Method of computing gradient vector and Jacobean matrix in arbitrarily connected neural networks

Bodgan M. Wilamowski and Nicholas J. Cotton,
Electrical and Computer Engineering
Auburn University
Auburn, AL 36849 United States
wilam@ieee.org, cottonj@auburn.edu

Okyay Kaynak and Günhan Dündar
Electrical and Electronic Engineering
Bogazici University
Istanbul, Turkey
dundar@boun.edu.tr, okyay.kaynak@ boun.edu.tr

Abstract—***The paper shows that it fully connected neural networks are used then the same problem can be solved with less number of neurons and weights. Interestingly such networks are trained faster. The problem is that most of the neural networks terming algorithms are not suitable for such network. Presented algorithm and software allow training feedforwad neural networks with arbitrarily connected neurons in similar way as the SPICE program can analyze any circuit topology. When the second order algorithm is used (for which Jacobean must be calculated) solution is obtained about 100 times faster***

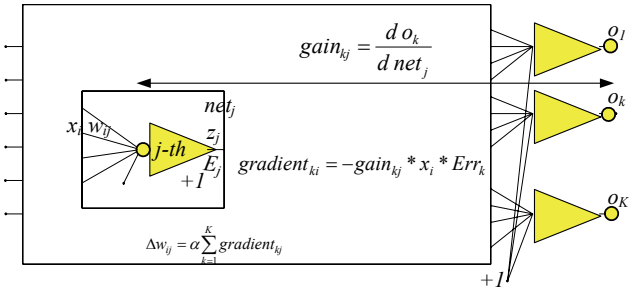

Fig. 1. Perturbation method for gradient evaluation

I. INTRODUCTION

Once the global error is defined the neural network training can be considered as a problem of finding a minimum of the error function. Various methods for neural network training were already developed ranging from random search through gradient-based methods. The best-known gradient method is the EBP - Error Back Propagation [2][5][6]. This method is characterized by very poor convergence. Several improvements for EBP were developed such as Quickprop algorithm, Resilient Error Back Propagation, Back percolation, Delta-bar-Delta etc. Much better results can be obtained using second order methods such as Newton or LM – Levenberg Marquet [2]. In the latter not only gradient but also Hessian or Jabobian should be found. The EBP algorithms propagate errors layer by layer from outputs to the inputs. If neural network has connections across layers the EBP algorithm becomes more complicated and even more difficult to adopt LM algorithm for such networks.

Another method of calculating gradients is forward calculations, also known as the perturbation method. In this case small changes on neuron net values are introduced and the resultant changes on the outputs are recorded. The perturbation method is especially useful for VLSI implemented network structure, where instead of analytical computation of gradients only various signal gains are measured [1][4].

EBP can work best for feedforward neural networks. Similar to EBP, the LM algorithm was adapted only to feedforward networks organized in layers (without weights across layers).

In this work we present alternative method for calculation of gradients and Jacobians to arbitrarily connected neural networks. The network topology is entered to the system in a similar way as it is done in SPICE program. It means that all neurons are listed together with node numbers where there neurons are connected. Each line of the list has the form:
neuron_name, model, out_node, inp_node1, inp_node2, ...inp_node.
Any neural network topology is possible as long as there are no feedback loops. This list is used to compute gradients and Jacobians at each iteration step

II. WHY USE FULLY CONNECTED NEURAL NETWORKS

Using a fully connected neural network can simplify the architecture and produce better results. When using traditional neural network with one hidden layer, an N bit parity problem would require $N+1$. This structure can be simplified by using a fully connected network with one hidden layer. The minimum number of neurons needed is shown in (1) [BMW Solving parity].

$$N_{\min} = \begin{cases} \frac{N-1}{2}+1 & \text{for odd number parity problems} \\ \frac{N}{2}+1 & \text{for even number parity problems} \end{cases} \quad (1)$$

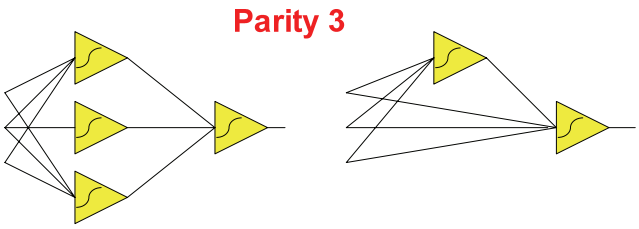

Fig. 2. Minimal network architectures to solve parity- 3 problems with traditional feedforward neural networks and with fully connected network.

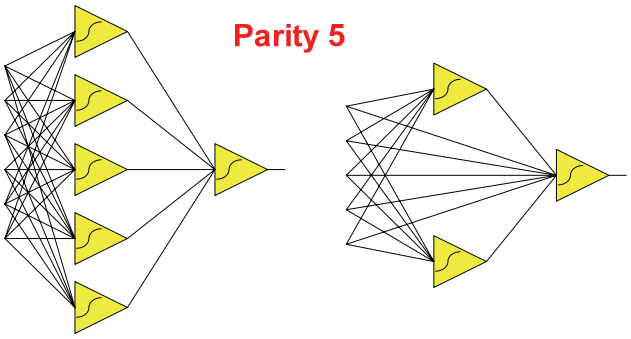

Fig. 3. Minimal network architectures to solve parity- 5 problems with traditional feedforward neural networks and with fully connected network.

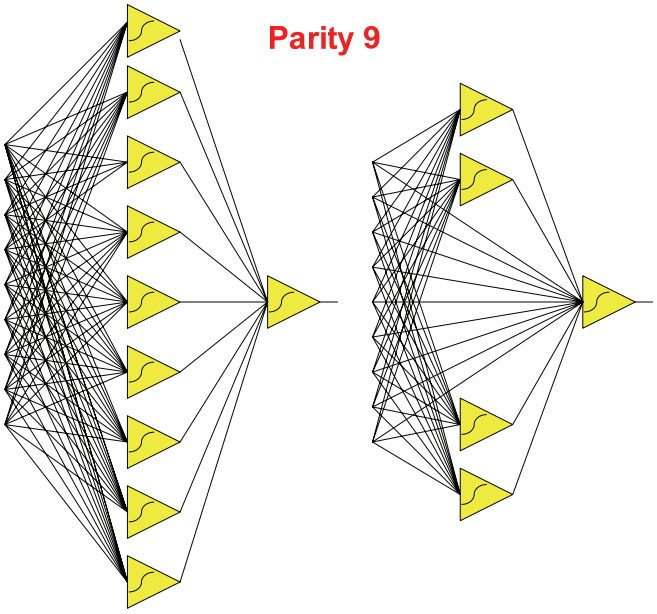

Fig. 4. Minimal network architectures to solve parity- 9 problems with traditional feedforward neural networks and with fully connected network.

This simple feedforward network is more efficient and quicker than the traditional method. If implemented in hardware this structure will have a drastically smaller footprint, and if implemented in software it will run significantly faster due to the fewer number of calculations needed.

Another example of where a feedforward fully connected network outperforms a traditionally connected network is in a nonlinear control system. Figure 5 is the desired control surface.

Using the network shown in Figure 6 the nonlinear surface could not be recreated. The network actually created a much more linear surface, shown in figure 7. This problem can be overcome by using a fully connected network with one less neuron. Figure 9 shows the new surface and Fig 8 shows the network used.

These examples show why this fully connected neuron structure should be adopted. This connection method does however require a different Algorithm to calculate the error as it is propagated back through the network.

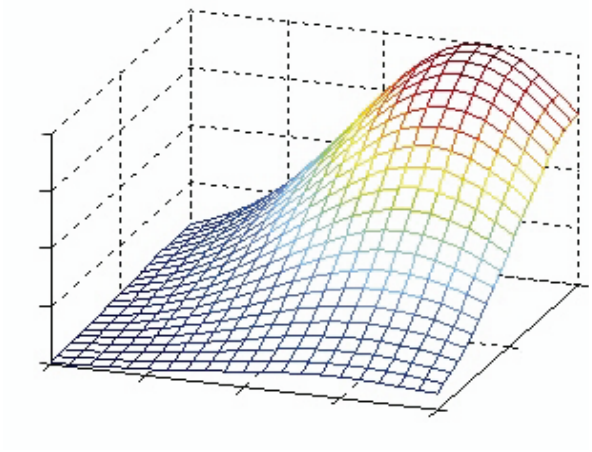

Fig 5 The desired nonlinear control surface that the neural networks are trained to.

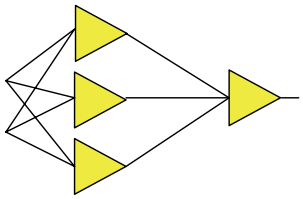

Fig 6 Traditionally connected 4 neural network

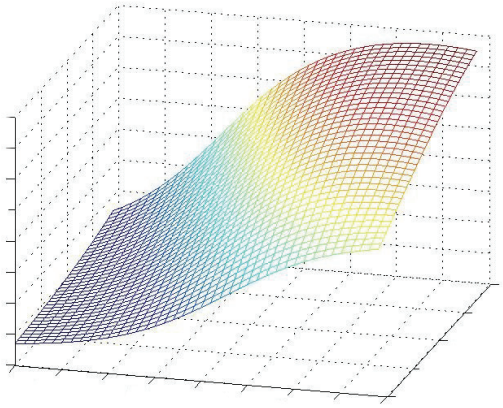

Fig 7 Surface produced by the network shown in Figure 6 Notice the surface lacks the desired curves.

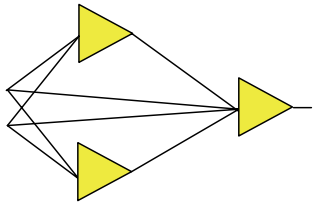

Fig. 8. Fully connected three neural network.

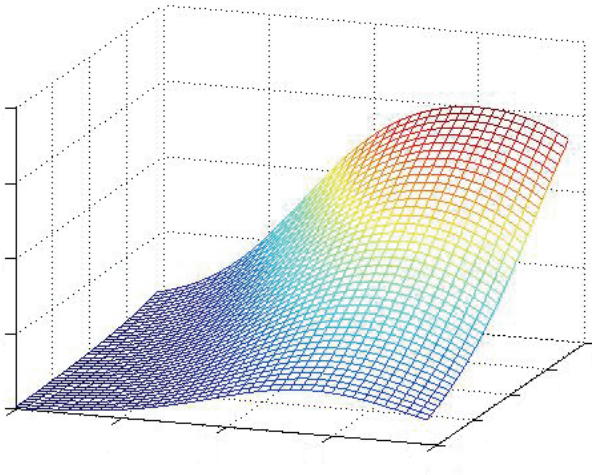

Fig. 9. Output surface from three neuron fully connected network.

III. COMPARISON OF FIRST AND SECOND ORDER ALGORITHMS

The first order methods use gradient computations. The most known algorithm is Error Back Propagation [EBP.] Second order methods can be represented by the Levenberg-Marquardt Algorithm [LM]

In EBP only gradient vector has to be calculated:

$$\mathbf{g} = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_n} \end{bmatrix} \tag{2}$$

while in LM algorithm all elements of the Jacobean matrix has to be found:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial e_{11}}{\partial w_1} & \frac{\partial e_{11}}{\partial w_2} & \cdots & \frac{\partial e_{11}}{\partial w_n} \\ \frac{\partial e_{21}}{\partial w_1} & \frac{\partial e_{21}}{\partial w_2} & \cdots & \frac{\partial e_{21}}{\partial w_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{M1}}{\partial w_1} & \frac{\partial e_{M1}}{\partial w_2} & \cdots & \frac{\partial e_{M1}}{\partial w_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{1P}}{\partial w_1} & \frac{\partial e_{1P}}{\partial w_2} & \cdots & \frac{\partial e_{1P}}{\partial w_N} \\ \frac{\partial e_{2P}}{\partial w_1} & \frac{\partial e_{2P}}{\partial w_2} & \cdots & \frac{\partial e_{2P}}{\partial w_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{MP}}{\partial w_1} & \frac{\partial e_{MP}}{\partial w_2} & \cdots & \frac{\partial e_{MP}}{\partial w_N} \end{bmatrix} \tag{3}$$

The length of the gradient vector is equal to the number of all weights in the network. Jacobean is a rectangular matrix with number of columns equal to number of weights (the same as gradient) and number of rows is equal to the product of number of training patterns times number of outputs. Most neural networks have only one output (for multiple output cases several neural networks with one output are used) but number of training patterns can be very large. This makes computation of Jacobean relatively complex. In this paper we are presenting method of calculating Jacobean matrix which has similar complexity to gradient calculation in EBP. Obviously LM algorithm would require more space to store values of the Jacobean matrix. Description of this method is given in section IV

In case of EBP weights change at every iteration are calculated

$$\Delta \mathbf{w} = -\alpha \mathbf{g} \tag{4}$$

Where α is the learning constant.

In Levenberg-Marquardt Algorithm (LM)

$$\Delta \mathbf{w} = -\left(\mathbf{J}_k^T \mathbf{J}_k + \mu \mathbf{I}\right)^{-1} \mathbf{g} \tag{5}$$

If Jacobean is known the gradient can be calculated using the formula:

$$\mathbf{g} = \mathbf{J}^T \mathbf{e} \tag{6}$$

where $\mathbf{e}$ is error vector of the length equal to the product of number of patterns times number of outputs.

IV. METHOD OF CALCULATION OF GRADIENT AND JACOBEAN

Let us consider a simple example of the neural network shown in Fig. 7. Network consists of three input nodes (1,2,3), three hidden nodes (4,5,6), and two output nodes (7,8). Please

notice that neurons must be numbered from inputs to outputs. It is important that for any given neuron, neuron inputs have always numbers smaller then the output number of the neuron.

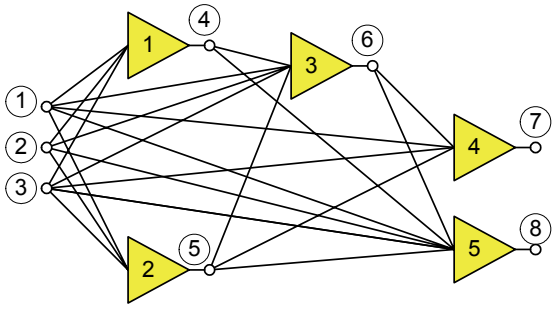

Fig. 10. Arbitrarily connected neural network.

For the network of Fig. 3 the network topology is listed

Network topology
N1 model 4 1 2 3
N2 model 5 1 2 3
N3 model 6 1 2 3 4 5
N4 model 7 1 3 5 6
N5 model 8 1 2 3 4 5 6 7

Notice that each line corresponds to one neuron. The first number is the neuron name and it is followed by model name for that neuron. The model includes each neuron's parameters such as type, activation function, and gain. The first number after the model is the output node which is followed by node numbers that act as inputs to that neuron. To save computer memory network topology can be stored in the form of linked list.

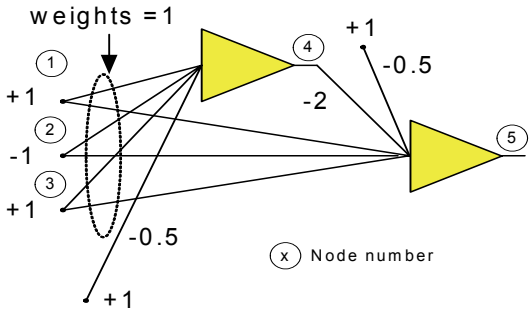

Fig. 11 Two neuron network displaying a parity-3 problem.

A. Forward computation

For each learning pattern computation is performed for all neurons. The following is pseudo code for the feedforward calculation

```
for each pattern p
  for each neuron n (starting from I to N)
    net=0;
    for each neuron input i
      net =net + w(i)*node(p, ind(i,n))
    end inputs
    node(p,n)=actfun(net)   //calculate neuron outputs
    slop(p,n)=(1-node(p,n)2) // for tanh activation function
  end neurons
  for each output neuron   (starting from N-O to N)
    err(p,o) =desired(p,o) – node(p,o)
  end output neurons
end patterns
```

where:
I is number of input nodes
O is number of output nodes
N is number of all nodes (inputs and neuron outputs)
P is number of patterns
node(p,n) – is the *P* * *N* array, which store signal level on each node. Nodes are network inputs and outputs of every neuron. Values are stored for every pattern.
w(i) – weights values associated with neurons inputs *i*
ind(i,n) – integer array describing network topology. It is being used to find input node number from the network topology.
slop(p,n)- is the *P* * *N* array, which store slops of activation functions for every node (neuron output) and every pattern. Values for the input nodes are not calculated (not defined)
desired(p,o) – is desired value for each pattern on each output
err(p,o) – is error for each pattern on each output

The result of the forward calculation is the rectangular *P*N* array *node(p,n)* where *P* is number of patterns and *N* is number of nodes (inputs plus neurons). The next step is calculation of the same size *P*N* array of slops (derivatives) of neuron activation functions. The forward computation is concluded with evaluation of output errors defined as desired value minus actual value. Results are stored in rectangular *P*O* array *err(p,o)* where *P* is number of patters and *O* is number of outputs

Algorithm is illustrated using a three bit parity problem with two bipolar neurons fully connected as shown in Figure 11. The forward computation is calculated for each neuron individually. The calculations will start for the first node that is not an input which in this case it is node four. The inputs are multiplied by their corresponding weights and summed. The sum will be 0.5 which is greater than zero resulting in a one being output to node four. The calculations for the next node will be calculated based on the previous results. The net value for the second neuron will be -1.5 resulting in an output of -1.

B. Gradient computation

After the forward calculations are made, the error and gradient are to be calculated. They cannot be calculated in a traditional layer by layer manner instead they are calculated neuron by neuron.

The following case is the gradient computation when the Jacobian is not needed.

$$gradient \Rightarrow \frac{d(TE)}{dw_i} = -\sum_{p=1}^{np} \Delta_{pn}\, x_{ip} \qquad (7)$$

```
errnode(n)= err(p,o)
for each pattern p
  for each output neuron o
    for each neuron n (starting from N down to I)
```

```
        delta(n)=slop(p,n)* errnode(n)
        for each neuron input  i
              errnode(ind(i,n))= errnode(ind(i,n))+
                delta(n)*w(i)
        end inputs
      end neurons
    end output neurons
  end patterns
```

where:

errnode(n) - is the vector with global errors on outputs of every neuron. Elements of this vector are used to sum errors.

delta(n)= is the vector with global errors on inputs of every neuron.

C. Jacobean computation

The complexity of Jacobean computations are not much different that complexity of gradient computations. The difference is that much more data has to be stored.

```
  for each pattern p
    for each output neuron o
      errnode(p,o,n ) = err(p,o)
      for each neuron n (starting from N down to I)
        delta(p,o,n ) = slop(p,n)* errnode(p,o,n )
          for each neuron input  i
            errnode(p,o,ind(i,n))=errnode(p,o,ind(i,n))+delta
              (p,o,n ) *w(i)
          end inputs
      end neurons
    end output neurons
  end patterns
```

where

errnode(p,o,n) - is the P*O*(N-I) array, which is summing errors on outputs of every neuron.

delta(p,o,n) is the P*O*(N-I) array, which is summing errors on inputs of every neuron.

As the result of this computation input errors delta are obtained for every pattern, every output and every neuron.

In order to calculate elements of Jacobean matrix for every neuron the input error delta is used to calculate Jacobean elements for every weight associated with this neuron.

```
  for each pattern p
    for each output neuron o
      for each neuron input i
        J(n,i)=delta(n) * node(p, ind(i,n ))
      end neuron inputs
    end output neurons
  end patterns
```

V SOFTWARE DESCRIPTION

MATLAB was used to create software that effectively trains arbitrarily connected neural networks. The program obtains its training parameters from several files including the data, input, the algorithm, and a linking file.

The data file contains the training set for the network. The user inputs his data into an $M * N$ array with each row being a set of inputs followed by the outputs. The topology determines how many of these columns to be used as outputs. If a neuron's output is not connected to the input of another neuron it is assumed to be an external output.

The software also allows the user to input initial weights into the input file or allow random weights to be generated. The weight matrix has the same irregular form as the network topology. With one more element representing biasing

Network weights

```
w4bias w41 w42 w43
w5bias w51 w52 w53
w6bias w61 w62 w63 w64 w65
w7bias w71 w73 w75 w76
w8bias w81 w82 w83 w84 w85 w86 w87
```

Initial weights could be randomly generated or they can be entered from the list of the network weights. Please notice that topology data and weight data have exactly the same structure. The difference is that node numbers are integer values while weights have real values.

```
// Network Architecture for Network shown in figure 10
N1 mbip 4 1 2 3
N2 mu 5 1 2 3
N3 mu 6 1 2 3 4 5
N4 mbip 7 1 3 5 6
N5 mlin 8 1 2 3 4 5 6 7
//Optional Initial Weights
W1  -3.65  3.65  3.65  1.83
W2  1  -27.6  27.6  27.6
W3  1.01  -97.2  81.5  5.89  46.2  69.8
W4  5.24  26.58  1.14  7.13 4.8
W5  2.20  1.98  4.88  2.25  3.12  8.47  1.74  8.97
//Model Definitions
.model mbip fun=bip, gain=2, der=0.01
.model mu fun=uni, gain=2, der=0.01
.model mlin fun=lin, gain=1, der=0.05
//Training Parameters
alpha =  1.5
maxite =10000
maxerr =0.01
gain =0.1
//LM training Parameters
mu = 0.01
mu_min =1e-50
mu_max= 1e+50
algorithm = 1
scal= 10
datafile = parity3.dat
```

Fig. 12 MATLAB input file.

VI. EXPERIMENTAL RESULTS

Several simulations were run to test the architectures and algorithms. Four examples are given demonstrating different ways to solve a parity-3 problem. The Levenburg Marquet outperformed the Error Back Propagation in multiple categories. The LM method converged to an error of less than 10^{-4} every time, on both architectures, in less than 30

iterations. EBP for the same accuracy of 10^{-4} requires about 5000 iterations. Figures 12 through 16 show the training algorithms converging twenty times. All of the networks were ran until they had twenty successful iterations.

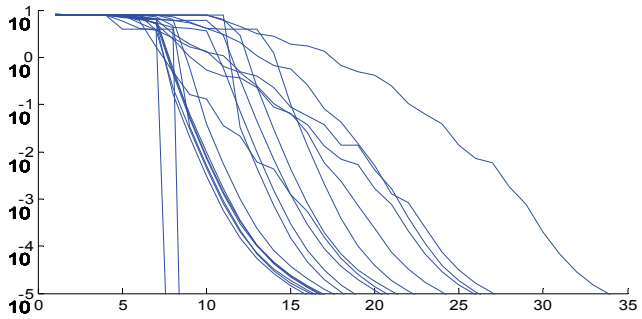

Fig. 12 LM training a traditionally connected network for a parity-3 problem.

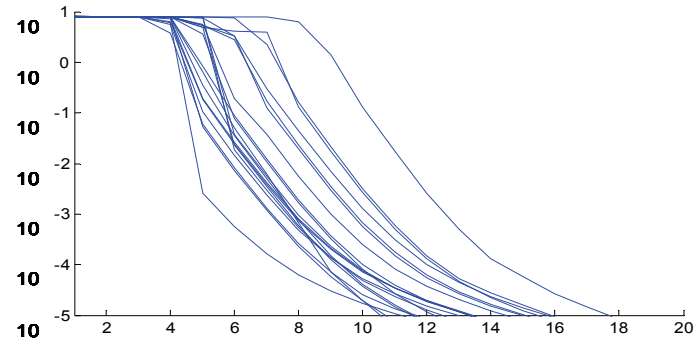

Fig. 13 LM training a fully connected network for a parity-3 problem.

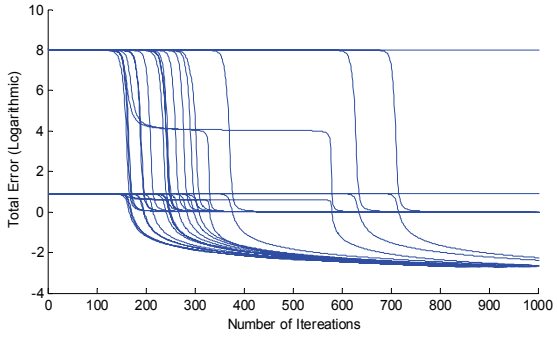

Fig. 16 EBP training a traditionally connected network for a parity-3 problem.

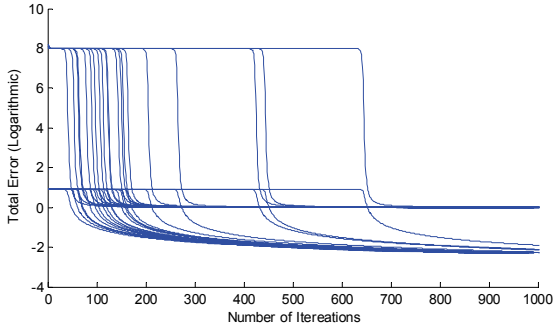

Fig. 17 EBP training a fully connected network for a parity-3 problem.

The LM fully connected network had no failures in twenty successful convergences, but the LM traditionally connected network failed twice. The EBP fully connected network failed four times, and the traditionally connected failed fourteen. Overall the LM training algorithm and the fully connected architecture outperformed EBP and traditionally connected networks.

Training algorithms comparison For parity-3 problem		
Algorithm	Success Rate	Failure Rate
LM fully connected	98%	2%
LM Traditionally connected	88%	12%
EBP fully conected	80%	20%
EBP Traditonally Connected	61%	31%

Fig. 18 Success rates for the different training algorithms and architectures.

VII. CONCLUSION

The algorithm of calculating gradient and Jacobean for fully connected neural network was presented. This way we were able to apply the LM for arbitrarily connected neural network and training time was reduced over 100 times. The paper also described very flexible method of entering network topology.

References

[1] Andersen, T. J. and Wilamowski, B.M. "A. Modified Regression Algorithm for Fast One Layer Neural Network Training", World Congress of Neural Networks, vol. 1, pp. 687-690, Washington DC, USA, July 17-21, 1995.

[2] Hagan, M. T. and Menhaj, M., "Training feedforward networks with the Marquardt algorithm", IEEE Transactions on Neural Networks, vol. **5**, no. 6, pp. 989-993, 1994.

[3] Rumenhart, D. E., Hinton, G. E. and Wiliams, R. J, "Learning representations by back-propagating errors", Nature, vol. 323, pp. 533-536, 1986

[4] Salvetti, A. and Wilamowski, B. M., "Introducing Stochastic Process within the Backpropagation Algorithm for Improved Convergence", presented at *ANNIE'94 - Artificial Neural Networks in Engineering*, St. Louis, Missouri, USA, November 13-16, 1994;

[5] Embrechts, M.J.; Benedek, S.; "Hybrid identification of nuclear power plant transients with artificial neural networks," *IEEE Transactions on Industrial Electronics* Volume 51, Issue 3, June 2004 pp. 686 – 693

[6] Da Silva, L.E.B.; Bose, B.K.; Pinto, J.O.P.; "Recurrent-neural-network-based implementation of a programmable cascaded low-pass filter used in stator flux synthesis of vector-controlled induction motor drive," *IEEE Transactions on Industrial Electronics* Volume 46, Issue 3, June 1999 pp. 662 - 665