Influence of the transfer function of the data classification process in a two - layer neural network

Bojan Vasovic[1], Ivan Garvanov[2]

[1]Academy of Professional Studies Southern Serbia, Leskovac, Serbia
[2]University of Library Studies and Information Technologies, Sofia, Bulgaria
bojan.vasovic@yahoo.com
i.garvanov@unibit.bg

Abstract: *This paper analyses the influence of transfer function choice and learning rate on a multilayer neural network in the classification process. The neural network is implemented in the Java programming language.*

Keywords: *Transfer Function, Neural Network, Data Classification, Characters Recognition*

1. Introduction

Artificial neural networks (ANNs), usually simply called neural networks (NNs), are computing systems vaguely inspired by the biological neural networks that constitute animal brains.

Neural networks learn (or are trained) by processing examples, each of which contains a known "input" and "result", forming probability-weighted associations between the two, which are stored within the data structure of the net itself. The training of a neural network from a given example is usually conducted by determining the difference between the processed output of the network (often a prediction) and a target output. This is the error. The network then adjusts its weighted associations according to a learning rule and using this error value. Successive adjustments will cause the neural network to produce output which is increasingly similar to the target output. After a sufficient number of these adjustments the training can be terminated based upon certain criteria. This is known as supervised learning.

Such systems "learn" to perform tasks by considering examples, generally without being programmed with task-specific rules. For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually labeled as "cat" or "no cat" and using the results to identify cats in other images. They do this without any prior knowledge of cats, for example, that they have fur, tails, whiskers, and cat-like faces. Instead, they automatically generate identifying characteristics from the examples that they process.

2. Elements of a Neural Network

Input Layer: This layer accepts input features. It provides information from the outside world to the network, no computation is performed at this layer, nodes here just pass on the information(features) to the hidden layer.

Hidden Layer: Nodes of this layer are not exposed to the outer world, they are the part of the abstraction provided by any neural network. Hidden layer performs all sort of computation on the features entered through the input layer and transfer the result to the output layer.

Output Layer: This layer brings up the information learned by the network to the outer world.

Definition of activation function: Activation function decides, whether a neuron should be activated or not by calculating weighted sum and further adding bias with it. The purpose of the activation function is to introduce non-linearity into the output of a neuron.

We know, neural network has neurons that work in correspondence of weight, bias and their respective activation function. In a neural network, we would update the weights and biases of the neurons on the basis of the error at the output. This process is known as back-propagation. Activation functions make the back-propagation possible since the gradients are supplied along with the error to update the weights and biases.

A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.

3. Network Architecture

A neural network with l layers is observed (Fig. 1). Inputs x_1, $x_2, \ldots, x_{\mathrm{nl}}$ are connected to the first layer neurons. Each connection corresponds to the weight $w_{\mathrm{i,j}}$. Each neuron represents a processor unit where neuron input to n1 is calculated as:

$$z_1^1 = w_{1,1}^1 x_1 + w_{1,2}^1 x_{2+} \ldots + w_{1,n1}^1 x_{n1} + b_1^1 \quad (1)$$

respectively, the input to the first layer *i*-th neuron is:

$$z_i^1 = \sum_{i=1}^{n2} w_{1,i}^1 x_i + b_i^1 \quad (2)$$

An artificial neuron may have an independent component which adds an additional signal to the transfer function. This component is called bias (b_i). This component has a value of 1, so if the weight w_0 is introduced, then the sum (2) can be simplified with

$$z_i^1 = \sum_{i=0}^{n2} w_{1,i}^1 x_i \text{ with } x_0 = b. \quad (3)$$

The output from the first layer *i*-th neuron, which is the input for the second layer neurons is:

$$a_i^1 = f(z_i^1) \quad (4)$$

The output from the neural network is calculated in the for-loop. For-loop is often slow to perform when it comes to processing large data sets, so the solution is to vectorize these equations.

For the architecture shown, the matrix equation can be generalized as follows:

$$Z = WX^T + b \quad (5)$$

$$A = f(Z) \quad (6)$$

The f function is called the transfer function. This function should provide a nonlinear complex functional mapping between the inputs and desired data target outputs.

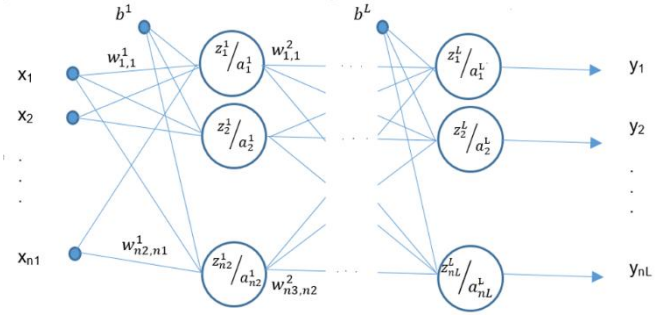
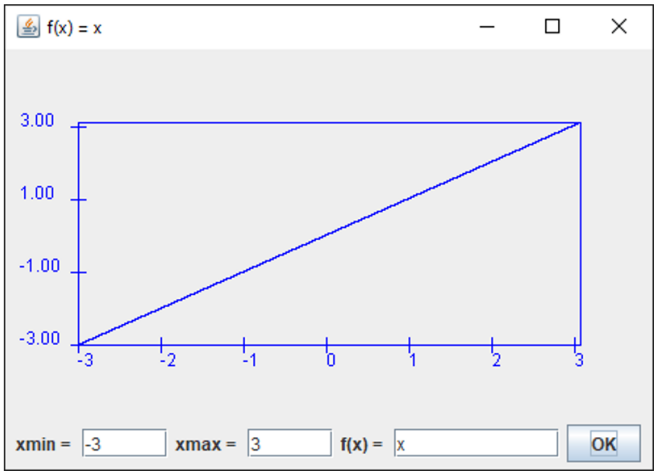
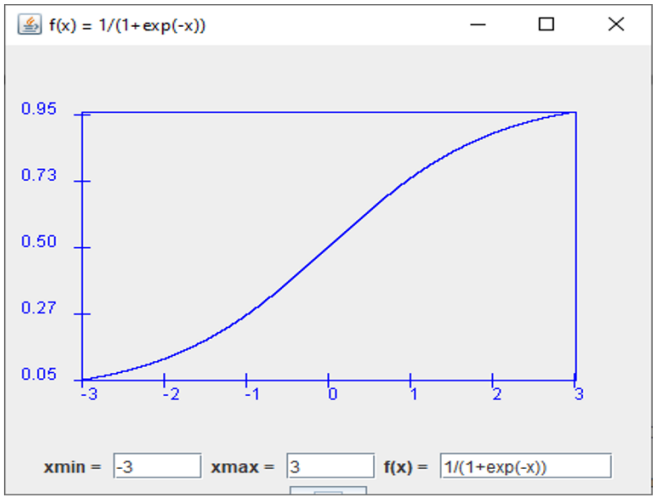


Fig. 1 *Neural network*

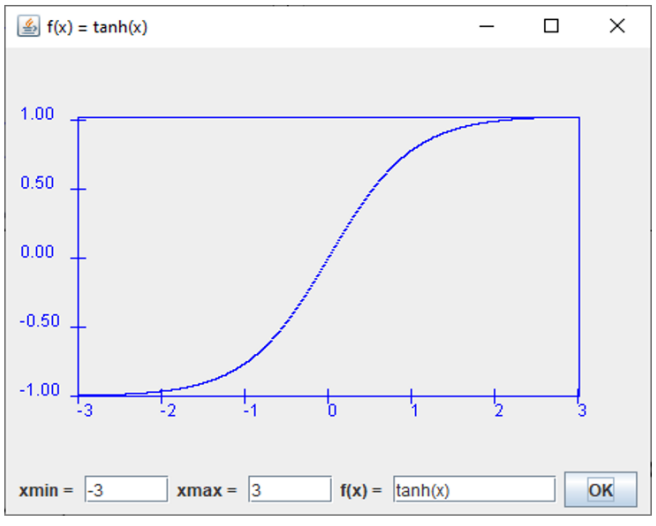
The most commonly used transfer functions are: Linear, Sigmoid and Hyperbolic tangent. The distinctive output of a single bias input linear neuron is shown in Figure 2 (a).



a) Linear



b) Sigmoid



c) Hyperbolic tangent

***Fig. 2** Transfer functions*

The sigmoid function is shown in Figure 2 (b). This transfer function takes an input, which can have any value between plus and minus infinity, and gives an output in the range of 0 to 1, according to the expression:

$$f(x) = \frac{1}{1+e^{-x}} \quad (7)$$

Hyperbolic tangent function is shown in Figure 2 (c).

$$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \quad (8)$$

In paper [1], several possibilities of using transfer functions of different types in neural networks are presented, as well as regularization of large networks with heterogeneous nodes and constructive approaches. Paper [2] aims to analyze the influence of the selection transfer function and training algorithms on neural network flood runoff forecast. Sigmoid function, used in this paper is described in paper [3].

4. Algorithm Backpropagation

Rummelhart [4] proposed an algorithm inspired by the gradient method and was called backpropagation. Based on the gradient method, the output error should be returned to the previous layers, find the influence of individual weights on the obtained error and determine the weight gain in individual layers.

The goal is to minimize the overall error. We can now calculate the error for each output neuron using the error function and sum them to get the total error:

$$J = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{nL} y_k^i \log(h_w(x^i))_k + (1-y_k^i)\log(1-(h_w(x^i))_k\right] \quad (9)$$

Where y_k^i is the desired output for the *i*-th training set and the *k*-th class from the neural network, and $h_w(x^i)$ is the output from the *i*-th layer, so it is the actual result of the network.

The error in the last layer is:

$$\delta^L = \frac{\partial J}{\partial z^L} = \frac{\partial J}{\partial z^L} * \frac{\partial a^L}{\partial z^L} = a^L - y \quad (10)$$

Backpropagation allows calculations of δ^l for each layer, and then with the help of these errors the values of the real interest $\frac{\partial J}{\partial w_{ij}^l}$, are calculated, respectively the influence of the weights of each layer on the total error.

The error of neurons in layer l can be expressed as follows:

$$\delta^l = (w^l)^T \delta^{l+1} \odot f'(z^l) \quad (11)$$

Here $\odot$ represents the Hadamard's product operator. Hadamard's product of two matrixes is very similar to the matrix's addition, the elements corresponding to the same row and column of given matrices are multiplied by each other to form a new matrix.

The influence of each layer weights on the total error is:

$$\frac{\partial J}{\partial w^i} = \Delta W_i = \delta^{l+1}(a^l)^T \quad (12)$$

The influence of each layer bias on the total error is:

$$\frac{\partial J}{\partial b^l} = \Delta B_i = \delta^{l+1} \quad (13)$$

New weights and bias are calculated as:

$$W_i = W_i - \alpha * \Delta W_i \quad (14)$$

$$\mathrm{B}_i = B_i - \alpha * \Delta B_i \quad (15)$$

The parameter α represents the learning rate. Learning rate is a small positive value that controls the magnitude of the parameters change at each run. Learning rate controls how quickly a neural network learns a problem.

5. Implementation

The base classes in the system are Neural Network and Layer, which should implement a general neural network model. Auxiliary class Matrix is also used, which implements basic operations with matrices. The Layer class is presented in the Figure 3.

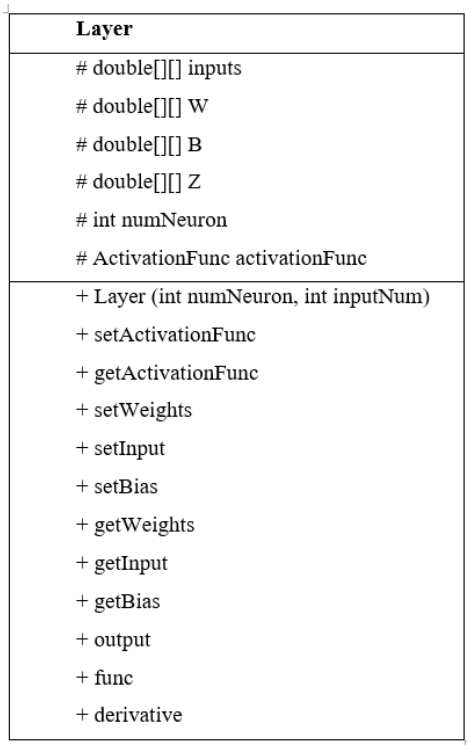


***Fig. 3** UML model of class Layer.*

Parameter NUMBER_LAYERS defines the number of layers and it is placed inside constructor of class Neural Network, where the numbers of neurons and number of entries in each layer are also defined by numNeurons and inputNum. Matrix desiredInputs defines the inputs to the neural network, and the matrix desiredOutputs the desired outputs.

```
int NUMBER_LAYERS = 2;
int numNeuron[] =  {15,10};
int inputNum[]  = {25,15};
private double[][] desiredInputs;
private double[][] desiredOutputs;
private List<Layer> layers = new
ArrayList<Layer>(NUMBER_LAYERS+1);
private List<double[][]> W = new
ArrayList<double[][]>(NUMBER_LAYERS+1);
private List<double[][]> B = new
ArrayList<double[][]>(NUMBER_LAYERS+1);
```

Lists layers, W and B represent the neural network elements (Figure 1) and they are initialized inside the constructor. Initialization is followed by neural network training in the training method. The number of iterations is determined by the variable NUM_ITERATION, and the weights are adjusted by the backpropagation algorithm according to the previously defined description.

```
double[][] dApom = null;
if(error_type == ERROR1)
{dApom=Matrix.sub(layers.get(NUMBER_LAYERS).out
put(),desiredOutputs);}
else if(error_type == ERROR2){
dApom=Matrix.sub(Matrix.div(Matrix.sub(1.0,desiredOut
puts),Matrix.sub(1.0,layers.get(NUMBER_LAYERS).out
put())), Matrix.div(desiredOutputs,
layers.get(NUMBER_LAYERS).output()));}
double[][] dZpom = Matrix.hadamardProduct(dApom,
layers.get(NUMBER_LAYERS).derivative(layers.get(NU
MBER_LAYERS).getZ()));
double[][] dWpom = Matrix.div(Matrix.mul(dZpom,
Matrix.T(layers.get(NUMBER_LAYERS).getInput())),
m); ///
double[][] dBpom = Matrix.div(dZpom, m);
dA.set(NUMBER_LAYERS, dApom);
dZ.set(NUMBER_LAYERS, dZpom);
dW.set(NUMBER_LAYERS, dWpom);
dB.set(NUMBER_LAYERS, dBpom);
```

m is the number of training data;

```
m = desiredInputs.length;
```

Then, the total error is calculated using the error method:

```
public static double error(int brojKlasa, int m, double[][]
y, double[][] a ){double J = 0.0;
for(int i = 0 ; i < m; i++){
for(int k = 0; k < brojKlasa; k++){
J=J+(y[i][k]*Math.log(a[i][k]))+(1-y[i][k])*Math.log(1-
a[i][k]);}}
return (-1.0/m) *J;}
```

The influence of weights on the neurons error in the layer layer (layer = NUMBER_LAYERS-1; layer >0; layer--) can be expressed as follows:

```
double [][] dA2 =
Matrix.mul(Matrix.T(layers.get(layer+1).getWeights()),
dZ.get(layer+1));
double[][] dZ2 = Matrix.hadamardProduct(dA2,
layers.get(layer).derivative(layers.get(layer).getZ()));
double[][]dW2;
if (layer == 1){
dW2 = Matrix.div(Matrix.mul(dZ2,(desiredInputs)), m);}
else{dW2 =
Matrix.div(Matrix.mul(dZ2,Matrix.T(layers.get(layer).get
Input())), m);}
double[][]dB2 = Matrix.div(dZ2, m);
dA.set(layer, dA2);
dZ.set(layer, dZ2);
dW.set(layer, dW2);
dB.set(layer, dB2);
```

At the end of each iteration, new weights are determined for each layer, considering the alfa learning rate. Weight gain and new weight and bias values by layers are obtained:

```
for(int layer = 1; layer < NUMBER_LAYERS+1;
layer++){
double[][] W1 =
Matrix.sub(layers.get(layer).getWeights(),
Matrix.hadamardProduct(alfa, dW.get(layer)));
double[][] B1 = Matrix.sub(layers.get(layer).getBias(),
Matrix.hadamardProduct(alfa, dB.get(layer)));
W.set(layer,W1);
B.set(layer,B1);}
for(int layer = 1; layer < NUMBER_LAYERS+1;
layer++){
layers.get(layer).setWeights(W.get(layer));
layers.get(layer).setBias(B.get(layer));}
```

Regularization was not used in the implementation.

6. Results

A set of ten characters is used to test the network. Each character is represented by a 5x5 matrix, so the neural network has 25 inputs and ten outputs. Also, the network contains one hidden layer that has 45 neurons.

```
int numNeuron[] =  {45,10};
int inputNum[]  = {25,45};
```

For the network training process, the test data shown in Figure 4 are used. The Figure 4 also shows the desired outputs, for the given training data.

We see that each letter is represented by a 5x5 matrix (Figure 5). Matrix elements have a value of 0.0 or 1.0, depending on the appearance of a given character

Fig. 4 *Training data*

After the training, the network testing was made for the input data, which represent the letter B, with intentionally introduced noise (Figure 6).

A	B	C	D	E
0 0 1 0 0	1 1 1 1 1	0 1 1 1 1	1 1 1 1 0	1 1 1 1 1
0 1 0 1 0	1 0 0 0 1	1 0 0 0 0	1 0 0 0 1	1 0 0 0 0
0 1 0 1 0	1 1 1 1 0	1 0 0 0 0	1 0 0 0 1	1 1 1 1 0
1 1 1 1 1	1 0 0 0 1	1 0 0 0 0	1 0 0 0 1	1 0 0 0 0
1 0 0 0 1	1 1 1 1 1	0 1 1 1 1	1 1 1 1 0	1 1 1 1 1
F	G	H	I	J
1 1 1 1 1	0 1 1 1 1	1 0 0 0 1	0 0 1 0 0	1 1 1 1 1
1 0 0 0 0	1 0 0 0 0	1 0 0 0 1	0 0 1 0 0	0 0 0 0 1
1 1 1 1 0	1 0 0 1 1	1 1 1 1 1	0 0 1 0 0	0 0 0 0 1
1 0 0 0 0	1 0 0 0 1	1 0 0 0 1	0 0 1 0 0	1 0 0 0 1
1 0 0 0 0	0 1 1 1 0	1 0 0 0 1	0 0 1 0 0	0 1 1 1 0

Fig. 5 *Characters*

1 0.7 1 1 1
1 0.3 0 0 1
1 1 1 0.9 0
1 0 0 0 1
1 1 1 1 1

Fig. 6 *Test data*

The results shown in Figure 7 show that the network correctly performed the classification and recognized the letter B, since only the second element of the vector has a value of 1.0, and all other elements are approximately equal to zero.

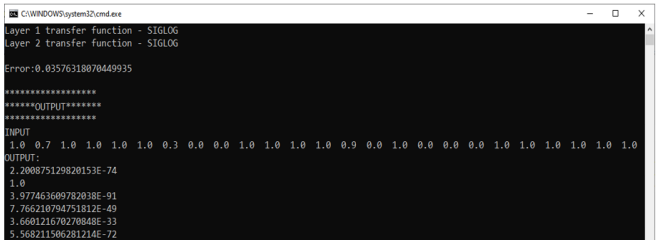


Fig. 7 *Results*

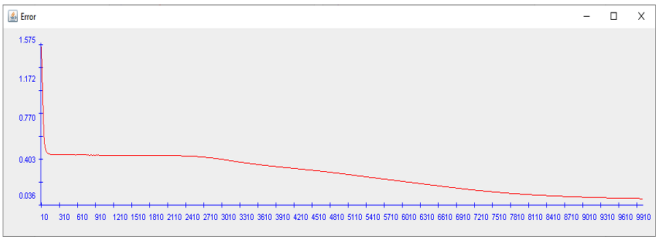


Fig. 8 *Dependence of the total error on the number of iterations*

Fig. 9 *Results for a neural network with two hidden layers*

In the case of a neural network, which has two hidden layers and which uses the tangent hyperbolic transfer function, the results are shown in Figure 9 and 10.

```
int numNeuron[] =  {45,25,10};
int inputNum[]   = {25,45,25};
```

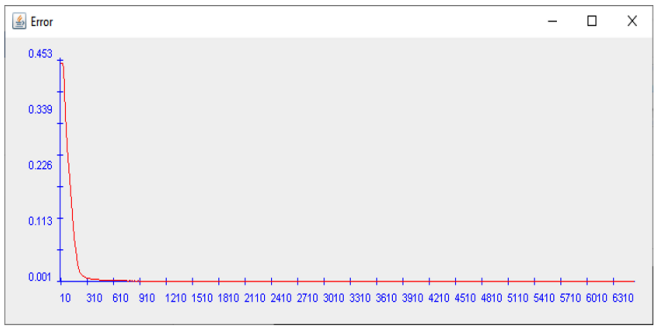


Fig. 10 *Error for a neural network with two hidden layers and Hyperbolic tangent transfer function*

7. Conclusion

This paper presents the neural network implementation into Java programming language. The implementation is generally performed for an arbitrary network having L layers and with the indicated number of neurons in each layer. The sigmoid function was used as a transfer function in the first example and tangent hyperbolic in second. A 10x25 matrix was used to train the net. Each row in the matrix represents a letter measuring 5x5. A matrix of desired outputs is also given. Network testing shows that the network correctly classifies the data and minimizes the error after some 300 iterations (Figure 10), in case of a neural network that has two hidden layers. In the case of a network with one hidden layer, the convergence is a bit slower. In the example, regularization was not used, so if the number of selected parameters is too large [5], the neural network may begin to describe noise, which may result in useless parameter adjustments.

Acknowledgment

This work is supported by: National Science Fund at the Bulgarian Ministry of Education and Science Project: "Synthesis of a dynamic model for assessing the psychological and physical impacts of excessive use of smart technologies", KP-06-N 32/4/07.12.2019.

8. References

1. Duch, W., Jankowski N, (2001). Transfer functions: hidden possibilities for better neural networks. 9th European Symposium on Artificial Neural Networks, Bruges, Belgium.

2. Maca P., Pech P., Pavlasek J., (2014) Comparing the selected transfer functions and local optimization methods for neural network flood runoff forecast. Mathematical Problems in Engineering. https://doi.org/10.1155/2014/782351

3. Yonaba, H., F. Anctil, and V. Fortin. (2010). Comparing sigmoid transfer functions for neural network multistep ahead streamflow forecasting. Journal of Hydrologic Engineering 15(4). DOI: 10.1061/(ASCE)HE.1943-5584.0000188

4. Rumelhart, D. et. al. (1996). Backpropagation: The basic theory. In P. Smolensky, M. C. Mozer, and D. E. Rumelhart(eds.) Mathematical Perspectives on Neural Networks. Erlbaum: NJ, 533-566.

5. Garvanova, M., Ivanov, V. (2020). Quality assessment of defocused image recovery algorithms. 3rd International Conference on Sensors, Signal and Image Processing, Prague, Czech Republic, doi: https://doi.org/10.1145/3441233.3441242.