HYPERCUBE AND DISTRIBUTED COMPUTERS
J.P. Verjus, F. André (North-Holland, 1989)

PARALLEL IMPLEMENTATIONS OF NEURAL NETWORK SIMULATIONS

A. PETROWSKI °*, L. PERSONNAZ *, G. DREYFUS *, C. GIRAULT °

° Laboratoire Méthodologie et Architecture des Systèmes Informatiques
Université Pierre et Marie Curie (Paris 6)
4, place Jussieu
75252 PARIS CEDEX 05, FRANCE

* Ecole Supérieure de Physique et de Chimie Industrielles de la Ville de Paris
Laboratoire d'Electronique
10, rue Vauquelin
75005 PARIS, FRANCE

At the present time, backpropagation is the most popular learning algorithm for multilayer feedforward neural networks. It can be expressed in the formalism of vector or matrix algebra : extensions of scalar operations and matrix products. These operations can be performed with a high degree of parallelism. Three architectures involving loosely coupled processors are considered: torus, mesh and ring of processors. The performances of these architectures are assessed and their privileged applications are discussed. The experiments which were performed on a hypercube of Transputers are in agreement with the theoretical predictions. It is shown analytically and experimentally that the implementation of neural networks on a multiprocessor architecture, if performed properly, can be efficient.

1. INTRODUCTION

Neural networks, unlike standard computers, do not run programs. Their behaviour is determined by parameters referred to as "synaptic weights". These parameters are computed by a learning procedure, during which a set of examples (input-output pairs) are presented to the network; the weights are computed so that the output responses to each input belonging to the training set are as close as possible to the desired outputs. When the training phase is completed, the network can be input some unknown data and is supposed to be able to generalize from the data of the training set and give a correct response.
For instance, in a pattern recognition task, the training set is made of examples of handwritten numerals (inputs) and their binary code (output). After training, the network should be able to output the correct code for each numeral of the training set. Moreover, when presented with a numeral which is not part of the training set, the network should be able to output the correct code.
The use of neural networks to perform such tasks as artificial vision, speech recognition, signal processing, and the like, is promising, but training algorithms such as backpropagation tend to be very time-consuming. It is therefore natural to try to capitalize on the intrinsic parallelism of these systems in order to speed up the computations.
The regular architecture of neural networks suggests that simulations can be implemented on parallel multiprocessor machines with optimal, simple placement rules. However, the processors which are currently available are much more powerful, and have a much smaller number of input-output ports, than an individual neuron.Thus, the placement problem is not a trivial one and deserves much attention, since the communication problem is obviously of crucial importance.
We first present the error backpropagation learning rule; in the subsequent sections, several parallel architectures based on networks of "Transputers" are described and analyzed: the torus, the mesh and the ring. We show that each architecture is suited to specific situations. The performances of these architectures are evaluated analytically, and experiments are presented to validate the theoretical predictions.

2. NETWORKS OF FORMAL NEURONS

The first model of formal neuron was proposed in 1943 by Mc Culloch and Pitts. Its evolution is described by the following expressions:

$$\sigma(t+\tau) = f[v(t)] \quad \text{with} \quad v(t) = \sum_j W_j x_j(t)$$

where $\sigma(t)$ is the state of the neuron at time t, $x_j(t)$ is the value present at the input j of the neuron, W_j is the "synaptic" weight of input j, and f is the transfer function of the neuron. The transfer function of the original McCulloch-Pitts neuron was a step function. It will be shown below that the backpropagation algorithm requires the transfer function to be differentiable, so that a sigmoidal transfer function must be used.

Each input of a given neuron of a network can be connected to the output of any neuron j. Let x_j be the value present at the considered input : $x_j = \sigma_j$. The state of a neural network of N neurons at time t can be defined by a vector $\underline{\sigma}(t)$ the components of which are the states of each neuron; similarly, a potential vector $\underline{v}(t)$ and a (N x N) synaptic matrix W can be defined; they are related by:

$$\underline{v}(t) = W * \underline{\sigma}(t) \qquad \underline{\sigma}(t+\tau) = f(\underline{v}(t))$$

where the symbol * denotes the matrix product. The function f is the vector extension of the transfer function.

Multilayer networks (figure 1) have been widely used in recent years [PERSONNAZ 1989]. They have input units which just transmit the input information to the network, and output neurons. All other neurons are referred to as "hidden neurons". In the present paper, we shall deal with feedforward networks only, i.e. nets in which information flows from the input to the output without any feedback: the response of the network to an input information is instantaneous inasmuch as the transmission time through the layers of the network is negligible.

We number the layers from 0 (input layer) to L (ouput layer).The synaptic matrix can be written as a block-triangular matrix (figure 2a). In the current case where the neurons of a layer are connected only to those of the previous layer, the synaptic matrix is sub-diagonal by blocks (figure 2b).

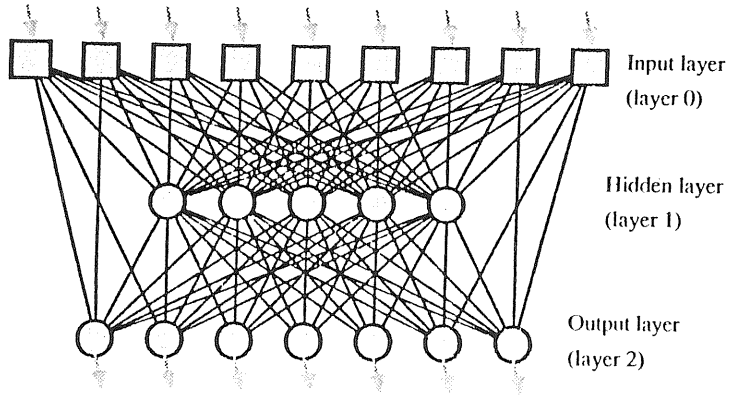

Figure 1 : multilayer network

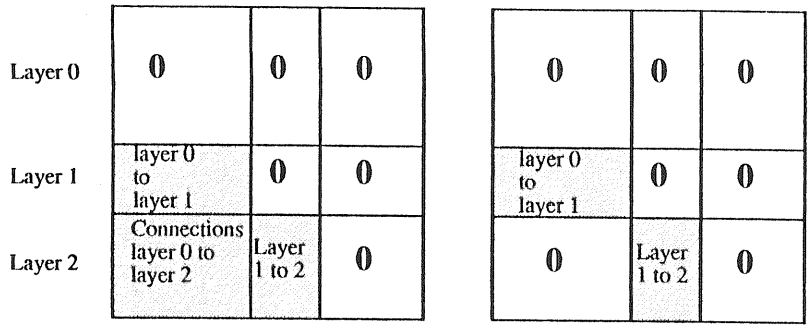

Figure 2 : Synaptic matrix of multilayer networks

2.2 Learning by backpropagation

During the backpropagation phase [PARKER 1985][RUMELHART 1986], a training set, made of an input data set $\{\underline{s}(k)\}$ (for instance, a representation of handwritten digits) and a set of output data $\{\underline{d}(k)\}$ (for example, $\underline{d}(k)$ would be the binary code of digit $\underline{s}(k)$). Learning is the computation of the synaptic weights so that, if a picture $\underline{s}(k)$ is presented on the input units, the corresponding code $\underline{d}(k)$ appears on the output neurons. Thus, the network learns to "associate" the picture of the numeral to the corresponding code. Once training has been completed, the network is expected to produce the correct output code if an unknown digit is presented at the input.

We denote by σ_i^l be the state of neuron i on layer l (i=1, 2, ..., N(l)-1, where N(l) is the number of neurons on layer l). Similarly, we denote by W_{ij}^{lm} the weight of the synapse transmitting information from neuron j of layer m to neuron i on layer l, where $l > m$. W^{lm} is the block of matrix W which corresponds to synapses transfering information from layer m to layer l.
For all vectors $\underline{s}(k)$ presented at the input of the network, learning consists of computing vector $\underline{\sigma}^L(k)$ on the output layer and to modify the synaptic weight in order to minimize a measure of the dissimilarity between $\underline{\sigma}^L(k)$ and the desired state vector $\underline{d}(k)$. Two minimization methods can be used: the true gradient rule and the stochastic gradient rule.

2.2.1 True gradient method

This method defines a cost function which takes all examples into account:

$$J = \frac{1}{2} \sum_{k=0}^{p-1} \sum_{i=0}^{N(L)-1} [d_i(k) - \sigma_i^L(k)]^2 \qquad [1]$$

where p denotes the number of examples in the training set. The cost function is minimized by a gradient method. Thus, for each presentation of the whole training set, the synaptic weights are modified by an amount ΔW_{ij} given by:

$$\Delta W_{ij}^{lm} = - \eta \frac{\partial J}{\partial W_{ij}^{lm}}$$

where η is a positive real number.

An error $\delta_i^l(k)$ can be defined for each neuron and example of the network such that:

$$\Delta W_{ij}^{lm} = \eta \sum_{k=0}^{p-1} \delta_i^l(k)\, \sigma_j^m(k)$$

Two cases must be considered to compute $\delta_i^l(k)$:
- if l denotes the output layer (l=L):

$$\delta_i^l(k) = 2\, f'\left[v_i^l(k)\right]\left[d_i(k) - \sigma_i^L(k)\right]$$

- if l denotes a hidden layer:

$$\delta_i^l(k) = f'\left[v_i^l(k)\right] \sum_{k<q<L} \sum_{n=0}^{N(q)-1} \delta_i^q(k)\, W_{ni}^{ql}$$

where f' is the derivative of the transfer function.

The only constraint on the transfer function is its differentiability. In general, a sigmoidal transfer function is used in order to approximate the threshold function of the Mc Culloch and Pitts neurons.
At each learning cycle, the whole training set is presented once. After the cycle has been performed, the cost function is computed and the weights updated accordingly. The learning cycles are iterated until a suitably small value of the cost function is obtained.

2.2.2 Stochastic gradient method

B. WIDROW [WIDROW 1985] used a gradient algorithm for the design of adaptative filters built with single-layer networks of linear neurons. This algorithm can be considered a simplification of the algorithm described in section 2.2.1, in which the value of J(k), for each example k, is taken as the average value of the cost function:

$$J(k) = 1/2\ \Sigma_i\, [d_i(k) - v_i(k)]^2 \qquad [2]$$

The mathematical relations are similar to those obtained for the true gradient algorithm, but, in this case, the weights are updated after the presentation of each example. Thus, during a learning cycle, the weights are altered p times, whereas the true gradient method involves one updating only per cycle. This algorithm is referred to as the "stochastic gradient algorithm".

2.3 True gradient and stochastic gradient algorithms

The algorithm uses the vector formalism to make mathematical notations more compact and to express the parallelism clearly. Conditional notations allow to separate the stochastic gradient and the true gradient algorithm. In order to make notations simpler, we consider the particular case where the neurons on a layer are connected to the previous layer only; thus, the synaptic matrix is sub diagonal by blocks (§ 2.1). This can easily be generalized.

Notations:

J	: cost of the solution developed by the network
J_{max}	: maximum acceptable cost
η	: gradient step
p	: number of examples
N(l)	: number of neurons on layer l
W^l	: block of the synaptic weights which tranfers information from layer l - 1 to layer l (N(l) x N(l-1) matrix)
$\sigma_l^l(k)$	: state column vector of layer l for example k
σ^l	: matrix whose p columns are vectors $\sigma^l(k)$
$V^l(k)$	: potential column vector of layer l for example k
f()	: vectorial extension of the transfer function
f'()	: vectorial extension of the derivative of the transfer function

$d(k)$: vector of desired outputs for example k
$\delta^l(k)$: error vector of layer l for example k
δ^l : matrix whose p columns are vectors $\delta^l(k)$
${}^tA$: transpose of matrix A

While $J > J_{max}$ do

For k = 0 à p-1 do
For l = 1 à L do -- **state evaluation**
$V^l(k) := W^l * \sigma^{l-1}(k)$
$\sigma^l(k) := f(V^l(k))$
End for
J = cost() -- **cost evaluation**
If true gradient -- **backpropagation**
$\delta^L(k) := f'(V^L(k)).(d(k) - \sigma^L(k))/p$
else
$\delta^L(k) := f'(V^L(k)).(d(k) - \sigma^L(k))$
End if
For l = L-1 à 1 do
$\delta^l(k) := f'(V^l(k)).({}^tW^{l+1} * \delta^{l+1}(k))$
End for
For l = 1 à L do -- **update of ΔW**
If true gradient
$\Delta W^l := \Delta W^l + \eta\ \delta^l(k) * {}^t\sigma^{l-1}(k)$
else
$W^l := W^l + \eta\ \delta^l(k) * {}^t\sigma^{l-1}(k)$
End if
End for
End for
If true gradient -- **if true gradient : update of W**
For l = 1 à L do
$W^l := W^l + \Delta W^l$
End for
End if

End while

The above algorithm involves two matrix/vector products and a vector product. The true gradient algorithm can be fully written in the matrix form. This increases the parallelism of the problem. The algorithm can thus be simplified to :

While $J > J_{max}$ do

For l = 1 to L do -- **state evaluation**
$V^l := W^l * \sigma^{l-1}$
$\sigma^l := f(V^l)$
End for
J := cost() -- **cost evaluation**
$\delta^L := f'(V^L).(d - \sigma^L)/p$ -- **backpropagation**
For l = L-1 to 1 do
$\delta^l := f'(V^l).({}^tW^{l+1} * \delta^{l+1})$
End for
For l = 1 to L do
$\Delta W^l = \eta\ \delta^l * {}^t\sigma^{l-1}$
End for
For l = 1 to L do -- **update of W**
$W^l := W^l + \Delta W^l$
End for

End while

3. PARALLELIZATION OF THE LEARNING ALGORITHM

3.1 Choice of the gradient algorithm

The parallelism of the above algorithms is clear from the vector and matrix relations, which involve either vector extensions of scalar operations, or matrix products. It is interesting to compare them in terms of required memory space and degree of parallelism.
In the case of the true gradient, the number of synaptic "sub-matrices" (N(l) x N(l-1)) is equal to the number of layers. Moreover, the computation uses L state matrices (N(l) x p), the matrix of desired states (N(L) x p) and the matrix of example to be presented to the network (N(0) x p). Backpropagation generates L matrices δ^l (N(l) x p) and L matrices ΔW^l. Thus, the required memory space to store all the matrices of floating point numbers is:

$$\mu_T = 2 \sum_{l=1}^{L} N(l)\, N(l\text{-}1) + 2\, p \sum_{l=1}^{L} N(l) + p\,[N(0) + N(L)]$$

In the case of the stochastic gradient algorithm, matrices ΔW^l are not to be stored since the weights are updated immediately after presentation of each example. Therefore, the column vectors $\sigma^l(k)$ et $\delta^l(k)$ only are requested. The required memory space thus reduces to:

$$\mu_S = \sum_{l=1}^{L} N(l)\, N(l\text{-}1) + 2 \sum_{l=1}^{L} N(l) + p\,[N(0) + N(L)]$$

In general, the number of examples is much larger than the number of layers. Therefore, μ_T/μ_S ranges from 2 to L+1. For a classification problem, where the number of examples is greater than the number of neurons per layer, μ_T/μ_S is close to L+1. Thus, for the usual case where L = 2, the true gradient algorithm requires a maximum of three times as much memory space as the stochastic gradient algorithm.
From the viewpoint of parallelism, the stochastic gradient algorithm involves matrix/vector products whereas the true gradient uses matrix/matrix products. A product of (N x N) matrices by vectors of N components can be performed by N^2 simultaneous additions and multiplications with a suitable architecture. A product of two (N x N) matrices can be performed by N^3 simultaneous operations. Therefore the true gradient algorithm allows a larger speedup than the stochastic gradient. These considerations will be developed in the following.

In spite of the large memory requirements, it seems more interesting to use the true gradient method in order to obtain better performances. However, some problems, or some types of networks are not suitable for this rule for practical reasons. The stochastic gradient method is more adequate for problems which involve large training sets. However, the difficulty has been claimed to be overcome by dividing the set of examples into blocks, the elements of which are presented simultaneously to the network [PLAUT 1986].
More fundamentally, when linear neurons are used, a value of the gradient step close to the optimum can be estimated: $\eta = 1/N$ if the cost function is J(k) (relation [2]) in the the case of the stochastic gradient method, or J/p (expression [1]) in the case of the true gradient method. Therefore, the true gradient method requires roughly p times more computation than stochastic gradient. It was one of the main motivations for the development of the stochastic gradient method [WIDROW 1985]. But if the units of the network are sigmoidal, no optimal value the gradient step has been found, and experiments showed that the convergence speed of both methods can be comparable.

In the following, we consider mainly learning by the true gradient method, unless otherwise noted, because of the larger speedup which can be achieved. The parallelization of the stochastic gradient method will be presented in a follow-up paper.

3.2 Architectures of the processor network, task placement

A Transputer network (INMOS) was used in our experiments. Communication times are the main problem with loosely coupled architectures. The matrix operations required to perform learning are extension of scalar operations and matrix products. The latter involve communications between processors and impose constraints on the architecture. The algorithms used here are inspired by [FOX 1987]. Since a Transputer has four communication links only, we shall investigate two-dimension architectures (mesh and torus), and a one-dimension architecture (ring), shown on Figure 3.

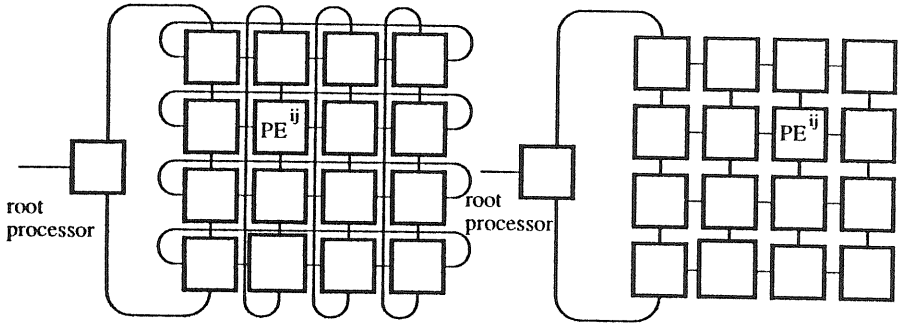

Figure 3a : torus of 16 processors

Figure 3b :
mesh of 16 processors

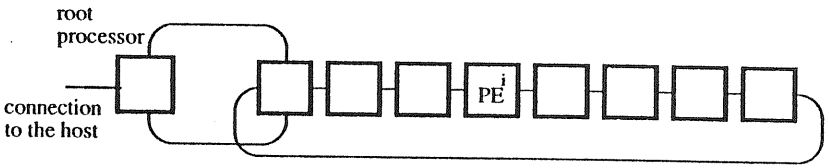

Figure 3c : ring of 8 processors

The implementation of a neural network on a processor network raises various issues. First, the regular sequence of the layers of a multilayer network make up a pipe-line that we one might take advantage of. However, for the stochastic gradient, backpropagation and the weight updates require a pipeline initialization for each presented example. This problem is also present with the true gradient method, to a lesser extent, because the initialization occurs once per learning cycle. Thus making use of the pipeline would be inefficient. Executing the computations related to various layers in sequence by the same processors is a more satisfactory solution [ROBERT 1988] [ERNOULT 1988]. All the blocks W^l of the synaptic matrix for the L layers are evenly distributed among all processors to allow a balanced load for the matrix products, according to FOX's algorithm (figure 4).

3.3 Algorithms for matrix products on a two-dimensional torus

Error back-propagation learning makes use of three types of matrix products:

$$V = W * \sigma, \quad C = {}^tW * \delta, \quad D = \delta * {}^t\sigma \ .$$

The matrices are not explicitely transposed to minimize the communication times. Therefore, these products are implemented by specific algorithms which guarantee an optimal sequence of computations.
The number of rows and columns of the torus is denoted by Q:
$Q = \sqrt{P}$.
A matrix W is divided into Q x Q blocks W^{ij}, each of them being stored in the memory of processor PE^{ij}. This placement on a "square" torus (figure 3a) allows the best communication performances.[FOX 1987].

Computation of the product $V := W * \sigma$:

The matrix product requires Q broadcasts of blocks $W^{i,k+i}$ along the rows and Q shifts of matrix σ along to the columns. Figure 4 illustrates the operation of the first step of the algorithm.

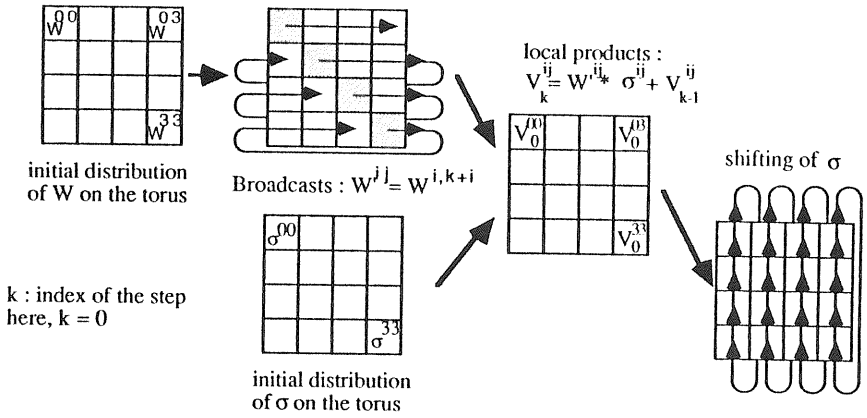

Figure 4 : Step 0 of the matrix product algorithm $V = W * \sigma$ on a torus

Duration of the computation:

We denote by t_f the time required for an elementary floating point operation (addition or multiplication), t_c the time required for the communication of a 32 bit number between two neighbouring processors and t_s the time required to initialize a pipelined communication of one data packet between two neighbours.
In the following, we assume that matrices W and σ are the blocks of the synaptic and state matrices respectively related to a given layer of a neural network, and that the number of neurons per layer is constant: N = N(l) for $0 < l \leq L$ (the latter is an unrealistic assumption).

$$t_{1t} = 2\left(\frac{\overline{N}}{Q}\right)^2 \frac{\overline{P}}{Q} Q\, t_f + \left[\left(\frac{\overline{N}}{Q} + \frac{\overline{P}}{Q}\right)\frac{\overline{N}}{Q} + \frac{Q}{2}\right] Q\, t_c + Q^2 \frac{t_s}{2}$$

where an horizontal bar above a fraction denotes the smallest integer larger than the fraction.

The computation of the product $C = {}^tW * \delta$ requires Q broadcasts of blocks of matrix W along the rows and Q shifts of matrix C along the columns. The product $D = \delta * {}^t\sigma$ requires Q broadcasts of blocks of matrix σ along the columns and Q shifts of matrix D along to the rows. The complexity of both algorithms is equal to t_{1t}:

$$t_{2t} = t_{3t} = t_{1t} \ .$$

3.4 Matrix products on a mesh of processors

Theoretically, a mesh is always less efficient than a torus. However, a Transputer has only four links which are all used on a torus. In this case, an extra Transputer, referred to as the "root Transputer" must be inserted on a link in order to provide a connection to the host computer (figure 3a). This root Transputer is active while loading the network only. It must be transparent during the learning cycle and it increases the communication times. Therefore, the mesh, which allows loading from one of the border Transputers (figure 3b), is more efficient than the torus for machines with a small number of processors.
Unlike the torus, which allows easy matrix shifts, communications within a mesh are broadcasts or concentrations of data only, which are more time-consuming. For the sake of brevity, the algorithms will not be described.

Duration of product V:= W * σ on a mesh:

$$t_{1m} = 2\left(\frac{\overline{N}}{Q}\right)^2 \frac{\overline{p}}{Q} Q\, t_f + \left[\left(\frac{\overline{N}}{Q} + \frac{\overline{p}}{Q}\right)\frac{\overline{N}}{Q} + 3\,\frac{Q}{2}\right] Q\, t_c + 3\,Q^2\,\frac{t_s}{2}$$

Duration of the computation of product C:= ${}^tW$ * δ on a mesh:

$$t_{2m} = \left(2\left(\frac{\overline{N}}{Q}\right)^2 \frac{\overline{p}}{Q} + \frac{3Q}{4}\right) Q\, t_f + \left[\left(\frac{\overline{N}}{Q} + \frac{\overline{p}}{Q}\right)\frac{\overline{N}}{Q} + 3\,\frac{Q}{2}\right] Q\, t_c + 3\,Q^2\,\frac{t_s}{2}$$

Duration of the computation of product D:= δ * ${}^t\sigma$ on a mesh:

$$t_{3m} = t_{2m}$$

3.5 Matrix products on a ring

Within the torus or the mesh, the communication time is proportional to the smallest integer larger than p/Q. Therefore, when p = 1, (p is the number of columns of the example matrix), the efficiency will be very low. Such is the case with the stochastic gradient algorithm, in which the examples are presented sequentially. Therefore, a parallel implementation of the stochastic gradient will be more efficient on a ring of processors; the adaptation of the matrix product algorithms of the torus architecture is trivial (figure 3c).
Within a ring, the above three types of matrix products require only matrix shifts. The computation times are identical:

$$t_{1r} = t_{2r} = t_{3r} = 2\left(\frac{\overline{N}}{P}\right)^2 p\,P\,t_f + P\,t_s + \frac{\overline{N}}{P}\,p\,P\,t_c$$

where P is the number of processors present in the ring.

3.6 Duration of a learning cycle

A learning cycle consists of a single presentation of the whole training set to the network. The complexity of the extensions of the scalar operations, and the computation of the state matrix of the network from the potential matrix, are $O(N^2)$, where N is the number of neurons per layer; the complexity of matrix products is $O(N^3)$. Therefore, the latter time is dominant if the processor network is large enough.
At the end of a learning cycle, the termination condition is checked. This requires the concentration of data towards a "master processor" which makes the decision and broadcasts it to the network. Alternately, the termination check might be distributed. However, the contribution of communication time for these algorithms is similar ($P^{1/2}.t_c$ or $P.t_c$, depending on the architecture), so that this contribution is negligibly small.
For a neural network of L layers, state evaluation requires L products W * σ. Backpropagation and the weight updatings require L-1 products ${}^tW$ * δ and L products δ * ${}^t\sigma$. The duration of a learning cycle is therefore given by:

$$t_t = (3L-1)\left\{2\left(\frac{\overline{N}}{Q}\right)^2 \frac{\overline{p}}{Q} Q\, t_f + \left[\left(\frac{\overline{N}}{Q} + \frac{\overline{p}}{Q}\right)\frac{\overline{N}}{Q} Q + \frac{Q^2}{2}\right] t_c + Q^2\,\frac{t_s}{2}\right\}$$

$$t_m = \left[(6L-2)\left(\frac{\overline{N}}{Q}\right)^2 \frac{\overline{p}}{Q} + (2L-1)\,\frac{3Q}{4}\right] Q\, t_f + (3L-1)\left[\left(\frac{\overline{N}}{Q} + \frac{\overline{p}}{Q}\right)\frac{\overline{N}}{Q} + \frac{3Q}{2}\right] Q\, t_c + \frac{3}{2}(3L\text{-}1)\,Q^2\, t_s$$

$$t_r = (3L-1)\left(2\left(\frac{\overline{N}}{P}\right)^2 p\,P\,t_f + \frac{\overline{N}}{P}\,p\,P\,t_c + P\,\frac{t_s}{2}\right)$$

where P is the number of processors and $Q = P^{1/2}$.

3.7 Speedup analysis:

The speedup is the ratio $S = t_{seq}/t_{par}$ where t_{seq} and t_{par} are the sequential and parallel computation times respectively.

It can easily be shown that:

$$t_{seq} = (6L\text{-}2)N^2.p.t_f$$

In order to clarify the discussion, the speedup will be estimated in two cases:
- to emphasize the importance of communication time, we assume that N and p can both be divided by Q, in the case of the torus as well as in the case of the ring. Then:

$$S_t = \frac{Q^2}{1 + \frac{Q^4}{4N^2p}\frac{t_s}{t_f} + \left(\frac{Q}{2N} + \frac{Q}{2p} + \frac{Q^4}{4N^2p}\right)\frac{t_c}{t_f}}$$

$$S_m = \frac{Q^2}{\left(1 + \frac{2L-1}{3L-1}\frac{3Q^4}{8N^2p}\right) + \frac{3Q^4}{4N^2p}\frac{t_s}{t_f} + \left(\frac{Q}{2N} + \frac{Q}{2p} + \frac{3Q^4}{4N^2p}\right)\frac{t_c}{t_f}}$$

$$S_r = \frac{p}{1 + \frac{P^2}{4N^2p}\frac{t_s}{t_f} + \left(\frac{P}{2N} + \frac{P^2}{4N^2p}\right)\frac{t_c}{t_f}}$$

As anticipated, the communication costs are smaller on a two-dimensional architecture than on a ring. These architectures will be used efficiently if:

$$P^2 < N^2.p$$

- to assess the efficiency loss due to the fact that the processors are not saturated, we assume that the communication times are negligible and we compute the average speedup |S|. In order to do this, we assume that

$\overline{X} \approx (X + 1/2)$

and we assume that the number of neurons per layer and the number of examples are independent. From the point of view of processor saturation, the torus and the mesh are equivalent:

$$|S_t| = \frac{N^2}{(N+Q/2)^2}\frac{p}{(p+Q/2)}Q^2 = \frac{N^2}{(N+\sqrt{P}/2)^2}\frac{p}{(p+\sqrt{P}/2)}P$$

$$|S_r| = \frac{N^2}{(N+P/2)^2}P$$

One has $|S_l| > |S_r|$ if :

$$\frac{(N+P/2)^2}{(N+\sqrt{P}/2)^2}\,\frac{p}{(p+\sqrt{P}/2)} > 1$$

This inequality defines a partition of the N vs. p plane into two areas, the border of which can be approximated by the line

$$p = \frac{1}{2}\,\frac{1}{\sqrt{P}-1}\,N$$

if N and p are much larger than P.
The area where the ring is more efficient than the torus lies between the border and the N axis (figure 5).
If the number of examples p is larger than N, the mesh or the torus are always more efficient than the ring. This is precisely the case in most practical situations. Otherwise, the efficiency of the torus increases with the number of processors. However, as previously mentioned, the implementation of the stochastic gradient algorithm requires the use of the ring.

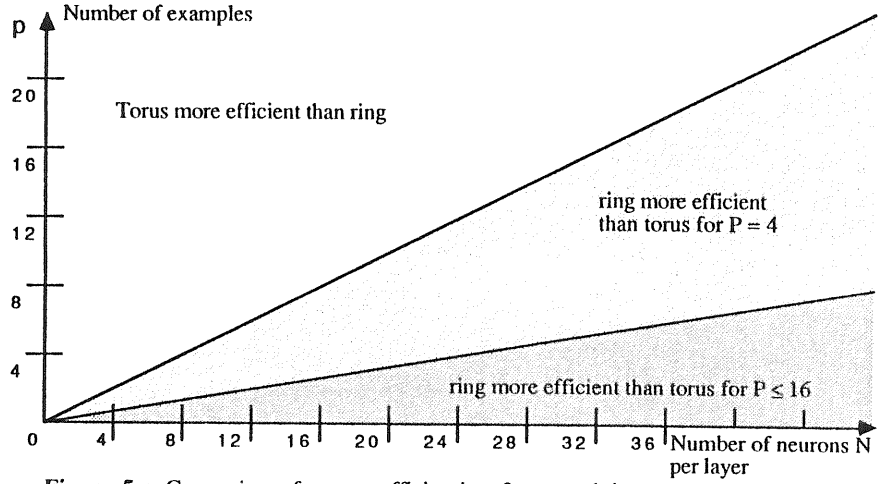

Figure 5 : Comparison of average efficiencies of torus and ring

4. EXPERIMENTAL RESULTS

4.1 The Transputer networks

A Transputer is a 32 bit microprocessor with four 20 Mbit/s serial bidirectional communication links. The Transputer can access 4 Gigabytes of memory through a 32 bit private bus. Part of this space is on-chip memory (T800: 4 Kbytes, access time: 50ns). The computation speed reaches 10 MIPS for a clock rate of 20 MHz. In addition, the T800 provides an internal floating point unit running at 1.5 MFlops for 64-bit real numbers.
The Transputer was designed for efficient implementation of the high level OCCAM language [INMOS] which stemmed from CSP [HOARE 78]. OCCAM allows the implementation of sequential processes which can communicate by sending and receiving messages on logical channels, some of which corespond to physical links. A Transputer can run several processes by timeslicing. The code generated by OCCAM is compact and efficient, making direct algorithm implementation in machine language unnecessary.

4.2 Experimental conditions

The experiments were performed on a hypercube of sixteen T800 Transputers, from which a torus of 4 x 4 processors, a mesh, and a ring of 16 processors, were extracted (figure 3). The computation time was measured in sequential and parallel modes for 100 learning cycles and for neural networks of various sizes. The latter had one hidden layer only. For simplicity, the number of neurons per layer was the same for each layer. In actual classifications tasks, the number of examples is usually much larger than the number of neurons; since the relations presented in section 3.7 show that the speedup is an increasing function of the number of examples, the cases investigated here are not very favourable.
The examples are binary vectors with random components. The synaptic weights are also initialized randomly. The measurements were performed for 16, 20, 24, 32, 48 et 64 neurons per layer. The results are shown below:

dim. N = p =	16	20	24	32	48	64
seq. time.:	27.04 s	51.00 s	86.03 s	202.1 s	675.0 s	1574 s
torus time:	3.094 s	5.237 s	8.183 s	16.75 s	52.64 s	115.7 s
mesh time:	2.973 s	5.048 s	7.900 s	16.39	50.88 s	111.7 s
ring time:	5.929 s	18.24	21.81	28.96	78.38	163.4
torus speedup:	8.74	09.74	10.5	12.1	12.8	13.6
mesh speedup:	9.10	10.1	10.9	12.3	13.3	14.1
ring speedup:	4.56	02.97 *	03.94 *	6.98	08.61	09.63

* The processors of the ring are not saturated for N=p=20 and N=p=24.

It was shown in section 3.7 shows that the speedup could be put under the form $S = P/(a + b/N)$ when N is large enough, where P is the number of processors and N is either the number of neurons per layer or the number of examples. The values of a and b for the three architectures are estimated by a least mean squares method. The estimated values of a and b are shown below. The curves of figure 6, which express the efficiency vs. the number of neurons by layers and examples (N = p), shows the validity of the expression of S.

torus:	a = 0.94	b = 14.1
mesh:	a = 0.92	b = 13.3
ring:	a = 1.05	b = 39.5

The value of t_f can be obtained from the computation time in the sequential mode: $t_f \approx 6\ \mu s$. The values of b for the three architectures give an estimate of t_c given t_f and P:

mesh: $t_c \approx 20\ \mu s$ torus: $t_c \approx 21\ \mu s$ ring: $t_c \approx 30\ \mu s$

From the technical notes of the manufacturer, a value of 10 µs could be expected for t_c. The observed differences seem to stem from some elements which were not taken into account, such as the delay time through the crossbar switches of the hypercube, and the efficiency of DMA when sending or receiving information on a link.
In order to approach more realistic situations, a toy version of a problem of handwritten numeral recognition was implemented. The training set included 80 examples. The input layer had 64 units, each unit coding for a pixel. The output layer was expected to provide the binary code of the digit by activating one neuron out of ten. The hidden layer included 45 neurons. The results are presented below:

gradient step: 0.06; number of learning cycles: 368;
cost at the end of the learning: $J/p = 6.55 \cdot 10^{-3}$.

sequential implementation:	3007,0 seconds		
torus implementation:	255,7 seconds;	speedup:	11.8
mesh implementation:	244,7 seconds;	speedup:	12.3
ring implementation:	467,8 seconds;	speedup:	6.43

The numbers of neurons on the hidden layer and the ouput layer are not multiples of the square root of the number of processors (which is equal to 4). Therefore, the computation load is not balanced and the efficiency is not optimal.

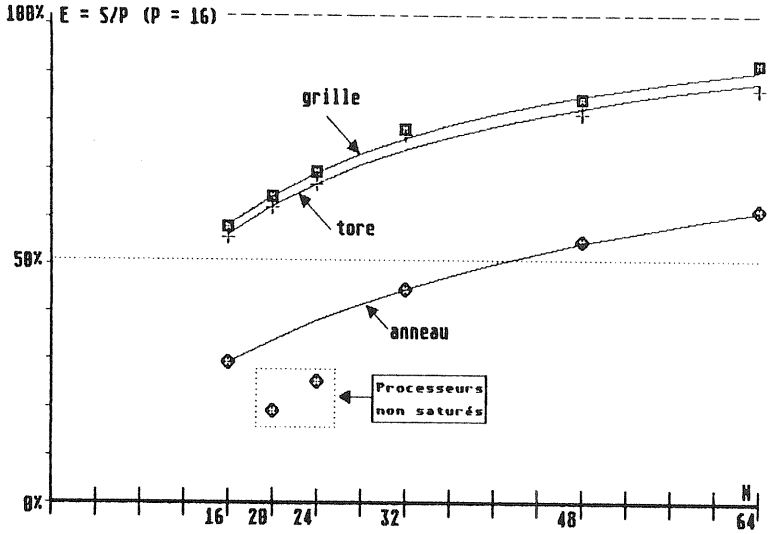

Figure 6 : Efficiency of the simultation of neural networks implemented on different architectures of 16 processors according to the number of neurons per layers

CONCLUSION

We showed that the simulation of neural networks, during the learning or the classification phases, can be implemented efficiently on a loosely coupled multiprocessor machine.
Experiments were performed with networks of small size. Our theoretical results show that the efficiency increases when either the size of the network or the size of the problem increases.
Three parallel architectures were considered: the torus, the mesh and the ring. In principle, the first architecture should be the most efficient. For practical reasons, however, the mesh turns out to be better for small processor networks. These architectures are used for the "true gradient" algorithm only. If the "stochastic gradient" method is used, the ring is the single suitable architecture in spite of its lower efficiency. The optimal implementation of the stochastic gradient on a two-dimensional architecture is still under investigation.

ACKNOWLEDGEMENTS

Part of the experiments reported in this paper were performed at the Institut National des Télécommunications. The authors are grateful to Prof. BECKER et Dr. VINCENT for their support.

LITERATURE REFERENCES

C. ERNOULT
Performance of backpropagation on a parallel Transputer-based machine
Actes du colloque Neuro-Nimes'88, p 311 (1988).

G. FOX, S. OTTO, A. HEY
Matrix algorithm on a hypercube: matrix multiplication
Parallel Computing. 4 17 (1987).

C. A. R. HOARE
Communicating Sequential Processes
Communication of the ACM 21 666 (1978).

R.W. HOCKNEY, C.R. JESSHOPE
Parallel Computers
ADAM HILGER LTD, BRISTOL (1981).

INMOS; OCCAM 2; reference Manual
Prentice Hall

D. B. PARKER
Learning-Logic", TR-47
Center for Computational Research in Economics and Management Science, MIT, (1985).

L. PERSONNAZ, G. DREYFUS, eds.
Neural Networks from Models to Applications
(I.D.S.E.T., Paris, 1989).

D. C. PLAUT, S. J. NOWLAN, G. E. HINTON
Experiments on Learning by Backpropagation
Technical report CMU-CS-86-126 (1986).

D. E. RUMELHART, G. E. HINTON, R. J. WILLIAMS
Learning Internal Representation by Error Propagation
Parallel Distributed Processing; MIT PRESS , p 318 (1986).

F. ROBERT, S. WANG
Implementation of a neural network on a hypercube F.P.S. T20
Parallel processing; M. COSNARD, M.H. BARTON and M. VANNESCHI (Editors); p 189
North Holland (1988).

B. WIDROW, S. D. STEARNS
Adaptative Signal Processing
Prentice Hall (1985).