A Distributed Discrete-Time Neural Network Architecture for Pattern Allocation and Control*

Anthony T. Chronopoulos,
Dept. of Computer Science,
Univ. of Texas at San Antonio,
6900 N. Loop 1604 West,
San Antonio, TX 78249.
atc@cs.utsa.edu

Jagannathan Sarangapani
Dept. of Electrical and Computer Engineering,
Univ. of Missouri-Rolla,
1870 Miner Circle,
Rolla, Missouri 65401.
sarangap@umr.edu

Abstract

The focus of this study is how we can efficiently implement a novel neural network algorithm on distributed systems for concurrent execution. We assume a distributed system with heterogeneous computers and that the neural network is replicated on each computer. We propose an architecture model with efficient pattern allocation that takes into account the speed of processors and overlaps the communication with computation. The training pattern set is distributed among the heterogeneous processors with the mapping being fixed during the learning process. We provide a heuristic pattern allocation algorithm minimizing the execution time of neural network learning. The computations are overlapped with communications. Under the condition that each processor has to perform a task directly proportional to its speed, we show that the pattern allocation is a polynomial-time problem, solvable by dynamic programming.

1 Introduction

There is no consensus on how to simulate artificial neural networks on parallel machines. During the last years, researchers have been trying to achieve maximal performance on their favorite (or available) parallel machine. Neural networks were implemented on many parallel architectures (see e.g. [5, 6, 7, 9]).

Backpropagation or other multilayer neural networks can be parallelized by *network-partitioning*, by *pattern-partitioning*, or by a combination of these two schemes. In *network-partitioning*, nodes and weights of the neural network are partitioned among different processor s, and thus the computations of node activations, node errors, and weight changes are parallelized. The idea of *pattern-partitioning* [11] is to distribute the training examples over the processors, i.e. it slices the training set and it as signs one slice to each processor while keeping a complete copy of the whole network in each processor node.

The implementation of a neural network on a heterogeneous parallel architecture gives rise to a hard problem. This problem concerns the optimal mapping of the network and of the training patterns among the heterogeneous processors. This optimization model is generally a NP-complete integer (or mixed) programming problem which can be solved either directly (for instance, by branch-and-bound), or simplified heuristically to a polynomial problem. The mapping algorithms can be *static* or *dynamic*. In the *static* case, we assume that the mapping is unchanged throughout the learning process. In the *dynamic* case, we assume that the background workload is time varying; hence, it may be necessary to perform a remapping as workload changes.

Only a few mapping schemes have been reported to implement neural network algorithms on parallel architectures with heterogeneous processors. Chu and Wah [5] presented an approximation algorithm for the mapping of large neural networks on multi-computers, given a user-specified error degree that can be tolerated in the final mapping. Saratchandran *et al.*, [7] optimized pattern parti tioning in backpropagation learning on a heterogeneous array of transputers. They solved the optimization problem in two ways: by branch-and-bound and by genetic algorithms.

There exist other approaches for optimal data partitioning in distributed systems. Notable are some works in divisible load theory [3, 12] where a divisible load can

*This research was supported, in part, by research grants from (1) NASA NAG 2-1383 (1999-2001), (2) State of Texas Higher Education Coordinating Board through the Texas Advanced Research/Advanced Technology Program ATP 003658-0442-1999, (3) NSF ECS 9985739.

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 IEEE

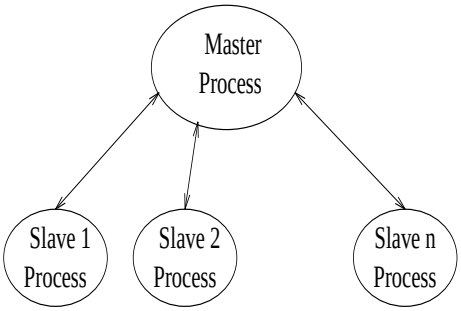

Figure 1. The master-slave model.

be arbitrarily partitioned and distributed to more than one computer to achieve a faster execution time. For non-divisible or discrete loads there are (non-optimal) strategies which have been used such as the *equal* and *rectilinear* allocation [10]. These strategies have been used for grid problems but not for neural networks.

In this paper, we consider the pattern-partitioning scheme. This scheme is particularly suited for feedforward neural networks where the size of the training set is large compared to the size of the network. The pattern-parti tioning is a coarse grained method because the number of processors is limited by the number of patterns.

We next propose a novel feedforward architecture for a neural network pattern association algorithm based on dynamic programming.

Distributed Architecture Model:

(1) We consider a dedicated master-slave architecture based on a network of heterogeneous computers (see Figure 1). The computers are assumed to have the capability to perform computation and communication simultaneously which is the case in most existing systems.

(2) We consider the pattern partitioning scheme for mapping our multilayer neural network algorithm for learning pattern associations onto the computers.

Pattern allocation schemes:

(1) We propose a straightforward proportional allocation which takes into account the CPU speed of the computers but does not overlap communication with computation.

(2) We then propose a novel pattern allocation algorithm which takes into account the CPU speed of the processors and overlaps computation and communication.

2 A Neural Network Pattern Association Model

In pattern association, the objective is to learn the associations via training or by physically associating the patterns with features. One of the commonly used method in the literature for learning the association or a relationship among inputs(patterns) and outputs(features or mapping) is using neural networks as they have proven to have the universal approximation property. Typically, the training process of a NN (neural network) is quite involved as the patterns could be taken from different regions and the mapping(or association) could be nonlinear. Therefore, NN training via updating the weights take considerable time and further the weights may not converge unless a suitable and mathematically proven training algorithm is deployed. If the weights do not converge, the pattern associations are not possible. To address these problems, we are proposing (i) a Distributed Neural Network learning Algorithm for faster learning and (ii) a proof of convergence of the NN weights when the proposed weight tuning scheme is used.

In our past work [2], we have used Backpropagation and implemented the parallelized backpropagation algorithm via network partitioning, or by pattern partitioning or by combination of these schemes. Since the Backpropagation algorithm is not proven to converge, in this paper , a novel NN weight tuning scheme is described. The proposed NN training algorithm via pattern partitioning will not only improve the learning time (execution time) but also the NN learns the nonlinear pattern association(or mapping), which is proven using a Lyapunov based analysis.

In this paper, a novel learning scheme is proposed for a multilayer NN. Patterns are partitioned and allocated to different processors and trained using the proposed algorithm. It was shown using the Lyapunov stability analysis that the proposed NN weight tuning algorithm converges to a bounded set (error) when initialized at zero initially and to an average value thereafter. Here we have considered two examples one for learning a mapping via pattern partitioning and the other for real-time control.

Pattern Association

The ability of neural networks to approximate large class of nonlinear systems makes them prime candidates for the identification of nonlinear pattern associations [8]. Let us consider a multi-input multi-output nonlinear association, to be constructed by a NN, as

$$x(k+1) = f(\, x(k), x(k-1), \ldots, x(k-n+1)\,) + d(k) \quad (1)$$

where: $f(.) \in \mathbf{R}^n$ is the nonlinear mapping or association to be constructed, $x(k) \in \mathbf{R}^n$ is a vector consisting of patterns, $d(k) \in \mathbf{R}^n$ is the disturbance or noise whose bound is given by $||d(k)|| \leq d_M$. Here the patterns are a function of time if k is a time index. If the

association is not a function of time, then k could be a pattern number.

The objective is to construct a suitable model that when subjected to the same input $x(k)$ will produce an output $\hat{x}(k+1)$ such that the actual output $x(k+1)$ and the estimated output $\hat{x}(k+1)$ are very close in some sense. Now taking the structure of the model same as that of the mapping to be constructed, the model can be written as:

$$\hat{x}(k+1) = \hat{f}(x(k), x(k-1), \ldots, x(k-n+1)) \quad (2)$$

Define the error in association as:

$$e(k) = x(k) - \hat{x}(k) \quad (3)$$

Then the error in pattern association in the next interval is given by

$$\begin{aligned} e(k+1) &= x(k+1) - \hat{x}(k+1) = \\ &= f(x(k), \ldots, x(k-1)) \\ &- \hat{f}(x(k), \ldots, x(k-1)) + d(k) \end{aligned} \quad (4)$$

Rewriting the above equation in a compact form as

$$e(k+1) = \tilde{f}(.) + d(k) \quad (5)$$

where the functional estimation errors are given by:

$$\tilde{f} = f(.) - \hat{f}(.) \quad (6)$$

This is an error system where the association error system is driven by functional estimation error. In the remainder of this paper we focus on selecting NN training algorithms that guarantees the convergence of the pattern association error.

3 Distributed NN Architecture

One of the most common programming model used in developing distributed systems application is the master-slave model. In this model we have a control program called master and a number of slave programs. The master program is responsible for spawning sla ve programs, initialization and collection of results. The slaves programs perform the computation on data allocated by the master or by themselves. The master-slave model involves no communication among the slaves. Only the master can communicate with sl aves by message-passing. The structure of the master-slave model is shown in Figure 1.

We now describe the mapping of our proposed multilayer NN algorithm onto the computers. Our NN algorithm trains a given feedforwarding neural network for a given set of learning patterns. The training of the neural network can be viewed as discovering values for its weights in order to match the effective outputs of the network with the desired outputs, for each input pattern.

In our proposed NN learning, weights can be updated in two ways:

1. In the *per-pattern regime* the weights are updated after each training pattern is presented;
2. In the *set-training regime* the weight increments are computed for each training pattern. The increments are summed for all patterns and the weights are updated with the total increment after all patterns have been presented one time [13].

Pattern-partitioning schemes for parallelization are applicable only to set-training updating [9]. However, our pattern-partitioning scheme can be based on either a per-training regime or a set-training regime. Here we intend to use both and evaluate the differences.

We assume a master-slave model with n slaves (processors). The training set S is partitioned into n subsets, $S_i,\ i = 1, 2, ..., n$. These training subsets are distributed to the n processors. Each slave process contains a complete copy of the whole neural network.

One *epoch* of our proposed NN algorithm has the following coarse description:

1. The weight changes and bias for the current epoch are initialized to zero.
2. Each slave process (P_i) carries out the training phase for each pattern assigned to it.
3. Each slave process also accumulates the weight changes and error according to the local patterns.
4. Each slave process sends the weight changes and errors to the master. The master process computes the sum of all weight changes and of all errors.
5. The master broadcasts the new weights to all slaves. The weights are updated on each slave. The master checks if the convergence is reached.

Remark: We do not use a reduce operation for implementing the communication required in 4. above, because in a LAN the reduction tree must be mapped to a single bus and this is very inefficient. More importantly, the master is needed to assign patterns according to our algorithm.

The timing diagram for an epoch is shown in Figure 2. In this diagram we used the following notations:

- $P_1, P_2, \ldots, P_n$ are the slave processes.
- M is the master process.

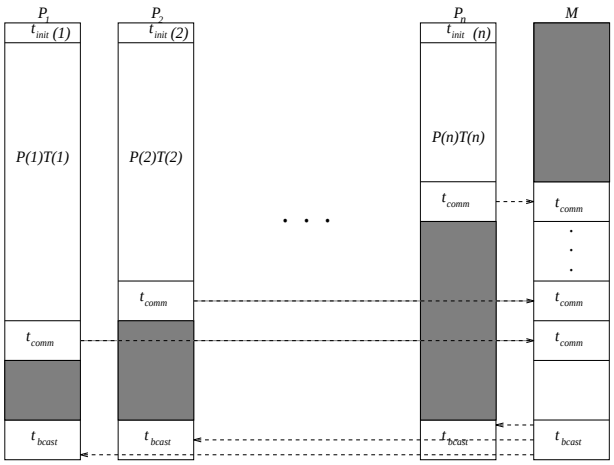

Figure 2. The timing diagram for an epoch

- $t_{init}(i)$ is time taken to initialize the weight changes and error.
- $P(i)$is the number of patterns allocated to the slave process P_i.
- $T(i)$is time taken to perform the training phase of the algorithm for a single pattern.
- $t_{comm}$is time taken to send the weight changes and errors from the slaves to the master.
- t_{bcast} is time taken to broadcast the updated weights.
- T_n is the parallel execution time on n processors.

In order to obtain the minimum epoch time we have to overlap communication time (t_{comm}) with computation time and to find a proper pattern distribution among the processors.

4 Optimization of pattern mapping

We present next an optimization approach (see [2] for details). The allocation of tasks (or jobs) in distributed systems may be considered a special case of task scheduling, without imposed precedence relations in the execution of tasks. The purpose of a task allocation technique is to find some task assignment in wh ich the total cost due to interprocessor communication and task execution is minimized. The task allocation problem is known to be NP-complete [1]. Optimal algorithms are obtained in very restricted cases. For instance, simplified versions of task allocation could be solved by dynamic programming [4], a method which leads usually to a polynomial solution. The intractability of the problem has led to the introduction of many heuristics.

Our pattern-mapping optimization problem is a task allocation problem. We have to distribute p patterns among n heterogeneous processors, minimizing T_n. In other words, we have to minimize T_n for one epoch, on n heterogeneous processors, consi dering also the weights transmission from each slave to the master. The computations are overlapped with communications, considering the following strategy: each processor has to perform a task directly proportional to its speed. The fastest processor is always the latest one in computing one epoch.

We shall cascade the computation times on the n processors in the following way. The computation time for one epoch on a faster processor has to overlap (as much as possible) with computation plus message passing times on a slower processor. This would make the faster processor to be the last one sending its weights to the master after one epoch. This means also that we have to map more patterns to a faster processor than to a slower one. Hence, we prefer to use the fastest processors over the slowest o nes. Moreover, we shall actually use a subset of the available n processors. This subset consists of the fastest processors.

We use the following notations:

- $\mathbf{P}$ = a vector of n elements, where $P(i) \geq 0$ is the number of patterns assigned to processor i, $1 \leq i \leq n$, $P(1) + \ldots + P(n) = p$.
- $\mathbf{T}$ = a vector of n elements, where $T(i)$ is the processing time for one pattern (and one cycle) assigned to processor i, $1 \leq i \leq n$.

We shall suppose from now on, without restricting the generality, that $T(1) \leq T(2) \leq \ldots \leq T(n)$.

Our objective is to find an optimal vector $\mathbf{P}^*$ $(P^*(i) \geq 0, 1 \leq i \leq n, P^*(1) + \ldots + P^*(n) = p)$, minimizing T_n. In this case, it is obviously better to send more work to the faster processors (those with a low $T(i)$). The first processor (the fastest) has to be the last one sending the weights to the master. Intuitively, T_n is proportional to $T(1)P(1)$ (and this means to $P(1)$, since $T(1)$ is constant). Therefore, we have to minimize $P(1)$. The same result can be achieved if, instead of maximizing the number of patterns processed by the fastest processor, we minimize the number of patterns processed by the slowest processor.

This task allocation problem can be simplified by considering the following assumption.

The vector $\mathbf{P}^*$ $(P^*(i) \geq 0, 1 \leq i \leq n, P^*(1) + \ldots + P^*(n) = p)$ is *optimal* if it minimizes (maximizes) the number of patterns allocated to the fastest (respectively, the slowest) available processor, under the following assumption:

Assumption 1:

$$T(i)P^*(i) \geq T(i+1)P^*(i+1)+t_{comm},\ 1 \leq i \leq n-1 \quad (7)$$

□

This assumption reduces the size of the search space, giving us the possibility to find a polynomial solution to the optimization problem. The meaning of the restrictions is that we always try to have no idle time periods for the fastest processors, by ke eping them busy as much as possible. The result is an unequal pattern distribution with overlapping of communication time and computation time.

For any vector $\mathbf{P}$ $(P(i) \geq 0, 1 \leq i \leq n, P(1)+\ldots+P(n) = p)$, respecting the restrictions:

$$T(i)P(i) \geq T(i+1)P(i+1)+t_{comm}, \qquad 1 \leq i \leq n-1 \quad (8)$$

we have the following two properties (the proofs are obvious, since $T(1) \leq T(2) \leq \ldots \leq T(n)$):

Property 1:

$$P(i) \geq P(i+1), \qquad 1 \leq i \leq n-1 \quad (9)$$

□

Property 2:

$$P(i) = 0 \ \Rightarrow\ P(i+1) = 0, \qquad 1 \leq i \leq n-1 \quad (10)$$

□

Remark: We solve:

$$\min \max_{\mathbf{P}} (T(i)P(i) + t_{comm}) \quad (11)$$

which simplifies to:

$$\min_{P(1)} (T(1)P(1) + t_{comm}) = \min_{P(1)} T(1)P(1) + t_{comm} \quad (12)$$

because of the assumption on ordering the processors according to their CPU speed.

Based on Assumption 1, we can find an optimal solution $\mathbf{P}^*$ by dynamic programming. However, $\mathbf{P}^*$ is not an optimal solution of the general optimization problem (i.e., the pattern mapping optimization without Assumption 1) and this can be easily shown by an example (see Example 2).

4.1 Dynamic programming solution (DP):

We will maximize the number of patterns allocated to the fastest processor, building a gain array g of $n \times p$ elements, where $g(i,j)$, for $1 \leq i \leq n$ and $1 \leq j \leq p$, is the maximum number of patterns allocated to processor i (the slowest) if we distribute j patterns on processors $1, 2, \ldots, i$.

We initialize the array as follows:

$$g(1,j) = j, \qquad j = 1,\ldots,p \quad (13)$$

$$g(i,1) = 0, \qquad i = 2,\ldots,n \quad (14)$$

We have to compute the rest of the elements of g. The optimality principle holds, since Assumption 1 is a recursive relation, and we have:

$$g(i,j) = \quad (15)$$

$$\begin{cases} 0 & \text{if } g(i-1,j-1)T(i-1) \leq T(i)+t_{comm} \\ \max\{k \mid g(i-1,j-k)T(i-1) > kT(i)+t_{comm}\} & \text{otherwise} \end{cases} \quad (16)$$

We can complete g line by line, or column by column. We can reduce the time to compute the gain array using the following observation.

If, for some j, we have $g(i,j) = 0$, then:

$$\text{i.} \qquad g(i+1,j) = 0 \quad (17)$$

(This is in accordance with Property 2.)

$$\text{ii.} \qquad g(i+1,j+1) = 0 \quad (18)$$

After completing g, the solution to our optimization problem can be obtained backwards:

$$P^*(n) = g(n,p)$$
$$P^*(n-1) = g(n-1, p-P^*(n))$$
$$\vdots$$

In general:

$$P^*(i) = g(i, (p-P^*(n)-P^*(n-1)-\ldots -P^*(i+1))), \qquad 1 \leq i \leq n-1 \quad (19)$$

Fact:

The complexity of the DP algorithm is $O(np\log p)$. This follows from: (1) An element $g(i,j)$ can be computed in $O(\log p)$ time using binary search. (2) The whole array can be completed in $O(np^2)$ time. Subsequently, the backward phase of the dynamic programming algorithm is in $\Theta(n)$ time.

4.2 Reducing the complexity of DP algorithm

The complexity of DP algorithm can be reduced based on the following observation. If there is a sufficient number of patterns we can allocate them in two phases. Phase 1: We allocate most patterns in constant time so that all the processors will finish execution at

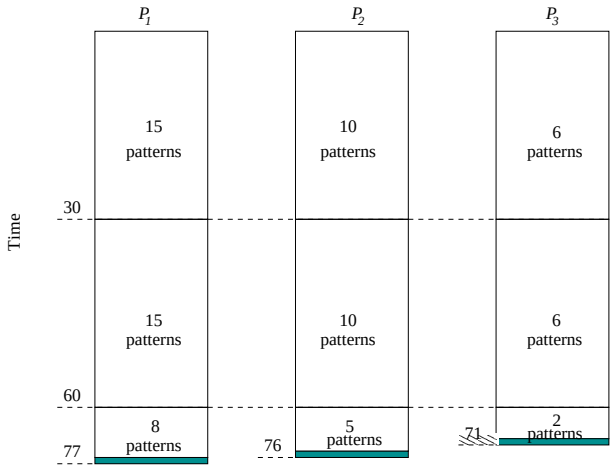

Figure 3. Pattern allocation for Example 1

the same time. Phase 2: We apply DP on the remaining patterns.

We make the following assumptions: (i) all n processors will be used in the most efficient way; (ii) t_{comm} is not negligible compared to the execution time of one pattern on any processor and so we want to overlap communication and computation.

The following example illustrates this technique:

Example 1:

We consider a system of $n = 3$ processors. The total number of patterns is $p = 77$. The execution times for one pattern on each processor are the following: $T(1) = 2, T(2) = 3, T(3) = 5$. We assume the communication time: $t_{comm} = 1$. We compute the least common multiplier (lcm) of $T(i)$, $i = 1, 2, 3$. In this case $lcm = 30$ and it represents the execution time for one time block. We allocate 2 time blocks on each processor as in Figure 3. In this figure the dark segments represents t_{comm}.

A formal description of this procedure is as follows. Let T_{block} be the lcm of $T(i)$, $i = 1, \ldots, n$. We have two cases: (I) when $t_{comm} \leq T_{block}$ and (II) when $t_{comm} > T_{block}$.
Case I:

1. Since the communication is to be overlapped with computation, we subtract the number of patterns $p_c = \sum_{i=1}^{n} \lfloor \frac{(i-1)t_{comm}}{T(i)} \rfloor$ whose execution will overlap with communication. Then, we determine the number k of possible stages of time blocks that can be formed:

$$k = \left\lfloor \frac{p - p_c}{nT_{block}} \right\rfloor$$

2. We allocate $k \lfloor \frac{T_{block}}{T(i)} \rfloor$ patterns to processor i. Then apply DP algorithm on p_c patterns.

Case II:

We find the minimum k_c such that $T'_{block} = k_c T_{block} > t_{comm}$. Then apply the procedure (I) with T'_{block} instead of T_{block}.

Using the procedures described above we can reduce the complexity of DP algorithm to $O(n^5)$. This can be showed as follows:

$$p_c \leq \sum_{i=1}^{n} \frac{(i-1)t_{comm}}{T(i)} \leq$$

$$\sum_{i=1}^{n} \frac{(i-1)t_{comm}}{T(1)} \leq \frac{T_{block}}{T(1)} O(n^2) \qquad (20)$$

From the remark above the complexity of DP is $O(np_c \log p_c)$. This implies that the complexity is $O(n^3 \log n)$. This is an important reduction in complexity because in most practical situations $p \gg n$.
*Remark:*One can generalize this approach to consider the case when it is more efficient to use fewer than n processors. Then all $p_c = \sum_{i=1}^{k} \lfloor \frac{(i-1)t_{comm}}{T(i)} \rfloor$ for $k = 1, \ldots, n$ must be examined.

5 Multilayer NN design

In this paper, a three-layer NN is considered and the convergence analysis is carried out for the error system in equation (6). A novel learning scheme is proposed for the NN. Assume that there exists some constant weights W and V for the three-layer NN such that the nonlinear mapping $f(.)$ in (1) can be written as:

$$f(.) = W^T \phi(V^T \theta(x(k)) + \epsilon(k) \qquad (21)$$

where W is a matrix of hidden-to-output layer weights, V is a matrix of input-to-hidden layer weights $\theta(.)$ is the input layer activation function vector, $\phi(.)$ is the hidden-layer activation function vector, and $\epsilon(k)$ is as approximation error given by $||\epsilon(k)|| < \epsilon_N$.

Assumption: The ideal weights are bounded by $||W|| \leq w_{max}$ and $||V|| \leq v_{max}$ and the hidden layer and the input activation functions are bounded by $||\sigma|| \leq \sigma_{max}, and ||\theta(x)|| \leq \theta_{max}$. Let the NN functional estimate as:

$$\hat{f}(.) = W^T \sigma(\hat{V}^T \theta(x)) = \hat{W}^T \hat{\sigma}(.) \qquad (22)$$

and the errors on weights are:

$$\tilde{W} = W - \hat{W} \qquad (23)$$

$$\tilde{V} = V - \hat{V} \qquad (24)$$

with the hidden layer output error

$$\tilde{\sigma} = \sigma - \hat{} \sigma \tag{25}$$

Using the above equation the error system in (6) can be written as:

$$e(k+1) = e_f(k) + \delta(k) \tag{26}$$

where

$$e_f(k) = \tilde{W}^T(k)\hat{\sigma}(k) \tag{27}$$

and

$$\delta(k) = W^T(k)\tilde{\sigma}(k) + \epsilon(k) + d(k) \tag{28}$$

The next step is to determine the NN weight updates so that the boundedness of the associated error is guaranteed.

6 Weight updates for guaranteed performance

Theorem: Given the nonlinear pattern association system with the NN weight updates, and let the weight tuning be given by:

$$\hat{V}(k+1) = \hat{V}(k) - \alpha\theta(x(k))[\hat{y}(k) + Be(k)]^T \tag{29}$$

$$\hat{W}(k+1) = \hat{W}(k) + \beta\hat{\sigma}(k)e^T(k+1) \tag{30}$$

with $\alpha, \beta > 0$ denoting constant learning rate parameters or adaptation gains, provided that the following conditions hold

$$\alpha||\theta(x(k))||^2 < 2, \quad c_0 < 1 \tag{31}$$

$$\beta||\hat{\sigma}(.)||^2 < 1, \quad \text{where} \quad c_0 = \frac{k_i^2}{2 - \alpha||\theta(x(k))||^2} \tag{32}$$

Then the pattern association error, e(k), and the weight updates are uniformly ultimately bounded.

Proof: Define a Lyapunov function candidate

$$J = e(k)^T e(k) + \frac{1}{\alpha}\tilde{W}^T(k)\tilde{W}(k) + \frac{1}{\beta}\tilde{V}^T(k)\tilde{V}(k) \tag{33}$$

whose first difference ΔJ is given by

$$\Delta J = e^T(k+1)e(k+1) - e^T(k)e(k) + \frac{1}{\alpha}[\tilde{W}^T(k+1)\tilde{W}(k+1) - \tilde{W}^T(k)\tilde{W}(k)] + \frac{1}{\beta}[\tilde{V}^T(k+1)\tilde{V}(k+1) - \tilde{V}^T(k)\tilde{V}(k)] \tag{34}$$

Using the NN weight update laws from (29), (30) and simplifying, one gets:

$$\Delta J \leq (1 - c_0)[||e(k)||^2 - 2\frac{c_1}{1 - c_0}||e(k)|| - \frac{c_2}{1 - c_0}]$$
$$-(1 - \beta||\hat{\sigma}(.)||^2||e_f - \frac{\beta\hat{\sigma}^T\hat{\sigma}\delta(k)}{1 - \alpha\theta(x(k))^T\theta(x(k))}||^2$$
$$-(2 - \alpha\theta(x(k))^T\theta(x(k))||\tilde{V}^T(k)\theta(x(k))$$
$$-\frac{1 - \alpha\theta(x(k))^T\theta(x(k))}{2 - \alpha\theta(x(k))^T\theta(x(k))}(V^T\theta(x(k)) + Be(k)||^2$$

where:

$$\delta_{max} = W_{max}\tilde{\sigma}_{max} + \epsilon_N + d_M$$

$$c_1 = \frac{k_1\theta_{max}V_{max}}{2 - \alpha\theta_{max}^2}$$

$$c_2 = \frac{\delta_{max}^2}{1 - \beta\tilde{\sigma}_{max}^2} + \frac{V_{max}\theta_{max}^2}{2 - \alpha\theta_{max}^2}$$

Since c_0, c_1, c_2 are positive constants, $\Delta J \leq 0$ as long as:

$$||e(k)|| > \frac{1}{1 - c_0}[c_1 + \sqrt{c_1^2 + c_2(1 - c_0)}] \tag{35}$$

Now summing the change in the Lyapunv function from the initial time to the final time

$$|\sum_{k=k_0}^{\infty}\Delta J(k)| = |J(\infty) - J(0)| < \infty \tag{36}$$

since $\Delta J \leq 0$ as long as (31) then (32) hold. The definition of J and the inequality (36) imply that every initial condition in the set X will evolve entirely in X. That is whenever the association error $||e(k)||$ is outside the region defined by (35), $J(e, \tilde{W}_i)$ will decrease. This further imply that $||e(k)||$ will not increase and will remain in X. This demonstrates that the association error $e(k)$ is bounded for all $k \geq 0$ and it remains to show that the weight estimates $\hat{W}$, $\hat{V}$ are bounded. The dynamics relative to error in the weight estimates are:

$$\tilde{V}(k+1) = (I - \alpha\theta(x(k))^T\theta(x(k)))\tilde{V}(k) + \alpha\theta(x(k))[W^T\theta(x(k)) + Be(k)]^T \tag{37}$$

and

$$\tilde{W}(k+1) = (I - \beta\hat{\sigma}(.)^T\hat{\sigma}(.))\tilde{W}(k) + \beta\hat{\sigma}(.)[\delta(.)]^T \tag{38}$$

where the association error is considered bounded. Applying the PE condition, the boundedness of $\tilde{W}(k)$, $\tilde{V}(k)$ and hence boundedness of $\hat{W}(k)$, $\hat{V}(k)$ are assured.

Remark 1: In the above theorem, for the case of patterns that depend upon time, the theorem shows that with

the developed NN weight tuning scheme, the pattern association and weight estimation errors converge to a set as time increases. For the case of patterns that are not a function of time, the time index k now becomes the pattern numbers. In this case, with the patterns used as inputs to the NN over and over again during training, the NN will learn the association since the pattern association error and the weight estimation error converge.

Remark 2: This theorem shows that even when the weights of the NN are initialized at zero or any other value, the pattern association and weight estimation errors converge to a small set. This further implies that if the initial weight values are close enough to their target, the convergence of the pattern association and weight estimation errors become faster than if the initial weights are far off from their targets. However, in the case of pattern association, there is no guarantee that the initial weights will be selected close to their target values except that patterns are allocated based on their locations or regions. In such a case, based on their regions, actual weights from the individual slave processors, can be quite different. By averaging the weight values from several slave processors and subsequently using them as initial weights will push the actual weights closer to their targets in the subsequent cycles of training and hence the overall convergence will be faster. Therefore, by averging the actual weights in the subsequent cycles and using the proposed weight updates, the NN learns the patterns associations efficiently and this leads to a faster error convergence.

Control: In closed-loop adaptive control application, NN identifiers are used to learn the nonlinear mapping and subsequently to use this information for control. Here the $x(k)$ will become the states of the unknown nonlinear system an $f(x(k))$ will be the unknown nonlinear function to be approximated. For the case of identification of such unknown nonlinear systems, the proposed algorithm can be employed. By distributing the patterns, here the states, the nonlinear function can be approximated globally.

References

[1] H. Ali and H. El-Rewini. On the intractability of task allocation in distributed systems. *Parallel Processing Letters*, 4:149–157, 1994.

[2] R. Andonie, A. T. Chronopoulos, D. Grosu, and H. Galmeanu. Distributed backpropagation neural networks on a PVM heterogeneous system. In *Proc. 10th IASTED Intl. Conf. on Parallel and Distributed Systems (PDCS'98)*, pages 555–560, October 1998.

[3] V. Bharadwaj, D. Ghose, V. Mani, and T. G. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems.* IEEE Computer Society Press, Los Alamitos, CA, 1996.

[4] B. Boffey. *Distributed Computing-Associated Combinatorics Problems.* Blackwell Scientific Publications, Oxford, 1992.

[5] L. C. Chu and B. W. Wah. Optimal mapping of neural-network learning on message-pasing multicomputers. *J. of Parallel and Distributed Computing*, 14:319–339, 1992.

[6] M. Crespo, F. Piccoli, M. Printista, and R. Gallard. Parallel shaping of backpropagation neural networks in a workstations-based distributed system. In *Proc. EIS'98 Int. ICSC Symp. on Engineering of Intelligent Systems*, pages 709–715. ICSC Academic Press, February 1998.

[7] S. K. Foo, P. Saratchandran, and N. Sundararajan. Parallel implementation of backpropagation neural networks on a heterogeneous array of transputers. *IEEE Trans. on Syst., Man and Cybern. Part B: Cybernetics*, 27(2):118–126, February 1997.

[8] S. Jaganathan and F. Lewis. Multi layer discrete-time neural-net controller with guaranteed performance. *IEEE Trans. on Neural Networks*, 7(1):107–129, 1996.

[9] V. Kumar, S. Shashi, and M. B. Amin. A scalable parallel formulation of the backpropagation algorithm for hypercubes and related architectures. *IEEE Trans. Parallel and Distributed Syst.*, 5:1073–1090, 1994.

[10] D. M. Nicol. Rectilinear partitioning of irregular data parallel computations. *J. of Parallel and Distributed Computing*, 23:119–134, 1994.

[11] H. Paugam-Moisy. Parallel neural computing based on network duplicating. In I. Pitas, editor, *Parallel Algorithms for Digital Image Processing, Computer Vision, and Neural Networks*, pages 305–340. John Wiley & Sons, 1993.

[12] J. Sohn, T. G. Robertazzi, and S. Luryi. Optimizing computing costs using divisible load analysis. *IEEE Trans. Parallel and Distributed Syst.*, 9(3):225–234, March 1998.

[13] J. M. Zurada. *Artificial Neural Systems.* PWS Publishing Company, Boston, 1992.