Problem 3: Perceptrons

Working group assignment

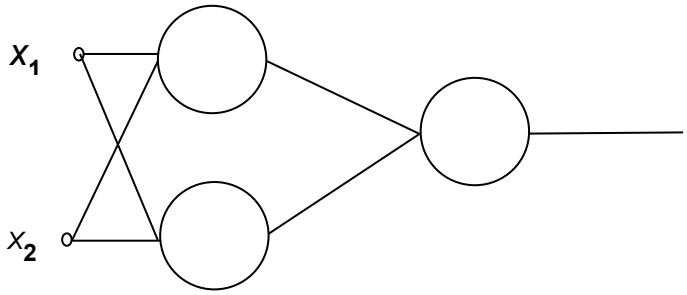
Submit your code and report (within a zip file) to
http://deei-mooshak.ualg.pt/~jvo/IA/Entregas/
as TP1

Submit to Mooshak problem D the task defined below.

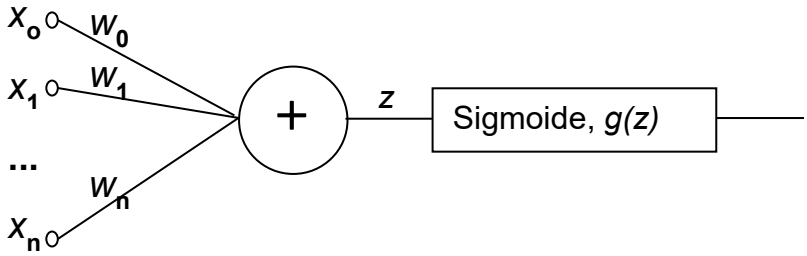
up to: November 24, 2025

Problem

Consider the following multilayer perceptron:



where each circle represents a neuron of the type:



Where $x_0=1, x_1, \ldots x_j, \ldots, x_n$ are the inputs, and $w_0, w_1, \ldots, w_j, \ldots w_n$ are their respective weights, and the sigmoid g is the logistic function, i.e.,

$$g(z) = \frac{1}{1+e^{-z}}$$

Task

1. Propose a data set $\{(x_1^{(i)},x_2^{(i)}, y^{(i)}) \mid i=1,\ldots, 4\}$ where x_1 and x_2 are independent binary variables and y is a linearly separable dependent binary variable.

2. Compute, without training, the weights of a single neuron for obtaining a zero-training error in the data set proposed in 1. Briefly explain.

3. Implement and test the neuron configured in 2.

4. Configure, without training, the above multilayer perceptron so that it works as a NOT XOR binary function. Briefly explain.

5. Implement the training of the network with the back propagation algorithm using the dataset of the NOT XOR function. Plot the MSE in each iteration of the training process. How similar are the obtained weights to those computed in 4.? Briefly explain.

Mooshak Task
Your task is to write a program **based on the approach presented in Appendix A's tutorial**, that trains an MLP on the NOT XOR.

Please note that any other approach, however meritorious it may be, will be quoted with 0 (zero).

Input
The input consists of an integer *n* defining the number of input lines that follow. Then, each line has a pair of variables that define the input for the NOT XOR logic function. These have been corrupted by noise, so these inputs are *double*.

Output
A set of *n* lines, each with a 0 or 1, the result of the NOT XOR function for each input pair.

Sample Input 1
1
1.0 0.9

Sample output 1
1

Sample Input 2
3
1.1 0.1
0.9 0.89
0.8 -0.1

Sample output 2
0
1
0

Appendix: MLP training tutorial

1. Introduction

This tutorial shows a vectorial implementation of a training algorithm for a multi-layer perceptron. The vectorial implementation has several advantages. It can handle many input samples simultaneously, it is easy to implement when supported by a linear algebra library, and it can be easily adapted to run on multi-core devices.

To support this vectorial algorithm a small set of linear algebra functions is defined in the Matrix class. Also, a Sigmoid class implements a differentiable activation function to apply at the neuron's output.

Note that the following code is incomplete. It is needed to add the missing code where it is commented:

//insert code here to …

2. Main class

First, let's define the main class. Note that you must define at the top of **main()** function a suitable learning rate and the number of epochs (epochs). You can also change the configuration of the MLP layers by playing with the **topology** array. Where the first element is the number of inputs – 2 in this problem – and the last element is the number of outputs, which, for this problem, should always be 1. The inner elements define the number of neurons in each hidden layer.
You must also insert code at the end to print the Matrix with the prediction.

```
import math.Matrix;
import neural.activation.IDifferentiableFunction;
import neural.MLP;
import neural.activation.Sigmoid;
import neural.activation.Step;
import java.util.Scanner;

public class MLPNXOR {

    public static void main(String[] args) {

        double lr     = //define a learning rate smaller than zero
        int   epochs = //define the number of epochs in the order of thousands
        int[] topology = {2, 2, 1};

        //Training set
        Matrix trX = new Matrix(
            new double[][] {
                    {0,0},
                    {0,1},
                    {1,0},
                    {1,1} } );

        Matrix trY = new Matrix(
            new double[][] {
                {1},
                {0},
                {0},
                {1} } );

            //Get input and create evaluation Matrix
            Scanner sc = new Scanner(System.in);
            int n  = sc.nextInt();
            double[][] input = new double[n][2];
```

```
            for (int i = 0; i < n; i++) {
                input[i][0] = sc.nextDouble();
                input[i][1] = sc.nextDouble();
            }
            Matrix evX = new Matrix(input);

            //Train the MLP
            MLP mlp = new MLP(   topology,
                                 new IDifferentiableFunction[]{
                                     new Sigmoid(),
                                     new Sigmoid() },
                                  -1);
            mlp.train(trX, trY, lr, epochs);

            //Predict and output results
            Matrix pred = mlp.predict(evX);

            //convert probabilities to integer classes: 0 or 1
            pred = pred.apply(new Step().fnc());

            //print output
            //insert code here to print the pred Matrix as integers
            //…

            sc.close();
        }
```

Next, we will define the supporting classes.

3. Activation functions

We need an interface for differentiable functions:

```
import java.util.function.Function;

public interface IDifferentiableFunction {
    Function<Double, Double> fnc();
    Function<Double, Double> derivative();
}
```

The **fnc()** method computes the function on the input, whereas the **derivative()** method returns the derivative of the function on the input.
Class Sigmoid implements this interface:

```
import java.util.function.Function;

public class Sigmoid implements IDifferentiableFunction  {

    @Override
    public Function<Double, Double> fnc() {
        return (z) -> 1.0 / (1.0 + Math.exp(-z));
    }

    @Override
    public Function<Double, Double> derivative() {
        return (y) -> y * (1.0 - y);
    }
}
```

We can see at the end of the **main()** function that a Step function is used to convert probabilities to the classes 0 or 1. This function is not continuous and thus it is not differentiable, but its derivative can be approximated by the Dirac (or impulse) function. However, for our purposes, we do not need to define its derivative, so we can just send an exception in the **derivative()** method:

```
public class Step implements IDifferentiableFunction {

    static private double threshold = 0.5;

    @Override
    public Function<Double, Double> fnc() {

        // insert code here to return a lambda function that returns
        // 0 if its argument is < 0 and 1 otherwise
    }

    @Override
    public Function<Double, Double> derivative() {
        throw new UnsupportedOperationException("Step function is not
differentiable.");
    }

}
```

4. Linear algebra support

It is also needed some elementary linear algebra functions to support the implementation of the MLP training and prediction algorithms. This will be implemented in the Matrix class.

It is suggested to use unit tests to make sure all functions are properly implemented.

Let's start with the constructors and accessors. Note that we need 2 constructors: One that initialises a matrix given the number of rows and columns, setting all elements to zero, and another that initialises a Matrix from an array of double[][] with each element value. This one was used in the **main()** function to create a Matrix from the input:

```
import java.util.List;
import java.util.function.BiFunction;
import java.util.function.Function;
import java.util.Random;

public class Matrix {

    private double[][] data;
    private int rows, cols;

    public Matrix(int rows, int cols) {
        data = new double[rows][cols];
        this.rows = rows;
        this.cols = cols;
    }

    public Matrix(double[][] data) {
        this.rows = data.length;
        this.cols = data[0].length;
        this.data = new double[rows][cols];
        for (int i = 0; i < rows; i++)
            System.arraycopy(data[i], 0, this.data[i], 0, cols);
```

```
    }

    //accessors
    public double get(int row, int col) { return data[row][col]; }
    public int rows() { return rows; }
    public int cols() { return cols; }
```

It is also needed a static factory function to initialise a Matrix with random numbers. This will be needed to initialise the weight and bias Matrices for MLP training:

```
static public Matrix Rand(int rows, int cols, int seed) {
    Matrix out = new Matrix(rows, cols);

    if (seed < 0)
        seed = (int) System.currentTimeMillis();

    Random rand = new Random(seed);
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            out.data[i][j] = rand.nextDouble();

    return out;
}
```

The next function applies a math function to all the elements of a Matrix and returns a new Matrix:

```
//Apply Function<Double, Double> to all elements of the matrix
//store the result in matrix result
private Matrix traverse(Function<Double, Double> fnc) {
    Matrix result = new Matrix(rows, cols);

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result.data[i][j] = fnc.apply(data[i][j]);
        }
    }
    return result;
}
```

Now we can use this traverse function to implement several others, like operations between a Matrix and a scalar: add a scalar to all elements, subtract a scalar from all elements:

```
//Apply a function to all elements
public Matrix apply(Function<Double, Double> fnc) { return traverse(fnc); }

//multiply matrix by scalar
public Matrix mult(double scalar) {

    // insert code here to multiply all elements by scalar
}

//add scalar to matrix
public Matrix add(double scalar) {
    // insert code here to add to all elements the scalar
}
```

```
//sub matrix from scalar:   scalar - M
public Matrix subFromScalar(double scalar) {

    // insert code here to subtract from the scalar all the elements
}
```

Some operations are performed element-wise between 2 matrices, for instance, adding or subtracting elements in the same positions. We can have a general function to traverse all the elements of the Matrix and apply a lambda function that defines the intended operation to each element.

```
//Element wise operation
public Matrix elementWise(Matrix other, BiFunction<Double, Double, Double>
fnc) {
    if (this.rows != other.rows || this.cols != other.cols) {
        throw new IllegalArgumentException("Incompatible matrix sizes for
element wise.");
    }

    Matrix result = new Matrix(rows, cols);

    //add element by element
    //store the result in matrix result
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++)
            result.data[i][j] = fnc.apply(this.data[i][j], other.data[i][j]);
    }
    return result;
}


//add two matrices
public Matrix add(Matrix other) {
    return this.elementWise(other, (a, b) -> a + b);
}


//multiply two matrices (element-wise)
public Matrix mult(Matrix other) {

    // insert code here to multiply element-wise
}


//subtract two matrices (element-wise)
public Matrix sub(Matrix other) {

    // insert code here to subtract (this-other) element-wise
}
```

Other math operations on matrices include the sum of all elements:

```
//sum all elements of the matrix
public double sum() {

    // insert code here to return the sum of all elements

    return total;
}
```

The dot product:

```
//multiply two matrices (dot product)
public Matrix dot(Matrix other) {
    if (this.cols != other.rows) {
        throw new IllegalArgumentException("Incompatible matrix sizes for
multiplication.");
    }

    Matrix result = new Matrix(this.rows, other.cols);

    //multiply 2 matrices
    //store the result in matrix result
    for (int i = 0; i < this.rows; i++) {
        for (int j = 0; j < other.cols; j++) {
            for (int k = 0; k < this.cols; k++) {
                result.data[i][j] += this.data[i][k] * other.data[k][j];
            }
        }
    }
    return result;
}
```

Some operations only manipulate the position of elements, rows and columns:

```
//transpose matrix
public Matrix transpose() {
    Matrix result = new Matrix(cols, rows);

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result.data[j][i] = data[i][j];
        }
    }
    return result;
}
```

5. MLP algorithm

Finally, the implementation of the training algorithm in the class MLP.

Class MLP constructor gets 3 parameters:

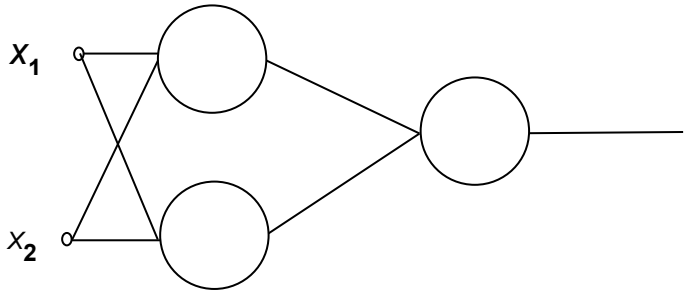
1) An array of integers that defines the number of neurons in each layer, from the input layer to the output layer, with the hidden layers in between.

2) An array of IDifferentiableFunction, one for each layer, except the input layer. This way, we can define a different activation function for each layer, although for the present case, it should be a Sigmoid.

3) The random seed for reproducibility. If seed = -1 it uses a random seed.

```
public MLP(int[] layerSizes, IDifferentiableFunction[] act, int seed)
```

This way, to define the MLP below, where all activation functions are Sigmoid:



we would use the following code snippet:

```
MLP mlp = new MLP(    new int[]{2, 2, 1},
                      new IDifferentiableFunction[]{
                          new Sigmoid(),
                          new Sigmoid() },
                -1);
```

To train it and use it to classify, we would use the code already presented in the **main()** function.

Now, the MLP class must define a list of weight matrices for each layer, except for the input layer. The number of rows is the number of neurons in the preceding layer, and the number of columns is the number of neurons in the current layer.
For the MLP above we will have 2 weight matrices with dims: 2x2 and 2x1.

It also defines an array of bias for each layer, except the input layer and an array of Matrices for the outputs of each layer. It is assumed that the output of the input layer is the input data.

```
public class MLP {

    private Matrix[] w;  //weights for each layer
    private double[] b;  //biases for each layer
    private Matrix[] yp; //outputs for each layer
    private IDifferentiableFunction[] act; //activation fnc for each layer
    private int numLayers;

    /* PRE: layerSizes.length >= 2
     * PRE: act.length == layerSizes.length - 1
     */
    public MLP(int[] layerSizes, IDifferentiableFunction[] act, int seed) {
        if (seed < 0)
            seed = (int) System.currentTimeMillis();

        numLayers = layerSizes.length;
        this.act = act;   //setup activation by layer

        //create output storage for each layer but the input layer
        yp = new Matrix[numLayers];

        //create weights and biases for each layer
        w = new Matrix[numLayers-1];
        b = new double[numLayers-1];

        Random rnd = new Random(seed);
        for (int i=0; i < numLayers-1; i++) {
            w[i] = Matrix.Rand(layerSizes[i], layerSizes[i + 1], seed);
            b[i] = rnd.nextDouble();
        }
    }
```

The training algorithm has 3 steps by epoch:

1) Shuffle input data (optional).

2) Feedforward, ie predict the classification for the input dataset.

3) Measure the error between the predicted output in the step above and the actual target output, and adjust the weight to minimise this error. We will use back propagation for this purpose.

Note:
In the formulas presented in the comments below:

the * symbol represents the **dot product** of 2 matrices
or the multiplication of a matrix by a scalar

the .* symbols represent the **element-wise multiplication** of 2 matrices

The feedforward or prediction phase is implemented by computing the output of each layer, with:

```
yp[0] = X
yp[l+1] = Sigmoid( yp[l] * w[l]+b[l] )
```

This is implemented by the function below. Note that the activation function is applied at the end and that the output of each layer is stored in array **yp**, for later use.

```
// Feed forward propagation
// also used to predict after training the net
// yp[l+1] = Sigmoid( yp[l] * w[l]+b[l] )
public Matrix predict(Matrix X) {
    yp[0] = X;
    for (int l=0; l < numLayers-1; l++)
        yp[l+1] = yp[l].dot(w[l]).add(b[l]).apply(act[l].fnc());
    return yp[numLayers-1];
}
```

The template for the training algorithm is below. Complete using the functions defined in the Matrix class:

```
public double[] train(Matrix X, Matrix y, double learningRate, int epochs) {
    int nSamples = X.rows();
    double[] mse = new double[epochs];

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

        //forward propagation
        Matrix ypo = predict(X);

        //backward propagation
        Matrix e = backPropagation(X, y, learningRate);

        //mse
        mse[epoch] = e.dot(e.transpose()).get(0, 0) / nSamples;
    }
    return mse;
}
```

Finally, implement the backpropagation phase, according to the notes below. Note that the derivative of the sigmoid function can be applied to a Matrix using:

```
public Matrix backPropagation(Matrix X, Matrix y, double lr) {
    Matrix e = null;
    Matrix delta = null;

    //back propagation using generalised delta rule
    for (int l = numLayers-2; l >= 0; l--) {
        if (l == numLayers-2)                       //output layer
            e = y.sub(yp[l+1]);                     //e = y - yp[l+1]
        else {                                      //propagate error

        // insert code here to compute the error of layer l:
        //
        //      e = delta * w[l+1]^T
        }

        // insert code here to compute Delta:
        //
        //      dy = yp[l+1] .* (1-yp[l+1])
        //      Note: to compute dy use Sigmoid class derivative
        //            in a similar way as in predict()
        //
        //      delta = e .* dy

         // insert code here to update the weights and biases
         //
         //      w[l] += yp[l]^T * delta * lr
         //      b[l] += sum(delta) * lr

    }
    return e;
}
```