International Journal of Parallel, Emergent and Distributed Systems
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713729127

ARTIFICIAL NEURAL NETWORKS AND ITERATIVE LINEAR ALGEBRA METHODS

K. G. Margaritis [a]; M. Adamopoulos [a]; K. Goulianas [a]; D. J. Evans [b]
[a] Informatics Centre, University of Macedonia, Thessaloniki, Greece
[b] Parallel Algorithms Research Centre, Loughborough University of Technology, UK

Online Publication Date: 01 January 1994

To cite this Article: Margaritis, K. G., Adamopoulos, M., Goulianas, K. and Evans, D. J. (1994) 'ARTIFICIAL NEURAL NETWORKS AND ITERATIVE LINEAR ALGEBRA METHODS', International Journal of Parallel, Emergent and Distributed Systems, 3:1, 31 — 44

To link to this article: DOI: 10.1080/10637199408962524
URL: http://dx.doi.org/10.1080/10637199408962524

Parallel Algorithms and Applications, Vol. 3, pp. 31–44
Reprints available directly from the publisher
Photocopying permitted by license only

ARTIFICIAL NEURAL NETWORKS AND ITERATIVE LINEAR ALGEBRA METHODS

K. G. MARGARITIS*, M. ADAMOPOULOS and K. GOULIANAS
Informatics Centre, University of Macedonia, Thessaloniki, Greece

D. J. EVANS
Parallel Algorithms Research Centre, Loughborough University of Technology, UK

(Received November 15, 1993)

This paper describes the usage of feed-forward artificial neural networks, for the implementation of a variety of iterative methods of numerical linear algebra for solving linear systems of equations. Extensions to matrix based iterative procedures are also presented and the application of those iterative methods in neural network training algorithms is discussed. Finally, some experimented results are presented, comparing the various methods discussed.

KEY WORDS: Artificial neural nets, matrix iterative methods, training algorithms.

C.R. CATEGORIES: G.1.3, I.2.6.

1. INTRODUCTION

Feedforward artificial neural networks have been studied extensively and have been proved capable of solving a wide variety of problems [4, 7]. Most applications of these networks use some type of training procedure in order to utilize associations of input patterns to output patterns. This relations can be either autoassociative or heteroassociative, i.e. they correlate a set of patterns either to themselves or to another set of patterns. The individual interconnection weights that are produced during the training process do not have a clear physical meaning and therefore the choice of training procedures, the number of layers as well as neuron configurations is quite arbitrary.

In this paper the development of simple feedforward neural networks with linear neuron functions is studied. The emphasis is placed in the study of different training procedures, i.e. weight adaptation methods, and the relation of those procedures to well known iterative methods of numerical linear algebra. Thus, in contrast to the majority of neural network applications, here the interest is focused onto the weight adaptation procedure in the sense that the result of the network operation is coded in the weights and not in the output vector. The effort is to establish direct relation

*Visiting Research Fellow, Parallel Algorithms Research Centre, Department of Computer Studies, Loughborough University of Technology, UK.

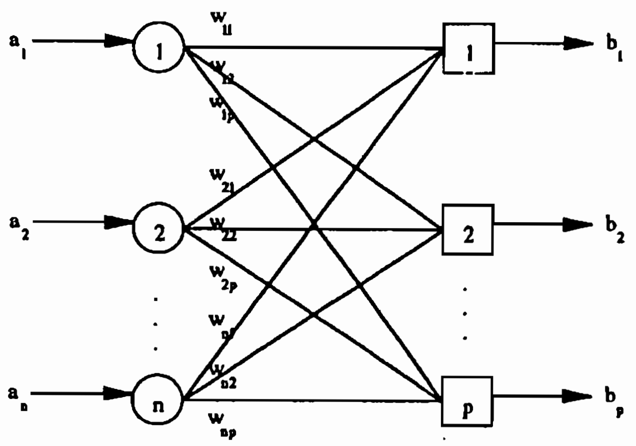

Figure 1 Two-layer feedforward ANN.

between well known iterative methods in numerical linear algebra and neural network structures that implement those methods. Thus, the interconnection weights and their modification procedures may have some relatively well defined physical meaning. Further to show the applicability of those iterative methods in general neural network training algorithms.

The term Neural Network (NN) is used herein as a shorthand for term Feedforward Artificial Neural Network. A generic NN is now described in some detail in order to serve as the basis of the NNs to be developed. This generic NN is based on two-layer feedforward artificial neural networks, such as Perceptron [5], Adaline [10] and Back-Propagation NNs [6].

The generic NN (GENN) is a two-layer, heteroassociative pattern matcher. It correlates pairs of pattern vectors $\mathbf{a}^{(k)}, \mathbf{b}^{(k)}$, $k = 1, 2, \ldots, m$ where $\mathbf{a}^{(k)}$, $\mathbf{b}^{(k)}$ are $(n \times 1)$ and $(p \times 1)$ vectors, respectively. It consists of two fully connected layers of n and p neurons, where the first layer is essentially an input node layer (Figure 1).

The training procedure is as follows: Initially, the interconnection weights, w_{ij}, $i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, p$ take random values. Then one pattern vector $\mathbf{a}^{(k)}$ is presented in the input layer and the corresponding outputs are calculated:

$$u_j^{(k)} = f\left(\sum_{i=1}^{n} w_{i,j} \cdot a_i^{(k)} - c_j\right), \qquad j = 1, 2, \ldots, p \quad \text{or} \quad \mathbf{u}^{(k)} = f(\mathbf{a}^{(k)T} W - \mathbf{c}) \tag{1}$$

in matrix-vextor format, and with $f(\,)$ a neuron function and c_j some input bias for neuron j. The discrepancy between calculated and desired output, i.e. the difference between $\mathbf{u}^{(k)}$ and $\mathbf{b}^{(k)}$ is calculated by means of the Delta Rule:

$$d_j^{(k)} = u_j^{(k)} - b_j^{(k)}, \qquad j = 1, 2, \ldots, p \quad \text{or} \quad \mathbf{d}^{(k)} = \mathbf{u}^{(k)} - \mathbf{b}^{(k)} \tag{2}$$

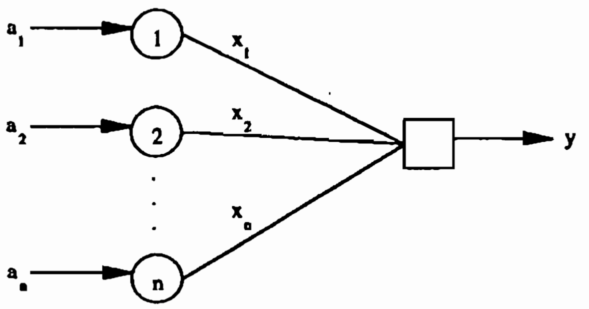

Figure 2 Inner Product Neural Network (IPNN).

The weight adaptation computation has the form:

$$\Delta w_{ij}^{(k)} = \omega \cdot a_i^{(k)} \cdot d_j^{(k)}, \qquad i = 1,2,\ldots,n, \quad j = 1,2,\ldots,p$$
$$w_{ij}^{(k+1)} = w_{ij}^{(k)} + \Delta w_{ij}^{(k)} \qquad \text{or} \qquad \Delta W^{(k)} = \omega \mathbf{a}^{(k)} \mathbf{d}^{(k)T}, \qquad W^{(k+1)} = W^{(k)} + \Delta W^{(k)}. \tag{3}$$

This procedure is repeated for all m pattern pairs and for a number of iterations, until the error between calculated and desired outputs is within acceptable limits. The recall function, i.e. the pattern matching operation, is exactly as equation (1), where the output is the pattern which is associated by the NN with the presented input pattern.

This generic NN outlines the operation of the majority of the two-layer feedforward artificial neural networks, as, for example, the Perceptron, or the Adaline or the Back-Propagation NNs. Differences between those NNs exist in several aspects, e.g.:

- the encoding method for the input and output patterns, as well as the interconnection weights,
- the neuron function used,
- the choice of input bias,
- the mathematical tools used for optimizing the training (i.e. the weight adaptation procedures).

The basic building block in the generic NN is the Inner Product NN (IPNN), shown in Figure 2.

In linear algebra terms the operation performed, is expressed as:

$$y^{(k)} = f(\mathbf{a}^{T(k)}\mathbf{x}) \qquad \text{or} \qquad y^{(k)} = f\left(\sum_{i=1}^{a} a_i^{(k)} \cdot x_i\right) \tag{4}$$

with a training scheme $d^{(k)} = y^{(k)} - b$, $\Delta \mathbf{x}^{(k)} = \omega d^{(k)} \mathbf{a}^{(k)}$, $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta \mathbf{x}^{(k)}$, where $\mathbf{a},\mathbf{x}$ are column vectors with n elements and b is the target output. The IPNN consists of n input nodes and a single output neuron. All inputs are connected to the single output neuron. The components of a vector **a** are used as inputs of the IPNN, while the elements of vector **x** are the interconnection weights. If the neuron function is chosen to be the threshold logic function with threshold equal to 0 then the relations in eq. (4) become linear. Of course the roles between vectors **a** and **x** are interchangeable. A further simplification of the IPNN is to keep the interconnection weights fixed, i.e. replace the training procedure with some pre-specified encoding scheme (e.g. in Hamming NNs). Compared to the generic NN it can be seen as a single output generic NN (i.e. $p = 1$) with linear or non-linear neuron function and with or without training procedure.

2. ITERATIVE METHODS FOR LINEAR SYSTEMS

Iterative methods based on matrix vector multiplication have been extensively used in numerical linear algebra for linear system solution. Using the generic NN described in Section 1 some of those methods are now presented for solving a $(n \times n)$ linear system of the form:

$$A\mathbf{x} = \mathbf{b}. \tag{5}$$

2.1. Successive Over-Relaxation and Gauss Seidel Methods

Although the Successive Over-Relaxation method (SOR) is not the simplest iterative method for solving linear systems, it can be realized by means of adding a simple training procedure onto the IPNN of Figure 2, based on a modification of Delta Rule. The SOR method can be derived from the Gauss Seidel (GS) method as follows. Given an initial approximation $\mathbf{x}^{(0)}$ to the solution vector $\mathbf{x}$, the iterative process for calculating the exact solution is given by the relation:

$$\mathbf{x}^{(k+1)} = (1-\omega)\cdot\mathbf{x}^{(k)} + \omega\cdot D^{-1}(L\mathbf{x}^{(k+1)} + U\mathbf{x}^{k} + \mathbf{b}) \tag{6}$$

with $A = D - M$, $M = L + U$ and D diagonal, L strictly lower and U strictly upper triangular matrices. The SOR method can be seen as a relaxation procedure applied on GS scheme, i.e.

$$\mathbf{x}^{(k+1)^*} = (1-\omega)\cdot\mathbf{x}^{(k)} + \omega\cdot\mathbf{x}^{(k+1)} \tag{7}$$

so that the new iterate $\mathbf{x}^{(k+1)^*}$ is a combination of the iterate $\mathbf{x}^{(k+1)}$ computed by the basic method, the GS in our case, and the previous iterate $\mathbf{x}^{(k)}$. In the case where $\omega = 1$ SOR is reduced to simple GS; for $0 < \omega < 1$ the method is sometimes called underrelaxation, while for $1 < \omega < 2$ it is called overrelaxation. SOR can be expressed point-wise as follows:

$$x_i^{(k+1)} = (1-\omega)\cdot x_i^{(k)} + \omega\cdot a_{ii}^{-1}\left(b_i - \sum_{j=1}^{i-1} a_{ij}\cdot x_j^{(k+1)} - \sum_{j=i+1}^{a} a_{ij}\cdot x_j^{(k)}\right) \tag{8}$$

in order to implement this iterative method, the IPNN of Figure 2 is considered, with the following configuration and training procedure (see Figure 3).

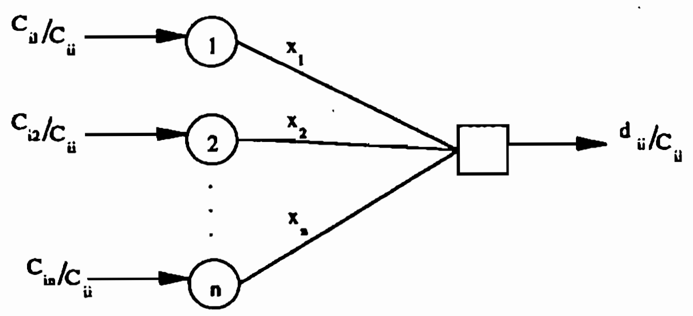

Figure 3 SOR neural network for *i*th iteration.

In the beginning of the calculation the interconnection weights are set equal to the initial approximation vector elements $\mathbf{x}^{(0)}$. Then, a series of matrix rows are presented to the input nodes, in the sequence $\mathbf{c}_1^T, \mathbf{c}_2^T, \ldots, \mathbf{c}_i^T, \ldots, \mathbf{c}_n^T$. The elements of matrix C are defined such that $C = D^{-1}A$. The presentation of all n rows of matrix C completes a SOR iteration step.

During the presentation of $\mathbf{c}_i^T$ the IPNN calculates the IP:

$$\mathbf{c}_i^T \mathbf{x}^{(0,i)} = d_i \qquad \text{or} \qquad \sum_{j=1}^{a} c_{ij} \cdot x_j^{(0,i)} = d_i \tag{9}$$

where the superscript i for the weight vector $\mathbf{x}^{(0)}$ denotes the index of the weight to be modified on this step. For each SOR iteration step i takes the values $i = 1, 2, \ldots, n$, with

$$\mathbf{x}^{(0,1)} = \mathbf{x}^{(0)} \qquad \text{and} \qquad \mathbf{x}^{(1,1)} = \mathbf{x}^{(1)} = \mathbf{x}^{(0,n+1)}.$$

The IP result is then compared to g_i, where vector $g = D^{-1}\mathbf{b}$. This comparison leads to the application of a modified Delta Rule, only for the ith element of $\mathbf{x}^{(0)}$. Thus the modified training procedure has the form:

$$\Delta\mathbf{x}^{(0,i)} = \omega(g_i - d_i)\mathbf{e}_i \qquad \text{and} \qquad \mathbf{x}^{(0,i+1)} = \mathbf{x}^{(0,i)} + \Delta\mathbf{x}^{(0,i)} \tag{10}$$

where $\mathbf{e}_i$ is a $(n \times 1)$ vector with all zero elements, except of a unity element in position i. Thus, in general there are n applications of the modified Delta Rule, for $i = 1, 2, \ldots, n$ for step k of the SOR method, with $k = 0, 1, 2, \ldots$. This procedure, expressed more generally for SOR iteration step k, takes the form:

$$\mathbf{c}_i^T \mathbf{x}^{(k,i)} = d_i^{(k)} \qquad \text{or} \qquad \sum_{j=1}^{a} c_{ij} \cdot x_j^{(k,i)} = d_i^{(k)} \tag{11}$$

$\Delta\mathbf{x}^{(k,i)} = \omega(g_i - d_i^{(k)})\mathbf{e}_i$ and $\mathbf{x}^{(k,i+1)} = \mathbf{x}^{(k,i)} + \Delta\mathbf{x}^{(k,i)}$ with $\mathbf{x}^{(0,1)} = \mathbf{x}^{(0)}$ and $\mathbf{x}^{(k+i,1)} = \mathbf{x}^{(k+i)} = \mathbf{x}^{(k,n+i)}$ for $k = 0, 1, 2, \ldots, n$ and $i = 1, 2, \ldots, n$.

RESULT 2.1 *The operation of the IPNN in Figure* 2 *as modified in Section* 2.1 *is equivalent to the SOR iterative method. This NN is termed SORNN.*

Proof 2.1

The value of $x_i^{(k,i+1)}$ at the beginning of step $i+1$, or at the end of step i, will be:

$$x_i^{(k,i+1)} = x_i^{(k,i)} + \Delta x_i^{(k,i)} = x_i^{(k,i)} + \omega(g_i - d_i^{(k)})$$

$$= x_i^{(k,i)} + \omega(g_i - \mathbf{c}_i^T \mathbf{x}^{(k,i)}) = x_i^{(k,i)} + \omega \left(g_i - \sum_{j=1}^{a} c_{ij} x_j^{(k,i)} \right).$$

Since the jth element of weight vector $\mathbf{x}^{(k)}$ changes only during step j, as denoted by vector $\mathbf{x}^{(k,i)}$ it can be said that

$$x_j^{(k,i)} = x_i^{(k,1)} = x_j^{(k)}, \qquad \text{if} \quad j < i \quad \text{and}$$

$$x_i^{(k+1,1)} = x_j^{(k+1)}, \qquad \text{if} \quad j \geq i.$$

Thus the weight adaptation equation can be written as

$$x_i^{(k+1)} = x_i^{(k)} + \omega \left(g_i - \sum_{j=1}^{i-1} c_{ij} \cdot \mathbf{x}_j^{(k+1)} - \sum_{j=i+1}^{a} c_{ij} \cdot x_j^{(k)} \right)$$

$$= \mathbf{x}_i^{(k)} + \omega \cdot a_{ii}^{-1} \left(b_i - \sum_{j=1}^{i-1} a_{ij} \cdot \mathbf{x}_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} \cdot x_j^{(k)} \right)$$

$$= \mathbf{x}_i^{(k)} + \omega \cdot a_{ii}^{-1} \left(b_i - \sum_{j=1}^{i-1} a_{ij} \cdot x_j^{(k+1)} - \sum_{j=i+1}^{a} a_{ij} \cdot x_j^{(k)} \right)$$

which is equivalent to the point notation of the SOR method.

The SORNN can be used for implementing the GS method with the choice of $\omega = 1$. ■

2.2. *Jacobi and Jacobi Extrapolated Methods*

Alternatively, a global training (weight adaptation) procedure can be followed. This procedure is obtained by a comparison of the SOR and Jacobi-based methods: while GS is sequential, in the sense that the ith element of the new vector iterate is calculated after element $i-1$, the Jacobi method is global, since all elements of the new vector are computed at once. This is expressed in the following equations

$$\mathbf{x}^{(k+1)} = (1-\omega) \cdot \mathbf{x}^{(k)} + \omega \cdot D^{-1}(M\mathbf{x}^{(k)} + \mathbf{b}) \tag{12}$$

with $A = D - M$. This is the Jacobi Extrapolated method, which leads to the basic Jacobi method for $\omega = 1$. The point notation is

$$x_i^{(k+1)} = (1-\omega) \cdot x_i^{(k)} + \omega \cdot a_{ii}^{-1} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{a} a_{ij} \cdot x_j^{(k)} \right). \tag{13}$$

In terms of the NNs discussed in Section 1 the global training procedure can be achieved by means of the n IPNNs, as in Figure 3 with the following configuration: Initially, the interconnection weights of each one of the n IPNNs are set equal to the initial approximation vector elements $\mathbf{x}^{(0)}$, so that all IPNNs have identical weight distribution. Then, the ith row of matrix C, i.e. $\mathbf{c}_i^T$, is presented to the input nodes of the ith IPNN, where matrix C is defined as $C = D^{-1}M$. Thus, all rows of C are presented simultaneously to the NN. This presentation, followed by the application of the weight adaptation procedure, completes a Jacobi iteration step.

During the kth step of the Jacobi method, the presentation of $\mathbf{c}_i^T$ in the ith IPNN calculates the IP:

$$\mathbf{c}_i^T \mathbf{x}^{(k,i)} = d_i^{(k)} \qquad \text{or} \qquad \sum_{j=1}^{a} c_{ij} \cdot x_j^{(k,i)} = d_i^{(k)} \tag{14}$$

where the superscript i of the weight vector denotes the IPNN involved. The IP result is then compared to g_i, where vector is defined as $g = D^{-1}\mathbf{b}$. This comparison leads to the application of a modified Delta Rule as follows. The ith IPNN computes the error that corresponds to the ith element of the vector iterate.

$$\Delta\mathbf{x}^{(k,i)} = \omega(g_i - d_i^{(k)})\mathbf{e}_i \qquad \text{or} \qquad \Delta x_i^{(k,i)} = \omega(g_i - d_i^{(k)})e_i \tag{15}$$

where $\mathbf{e}_i$ is a $(n \times 1)$ vector with all zero elements, except of a unity element in position i. Then, this error is used for updating the ith interconnection weight of all IPNNs, such that

$$x_i^{(k+1,j)} = x_i^{(k,j)} + \Delta x_i^{(k,i)} \qquad \text{for} \quad j = 1, 2, \ldots, n. \tag{16}$$

Therefore it can be said that the new weight vector, identical for all IPNNs is now

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \sum_{i=1}^{a} \Delta\mathbf{x}^{(k,i)}. \tag{17}$$

A new step of the Jacobi procedure can start with the application of the rows of matrix C in the inputs of the n IPNNs.

RESULT 2.2 *The operation of the n IPNNs as described in Section 2.2 is equivalent to the Jacobi Extrapolated iterative method. This NN is termed JENN.*

Proof 2.2 The value of $x_i^{(k+1,i)}$ at the beginning of step $k+1$, or at the end of step k, will be modified by the result of the ith IPNN:

$$x_i^{(k+1,i)} = x_i^{(k,i)} + \Delta x_i^{(k,i)} = x_i^{(k,i)} + \omega(g_i - d_i^{(k)})$$

$$= x_i^{(k,i)} + \omega(g_i - \mathbf{c}_i^T \mathbf{x}^{(k,i)}) = x_i^{(k,i)} + \omega \left(g_i - \sum_{j=1}^{a} c_{ij} x_j^{(k,i)} \right).$$

Since the ith element of the weight vectors of all IPNNs, denoted by $x_i^{(k,j)}$, $j = 1,2,\ldots,n$ undergoes the same modification for step k it can be said that $x_i^{(k,j)} = x_i^{(k)}$. Thus the weight adaptation equation can be written as

$$x_i^{(k+1)} = x_i^{(k)} + \omega \left(g_i - \sum_{j=1}^{a} c_{ij} \cdot x_j^{(k)} \right) = x_i^{(k)} + \omega \cdot a_{ii}^{-1} \left(b_i - \sum_{j=1}^{a} a_{ij} \cdot x_j^{(k)} \right)$$

$$= (1-\omega) \cdot x_i^{(k)} + \omega \cdot a_{ii}^{-1} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{a} a_{ij} \cdot x_j^{(k)} \right)$$

which is equivalent to the point notation of the Jacobi Extrapolated method.

The ith IPNN is responsible of producing the appropriate modification of the weight of the ith interconnection of all n IPNNs, so that the new weights are simultaneously calculated by all IPNNs. The same JENN with the modified Delta Rule can be used for implementing the basic Jacobi method with $\omega = 1$. ■

3. MATRIX EQUATIONS AND TRAINING PROCEDURES

The methods that have been previously discussed for linear system solution can be extended to cover the solution of matrix equations of the form $AX = B$, where A, X and B are $(n \times n)$, $(n \times m)$ and $(n \times m)$ matrices. A matrix equation can be seen as a set of m systems of linear equations with common coefficient matrix A. Therefore it is possible to use m NNs (of the SORNN or JENN types) in order to solve those equations. Notice that there is common input for all NNs involved, since the coefficient matrix is the same for all systems. Thus, in the case of the SORNN the configuration is exactly as the GENN of Figure 1 with the modified Delta Rule training procedure. In the case of JENN the configuration involves m GENNs working in parallel.

The weight adaptation computations are matrix expansion of the vector forms in eqs. (10)–(11) for SORNN and eqs. (15)–(17) for JORNN, and they should be compared to the GENN training procedure in eqs. (2)–(3). For SORNN the basic relations are in matrix form

$$\begin{aligned} &\mathbf{c}_i^T X^{(k,i)} = \mathbf{d}^{(k)T}, \qquad \Delta X^{(k,i)} = \omega(\mathbf{g}_i^T - \mathbf{d}_i^{(k)T})E_i \qquad \text{and} \\ &X^{(k,i+1)} = X^{(k,i)} + \Delta X^{(k,i)} \qquad \text{with} \quad X^{(0,i)} = X^{(0)} \quad \text{and} \\ &X^{(k+i,i)} = X^{(k+i)} = X^{(k,n+i)}, \qquad i = 1,2,\dots,n \quad \text{and} \quad k = 0,1,2,\dots \end{aligned} \tag{18}$$

where the superscript i of the weight matrix denotes the row of the synaptic matrix that is currently updated. The target output matrix is partitioned in row vectors g_i^T and E_i is a compatible matrix with 1's only in row i and all other elements equal to 0. Thus each full iteration k has n steps, each step updating one row of the weight matrix X. Similarly for the JENN the weight adaptation scheme is

$$\begin{aligned} &CX^{(k)} = D^{(k)}, \qquad \Delta X^{(k)} = \omega(G - D^{(k)}) \qquad \text{and} \\ &X^{(k)} = X^{(k)} + \Delta X^{(k)}, \qquad k = 1,2,\dots . \end{aligned} \tag{19}$$

A special case of the matrix equation problem is the solution of the system $AX = I$ 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 the previously discussed types in order to invert a matrix A. Alternatively the serial use of one NN is possible, where the resulting matrix is produced column by column in a series of n steps.

Following the same line of thinking the NNs developed therein can be used for the solution of general matrix equations of the form

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

where A, X and B are matrices of size $(m \times n)$, $(n \times p)$, $(m \times p)$ respectively. This can be achieved by solving the equivalent Normal Equations systems [1]

$$A^T AX = A^T B. \tag{21}$$

Of special interest is the solution of the system

$$A^T AX = A^T I \tag{22}$$

so that the solution $(n \times m)$ matrix X is equal to $(A^T A)^{-1} A^T$ and satisfies the Moore–Penrose conditions of the Generalized Inverse of the $(m \times n)$ matrix A, A^+ [1, 3]. Thus, equations (20) and (21), which are equivalent, have the general solution

$$X = A^+ B. \tag{23}$$

The importance of this result in neural computing has been underlined in many occasions [10, 3, 8, 7]. As explained in the discussion of the GENN, the operation of such NNs is outlined in terms of pattern matching. A set of pattern vectors $\mathbf{a}^{(k)}$, $k = 1,2,\dots,m$, is associated to another set $\mathbf{b}^{(k)}$, $k = 1,2,\dots,m$, a pattern matching function usually termed as heteroassociation, or, alternatively, a set of vectors may be associated to itself (autoassociation). Each input pattern vector $\mathbf{a}^{(k)}$ is encoded in n vector elements, while the output pattern vector $\mathbf{b}^{(k)}$ is be encoded in p elements.

The NN operation can be expressed in matrix terms as follows: Given an $(m \times n)$ input pattern matrix $A = [\mathbf{a}^{(1)}, \mathbf{a}^{(2)}, \ldots, \mathbf{a}^{(m)}]^T$, an $(m \times p)$ output matrix $B = A = [\mathbf{b}^{(1)}, \mathbf{b}^{(2)}, \ldots, \mathbf{b}^{(m)}]^T$ and an $(n \times p)$ weight matrix W that corresponds to the NN interconnection topology chosen. The training procedure, is an iterative method of modifying the weight matrix so that the discrepancy between actual and desired output is minimized

$$\text{minimize}(\mathbf{u}^{(k)} - \mathbf{b}^{(k)}) \qquad \text{or} \qquad \text{minimize}(\mathbf{a}^{(k)T} W - \mathbf{b}^{(k)T}), \qquad k = 1, 2, \ldots, m.$$

The recall, i.e. pattern matching, operation is the calculation of an output pattern for a given input.

It has been noticed that the training procedure is equivalent to the solution of equation (20) for $W = X$, through the calculation of the generalized inverse of A, A^+:

$$W = A^+ B, \qquad \text{for heteroassociation}$$

$$W = A^+ A, \qquad \text{for autoassociation.}$$

The recall procedure corresponds to the calculation

$$\mathbf{u}^{(k)T} = \mathbf{a}^{(k)T} W \qquad \text{or} \qquad \begin{aligned} \mathbf{u}^{(k)T} &= \mathbf{a}^{(k)T} A^+ B, \quad \text{for heteroassociation} \\ \mathbf{u}^{(k)T} &= \mathbf{a}^{(k)T} A^+ A, \quad \text{for autoassociation.} \end{aligned} \tag{24}$$

Further, notice that the least squares computation of the generalized inverse has been achieved through the solution of the Normal Equations system that corresponds to the general matrix equation system. Thus, it is concluded that iterative methods discussed herein can be applied as training procedures for NNs as follows: Given the pattern matrices A and B, defined as in equations (20) to (23), the Normal Equations system is calculated. Then the generalized inverse matrix is calculated by means of one NN of the SORNN or JENN types. Finally, the pattern matching operation is performed as in equation (24).

3.1. Experimental Results for Linear Systems

In the following sections the SORNN and JENN procedures are compared to IPNN of Figure 2 for a number of systems of various sizes.

A number of linear systems of equations of various sizes ($n = 5$, $n = 10$, $n = 20$) and with randomly chosen elements are solved by means of the three methods presented. The first experiment investigates the effect of ω on the speed of convergence for each method, with ω taking values between 0 and 2.

Three diagrams (Figures 4–6) are plotted for SORNN, JENN, and GENN for the three groups of systems, showing the number of iterations required, as ω varies between 0 and 2:

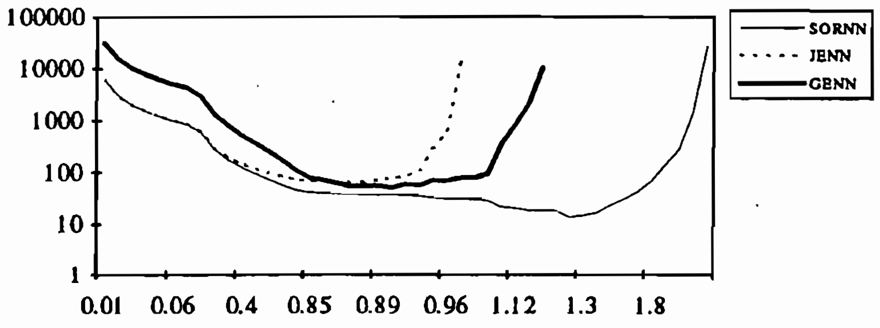

Figure 4 Diagram 1 (5 × 5 system, random values).

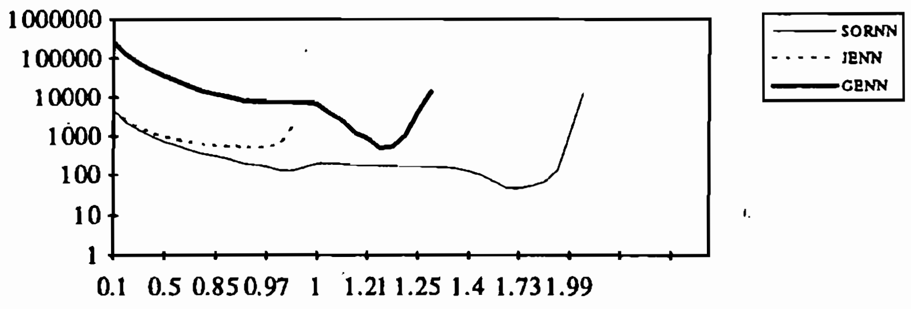

Figure 5 Diagram 2 (System 10 × 10, random values).

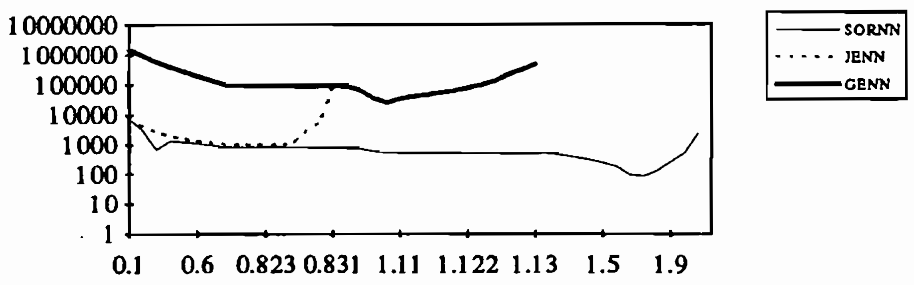

Figure 6 Diagram 3 (System 20 × 20 system, random values).

There exists an optimal ω, for which each one of the above networks requires the minimum number of iterations in order to converge. SORNN reaches a solution for any value of ω between 0 and 2, while JENN and GENN fail to converge for values of ω greater than an upper limit ω_{max}. Note that as problem size increases,

ω_{max} decreases for JENN, while it increases for GENN. As ω approaches the upper or lower limit, the number of iterations increases dramatically.

For all systems examined, it can be said that SORNN converges faster than the other NNs. JENN behaves better than GENN when ω is not close to the upper limit, GENN is more robust than JENN but the minima achieved are very poor compared to those of JENN in terms of the number of iterations.

Another experiment has been conducted, where for a fixed value of ω, i.e. $\omega = 0.4$, a number of randomly chosen linear systems of size $n = 5$ are solved by means of the three NNs. The results are plotted on Figure 7 in increasing order of iterations for SORNN:

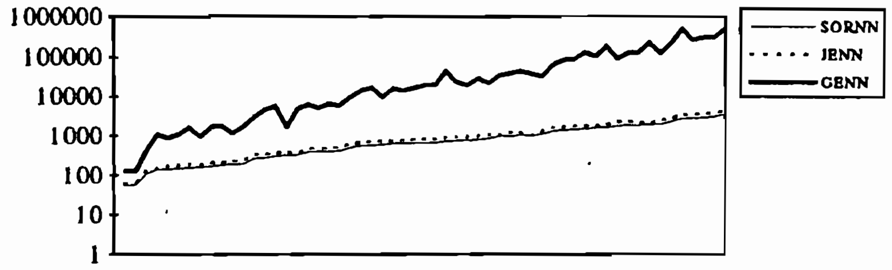

Figure 7 Diagram 4 (77 systems, 5 × 5, $w = 0.4$).

Again, SORNN is consistently better than JENN and GENN, (as it is expected from Figure 4), with a minor exception of a system, for which JENN converges faster than SORNN.

3.2. *Experimental Results for Matrix Equations*

Similar experiments have been conducted for the Matrix Equation system. The results are shown in Figures 8 and 9:

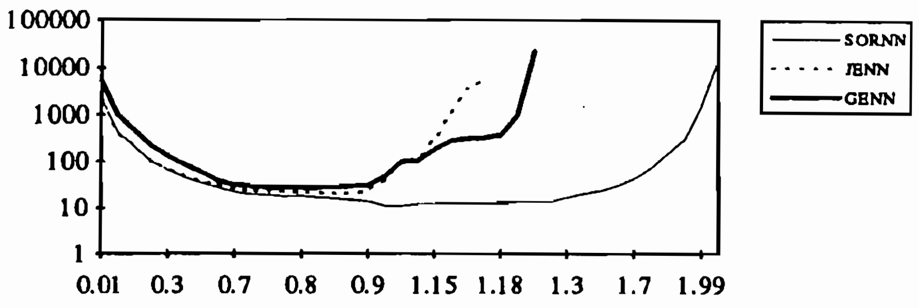

Figure 8 Diagram 5 (system 5 × 5 with random values).

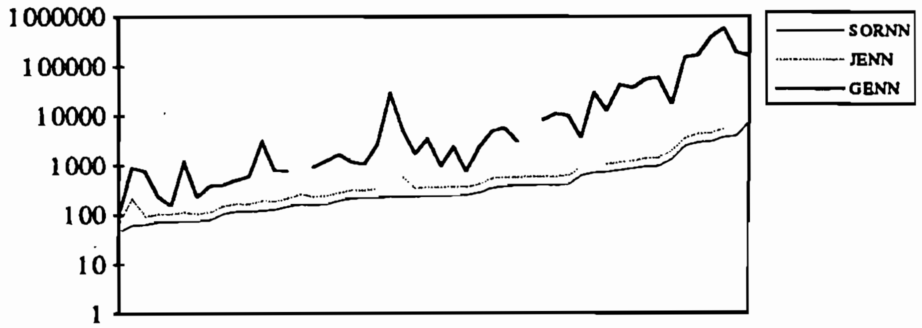

Figure 9 Diagram 6 (50 systems, 5×5, $\omega = 0.7$).

In the first diagram the behavior of the three NNs is plotted over different values of ω. In the second diagram the convergence of the three methods for a fixed value of ω is presented.

The results are identical to those obtained in the case of the linear systems of equations.

4. CONCLUSIONS

This paper describes the implementation of iterative methods for solving linear systems of equations by means of feed-forward artificial neural networks. Initially a generic neural network is described and then simple networks are derived for realizing basic matrix operations. Further the basic networks are combined and the Delta Rule for network training is modified in order to lead to the realization of iterative methods for solving systems of linear equations. Finally extensions to matrix based iterative procedures are presented, and the application of the iterative methods in general feed-forward artificial neural network training algorithms.

Further investigations include the comparison of the behaviour of the iterative methods to that of classical training procedures, such as the Delta Rule of the generic NN, using nonlinear neuron function, as well as multi-layer NNs. Conversely, it would be of interest the investigation of the numerical behaviour of the neural network training procedures if applied in model numerical problems normally being solved with traditional iterative methods.

Another recent concept is the development of multi-layer feed-forward neural networks, termed as Structured Networks, which are envisaged as a tool for the migration of massively parallel algorithms onto neural network architectures [11–13]. Following this method it would be of importance the investigation of the possible general migration of algorithms developed for fine-grain or massively parallel architectures (e.g. systolic, optical, cellular, connetionist) and their neural network implementations, and vice versa, i.e. the migration of neural network algorithms to other parallel architectures.

References

[1] G. Dahlquist and A. Biorg, Numerical methods, Prentice Hall, 1974.
[2] L. Hageman and D. Young, Applied iterative methods, Academic Press, 1981.
[3] T. Kohonen, Self-organization and associative memory, Springer-Verlag, 1984.
[4] A. J. Maren, C. T. Harston and R. M. Pap, Handbook of neural computing applications, Academic Press, 1990.
[5] F. Rosenblatt, Principles of neurodynamics, Spartan Books, 1962.
[6] D. Rumelhart and J. McClelland, Parallel distributed processing: Explorations in the microstructure of cognition, MIT Press, 1986.
[7] P. K. Simpson, *Artificial Neural Systems*, Pergamon Press, 1990.
[8] G. Stone, An analysis of the Delta Rule and the learning of statistical applications, in *Parallel Distributed Processing*, MIT Press, 1986.
[9] R. S. Varga, Matrix iterative analysis, Prentice Hall, 1962.
[10] B. Widrow, Learning phenomena in layered neural networks, *Proc. of IEEE 1st Int. Conf. on Neural Networks*, 1987.
[11] L. X. Wang and J. M. Mendel, Structured trainable networks for matrix albebra, *Proc. Int. Joint Conf. on Neural Networks*, 1990.
[12] L. X. Wang and J. M. Mendel, Three dimensional structured networks for matrix equation solving, *IEEE Trans. Comput.* **40**, 12 (1991).
[13] L. X. Wang and J. M. Mendel, Parallel structured networks for solving a wide variety of matrix algebra problems, *J. of Parallel and Distributed Computing* **14** (1992).
[14] D. M. Young, Iterative solution of large linear systems, Academic Press, 1971.