Random Weights Search in Compressed Neural Networks Using Overdetermined Pseudoinverse

Milos Manic [1], Member, IEEE, Bogdan Wilamowski [2], Fellow member, IEEE

[1] College of Engineering, University of Idaho, 800 Park Blvd., Boise, ID 83712, USA, e-mail: misko@ieee.org
[2] College of Engineering, University of Idaho, 800 Park Blvd., Boise, ID 83712, USA, e-mail: wilam@ieee.org

***Abstract*—Proposed algorithm exhibits 2 significant advantages: easier hardware implementation and robust convergence. Proposed algorithm considers one hidden layer neural network architecture and consists of following major phases. First phase is reduction of weight set. Second phase is gradient calculation on such compressed network. Search for weights is done only in the input layer, while output layer is trained always with pseudo-inversion training. Algorithm is further improved with adaptive network parameters. Final algorithm behavior exhibits robust and fast convergence. Experimental results are illustrated by figures and tables.**

***Index Terms*—neural networks, random weight search, overdetermine, pseudoinverse.**

I. Introduction

The proposed algorithm in this paper is motivated by two different approaches to iterative search methods. First approach is neural network iterative gradient search method. Second approach encompasses flexible approximate techniques for neighborhood search.

Error Back Propagation (EBP) neural networks and gradient methods generally provide very good results [1]. Though presented a breakthrough in neural network research, backpropagation algorithm has number of disadvantages, such as oscillation, slow convergence, sensitivity to network parameters, etc.

Numerous researchers targeted these problems over the years. In order to smooth the process, Rumelhart et al. [2] have proposed weight adaptation, also formulated by Sejnowski and Rosenberg [3]. To speed up the process, Fahlman [4] proposed a quickprop algorithm. Wilamowski and Torvik [5] proposed solution towards robustness in terms of activation function derivative modification. Still, no universal solution towards robust and fast training algorithm has been found.

Neighborhood search methods are iterative procedures in which for each feasible solution a neighborhood for searched for next solution is defined. Most popular neighborhood search methods for finding an approximation to the minimum value of a real value function are gradient method, Simulated Annealing and Tabu search.

Roots of Tabu search go back to the 1970s, first presented by Glover and Hansen [6,7], independently. Further was formalized by Glover [8,9] and De Werra & Hertz [10]. It is a flexible approximation technique that uses memories or tabu lists to forbid moves that might lead to recently visited solutions, tabu solutions. Tabu lists can help to intensify the search in 'good' regions or to diversify the search toward unexplored regions by variable tabu list size.

Overview of combination of neural network gradient search and Tabu search with applications can be found in [11-15]. The proposed algorithm combines these two approaches to achieve fast and robust convergence. Furthermore, easy hardware implementation is facilitated by simplified gradient calculation. Main characteristics of the algorithm will be briefly overviewed.

The algorithm considers one hidden layer neural network architecture and has been tested on parity problems. The algorithm consists of following phases. First phase is reduction of weight set. Second phase performs the forward calculation where second layer gets trained with Pseudo-Inversion algorithm. Therefore, search for weights is done only in the input layer, while the output layer is automatically trained. In third phase, gradient is estimated on such compressed network, i.e. reduced weight space.

Gradient is estimated from set of overdetermined equations. Each equation defines one pass through the network (randomly generated weights resulting in certain error). Only forward calculation is performed vs. EBP that performs both forward and backpropagation. The reduced weight set participates in gradient calculation, vs. EBP that includes complete weight set. Finally, the second layer is trained automatically with pseudo-inversion training for all patterns at once, vs. EBP that performs both forward and backpropagation on all layers for single pattern at the time.

The rest of the paper is organized as following. Second section discusses weight space reduction. Third section explains steps of proposed algorithm, and is followed by gradient calculation explanation. The fifth section illustrates test examples. The sixth section concludes this paper with directives for future work. Last, seventh section contains references used in this paper.

II. Weight Space Reduction

The proposed algorithm considers 2 layer neural networks of $n+1$ neurons, where 1st layer consists of n-neurons and last layer of single neuron. Each 1st layer neuron has all inputs. This network is shown at Fig. 1.

The weight reduction portion of algorithm will be explained on this general 2-layer architecture.

For the network given by Fig. 1, the reduction goes as follows. Initial number of weights in general case is:

$$1^{st}\,layer + 2^{nd}\,layer = n*(m+1)+1*n+1 \qquad (1)$$

where $n+1$ is number of neurons, and m number of inputs for 1st layer neurons, $m+1$ is because of added bias.

Last layer ($n+1^{st}$ neuron) is always trained by pseudo-inversion training algorithm. This way, initial number of weights is reduced to $n*(m+1)$. From now on, only these weights from the 1st layer are considered.

Second reduction step reduces $n*(m+1)$ first layer weights to $n*m$ weights. Now the considered weights are only input weights of the first layer. Bias for these 1st layer neurons is fixed to 1.

III. Proposed Algorithms (Steps)

The algorithm steps go as follows:

1. *Assign Input* values. For the network with one hidden layer (Fig.1.), those are $n*m$ weights, where n is number of neurons in 1st layer, and m is number of their input weights (1st layer bias is fixed to 1). In case of simple 3-neuron network (Fig.2.), these are only 4 weights for inputs of first layer neurons set ($w_{k,11}$, $w_{k,21}$, $w_{k,12}$, $w_{k,22}$). Weights for the second layer neuron can be arbitrary chosen and do not influence the calculation.
2. *Initial Pass.* One pass through the network is done. This means *net*, *output*, and finally *total error* (TE) is calculated for all patterns. This is done through forward calculation through the first layer, for each pattern, for all patterns. Now Pseudo inversion takes charge of training of last layer weights. This 2nd layer hypersonic training is done for all patterns at once. The outcome of this step is the initial total error E_0.
3. *Assign input values for next iteration.* These are total error E_0 same weight set from the 1st step ($n*m$ weights in general case, or $w_{k,11}$, $w_{k,21}$, $w_{k,12}$, $w_{k,22}$ in case of n-to-1 network. For this starting weight set, gradient will be estimated in next step. Go to a next step (start with iterations).
4. *Start iterations (perform cycles).* Each iteration consists of certain number of cycles. Number of cycles should be at least twice as number of "changeable" weights. Each cycle performs the following. First, one set of increments ($\Delta w_{c,k}$) is created, one increment per changeable weight. These delta values are randomly created within specified radius (in cycle c in k-th iteration):

$$\Delta W_{c,k} = \left[\Delta w_{c,k,11}, \Delta w_{c,k,12}, \ldots, \Delta w_{c,k,1m}, \ldots, \Delta w_{c,k,m1}, \ldots, \Delta w_{c,k,mn}\right]_{1,m*n} \quad (2)$$

or, in different form:

$$\Delta W_{c,k} = \left[\Delta w_{c,k,ij}\right]_{\substack{i=1,n.\\ j=1,m.}}. \quad (3)$$

Secondly, new weight set for this cycle $W_{repr\,c,k}$ is created by adding these increments to a weight set initially defined in step 3 for the whole iteration:

$$W_{repr\,k,c} = W_{init\,k,c} + \Delta W_{k,c} = \sum_{i=1,n;\, j=1,m;} w_{init\,k,c,ij} + \Delta w_{k,c,ij}, \quad (4)$$

which can be represented by the following weight vector:

$$W_{repr\,c,k} = \left[w_{c,k,11}, w_{c,k,12}, \ldots, w_{c,k,1m}, \ldots, w_{c,k,m1}, \ldots, w_{c,k,mn}\right], \quad (5)$$

where c-cycle index (goes from 1 to C-number of cycles), k-iteration index, n-number of neurons in 1st layer, and m-number of inputs of each 1st layer neuron. Thirdly, one forward pass, same as initial pass in step2 is performed. This forward calculation returns total error. For each cycle, deltas and a respective total error for such weight modification are stored. Output from k-iteration of C cycles is the total error vector TE_k and delta matrix ΔW_k. Those are given as:

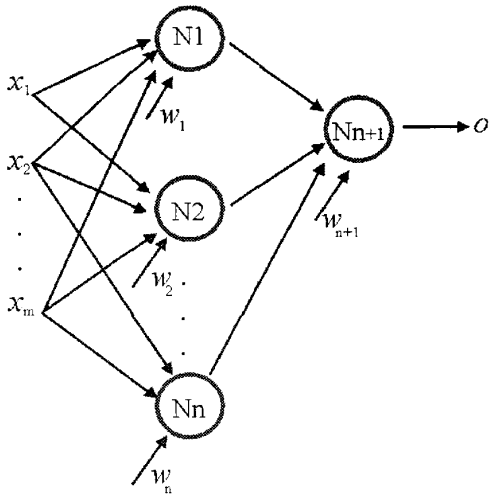


Fig. 1. General one hidden layer neural network architecture used in proposed algorithm.

$$TE_k = \left[TE_{k,11}\ TE_{k,21} \ldots TE_{k,m1} \ldots\ldots TE_{k,1n}\ TE_{k,2n} \ldots TE_{k,mn}\right]_{1\times m*n}, \quad (6)$$

or in different form:

$$TE_k = \left[TE_{k,ij}\right]_{\substack{i=1,n.\\ j=1,m.}}. \quad (7)$$

Delta matrix ΔW_k is given by:

$$\Delta_k = \begin{bmatrix} \Delta w_{1,k,11}\ \Delta w_{1,k,21} \ldots \Delta w_{1,k,m1} \ldots \Delta w_{1,k,1n}\ \Delta w_{1,k,2n} \ldots \Delta w_{1,k,mn} \\ \Delta w_{2,k,11}\ \Delta w_{2,k,21} \ldots \Delta w_{2,k,m1} \ldots \Delta w_{2,k,1n}\ \Delta w_{2,k,2n} \ldots \Delta w_{2,k,mn} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \Delta w_{C,k,11}\ \Delta w_{C,k,21} \ldots \Delta w_{C,k,m1} \ldots \Delta w_{C,k,1n}\ \Delta w_{C,k,2n} \ldots \Delta w_{C,k,mn} \end{bmatrix}_{C\times m*n} \quad (8)$$

or in different form:

$$\Delta W_k = \left[\Delta w_{c,k,ij}\right]_{\substack{c=1,C.\\ i=1,n.\\ j=1,m.}}. \quad (9)$$

5. *Calculate the gradient.* Gradient is numerically calculated from values from previous step. Such quasi-gradient is calculated in following way. First ΔE_k is calculated as:

$$\Delta E_k = TE_k - E_{k,0}. \quad (10)$$

where $E_{k,0}$ is the total error obtained before iterations began, i.e. TE with initial set of weights from step 2. Then, gradient is calculated as:

$$grad_k = MPInv(\Delta W_k) * \Delta E_k^T, \quad (11)$$

where $MPInv$ is Moore-Penrose pseudoinverse of ΔW_k.

6. Select the appropriate *learning constant alpha.* Different techniques for finding appropriate value for learning constant alpha can be applied. Those are alpha jump and random alpha probing. These techniques are explained separately. For each value of alpha, one forward pass is performed and obtained total error stored. Now, the alpha that resulted in minimal total error α_k is selected to be used in next step.
7. Now *new set of weights* is calculated based on the estimated gradient, previously selected alpha, and set of "changeable" weights from step 3. New set of weights is calculated based on following formula:

$$W_{k+1} = W_k - \alpha_k * grad_k. \quad (12)$$

Again, only "changeable" weights are updated (1st layer weights without bias). This set of weights is now ready to be used in next iteration. Go to next step where network parameters will be prepared for next iteration.

8. *Adapt network parameters:* network gain k, and search radius r. These steps are not tested in algorithm yet.
9. End of this iteration. *Start a new iteration* (go to Initial Pass (2nd step). Keep doing this until the criterion for the error is satisfied.
10. *End* of the Algorithm.

IV. Gradient Calculation

Gradient is calculated through overdetermined system of equations.

$$grad_k = MPInv(\Delta W_k) * \Delta E_k^T \quad (13)$$

More precisely, gradient is calculated as:

$$grad_k = MPInv(\Delta W_k) * \Delta E_k^T =$$

$$= MPInv\begin{bmatrix} \Delta w_{1,k,11} & \Delta w_{1,k,21} & \dots & \Delta w_{1,k,m1} & \dots & \Delta w_{1,k,1n} & \dots & \Delta w_{1,k,mn} \\ \Delta w_{2,k,11} & \Delta w_{2,k,21} & \dots & \Delta w_{2,k,m1} & \dots & \Delta w_{2,k,1n} & \dots & \Delta w_{2,k,mn} \\ \dots & & & & & & & \\ \Delta w_{C,k,11} & \Delta w_{C,k,21} & \dots & \Delta w_{C,k,m1} & \dots & \Delta w_{C,k,1n} & \dots & \Delta w_{C,k,mn} \end{bmatrix}_{C \times m*n} *$$

$$* \begin{bmatrix} \Delta E_{k,11} \\ \Delta E_{k,21} \\ \dots \\ \Delta E_{k,m1} \\ \dots\dots \\ \Delta E_{k,1n} \\ \Delta E_{k,2n} \\ \dots \\ \Delta E_{k,mn} \end{bmatrix}_{m*n \times 1} \quad (14)$$

where $MPInv$ is Moore-Penrose pseudoinverse of ΔW_k.

The reason for using Moore-Penrose pseudoinverse is the following.

If matrix ΔW_k is square and not singular, there is a single solution to inverse(ΔW_k), and then $MPInv(\Delta W_k)$ is an expensive way to compute ΔW_k^{-1} [16]. Unfortunately, problem here is that ΔW_k can have various dimensions and therefore does not necessarily have to have inverse matrix (ΔW_k matrix can be non-square matrix, can be singular matrix, etc.). ΔW_k is not square, or is square and singular, then ΔW_k^{-1} does not exist, which is exactly the case here. Matrix ΔW_k can have more rows than columns and is not of full rank. In these cases, $MPInv(\Delta W_k)$ has some though not all the properties of ΔW_k^{-1}.

V. Experimental Results

Parity problems, as one of most frequently used benchmarks for testing neural network training algorithms

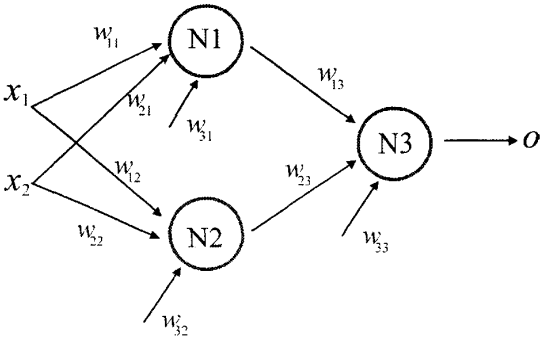


Fig. 2. Simple one hidden layer neural network architecture.

were chosen for testing of proposed algorithm. Activation function used was bipolar. Network parameters like initial *set of weights*, learning parameter *alpha*, and search radius r were randomly chosen.

Randomness of initial set of weights is in range (-1, +1); otherwise net values get saturated, causing algorithm oscillations or very slow convergence.

Tests were performed on Intel Pentium III 1GHz, with 512MB SDRAM memory, Windows 2000 Pro machine.

A. Parity 2 (XOR2) Problem

First problem was the XOR problem, tested on network architecture from Fig. 2. Examples show behavior with adaptive learning constant alpha, followed by acceleration achieved through adaptive search radius.

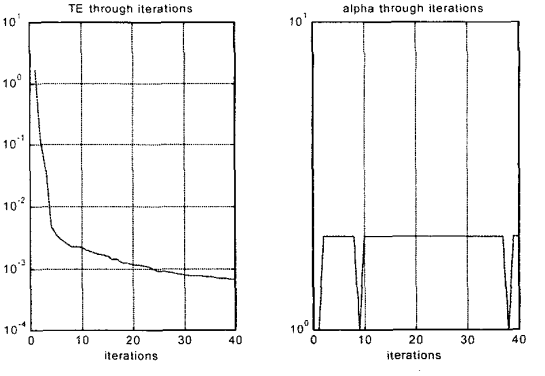


Fig. 3. Total error and alpha through iterations with initial parameters: gain_ini=1; alpha_ini=1; req_err=.5e-08; TERR=1e3; radius=.5; cycleNo=6;

Parameters through iterations:
iteration= 1 TERR=1.6598e+000 Alpha=1.0000e+000
iteration= 2 TERR=1.2256e-001 Alpha=2.0000e+000
……
iteration=38 TERR=7.1542e-004 Alpha=1.0000e+000
iteration=39 TERR=6.9201e-004 Alpha=2.0000e+000
iteration=40 TERR=6.8695e-004 Alpha=2.0000e+000
****
Elapsed time=2.2930e+000 (seconds)
CPU time=2.3030e+000 (seconds)
Stopwatch time=2.2830e+000 (seconds) (Time through iterations only)

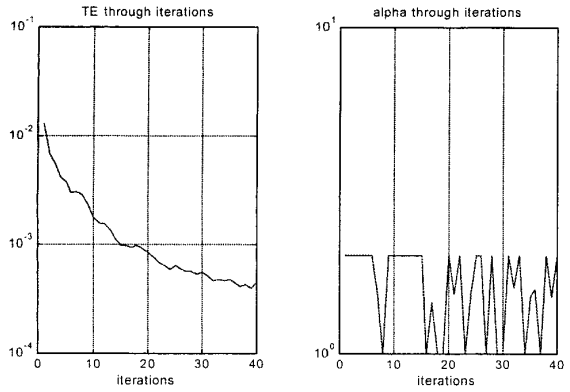


Fig. 4. Total error and alpha through iterations with initial parameters: gain_ini=1; alpha_ini=1; req_err=.5e-08; TERR=1e3; radius=.5; cycleNo=6;

Parameters through iterations:
iteration= 1 TERR=1.3056e-002 Alpha=2.0000e+000
iteration= 2 TERR=6.6966e-003 Alpha=2.0000e+000
iteration= 3 TERR=5.5186e-003 Alpha=2.0000e+000
iteration= 4 TERR=4.1863e-003 Alpha=2.0000e+000
iteration= 5 TERR=3.7718e-003 Alpha=2.0000e+000
.....
iteration=39 TERR=4.0052e-004 Alpha=1.4894e+000
iteration=40 TERR=4.4369e-004 Alpha=2.0000e+000
****
Elapsed time =2.4940e+000 (seconds)
CPU time=2.4940e+000 (seconds)
Stopwatch time=2.4740e+000 (seconds) (Time through iterations only):

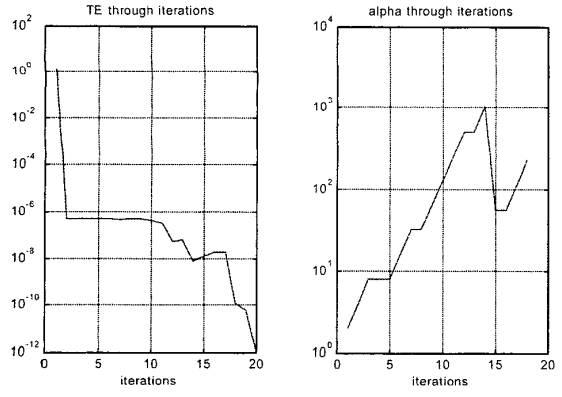


Fig. 5. Total error and alpha through iterations with initial parameters: gain_ini=1; alpha_ini=1; req_err=.5e-08; TERR=1e3; radius=.5; cycleNo=6;

Parameters through iterations:
iteration= 1;TE=1.3131e+000;alpha=2.0000e+000
iteration= 2;TE=4.9407e-007;alpha=4.0000e+000
iteration= 3;TE=4.9356e-007;alpha=8.0000e+000
.....
iteration=18;TE=1.2522e-010;alpha=2.2390e+002
iteration=19;TE=6.1849e-011;alpha=-6.4058e+001
iteration=20;TE=1.0387e-012;alpha=-1.2039e+001
****
Elapsed time=4.7100e-001 (seconds);
CPU time=4.7100e-001 (seconds)
Stopwatch time=4.6100e-001 (seconds) (Time through iterations only);

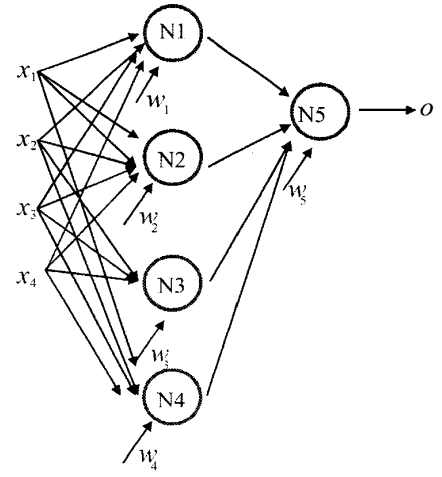


Fig. 7. Neural network architecture used for parity 4 testing.

First two examples (Fig. 3 & 4) illustrate algorithm's behavior with alpha reinitialized for every iteration. Convergence is smooth and takes about 40 iterations. Number of cycles is 8 per iteration.

With alpha cumulatively changed (not reinitialized at the beginning of each iteration), algorithm exhibits faster convergence (Fig. 5). Convergence is not smooth anymore, rather algorithm jumps following best found gradient, and it takes around half of iterations needed then with alpha reinitialized for every iteration.

Finally, algorithm's best performance is exhibited with adaptive radius introduced (Fig. 6).

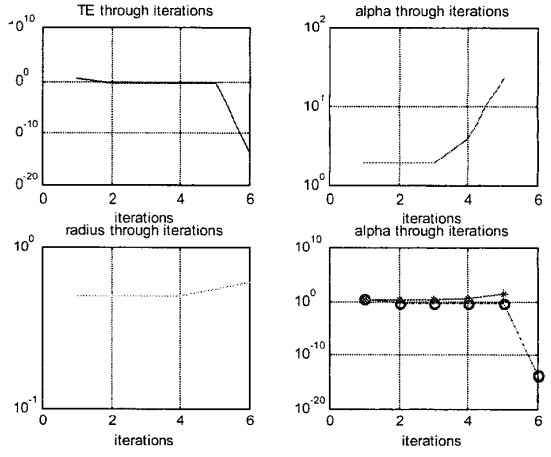
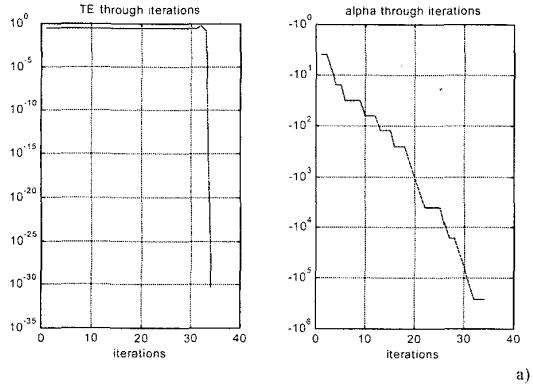


Fig. 6. Total error and alpha through iterations with initial parameters: gain=1; alpha=1; req_err=.5e-04; TERR=1e3; radius=.5; cycleNo=8;

Parameters through iterations:
iteration= 1;TERR=2.8769e+000;alpha=2.0000e+000;radius=5.0000e-001
iteration= 2;TERR=6.0568e-001;alpha=2.0000e+000;radius=5.0000e-001
iteration= 3;TERR=6.1051e-001;alpha=2.0000e+000;radius=5.0000e-001
iteration= 4;TERR=6.0624e-001;alpha=4.0000e+000;radius=5.0000e-001
iteration= 5;TERR=5.3890e-001;alpha=2.4000e+001;radius=5.5000e-001
iteration= 6;TERR=2.4647e-014;alpha=-1.1479e+003;radius=6.0500e-001
****
Elapsed time=3.3100e-001 (seconds);
CPU time=3.3100e-001 (seconds);
Stopwatch time=3.0000e-001 (seconds) (Time through iterations only);

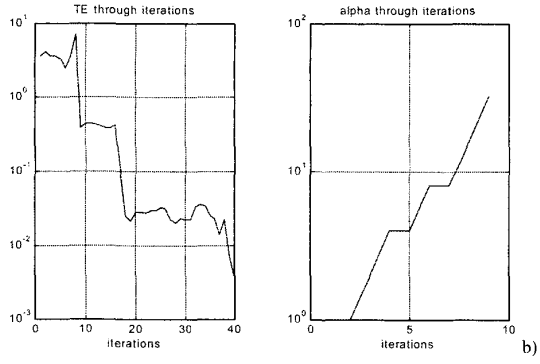
B. Parity 4 (XOR4) Problem

Second tested problem was parity 4. The architecture used for testing parity 4 problem is given by Fig.7. The algorithm proved all time convergence. However, the convergence rate and therefore minimum achievable total error significantly vary with respect to gain.



a)

Parameters through iterations:
iteration= 1 TE=3.0707e-001 alpha=-3.9466e+000
iteration= 2 TE=3.0707e-001 alpha=-3.9466e+000
iteration= 3 TE=3.0707e-001 alpha=-7.8933e+000
.....
iteration=32 TE=5.6896e-001 alpha=-2.5865e+005
iteration=33 TE=1.5690e-001 alpha=-2.5865e+005
iteration=34 TE=5.0536e-031 alpha=-2.5865e+005
****
Elapsed time =4.9870e+000 (seconds)
CPU time =4.9870e+000 (seconds)
Stopwatch time=4.9170e+000 (seconds) (Time through iterations only)



b)

Parameters through iterations:
iteration= 1 TE=3.5913e+000 alpha=1.0000e+000
iteration= 2 TE=4.0880e+000 alpha=1.0000e+000
iteration= 3 TE=3.5935e+000 alpha=2.0000e+000
.....
iteration=39 TE=7.4661e-003 alpha=-5.1760e+008
iteration=40 TE=3.6879e-003 alpha=-1.0352e+009
****
Elapsed time =5.8090e+000 (seconds)
CPU time =5.8190e+000 (seconds)
Stopwatch time=5.7580e+000 (seconds) (through iterations only)

Fig. 8. Total error and alpha through iterations with initial parameters:
gain_ini=1; alpha_ini=1; req_err=.5e-10;
TERR=1e3; radius=.5; cycleNo=6;

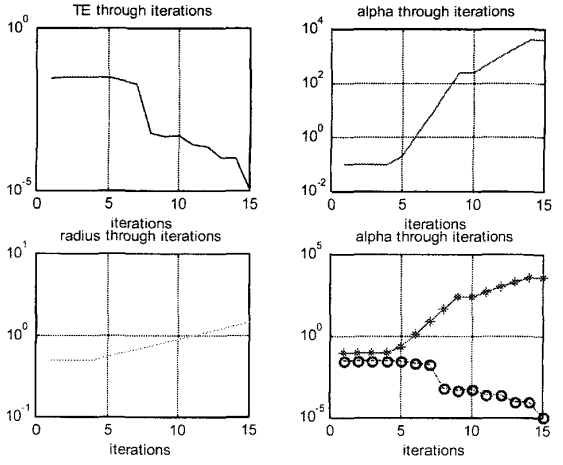


Fig. 9. With adaptive radius
Initial network parameters: gain=.2; alpha=.1; req_err=.5e-04; TERR=1e3;
radius=.5; cycleNo=8;

Parameters through iterations:
iteration= 1; TERR=3.0072e-002 alpha=1.0000e-01;radius=5.0000e-01
iteration= 2; TERR=3.0142e-002 alpha=1.0000e-01;radius=5.0000e-01
iteration= 3; TERR=3.0186e-002 alpha=1.0000e-01;radius=5.0000e-01
.....
iteration=13; TERR=1.0266e-004; alpha=2.0736e+003;radius=1.1790e+00
iteration=14; TERR=1.0001e-004; alpha=4.1472e+003;radius=1.2969e+00
iteration=15; TERR=1.1246e-005; alpha=4.1472e+003;radius=1.4266e+00
****
Elapsed time=2.8340e+000 (seconds)
CPU time=2.8440e+000 (seconds)
Stopwatch time=2.7840e+000 (seconds) (Time through iterations only)

Following examples (Fig.8. a & b) illustrate convergence with alpha cumulatively changed. Convergence is smooth and it takes about 40 iterations, with 8 cycles each.

However, with introduced adaptive search radius, algorithm achieves its best performance, in less than half as many iterations compared to previous examples (Fig. 9). This is the example with adaptive search radius and learning constant alpha. Algorithm converges nicely and smoothly.

VI. Conclusion

Apart from novel approach, certain modifications were made. However, "plain" method did not result in all-time convergence. Adaptive network parameters provided expected improvements.

Further research would include different directions. Dependence between algorithm's convergence and other parameters should be established. Those parameters are: degree of overdeterminance, learning constant, search radius, network gain, etc. Interdependence of these parameters would be a next step, and that interdependence should be formulated in form of certain rules. If these modifications prove further results, algorithm will be tested on more difficult problems.

VII. References

[1] J.M. Zurada. *Artificial Neural Systems*, PWS Publishing Company, St. Paul, MN, 1995.

[2] Rumelhart, D.E., Hinton, G.E., and Williams, R.J., *Learning internal representation by error propagation*, Parallel Distributed Processing, Vol.1, pp.318-362, MIT Press, Cambridge, MA., 1986

[3] Sejnowski T.J., Rosenberg, C.R., *Parallel networks that learn to pronounce English text*, Complex Systems 1:145-168, 1987.

[4] Fahlman S.E., "*Faster-learning variations on backpropagation: An empirical study*", Proceedings of the Connectionist Models Summer School, eds. D.Touretzky, G. Hinton, and T.Sejnowski, Morgan Kaufmann, San Mateo, CA, 1988.

[5] Wilamowski, M., Torvik, L., "*Modification of gradient computation in the back propagation algorithm*", Artificial Neural Network in Engineering, Nov., St.Louis, Missou., 1993.

[6] Glover, F., "Future paths for integer programming and links to artificial intelligence", *Computers & Operations Research* 13, pp.533-549, 1986.

[7] Hansen, P., "*The steepest ascent mildest descent heuristic for combinatorial programming*", Talk presented at the Congresss on Numerical Methods in Combinatorial Optimization, Capri, 1986.

[8] Glover, F., *Tabu Search:part I.* ORSA Journal on Computing 1, pp.190-206, 1989.

[9] Glover, F., *Tabu Search:part II.* ORSA Journal on Computing 2, pp.4-32, 1990.

[10] de Werra, D., Hertz, A., "*Tabu search techniques: a tutorial and an application to neural networks*", OR Spektrum 11, pp.131-141, 1989.

[11] Battiti, R; Tecchiolli, G, "*Training Neural Nets with the Reactive Tabu Search*", IEEE transactions on neural networks, no. 5, pp. 1185-1201, 1995.

[12] Vaithyanathan, S; Burke, L.I, Magent, M.A., "*Massively parallel analog tabu search using neural networks applied to simple plant location problems*", European Journ. of Operat. Research, 93, no. 2, pp. 317-321, 1996.

[13] Magent, M.A., *Combining neural networks and TABU search in a fast neural network simulation for combinatorial optimization*, Thesis manuscript, Lehigh University, 1997.

[14] Pham D.T., Karaboga, D., *Intelligent optimisation techniques : genetic algorithms, tabu search, simulated annealing and neural networks*, London ; New York : Springer, 2000.

[15] Laguna, M., Marti, R., "*Neural network prediction in a system for optimizing simulations*", IIE Transactions 34, no. 3, pp. 273-282, 2002.

[16] Rao, C., Mitra, S., *Generalized inverse of matrices and its applications*, John Wiley & Sons, New York, 1971.