Neural, Parallel & Scientific Computations 5 (1997) 549-562

Structured Artificial Neural Networks for Fast Batch LMS Algorithms

K. Goulianas[1,2], M. Adamopoulos[1,2], and K. G. Margaritis[1]
[1]Department of Informatics, University of Macedonia, Thessaloniki, Greece
[2]Department of Informatics, Technological Educational Institute of Thessaloniki, Greece

Abstract

This paper describes an artificial neural network architecture, which implements batch-LMS algorithms. The patterns are stored in the network in the form of interconnection weights, while the convergence of the learning procedure is based on Steepest Descent algorithm. The objective is to find a set of weights, so that the sum of the squares of the errors is minimized. In this paper we show that by using an adaptive learning rate, the network implements the Steepest Descent method of numerical linear algebra for solving linear systems of equations. With the application of Delta Rule in the learning procedure the system of normal equations is solved, and the set of weights generated by the learning procedure satisfies convergence to the optimal least squares solution for all kinds of systems (normal, overdetermined or underdetermined), while the number of iterations needed for convergence is significantly decreased. Extension to matrix inversion is also presented and convergence behaviour and performance by computer simulations are discussed.

1. INTRODUCTION

Feedforward artificial neural networks have been studied extensively and have been proved capable of solving a wide variety of problems [5],[12],[17]. Most applications of these networks use some type of training procedure in order to utilise associations of input patterns to output patterns. These relations can be either auto-associative or hetero-associative, i.e. they correlate a set of patterns either to themselves or to another set of patterns.

Recently, many feedforward neural networks architectures with linear neurons for solving systems of linear equations and matrix algebra problems have been studied and implemented [10]-[11],[13],[19]-[20]. The matrix algebra problem is represented with some architecture and a training algorithm (usually Back-Propagation [16]) is used, so that the network matches the desired patterns, and the solution to the problem is given by the trained weights of the network. In the above architectures, the networks of [10]-[11],[13],[19] are two-dimensional (2-D), whereas the network [20] is three-dimensional (3-D), a fact that introduces a higher degree of parallelism. When used for linear system equation solving, the networks in [10]-[11] use a simple architecture, with n input neurons and 1 output neuron, whereas the network [13] (called Orthogonalized Back Propagation) uses n input neurons, a hidden layer with m neurons, and 1 output neuron. The networks [19]-[20] are applied for finding the inverse of matrix A and a network with n input neurons, a hidden layer with n neurons, and n output neurons is used in [19], while the network [20] uses n networks, each one having n input neurons, a hidden layer with n neurons, and n output neurons. The lines of the matrix

Received May 21, 1997

involved in the matrix algebra problem are presented to the networks [10]-[11], while in [13],[19]-[20] the matrix involved is stored in the network in the form of interconnection weights and linearly independent input vectors are applied. The network in [11] is limited to linear systems with matrix A assumed to be Symmetric and Positive Definite (SPD), in [19]-[20] matrix A is assumed to be square and the network is used for finding the inverse of matrix A, while in [10],[13] matrix A can be of any kind. All the above architectures use the linear activation function, while in the training procedures the learning rate can be stochastic [11],[19] or adaptive [10],[13],[20].

In this paper the development of a simple two-layer feedforward neural network with linear neuron functions is studied. The emphasis is placed in the fact that the proposed architecture solves all types of linear equation systems, since the learning procedure generates the system of normal equations yielding a least square solution. The procedure used is the error function gradient, and it takes two forms: in the first alternative, the learning rate or stepsize α is determined in a heuristic way, hence the Heuristic Steepest Descent (HSD) algorithm; in the second and better alternative the stepsize α is adaptive yielding the Adaptive Steepest Descent (ASD) algorithm. The network is trained with vector b as targets, and without using the matrix A as inputs, since the matrix A with the patterns is stored as weights in the network. The trainable weights i.e. the vector x are updated, until the network converges, i.e. the outputs of the network match the desired patterns, and the final trainable weights give the solution of the problem.

The material is organised as follows. In Section 2 we formulate the problem along with the optimal solution (using pseudo-inverse), and discuss various algorithms and associated neural network architectures for obtaining estimates of the optimal solution vector x, such as the Least Mean Square (LMS) algorithm, both the incremental and batch version, [1],[8],[10],[21]. In Section 3, we introduce the new architecture along with the heuristic (equivalent to batch-LMS) and the adaptive learning procedure, and discuss convergence issues. The extension of the method for matrix inversion is also introduced. In Section 4, we study a few examples for systems of various dimensions comparing the convergence behaviour of the above three methods, as it concerns convergence, the number of iterations, and we compare the solutions of the above methods to the optimal least squares solution. along with some experimental results. Finally, in Section 5 we draw some final conclusions.

2. NEURAL NETWORK ALGORITHMS FOR SOLVING SYSTEMS OF LINEAR EQUATIONS

Given a matrix $A \in R^{m \times n}$ and a vector $b \in R^m$ the task is to find a vector $x \in R^n$, such that $Ax = b$. The minimization of the mean square error, or the cost function

$$E(x)=\sum_{i=1}^{m} E_i(x)=\sum_{i=1}^{m} \tfrac{1}{2}(a_i^T x - b_i)^2 = \tfrac{1}{2}\sum_{i=1}^{m}(a_i^T x - b_i)^2 = \tfrac{1}{2}(Ax-b)^T(Ax-b)=\tfrac{1}{2}\|Ax-b\|^2 \quad (1)$$

is the criterion of optimality. Using a general gradient approach for minimization of a function, the system can be mapped to the equation

$$\nabla E(x)=A^T(Ax-b)=0 \quad (2)$$

which is the corresponding system of normal equations $A^T Ax=A^T b$ or $Cx = d$, with $C = A^T A$ and $d=A^T b$, where $C \in R^{n \times n}$ is positive definite and symmetric (i.e. $x^T Cx > 0$ for all non-zero $x \in R^n$). For such systems the problem is equivalent to minimizing the functional $E(x)=\tfrac{1}{2}x^T Cx - d^T x$. The minimum value of $E(x)$ is $-\tfrac{1}{2}d^T C^{-1} d$ achieved by setting $x = C^{-1} d$. Thus, minimising $E(x)$ and solving (2) are

equivalent problems. The optimal least mean square solution of system (2) by using the Moore-Penrose generalised inverse A^+ is defined as

$$x = A^+ b \tag{3}$$

For the overdetermined system $Ax = b$, with A a $(m \times n)$ matrix, the resulting solution is the Least Squares solution. If $r(A) = n$, then the least squares solution is unique, given by

$$x = A^+ b, \text{ with } A^+ = (A^T A)^{-1} A^T \tag{4}$$

where A^+ is the pseudo-Inverse of the $(m \times n)$ matrix A, and satisfies the Moore-Penrose conditions of A^+ [4].

For the square or normal system $Ax = b$, with a square $(n \times n)$ non-singular full rank coefficient matrix A, the solution of (2) is unique. Similarly, the Generalised Inverse of the $(n \times n)$ matrix A, A^+ defined in (4) is equal to A^{-1}.

For the underdetermined system $Ax = b$, with A a $(m \times n)$ matrix, the resulting solution is one from the infinite least squares solutions.

One approach is to use a hetero-associative one-layer feedforward neural network, with n inputs and one output neuron (as shown in Figure 1), a special case of Kohonen Linear Associative Memory [6]. A better approach is to use the Moore-Penrose generalised inverse A^+, a special case of Kohonen Optimal Linear Associative Memory [7]-[8]. The Moore-Penrose generalised inverse A^+ is calculated, using for example the Greville's recursive algorithm if $m \geq n$, and the interconnection weights between the input layer and the output neuron, i.e. the solution of equation (2) defined from equation (3), or (4) are encoded to the network, yielding the optimal least mean square correlation of A and b.

The above scheme is easy to be implemented, but it needs off-line calculation of the pseudoinverse. However, pseudoinverse can be adaptivelly approximated with the network in Figure 1. The m lines of matrix A are presented to the network in a cyclical fashion, and the following LMS learning iterative algorithm [7],[10],[21] is used for adapting the weights at step $t+1$ (after the i^{th} line of matrix A has been presented to the input layer):

$$x^{(t+1)} = x^{(t)} - \alpha(a_i^T x^{(t)} - b_i)a_i^T = x^{(t)} - \alpha \nabla E_i(x^{(t)}) \tag{5}$$

with $\nabla E_i(x^{(t)})$ the instantaneous gradient defined as $\nabla E_i(x^{(t)}) = (a_i^T x^{(t)} - b_i)a_i^T$, derived from $E_i(x^{(t)}) = \frac{1}{2}(a_i^T x^{(t)} - b_i)^2$, where $E_i(x^{(t)})$ is the cost function for the i^{th} pattern, a_i^T the i^{th} line of matrix A, and α the learning rate.

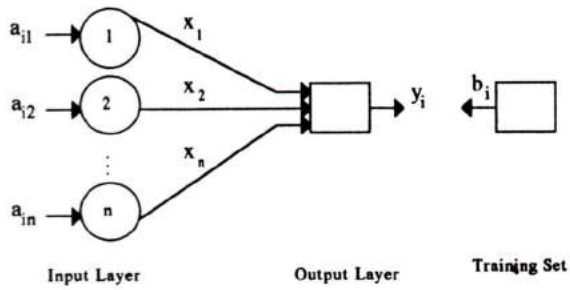

Figure 1. One-Layer Structured ANN for Linear System of Equation Solving

This procedure is repeated for a number of iterations, until the error between calculated and desired outputs is within acceptable limits, forcing the values x_j, $j = 1,2,\ldots,n$ of vector x to converge to an approximate solution of the system (1). It has been shown [7],[21] that if the learning rate is small and fixed, then the consequence of the vectors x generated by the LMS algorithm converge to some matrix close to the optimal solution of the system (1).

The batch version of the LMS algorithm [1],[5] uses the total gradient $\nabla E(x^{(t)})$ instead of the approximate $\nabla E_i(x^{(t)})$. In this version, the contributions to the gradient $\nabla E_i(x^{(t)})$ from the m different patterns (the m lines of matrix A) are calculated and summed in order to obtain the total gradient $\nabla E(x^{(t)})$. The learning procedure for $x_k^{(t+1)}$, $k = 1,2,\ldots,n$ at time $t+1$ has the form $x_k^{(t+1)} = x_k^{(t)} + \alpha\delta^{(t)}$, with $\delta^{(t)}$ defined as $\delta^{(t)} = A^T(b - Ax^{(t)}) = A^T b - A^T Ax^{(t)}$, and the adaptation of the weights has the form

$$x^{(t+1)} = x^{(t)} + \alpha\delta^{(t)} = x^{(t)} + \alpha A^T(b - Ax^{(t)}) = x^{(t)} - \alpha A^T(Ax^{(t)} - b) = x^{(t)} - \alpha\nabla E(x^{(t)}) \qquad (6)$$

The values x_k, $k = 1,2,\ldots,n$ of vector x converge to the solution of the system of normal equations (2), which minimises the residual error.

3. NEURAL NETWORK ARCHITECTURE FOR FAST BATCH-LMS

The LMS learning algorithm is an inexact version of the deterministic gradient descent algorithm. The gradient of the objective function E(x) is approximated by the gradient of an individual error function $E_i(x)$ for pattern i. Thus, the weight vector x is updated along the gradient direction of $E_i(x)$, a crude gradient estimate in place of the true gradient of E(x), which is difficult to obtain, since it involves averaging the instantaneous gradients associated with all patterns (the lines of matrix A). As a result, the total error E(x) may not decrease, (in some cases it may increase) and the convergence is very slow. In addition, the procedure converges to an approximate solution of the system (2).

We propose a new architecture, better than LMS for the same values of the learning rate. The network representing (2) is shown in Figure 2. As it can be seen, it is a two-layer structured neural network consisting of two layers: a hidden layer with n neurons, each one connected with the neuron of the input layer, and an output layer with m neurons, fully connected with the hidden layer (the input layer with 1 neuron is not considered as a distinct layer). As an alternative, instead of using the input neuron, we could use a bias threshold connected to every neuron at the hidden layer and discard the input neuron.

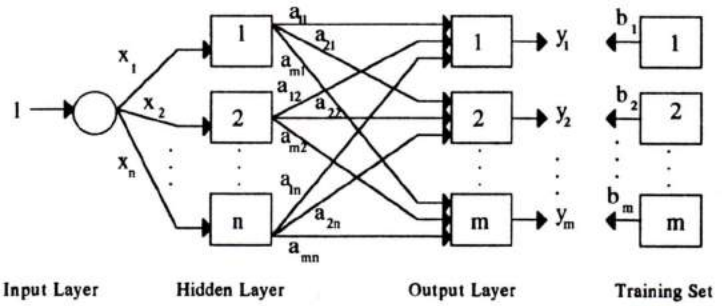

Figure 2. Two-Layer Structured ANN for Linear System of Equation Solving

The network uses the gradient descent algorithm [16] for minimising the residual error. The algorithm works as follows: starting at an arbitrary point $x^{(0)}$, a sequence of improved approximations $x^{(1)}, x^{(2)}, \ldots$, is generated, such that, for $t \geq 0$, $x^{(t+1)} = x^{(t)} + \alpha(-\nabla E(x^{(t)}))$, where $-\nabla E(x^{(t)})$ is the descent direction defined in (2) and α is the stepsize. The stepsize α can be constant (determined in a heuristic way), as in [1],[11],[16],[19], simulating the steepest descent method with heuristic line search, which leads to a classical Back-propagation algorithm, or adaptive during time, as in [10],[13],[20], simulating the steepest descent method with exact line search.

We define $w_{0j} = x_j$, $j = 1,2,\ldots,n$ to be the synaptic weight incoming to the hidden layer neuron j from the input neuron, so the connections of every hidden layer neuron j ($1 \leq j \leq n$) with the input neuron is the corresponding j^{th} value of a vector x, of size n (the solution of the system), and $w_{ji} = a_{ij}$, $1 \leq j \leq n$, $1 \leq i \leq m$ to be the synaptic weight incoming to the output layer neuron j from the hidden layer neuron i, to be the corresponding i^{th} row of matrix A.

Comment 3.1: A disadvantage of the proposed architecture is that the number of neurons and weights is significantly increased if the number of patterns (the size of matrix A) is too large. In this case, we can use block-LMS, with k blocks using an architecture having $\left\lceil \frac{m}{k} \right\rceil$ neurons.

3.1. The Heuristic Steepest Descent (HSD) Learning Algorithm

The training procedure is as follows: Initially, the interconnection bias weights, x_j, $j = 1,2,\ldots,n$ take random values in (-1, 1). An input with the value of 1 is presented in the input neuron and the corresponding outputs $u_j^{(t)} = f(x_j^{(t)}) = x_j^{(t)}$ $j = 1,2,\ldots,n$ of the hidden layer neurons are calculated, with $f(\,)$ the simple identity function for neuron j, x_j the connection between the input neuron and the hidden layer neuron j, and t the step of updating ($t = 0,1,\ldots$). Then, the actual outputs y_i for every output layer neuron i ($i = 1,2,\ldots,m$) at time t are calculated

$$y_i^{(t)} = f\left(\sum_{j=1}^{n} u_j^{(t)} w_{ji}\right) = f\left(\sum_{j=1}^{n} x_j^{(t)} a_{ij}\right) = \sum_{j=1}^{n} a_{ij} x_j^{(t)} \tag{7}$$

The discrepancy between desired and calculated output, i.e. the difference between b_i and $y_i^{(t)}$, for $i = 1,2,\ldots,m$ is calculated by means of the Delta Rule

$$d_i^{(t)} = b_i - y_i^{(t)}, \quad i = 1,2,\ldots,m \tag{8}$$

Since the connections between the hidden layer and the output layer is the matrix A, those connections are constants, and remain unchanged.

Following the back-propagation procedure [16], the calculation of δ_k, $k = 1,2,\ldots,n$ for the hidden layer has the form

$$\delta_k^{(t)} = \sum_{i=1}^{m} w_{ki}^{(t)} d_i^{(t)} = \sum_{i=1}^{m} a_{ik} d_i^{(t)}, \quad k = 1,2,..,n \tag{9}$$

The weight adaptation procedure only for the input weights (since input is always 1) has the form

$$x_k^{(t+1)} = x_k^{(t)} + \alpha\delta_k^{(t)}, \quad k = 1,2,...,n \tag{10}$$

where α is the learning rate.

This procedure is repeated for a number of iterations, until the error between calculated and desired outputs is within acceptable limits. The convergence of the above algorithm is proved by the following theorem:

THEOREM 3.1. The operation of the ANN in Figure 2 using the HSD algorithm converges to the Least Squares solution of the linear system (2).

Proof. The value of $\delta_k^{(t)}$, $(k = 1,2,...,n)$ at step t, using (7),(8),(9) will be

$$\delta_k^{(t)} = \sum_{i=1}^{m} a_{ik} d_i^{(t)} = \sum_{i=1}^{m} a_{ik} (b_i - y_i^{(t)}) = \sum_{i=1}^{m} a_{ik} (b_i - \sum_{j=1}^{n} a_{ij} x_j^{(t)}) = \sum_{i=1}^{m} a_{ik}\, b_i - \sum_{i=1}^{m} a_{ik} \sum_{j=1}^{n} a_{ij} x_j^{(t)} \tag{11}$$

or in matrix - vector form

$$\delta^{(t)} = A^T(b - Ax^{(t)}) = A^T b - A^T A x^{(t)} \tag{12}$$

The value of $x_k^{(t+1)}$, $(k = 1,2,...,n)$ at step $t+1$, using (11),(12) will be $x_k^{(t+1)} = x_k^{(t)} + \alpha\delta_k^{(t)}$ $= x_k^{(t)} + \alpha(\sum_{i=1}^{m} a_{ik} b_i - \sum_{i=1}^{m} a_{ik} \sum_{j=1}^{n} x_j^{(t)} a_{ij})$, or in matrix - vector form

$$x^{(t+1)} = x^{(t)} + \alpha A^T(b - Ax^{(t)}) = x^{(t)} - \alpha A^T(Ax^{(t)} - b) = x^{(t)} - \alpha \nabla E(x^{(t)}) \tag{13}$$

which is identical to equation (9) in the batch-LMS algorithm, so the values x_k, $k = 1,2,...,n$ of vector x converge to the solution of the system of normal equations (2), which minimises the residual error (i.e. $\lim_{t\to\infty}(E(x^{(t)})) = 0$, if $m \le n$, or $\lim_{t\to\infty}(E(x^{(t)}))$ becomes minimum, if $m > n$).

3.2. The Adaptive Steepest Descent (ASD) Learning Algorithm

The disadvantages of the HSD algorithm is that its convergence rate is very slow, compared to algorithms that use adaptive stepsize α as in [10],[13],[20], with neat convergence properties. Using the same architecture and training procedure as in HSD, the new learning procedure differs from HSD in that the weight adaptation procedure instead of (10) for the input weights will be $x_k^{(t+1)} = x_k^{(t)} + \alpha^{(t+1)}\delta_k^{(t)}$, $k = 1,2,...,n$ or in vector form $x^{(t+1)} = x^{(t)} + \alpha^{(t+1)}\delta^{(t)}$, with

$$\alpha^{(t+1)} = \frac{\delta^{(t)}\delta^{(t)}}{\delta^{(t)}A^T A\delta^{(t)}} = \frac{\delta^{(t)}\delta^{(t)}}{\delta^{(t)}C\delta^{(t)}} \tag{14}$$

where $\delta^{(t)}$ is defined in (12), the residual error of solving the corresponding system of normal equations (2).

This procedure is repeated for a number of iterations, until the error between calculated and desired outputs is within acceptable limits. The convergence of the ASD algorithm is proved by the following theorem:

THEOREM 3.2. Using $\alpha^{(t+1)}$ defined in (14) the ANN of Figure 2 simulates the Steepest Descent Method.

Proof. With $E(x)$ defined in (1), from (2) we have

$$\nabla E(x)=Cx-d=A^TAx-A^Tb \tag{15}$$

Using (15), $\delta^{(t)}$ defined in (14) becomes

$$\delta^{(t)}=A^Tb-A^TAx^{(t)}=d-Cx^{(t)}=-\nabla E(x^{(t)}) \tag{16}$$

If we choose $\alpha^{(t+1)}$ to minimise $E(x^{(t+1)})=E(x^{(t)}+\alpha^{(t+1)}\delta^{(t)})$, then $\alpha^{(t+1)}$ is given from equation $\nabla E(\alpha^{(t+1)})=0$. By the chain rule using (15),(16) we have

$$\nabla E(x^{(t+1)})=\nabla E(x^{(t)}+\alpha^{(t+1)}\delta^{(t)})=\nabla E(x^{(t)}+\alpha^{(t+1)}\delta^{(t)})\delta^{(t)}=[C(x^{(t)}+\alpha^{(t+1)}\delta^{(t)})-d]\delta^{(t)}$$
$$=[Cx^{(t)}+\alpha^{(t+1)}C\delta^{(t)}-d]\delta^{(t)}=[\alpha^{(t+1)}C\delta^{(t)}-\delta^{(t)}]\delta^{(t)}$$

By setting $\nabla E(\alpha^{(t+1)})=0$, we can see that $\alpha^{(t+1)}$ is given by $\alpha^{(t+1)}=\dfrac{\delta^{(t)}\delta^{(t)}}{\delta^{(t)}C\delta^{(t)}}$, as defined in equation (14), QED.

THEOREM 3.3. Using $\alpha^{(t+1)}$ defined in (14), the ANN of Figure 2 simulates the Optimal Steepest Descent Method [5] used in back-propagation, with $\alpha^{(t+1)}$ given by [18]

$$\alpha_{opt}^{(t+1)}\approx\frac{\left\|\nabla E(x^{(t)})\right\|^2}{\nabla E(x^{(t)})\nabla^2E(x^{(t)})\nabla E(x^{(t)})} \tag{17}$$

Proof. From (15) we have

$$\nabla^2E(x^{(t)})=\nabla(\nabla E(x^{(t)}))=\nabla(Cx^{(t)}-d)=C \tag{18}$$

and using (15),(18), $\alpha_{opt}^{(t+1)}$ becomes

$$\alpha_{opt}^{(t+1)}\approx\frac{\left\|\nabla E(x^{(t)})\right\|^2}{\nabla E(x^{(t)})\nabla^2E(x^{(t)})\nabla E(x^{(t)})}=\frac{\left\|-\delta^{(t)})\right\|^2}{(-\delta^{(t)})C(-\delta^{(t)})}=\frac{\delta^{(t)}\delta^{(t)}}{\delta^{(t)}C\delta^{(t)}} \tag{19}$$

as in equation (14), QED.

3.3 Matrix Inversion

The method that have been previously discussed for linear system solution can be extended to cover the solution of matrix equations

$$AX=B \tag{20}$$

where A, X and B are $(m\times n)$, $(n\times k)$ and $(m\times k)$ matrices. Equation (20) can be seen as a set of m systems of linear equations with common coefficient matrix A. The problem can be partitioned in solving k systems of linear equations formed by using the k columns of matrix B. Thus, using k times the ANN of Figure 2, we can generate the k columns of matrix X. Another alternative is to use a 3–D ANN with k NNs of Figure 2, in order to solve those equations. Notice that matrix A is stored in

the connections between the hidden and the output layer of all NNs involved, since the coefficient matrix is the same for all systems. The configuration involves k NNs working in parallel, as shown in Figure 3 with the same learning procedures in all methods presented for every 2-D ANN. A special case of the matrix equation problem is the solution of the system

$$AX=I \tag{21}$$

for A, X $(n \times n)$ matrices and I the $(n \times n)$ identity matrix. Then, the solution is $X = A^{-1}$. Thus, it is possible to use n NNs of Figure 2, in order to invert a matrix A.

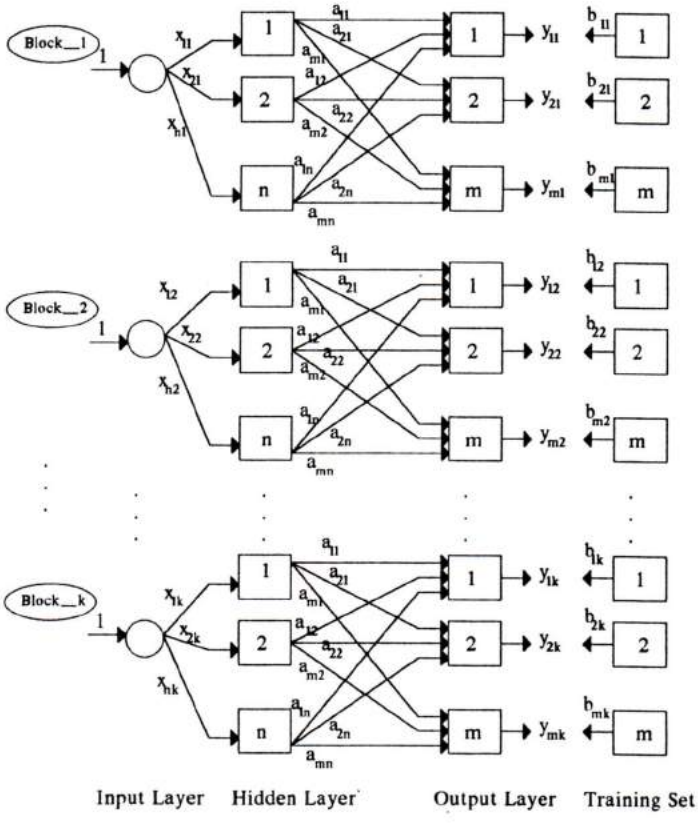

Figure 3. Two-Layer 3-D Structured ANN for solving AX=B

4. EXPERIMENTAL RESULTS

To check the performance and the convergence behaviour of the proposed algorithms for solving system $Ax = b$, we used some specific examples, and compare the solutions to the least square (LS) solution. Also, we have drawn the corresponding graphs showing the convergence (with respect to the minimisation of the Mean Square Error) of the three algorithms through time in a VAX 4200 machine.

Example 1: The square linear system of equations to be solved is :

$$\begin{bmatrix} 1.2 & 0.8 & 0.7 & 0.5 \\ 0.4 & 1.5 & 0.3 & 0.1 \\ 0.1 & 0.5 & 1.7 & 0.9 \\ 0.1 & 0.5 & 0.6 & 1.2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \tag{22}$$

Given zero initial values to the synaptic weights, and using

$$E(x) = \|Ax - b\|^2 \tag{23}$$

as the total mean square error, the same used by Wang L.X. and Mendel J. [20], the convergence behaviour of the above training algorithms is shown in Figure 4, where the horizontal axis denotes the training cycle t, and the vertical axis the mean square error given in (23). With $\varepsilon = 10^{-5}$, ASD algorithm converges in 50 training cycles to the solution

$$x_{ASD} = [x_1, x_2, x_3, x_4]^T = [-1.30014,\ 1.51324,\ -0.12329,\ 2.87085]^T$$

with

$$r_{ASD} = [r_1, r_2, r_3, r_4]^T = [0.00046,\ 0.00010, -0.00078,\ 0.00235]^T$$

which is close to the exact least squares solution of system (22)

$$x_{LS} = [-1.29991,\ 1.51346,\ -0.12535,\ 2.87372]^T \tag{24}$$

obtained by using Greville's algorithm, and better than the estimation obtained by Wang L.X. and Mendel J. [20]

$$x = [-1.21653,\ 1.47053,\ 0.16055,\ 2.35300]^T$$

$$r = [0.00548,\ 0.00265,\ 0.00465,\ 0.00215]^T$$

Given zero initial values to the synaptic weights, and using $\alpha = 0.01$ as learning rate for LMS and HSD, HSD algorithm (which is equivalent to the batch-LMS method) converges in 1351 training cycles, with $\varepsilon = 10^{-5}$, to the solution

$$x_{HSD} = [-1.29998,\ 1.51329,\ -0.12241,\ 2.87013]^T$$

$$r_{HSD} = [-0.00003,\ -0.00023,\ -0.00167,\ 0.00265]^T$$

whereas incremental LMS algorithm converges in 1341 training cycles, with $\varepsilon = 10^{-5}$, to the solution

$$x_{LMS} = [-1.30002,\ 1.51330,\ -0.12242,\ 2.87016]^T$$

$$r_{LMS} = [-0.00001,\ -0.00024,\ -0.00168,\ 0.00261]^T$$

Comment 4.1: In order to allow a more meaningful comparison between the incremental LMS method and HSD (or the equivalent batch-LMS method) one learning step of incremental LMS algorithm is taken to mean a full cycle through the m samples (the m lines of matrix A).

Comment 4.2: As it is shown in Figure 4, the behaviour of the incremental LMS and HSD method is similar and the number of iterations needed for LMS and HSD to converge is almost the same as stated in [1],[21], whereas with the application of ASD

algorithm the mean square error is decreased rapidly, and the number of iterations needed for convergence is too small compared to the other two methods.

Comment 4.3: For $\varepsilon = 10^{-9}$ the above algorithms converge to the least square solution (24) of the system (22) but the number of iterations needed to converge is increased.

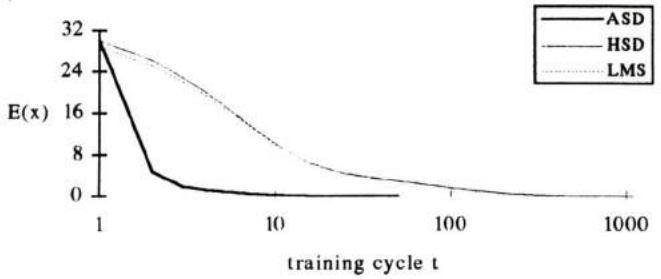

Figure 4. Convergence behaviour of LMS, HSD, and ASD for square linear system in Example 1

Example 2 : The underdetermined linear system of equations to be solved is:

$$\begin{bmatrix} 2 & -1 & 4 & 0 & 3 & 1 \\ 5 & 1 & -3 & 1 & 2 & 0 \\ 1 & -2 & 1 & -5 & -1 & 4 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ -4 \end{bmatrix} \tag{25}$$

Given zero initial values to the synaptic weights, and using $E(x)=\frac{1}{2}\|Ax-b\|^2$ for the total mean square error, as defined in (1), the convergence behaviour of the above training algorithms is shown in Figure 5, where the horizontal axis denotes the training cycle t, and the vertical axis the mean square error given in (1). With $\varepsilon = 10^{-5}$, ASD algorithm converges in 8 training cycles to the solution

$$x_{ASD} = [0.08826,\ 0.10826,\ 0.27321,\ 0.50457,\ 0.38275,\ -0.30965]^T$$

$$r_{ASD} = [0.00029,\ 0.00001,\ -0.00076]^T$$

which is a close estimation of the exact least square solution of the system (25), and close to the estimation achieved by Cichocki and Unbehauen [3]

$$x = [0.0882,\ 0.1083,\ 0.2733,\ 0.5047,\ 0.3828,\ -0.3097]^T$$

Given zero initial values to the synaptic weights, and using $\alpha = 0.01$ as learning coefficient for HSD and LMS, HSD algorithm converges in 24 training cycles, with $\varepsilon = 10^{-5}$, to the solution

$$x_{HSD} = [0.08832,\ 0.10832,\ 0.27278,\ 0.50444,\ 0.38254,\ -0.30960]^T$$

$$r_{HSD} = [0.00254,\ -0.00112,\ -0.00133]^T$$

whereas LMS converges in 23 training cycles, with $\varepsilon = 10^{-5}$, to the solution

$$x_{LMS} = [0.08827,\ 0.10834,\ 0.27281,\ 0.50451,\ 0.38255,\ -0.30965]^T$$

$$r_{LMS} = [0.00256,\ -0.00085,\ -0.00071]^T$$

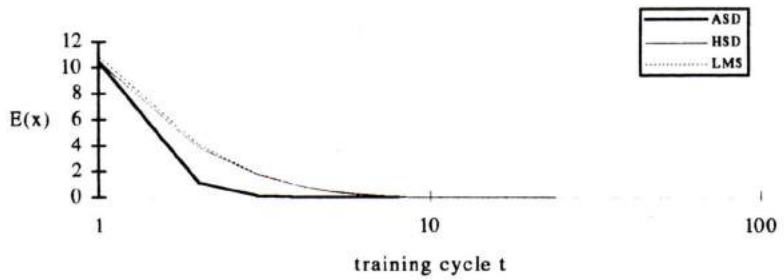

Figure 5. Convergence behaviour of ASD, HSD, and LMS for underdetermined linear system for Example 2

Example 3: The overdetermined linear system of equations to be solved is :

$$\begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 3 \end{bmatrix} \tag{26}$$

Since system (26) is overdetermined, the application of the algorithms will give a least squares solution, which minimises the mean square error defined in (1) but it will never be zero. Thus, we use the following convergence criterion

$$\Delta x = \left\| x^{(t+1)} - x^{(t)} \right\|_1 = \sum_{j=1}^{n} \left| x_j^{(t+1)} - x_j^{(t)} \right| \le 10^{-5} \tag{27}$$

in order to terminate the algorithms, with t=0,1,... the training cycle. The convergence behaviour of the above training algorithms is shown in Figure 6, where the horizontal axis denotes the training cycle t, and the vertical axis the mean square error given in (1). With $\varepsilon = 10^{-5}$, given zero initial values to the synaptic weights, ASD algorithm converges in 4 training cycles, with E(x) = 0.20000, to the solution

$$x_{ASD} = [x_1, x_2]^T = [0.20000,\ 0.60000]^T$$

which is in excellent agreement with the exact least square solution of the system (26)

$$x_{LS} = [x_1, x_2]^T = [0.20000,\ 0.60000]^T \tag{28}$$

obtained by using Greville's algorithm. The residual vector of the ASD solution is

$$r_{ASD} = [r_1, r_2, r_3, r_4, r_5]^T = [-0.20000,\ 0.40000,\ 0.00000,\ -0.40000,\ 0.20000]^T$$

This solution is better than the estimation achieved by Cichocki and Unbehauen [2]

$$x = [0.206,\ 0.598]^T$$

$$r = [-0.196,\ 0.402,\ 0,\ -0.402,\ 0.196]^T$$

Given zero initial values to the synaptic weights, and using $\alpha = 0.01$ as learning coefficient for HSD, the algorithm converges in 417 training cycles, with $E(x) = 0.20000$, to the solution

$$x_{HSD} = [0.19909,\ 0.60025]^T$$

$$r_{HSD} = [-0.20066,\ 0.39959,\ -0.00015,\ -0.39990,\ 0.20035]^T$$

Given zero initial values to the synaptic weights, and using $\alpha = 0.01$ as learning coefficient for LMS, the algorithm converges in 456 training cycles, with $E(x) = 0.22634$, to the solution

$$x_{LMS} = [0.22298,\ 0.59080]^T$$

$$r_{LMS} = [-0.18622, 0.40459, -0.00461, -0.41381, 0.17699]^T$$

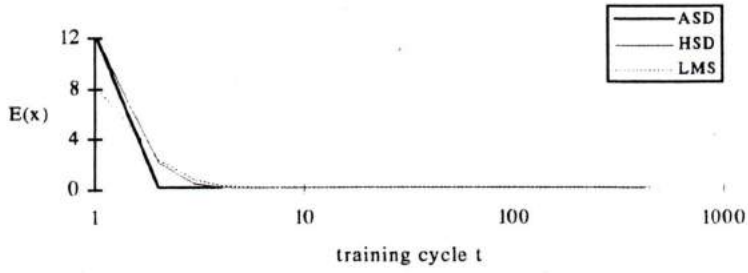

Figure 6. Convergence behaviour of ASD, HSD, and LMS for overdetermined linear system for Example 3

Comment 4.4: If we use $\Delta x = \left\| x^{(t+1)} - x^{(t)} \right\|_1 \leq 10^{-9}$ instead of (27) for HSD and LMS algorithm, HSD converges to the least square solution of the system (26) in 1502 training cycles, while the solution of LMS is getting worse, increasing the training cycles and the residual error.

Discussion: For square and underdetermined linear systems of equations as in (22),(25), all three algorithms for small values of the learning rate (used in HSD and LMS) converge to the same and only solution (the Least Square solution $A^{-1}b$ for square systems or $A^{+}b$ for underdetermined) or a close estimation of the Least Square solution. For such systems, the performance of ASD algorithm is better than HSD and LMS (the number of iterations needed for ASD in order to converge is very small compared to the others).

For overdetermined linear systems of equations as in (26), only ASD converge to the Least Square solution $A^{+}b$ using the total error as defined in (1). HSD and LMS converge to an approximation of $A^{+}b$, with HSD giving a better approximation than LMS. For such systems, the performance of ASD algorithm is better than HSD and LMS.

Example 4: Find the pseudo-inverse of the singular matrix

$$A = \begin{bmatrix} 1.2 & 0.8 & 0.7 & 0.5 \\ 0.4 & 1.5 & 0.3 & 0.1 \\ 0.1 & 0.5 & 1.7 & 0.9 \\ 0.1 & 0.5 & 0.6 & 1.2 \end{bmatrix} \tag{29}$$

Applying the ANN of Figure 3 or the ANN of Figure 2 four times, and using $[1,0,0,0]^T$, $[0,1,0,0]^T$, $[0,0,1,0]^T$, $[0,0,0,1]^T$ as vector b, we can find the four columns of matrix $X = A^+$.

Given zero initial values to the synaptic weights, and using α=0.01 as the learning rate for HSD and LMS, ASD algorithm converges with $\varepsilon = 10^{-5}$, in 30 training cycles for column 1, in 37 training cycles for column 2, in 36 training cycles for column 3, and in 40 training cycles for column 4, to the solution

$$X_{ASD} = A^+ = \begin{bmatrix} 0.99493 & -0.37772 & -0.28277 & -0.17232 \\ -0.26761 & 0.79883 & -0.06353 & 0.09261 \\ 0.00769 & -0.07447 & 0.80851 & -0.60289 \\ 0.02377 & -0.26244 & -0.35240 & 1.10766 \end{bmatrix}$$

which is a good estimation of the Least Square solution obtained by using Greville's algorithm

$$X_{LS} = A^{-1} = \begin{bmatrix} 0.99814 & -0.38069 & -0.28319 & -0.17177 \\ -0.26927 & 0.80037 & -0.06314 & 0.09285 \\ 0.00696 & -0.07242 & 0.81198 & -0.60585 \\ 0.02553 & -0.26555 & -0.35608 & 1.11188 \end{bmatrix} \tag{30}$$

HSD algorithm converges with $\varepsilon = 10^{-5}$ in 843 training cycles for column 1, in 841 training cycles for column 2, in 1059 training cycles for column 3, and in 1149 training cycles for column 4, to the solution

$$X_{HSD} = A^+ = \begin{bmatrix} 0.99357 & -0.37687 & -0.28241 & -0.17223 \\ -0.26638 & 0.79808 & -0.06333 & 0.09283 \\ 0.00809 & -0.07493 & 0.80786 & -0.60170 \\ 0.02376 & -0.26215 & -0.35099 & 1.10678 \end{bmatrix}$$

and LMS algorithm converges with $\varepsilon = 10^{-5}$ in 834 training cycles for column 1, in 834 training cycles for column 2, in 1049 training cycles for column 3, and in 1140 training cycles for column 4, to the solution

$$X_{LMS} = A^+ = \begin{bmatrix} 0.99357 & -0.37687 & -0.28238 & -0.17227 \\ -0.26641 & 0.79811 & -0.06334 & 0.09284 \\ 0.00794 & -0.07485 & 0.80790 & -0.60174 \\ 0.02394 & -0.26227 & -0.35107 & 1.10686 \end{bmatrix}$$

The convergence behaviour of the above training algorithms for finding the inverse matrix of A in Example 4 is shown in Figure 7. As it can be seen, the behaviour of the incremental LMS and HSD method is similar, while the number of iterations needed for LMS and HSD to converge is almost the same. With the application of ASD algorithm the Mean Squared Error is decreased rapidly, and the number of iterations

needed for convergence is too small compared to the other two methods. If we use $\varepsilon = 10^{-9}$, the above algorithms converge to the least square solution (30) of the system (29) but the number of iterations needed to converge is increased.

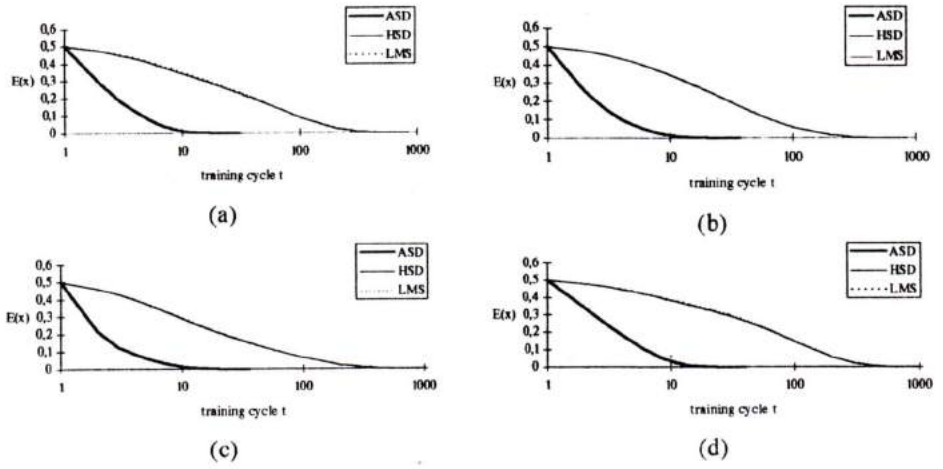

Figure 7. Convergence behaviour of ASD, HSD, and LMS algorithm for column 1 (a), column 2 (b), column 3 (c), and column 4 (d) of matrix A for matrix inversion in Example 4

5. CONCLUSIONS

In this paper we discussed the issue of a neural network design and implementation for solving linear systems of equations. Delta Rule for network training is modified in order to lead to a batch-LMS algorithm, whereas the use of an adaptive learning rate implements the steepest descent method. We have proven the capability of the proposed network for solving any kind of linear systems of equations. We used numerical examples in order to demonstrate operating characteristics of the proposed neural network. Simulations was performed to estimate the performance, i.e. the training cycles required to reduce the mean squared error by a fraction of $\varepsilon=10^{-5}$, and the comparison of the solutions taken to the least squares solution. The fact that distinguishes our implementation from previous ones is the simplicity of its architecture and its training algorithm, along with the fact that it exceeds the limitations of matrix A needed to be SPD. In addition, the ASD training algorithm proposed with adaptive stepsize α has neat convergence properties and guarantees fast network convergence to the optimal solution for any kind of linear system of equations but it is hard to be implemented in VLSI circuits because of the off-line calculations needed and the size of matrix A, which can be too large. On the other hand, the HSD algorithm (equivalent to batch-LMS) converges for sufficiently small values of the stepsize α [7],[21] to a solution close to the optimal solution but convergence analysis is difficult and its convergence rate is very slow.

REFERENCES

[1] Battiti, R., "First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method", *Neural Computation* (1992), Vol. 4, pp. 141-166.

[2] Cichocki, A. and Unbehauen, R., "Neural Networks for Solving Systems of Linear-Equations and Related Problems", *IEEE Transactions on Circuits and Systems I - Fundamental Theory and Applications*, (1992), Vol. 39, no. 2, pp. 124-138.

[3] Cichocki, A. and Unbehauen, R., "Simplified Neural Networks for Solving Linear Least Squares and Total Least Squares Problems in Real Time", *IEEE Transactions on Neural Networks*, (1994), Vol. 5, no. 6, pp. 910-923.
[4] Golub, G. H., and Van Loan C.F., "Matrix Computations", *The John Hopkins University Press*, Baltimore, MD, 1983.
[5] Hassoun, M. H., "Fundamentals of Artificial Neural Networks", *The MIT Press*, Cambridge, Massachusetts 02142, 1995.
[6] Kohonen T, "Correlation Matrix Memories", *IEEE Transactions on Computers*, C-21 (4), pp. 353-359, 1972.
[7] Kohonen T, "An Adaptive Associative Memory Principle", *IEEE Transactions on Computers*, pp. 444-445, 1974.
[8] Kohonen T, "Self-Organisation and Associative Memory", *Springer-Verlag*, 1984.
[9] Luenberger, "Introduction to Linear and Non-linear Programming", *Addison-Wesley*, Reading, MA, 1973.
[10] Luo Zhi-Quan, "On the Convergence of the LMS Algorithm with Adaptive Learning Rate for Linear Feedforward Networks", *Neural Computation* (1991), Vol. 3, pp. 226-245.
[11] Margaritis K.G., Adamopoulos M., Goulianas K., and Evans D.J., "Artificial Neural Networks and Iterative Linear Algebra Methods", *Parallel Algorithms and Applications* (1994), Vol. 3, pp. 31-44.
[12] Maren A.J., Harston C.T., Pap R.M., "Handbook of neural computing applications", *Academic Press*, 1990.
[13] Polycarpou M. and Ioannou P., "Learning and Convergence Analysis of Neural-Type Structured Networks", *IEEE Transactions on Neural Networks* (1992), Vol. 3, no. 1, pp. 39-50.
[14] Rosenblatt F, "Principles of neurodynamics", *Spartan Books*, 1962.
[15] Rumelhart D.E., McClelland J, "Parallel distributed processing: Explorations in the microstructure of cognition", *MIT Press*, 1986.
[16] Rumelhart D.E., Hinton G.E., and Williams R.J., "Learning Internal representation by error propagation", *Parallel Distributed Processing I*, Rumelhart D.E., and McClelland J, Eds., Cambridge, MA, MIT Press, 1986.
[17] Simpson PK, "Artificial Neural Systems", *Pergamon Press*, 1990.
[18] Tsypkin, Ya. Z., "*Adaptation and Learning in Automatic Systems*", translated by Z. J. Nikolic, Academic Press, New York, 1971.
[19] Wang L. X. and Mendel, J.M., "Parallel Structured Networks for Solving a Wide Variety of Matrix Algebra Problems", *Journal of Parallel and Distributed Computing* (1992), Vol. 14, pp. 236-247.
[20] Wang L. X. and Mendel, J. M., "Three-Dimensional Structured Networks for Matrix Equation Solving", *IEEE Transactions on Computers* (1991), Vol. 40, no. 12, pp. 1337-1346.
[21] Widrow B, and Lehr M, "30 Years of Adaptive Neural Networks: Perceptron, Madaline and Backpropagation", *Proceedings of the IEEE*, Vol. 78, No. 9, pp. 1415-1442, 1990.