Часопис из области економије, менаџмента и информатике
БизИнфо (БЛАЦЕ)
Година 2020, волумен 11, број 1, стр. 19-30

Journal of Economics, Management and Informatics
BizInfo (BLACE)
Year 2020, Volume 11, Number 1, pp. 19-30

Оригиналан научни рад/ Original scientific paper

УДК/UDC: 004.43.032.26JAVA

doi: 10.5937/bizinfo2001019J

Neural Network Implementation in Java

Implementacija Neuronske Mreže u Javi

Nenad Jovanović[a*], Bojan Vasović[b], Zoran Jovanović[b], Miloš Cvjetković[b]

[a] University of Pristina, Faculty of Technical Sciences, Kosovska Mitrovica, Serbia
[b] Academy of Professional Studies South Serbia, Department of Business Studies Blace, Serbia

Abstract: *Artificial neural networks are a powerful tool that engineers can use in variety of purposes. The most common tasks are classification and regression, as well as control, modeling and prediction. For more than three decades, the field of artificial neural networks has been the center of intensive research. A large number of software tools have been developed to train these types of networks, but there is still interest in implementing neural networks in different programming languages. This paper aims to present the implementation of an arbitrary neural network in the Java programming language.*
Keywords: *Artificial neural networks, Java, Neural network training, Neural networks prediction*

Sažetak: *Veštačke neuronske mreže moćan alat koji inženjeri mogu da koriste za razne svrhe. Najčešći zadaci su klasifikacija i regresija, kao i kontrola, modeliranje i predviđanje. Više od tri decenije polje veštačkih neuronskih mreža je centar intenzivnih istraživanja. Razvijen je veliki broj softverskih alata koji se koriste za obuku ovih vrsta mreža, ali i dalje postoji interesovanje za implementacijom neuronskih mreža u različitim programskim jezicima. Ovaj rad ima za cilj da predstavi implementaciju proizvoljne neuronske mreže u Java programskom jeziku.*
Ključne reči: *Veštačke neuronske mreže, Java, obuka neuronske mreže, predikcija neuronskih mreža*

1. Introduction

Artificial neural networks are systems inspired by biological neural networks. Neural networks learn to perform tasks by considering examples without programming of task-specific rules. For example, in image recognition, they can learn to identify

---

[*]Corresponding author.
E-mail address: nndjov@gmail.com

images by analyzing examples of images that have been manually tagged. They automatically generate identification characteristics from the examples they process.

The class of multilayer perceptron networks (MLP) (Bishop, 1995; Demuth et al., 2014) is one of the most frequently studied models of neural networks. The multilayer perceptron is a nonlinear model of data transfer, which organizes process units, neurons, into layers. Communication is possible only between neurons from different layers (Maca et al., 2014). Standard multilayer neural networks are capable of approximating any measurable function in any degree of accuracy (Hornik et al., 1989).

Despite the positive research results and the large number of papers, there is a need for presentation and clear methodological recommendations for neural network implementation into Java programming language.

The presented paper aims to present the implementation of a neural network with an arbitrary number of layers and an arbitrary number of neurons in those layers.

2. Network architecture

A neural network with l layers is observed (Figure 1). Inputs $x_1, x_2, ..., x_{nl}$ are connected to the first layer neurons. Each connection corresponds to the weight $w_{i,j}$. Each neuron represents a processor unit where neuron input to n_1 is calculated as:

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

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

$$z_i^1 = \sum_{i=0}^{n2} w_{1,i}^1 x_i \quad (3)$$

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

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

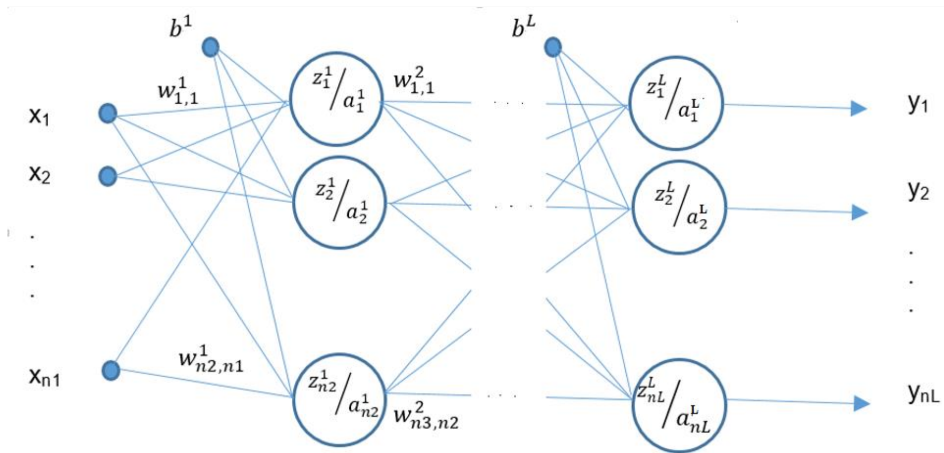


Figure 1. Neural network

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:

$$Z = WX^T + b \tag{5}$$

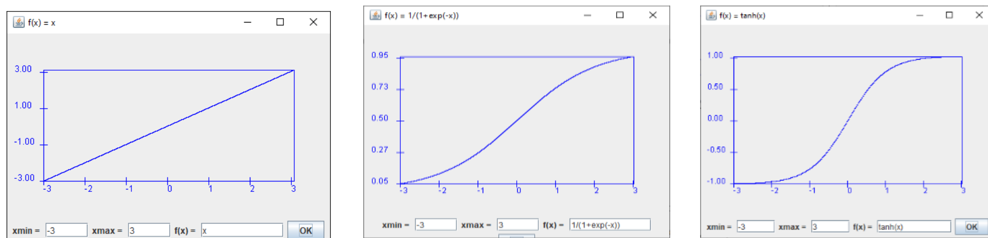
$$A = f(Z) \tag{6}$$

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.

The most commonly used transfer functions are:

- Linear
- Sigmoid
- Hyperbolic tangent

The distinctive output of a single bias input linear neuron is shown in Figure 2 (a).



Linear (a) Sigmoid (b) Hyperbolic tangent (c)

Figure 2. Transfer function

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:

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

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

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

In paper Duch and Jankowski (2001), 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 aims to analyze the influence of the selection transfer function and training algorithms on neural network flood runoff forecast (Maca et al., 2014). Sigmoid function, used in this paper is described in paper (Yonaba et al., 2010).

3. Algorithm Backpropagation

Rummelhart (1995) 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:

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

Where is y_k^i 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:

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

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:

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

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.

4. Implementation

The base classes in the system are `NeuralNetwork` 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.

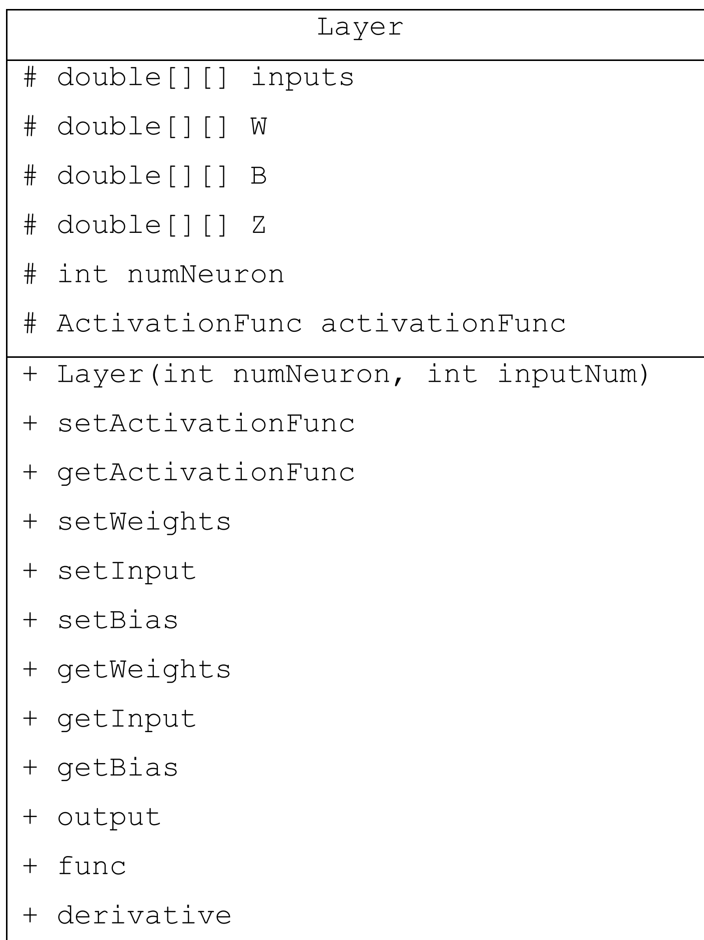


Figure 3. UML model of class `Layer`.

Parameter `NUMBER_LAYERS` defines the number of layers and it is placed inside constructor of class `NeuralNetwork`, 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).output(),
desiredOutputs);
} else if(error_type == ERROR2){
dApom = Matrix.sub(Matrix.div(Matrix.sub(1.0, desiredOutputs),
Matrix.sub(1.0,
     layers.get(NUMBER_LAYERS).output())),
Matrix.div(desiredOutputs,
     layers.get(NUMBER_LAYERS).output()));
}
double[][] dZpom = Matrix.hadamardProduct(dApom,

layers.get(NUMBER_LAYERS).derivative(layers.get(NUMBER_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).getInput())),
m);
}
double[][] dZ2pom = Matrica.sumByRows(dZ2);
double[][]dB2 = Matrix.div(dZ2pom, 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.

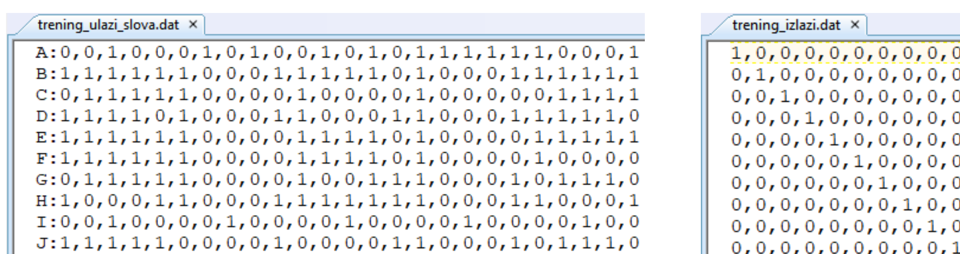
5. Results

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.



trening_ulazi_slova.dat

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

trening_izlazi.dat

```
1,0,0,0,0,0,0,0,0,0
0,1,0,0,0,0,0,0,0,0
0,0,1,0,0,0,0,0,0,0
0,0,0,1,0,0,0,0,0,0
0,0,0,0,1,0,0,0,0,0
0,0,0,0,0,1,0,0,0,0
0,0,0,0,0,0,1,0,0,0
0,0,0,0,0,0,0,1,0,0
0,0,0,0,0,0,0,0,1,0
0,0,0,0,0,0,0,0,0,1
```

Figure 4. 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.

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

Figure 5. Characters

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

```
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
```

Figure 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.

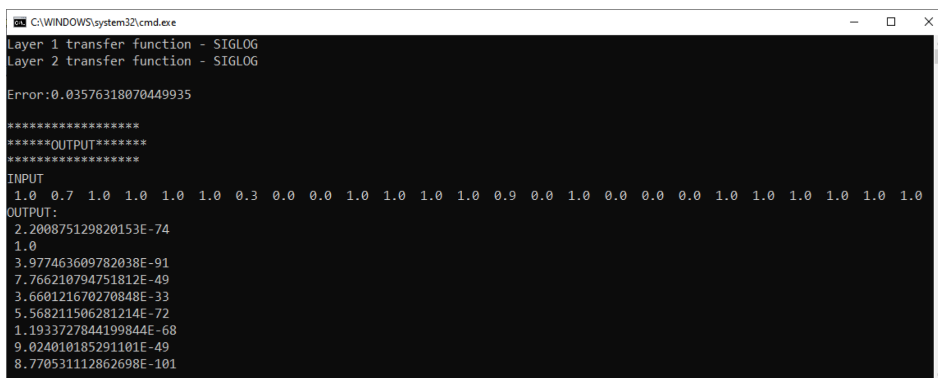


Figure 7. Results

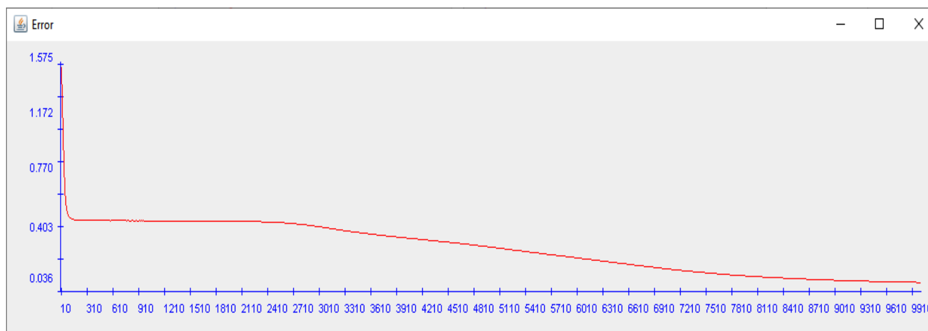


Figure 8. Dependence of the total error on the number of iterations

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 Figures 9 and 10.

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

```
C:\WINDOWS\system32\cmd.exe
Layer 1 transfer function - TANH
Layer 2 transfer function - TANH
Layer 3 transfer function - TANH

Error:7.000068426824189E-4

*****************
******OUTPUT*******
*****************
INPUT
 1.0  0.7  1.0  1.0  1.0  1.0  0.3  0.0  0.0  1.0  1.0  1.0  1.0  0.9  0.0  1.0  0.0  0.0  0.0  1.0  1.0  1.0  1.0  1.0
 1.0
OUTPUT:
 -0.9999660433826547
 1.0
 -0.9999999272761385
 0.05653769043753082
 0.9999206442074693
 -0.9998070857024085
 -0.9998058924599099
 -0.7680774895201294
 0.9998945860660678
 -0.9985446576925416
```

Figure 9. Results for a neural network with two hidden layers

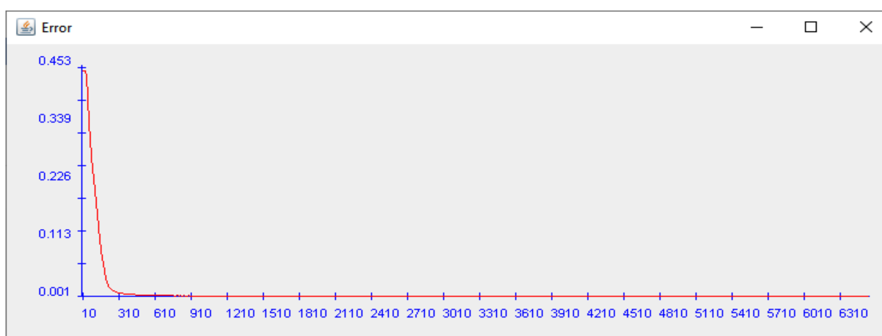


Figure 10. Error for a neural network with two hidden layers and Hyperbolic tangent transfer function

6. 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, the neural network may begin to describe noise, which may result in useless parameter adjustments.

References

Bishop, C. M. (1995). *Neural networks for pattern recognition*. Oxford university press.

Demuth, H. B., Beale, M. H., De Jess, O., & Hagan, M. T. (2014). *Neural network design*. Martin Hagan.

Duch, W., & Jankowski, N. (2001). Transfer functions: hidden possibilities for better neural networks. In *ESANN - European Symposium on Artificial Neural Networks Bruges* (pp. 81-94). Faculty of Physics, Astronomy and Informatics, Nicolaus Copernicus University, Poland.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, *2*(5), 359-366.

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*, *2014*, 1-10. http://dx.doi.org/10.1155/2014/782351

Rumelhart, D. E., Durbin, R., Golden, R., & Chauvin, Y. (1995). Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications*, 1-34.

Yonaba, H., Anctil, F., & Fortin, V. (2010). Comparing sigmoid transfer functions for neural network multistep ahead streamflow forecasting. *Journal of Hydrologic Engineering*, *15*(4), 275-283. https://doi.org/10.1061/(ASCE)HE.1943-5584.0000188

Received: 9 June, 2020; Accepted: 20 June, 2020
Rad je primljen: 09.06.2020; Prihvaćen za objavljivanje: 20.06.2020.