][] actual = Matrix.multiply(a, b).m;
        Assert.assertArrayEquals(expected, actual);
        a.multiply(c);
        Assert.assertArrayEquals(a.m, d.m);
        a.multiply(0.5);
        Assert.assertArrayEquals(a.m, c.m);
    }

    @Test
    public void testAdd(){
        Matrix a = new Matrix(2,3);
        Matrix b = new Matrix(2,3);
        Matrix c = new Matrix(2,3);
        Matrix d = new Matrix(2,3, 1);
        a.add(b);
        Assert.assertArrayEquals(c.m, a.m);
        a.add(1);
        Assert.assertArrayEquals(d.m, a.m);
```

```
    }

    @Test
    public void testTranspose(){
        Matrix a = new Matrix(2,3, 1);
        Matrix b = new Matrix(3,2,1);
        Assert.assertArrayEquals(Matrix.transpose(a).m, b.m);
    }

    @Test
    public void testSigmoid(){
        Matrix a = new Matrix(2, 2);
        a.msigmoid(0, 1);
        Assert.assertArrayEquals(new double[][]{{0.5, 0.5}, {0.5,
0.5}}, a.m);

        a = new Matrix(2,2);
        a.bipolarSigmoid();
        Assert.assertArrayEquals(new double[][]{{0, 0}, {0, 0}}, a.m);
    }

    @Test
    public void testParseArray(){
        Matrix a  = new Matrix(2, 1);
        double[] actual = new double[]{0,0};
        Assert.assertArrayEquals(Matrix.parseArray(actual).m, a.m);
    }

    @Test
    public void testToArray(){
        Matrix a = new Matrix(1, 2);
        double[] expected = new double[]{0, 0};
        Assert.assertArrayEquals(expected, Matrix.toArray(a), 1e-15);
    }
}
```

- NeuralNetTest.java

```
import org.junit.Assert;
import org.junit.Test;

import java.io.File;
import java.io.IOException;

public class NeuralNetTest {
    @Test
    public void testZeroWeight(){
        NeuralNet nn = new NeuralNet(2,4,0.2,0,0,1, true);
        Matrix zeroIH = new Matrix(4,2);
        Matrix zeroHO = new Matrix(1,4);
        nn.zeroWeights();
        Assert.assertArrayEquals(zeroIH.m, nn.getWeightIH().m);
        Assert.assertArrayEquals(zeroHO.m, nn.getWeightHO().m);
    }
```

```
    @Test
    public void testOutputfor(){
        NeuralNet nn = new NeuralNet(2,4,0.2,0,0,1, false);
        Matrix zeroIH = new Matrix(4,2);
        Matrix zeroHO = new Matrix(1,4);
        nn.zeroWeights();
        double[] res = new double[]{2,2};
        Assert.assertEquals(0.5, nn.outputFor(res), 1e-16);
    }

    @Test
    public void testSave(){
        NeuralNet nn = new NeuralNet(2,4,0.2,0,0,1, true);
        nn.zeroWeights();
        try {
            nn.save(new File("outputs/test.json"));
        } catch (IOException e) {
            Assert.assertTrue(false);
        }
        Assert.assertTrue(true);
    }

    @Test
    public void testLoad(){
        NeuralNet nn = new NeuralNet(2,4,0.2,0,0,1, true);
        nn.initializeWeights();
        try {
            nn.load("test.json");
        } catch (IOException e) {
            Assert.assertTrue(false);
        }
        Assert.assertArrayEquals(nn.getWeightIH().m, new
Matrix(4,2).m);
        Assert.assertArrayEquals(nn.getWeightHO().m, new
Matrix(1,4).m);

        nn = new NeuralNet(2,3,0.2,0,0,1, true);
        try {
            nn.load("test.json");
        } catch (IOException e) {
            Assert.assertTrue(true);
            return;
        }
        Assert.assertTrue(false);
    }

    @Test
    public void testGetGradient(){

    }
}
```