JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 14, 260–285 (1992)

Using and Designing Massively Parallel Computers for Artificial Neural Networks

TOMAS NORDSTRÖM AND BERTIL SVENSSON*

Division of Computer Science and Engineering, Department of Systems Engineering, Luleå University of Technology, S-95187 Luleå, Sweden

During the past 10 years the fields of artificial neural networks (ANNs) and massively parallel computing have been evolving rapidly. In this paper we study the attempts to make ANN algorithms run on massively parallel computers as well as designs of new parallel systems tuned for ANN computing. Following a brief survey of the most commonly used models, the different dimensions of parallelism in ANN computing are identified, and the possibilities for mapping onto the structures of different parallel architectures are analyzed. Different classes of parallel architectures used or designed for ANN are identified. Reported implementations are reviewed and discussed. It is concluded that the regularity of ANN computations suits SIMD architectures perfectly and that broadcast or ring communication can be very efficiently utilized. Bit-serial processing is very interesting for ANN, but hardware support for multiplication should be included. Future artificial neural systems for real-time applications will require flexible processing modules that can be put together to form MIMSIMD systems. © 1992 Academic Press, Inc.

1.0. INTRODUCTION

This paper is intended to provide a survey of the use and design of massively parallel computers for artificial neural networks (ANNs) and to draw conclusions based on reported implementations and studies. The simple control structure that characterizes massively parallel computers can be SIMD (Single Instruction stream, Multiple Data streams) or a highly restricted form of MIMD (Multiple Instruction streams, Multiple Data streams) that we call SCMD (Same Code for Multiple Data streams).

We try to identify the architectural properties that are important for simulation of ANNs. We also emphasize the importance of the mapping between algorithms and architecture. ANN computations are communication intensive, a fact which may put strong demands on the communication facilities of the architecture. Moreover, the requirements vary with the ANN model used and the mapping between the algorithm and the architecture.

The paper is organized into three parts: The first part (Sections 1 through 7) is ANN-oriented. It concentrates on ANN models and those characteristics of the models that are of interest when considering parallel implementation. In this part we first go through the basics of artificial neural networks and ANN algorithms. We then discuss some general computational topics that are relevant for the implementation of any ANN model, such as the precision of the calculations and the opportunities for parallel execution. We conclude the ANN-oriented part with a discussion of different measurements of speed for ANN computations.

The second part (Sections 8 and 9) is architecture-oriented. Here we define different classes of parallel computer architectures and give a review of the types of ANN algorithms that have been implemented on computers of these classes.

In the final part of the paper (Section 10) we analyze what experiences can be drawn from the reported implementations and try to determine what requirements will be placed on massively parallel computers for ANN simulation in the future—in batch processing, in real-time applications, and in action-oriented systems. In real-time applications, the speed of the input data flow and the requirements for output data are set by the environment. In action-oriented systems, sensory, motor, and processing parts, all possibly utilizing neural network principles, are seen as integrated systems capable of interacting with the environment. These systems are sometimes called "sixth-generation computers" [2, 3].

2.0. THE BASICS OF ARTIFICIAL NEURAL NETWORKS

In this section we describe the basic properties and terminology of biological neurons and networks. We also show some simple models of these biological structures.

It should be noted that ANNs are often far from being good biological models. Instead they may be seen as biologically inspired algorithms. Studying "the real thing" will give perspective on how simple our models are and how complex the brain is.

* Also at Centre for Computer Science, Halmstad University, S-30118 Halmstad, Sweden.

0743-7315/92 $3.00

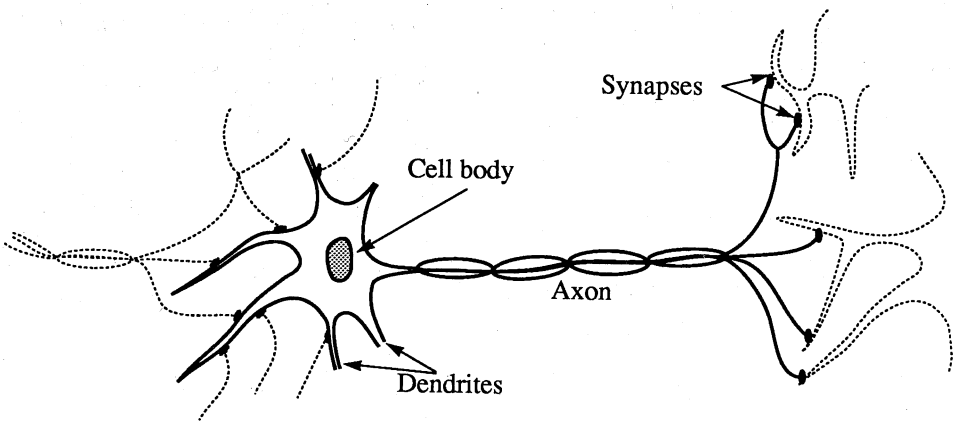

FIG. 1. The principal components of a basic neuron. The input comes to the neuron through synapses on the dendrites. If there are enough stimuli on the inputs there will be an activation (impulse) through the axon which connects to other cells via synapses.

2.1. The Biological Neuron

The basic building block of the brain is the nerve cell (neuron). In humans there are about 10^{12} neurons. Neurons come in many varieties. They are actually all different but can be grouped into at least 50 types of cells.

The principal components of a neuron are shown in Fig. 1. There is a cell body, a number of dendrites (input), and an axon (output). The axon splits and connects to other neurons (or muscles, etc.) The connections function like a sort of chemical resistor and are called synapses. Thus the complexity of the brain is not limited to the vast number of neurons. There is an even larger number of connections between neurons. One estimate is that there are a thousand connections per neuron on average, giving a total of 10^{15} connections in the brain.

Neurons can often be grouped naturally into larger structures (hundreds of thousands of neurons). It has been established that some groups/areas of the brain are organized in a way that reflects the organization of the physical signals stimulating the areas, i.e., topological order. The result is that nearby areas in the brain correspond to nearby areas in signal space. This order is accomplished even when the fibers that are transporting the signals do not exhibit any apparent order. The order seems also to be achieved without any guidance as to what is right or wrong. The resulting maps are therefore often called self-organizing maps. Examples are visual and somatosensory cortex. Each of these structures often connects to other structures at a higher level.

2.1.1. Adaptation and Learning

The brain would not be as interesting, nor as useful, without its ability to adapt and to learn new things. There are basically two ways in which adaptation takes place, by changing the structure and by changing the synapses. The first has the nature of long-term adaptation and often takes place only in the first part of an animal's life. The second, changes of synapses, is a more continuous process that happens throughout the animal's entire lifetime.

2.1.2. Information Processing

The information processing in a neuron is done as a summation or integration of information fed into it. The information is represented as brief events called nerve impulses.† The interval or frequency conveys the information. According to Hubel [50] the impulse rates may vary from one event every few seconds or even slower to about 1000 events per second at the extreme upper limit. The normal upper limit is often cited to be 100 to 200 impulses per second. The "speed" of the impulses along the axon is around 0.1 to 10 m/s. The length of an axon varies from less than a millimeter to more than a meter.

2.2. The Artificial Neuron

The first and very simple model, however much used, is the model in which information is contained as levels/values corresponding to the impulse frequencies. Then the integration of pulses is done as a summation. The synapses are represented as weights, w_j, multiplied by inputs i_j. To make the model more powerful, a nonlinear function, f, is applied to the sum, and the result, $o = f(\Sigma w_j i_j)$, is sent to the neurons connected to it (Fig. 2).

As with their biological counterparts the artificial neurons are not very interesting by themselves. A large number of artificial neurons are necessary for interesting computations. By changing the structure of the connections and adaptation rules it is possible to radically change the type of computations made by the network. Some of the models used are described in Section 3.0.

† This is not true for all neurons. There are, for example, neurons in the retina which have "graded" response. See e.g. [50] or [100] for more on this topic.

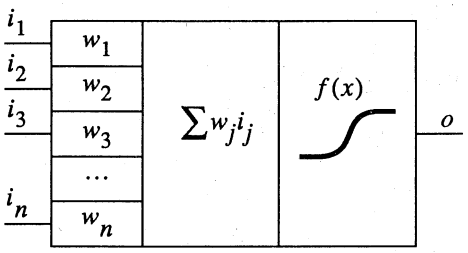

FIG. 2. The simplest model of a neuron. It can be seen as a model of Fig. 1. The output has the form $o = f(\sum w_j i_j)$.

2.3. Layered Models

In many models there are layers of neurons; see Fig. 3. There has been some confusion about how to count the number of layers. One method is to count the node layers including the input layer, and another method is to count weight layers (or node layers excluding the input layer). In this paper we use the word "node" or "weight" in front of the word "layer" when it is needed to avoid confusion. When we count layers we use weight layers, since this is the most relevant method when considering the computational effort. This method of counting implies that one (weight) layer is the smallest network possible. This single-layer network corresponds to the concept of perceptrons [109]. Node layers which have no connection to input or output are called hidden layers; e.g., in Fig. 3 there are two hidden layers.

A compact way of giving the size of a multilayer network is to present the sizes of the node layers with an "×" in between. For example, 203 × 60 × 26 states that the input node layer has 203 nodes, the hidden node layer has 60 nodes, and the output node layer has 26 nodes. Between each layer a fully connected weight layer is assumed. Thus, we consider this a two-layer network.

3.0. SOME OF THE MOST COMMONLY USED ANN ALGORITHMS

During the past 10 years the artificial neural networks area has developed into a rich field of research. Many new models or algorithms have been suggested. Not all these models have been implemented on parallel computers. This is not to say that some of them are not suitable for parallel execution. On the contrary, a common characteristic of all neural network algorithms is that they are parallel in nature. For the purposes of this paper, however, we review the most common ANN algorithms only, in order to be able to discuss their implementation on parallel computers.

The models are characterized by their network topology, node characteristics, and training rules [76]. We describe some frequently used and discussed models.

1. *Multilayer feedforward networks* with supervised learning by *error back-propagation* (BP), also called *generalized delta rule* [110]. The feedforward back-propagation model is used as a pattern classifier or feature detector, meaning that it can recognize and separate different features or patterns presented to its inputs.

2. *Feedback networks* (also referred to as *recurrent networks*). Different variations in node topology and node characteristics have been proposed: symmetric connectivity and stochastic nodes: *Boltzmann machines* [41, 42, 50]; symmetric connectivity and deterministic nodes: *Hopfield nets* [47, 48, 49, 95] and *mean field theory* [95, 96]; and nonsymmetric connectivity and deterministic nodes: *recurrent back-propagation* (RBP) [1, 99]. The feedback models can be used as hetero- or autoassociative memories, but also for solving optimization problems. Using an ANN as an autoassociative memory means that whenever a portion or a distorted version of a pattern is presented, the remainder of the pattern is filled in or the pattern is corrected.

3. *Self-organizing maps* (SOM), also called *self-organizing feature maps* (SOFM) or *topological feature maps* (TFM), developed by Kohonen [62, 63]. This is one of the more frequently used models with unsupervised learning. SOM, with its learning vector quantization variations (LVQ1-3), is used for vector quantization, clustering, feature extraction, or principal component analysis [63].

4. *Sparse distributed memory* (SDM) suggested by Kanerva [58], who argues that it is biologically plausible. The SDM model has been used, for example, in pattern matching and temporal sequence encoding [57]. Rogers [107] has applied SDM to statistical predictions, and also identified SDM as an ideal ANN for massively parallel computer implementation [106].

3.1. Feedforward Networks: Back-Propagation Learning

A feedforward net with three weight layers is shown in Fig. 3. The network topology is such that each node (neu-

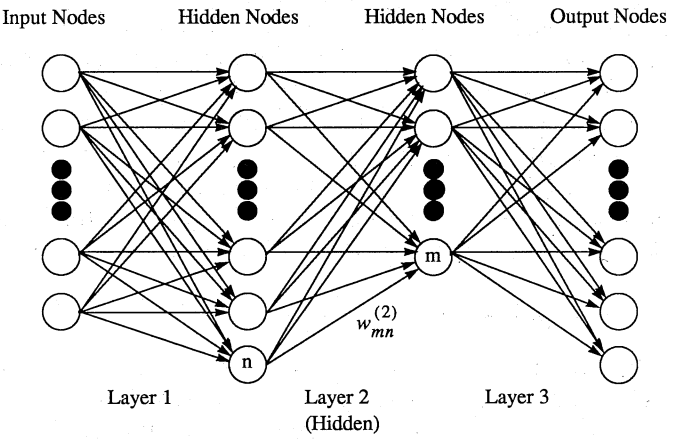

FIG. 3. A three-layer feedforward network.

ron) in a layer receives input from every node of the previous layer. As in most models each node computes a weighted sum of all its inputs. Then it applies a nonlinear activation function to the sum, resulting in an activation value—or output—of the neuron. A sigmoid function, with a smooth threshold-like curve (see Section 4.3), is the most frequently used activation function in feed-forward networks, but hard limiters are also used.

In the first phase of the algorithm the input to the network is provided and values propagate forward through the network to compute the output vector, O. The output vector of the network is then compared with a target vector, T, which is provided by a teacher, resulting in an error vector, E.

In the second phase the values of the error vector are propagated back through the network. The error signals for hidden units are thereby determined recursively: Error values for node layer l are determined from a weighted sum of the errors of the next node layer, $l + 1$, again using the connection weights—now "backward." The weighted sum is multiplied by the derivative of the activation function to give the error value, δ.

Now, finally, appropriate changes of weights and thresholds can be made. The weight change $\Delta w_{ij}^{(l)}$ in the connection to unit i in layer l from unit j in layer $l - 1$ is proportional to the product of the output value, o_j, in node layer $l - 1$, and the error value, δ_i, in node layer l. The bias (or threshold) value may be seen as the weight from a unit that is always on and can be learned in the same way. The algorithm is summarized in Algorithm 1.

Algorithm 1. Back-Propagation Training Algorithm

1. Apply input $O^{(0)} = I$.
2. Compute output $o_j^{(l)} = f(\text{net}_j^{(l)} + b_j^{(l)})$, where $\text{net}_j^{(l)} = \sum_i w_{ji}^{(l)} o_i^{(l-1)}$ for each layer.
3. Determine error vector $E = T - O$.
4. Propagate error backward.
   If node j is an *output* node then the jth element of the error value vector D is
   $\delta_j^{(l)} = o_j^{(l)}(1 - o_j^{(l)})(t_j^{(l)} - o_j^{(l)}) = o_j^{(l)}(1 - o_j^{(l)})e_j^{(l)}$
   else
   $\delta_j^{(l)} = o_j^{(l)}(1 - o_j^{(l)}) \sum_i \delta_i^{(l+1)} w_{ij}^{(l+1)}$.
   Here we have used the fact that the sigmoid function $f(x) = 1/(1 + e^{-x})$ has the derivative $f' = f(1 - f)$.
5. Adjust weights and thresholds:
   $\Delta w_{ij}^{(l)} = \eta \delta_i^{(l)} o_j^{(l-1)}$,
   $\Delta b_i^{(l)} = \eta \delta_i^{(l)}$.
6. Repeat from 1.

By remembering between iterations and adding a portion of the old change to the weight it is possible to increase the learning rate without introducing oscillations. The new term, suggested by Rumelhart and McClelland [110], is called the momentum term and is computed as $w_{ij}^{(l)}(n + 1) = w_{ij}^{(l)}(n) + \Delta w_{ij}^{(l)}(n) + \alpha \Delta w_{ij}^{(l)}(n - 1)$, where α is chosen empirically between 0 and 1. Many other variations of back-propagation exist and some of them have been studied by Fahlman [23].

3.2. Feedback Networks

A feedback network consists of a single set of N nodes that are completely interconnected; see Fig. 4. All nodes serve as both input and output nodes. Each node computes a weighted sum of all its inputs: $\text{net}_j = \sum_i w_{ji} o_i$. Then it applies a nonlinear activation function (see Section 4.3) to the sum, resulting in an activation value—or output—of the node. This value is treated as input to the network in the next time step. When the net has converged, i.e., when the output no longer changes, the pattern on the output of the nodes is the network response.

This network may reverberate without settling down to a stable output. Sometimes oscillation may be desired, otherwise oscillation must be suppressed.

Training or learning can be done in supervised mode with the delta rule [111] or back-propagation [1], or it can be done unsupervised by a Hebbian rule [111]. It is also used "without" learning, where the weights are fixed at start to a value dependent on the application.

3.3. Self-Organizing Maps

Relying on topological ordered maps and self-organization as important concepts, Kohonen developed the SOM [62, 63] which form mappings from a high-dimensional input space into a low-dimensional output space. These maps have been used in pattern recognition, especially in speech recognition, but also in robotics, automatic control, and data compression. The SOM algorithm proceeds in two steps: (i) the network node whose value is closest to the input vector is identified, and (ii) the nodes belonging to the neighborhood of this node (in the output space) change their values to become closer to

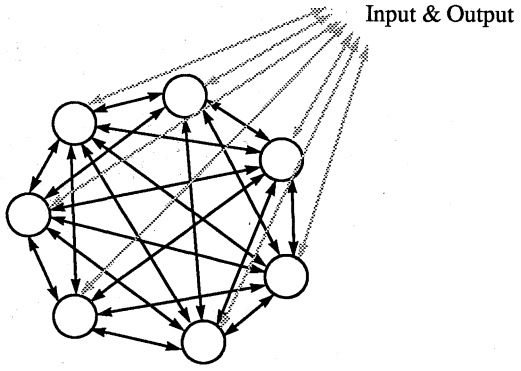

FIG. 4. A seven-node feedback network.

the input. These two steps are repeated with ever-decreasing neighborhood size. The resulting nodes or neurons will develop into specific detectors of different signal patterns.

Described below is the "shortcut version" of the basic SOM algorithm. This version is motivated by the reduction of the computational effort compared to the original one.

Algorithm 2. The SOM Algorithm "Shortcut Version"

1. Find the node (or memory) m_c closest to input x.
$$\|x(t_k) - m_c(t_k)\| = \min_i \|x(t_k) - m_i(t_k)\|.$$
2. Find the neighborhood $N_c(t_k)$.
3. Make the nodes in the neighborhood closer to input.
$$m_i(t_{k+1}) = m_i(t_k) + \alpha(t_k)[x(t_k) - m_i(t_k)] \quad \text{for } i \in N_c(t_k)$$
$$m_i(t_{k+1}) = m_i(t_k) \quad \text{otherwise.}$$
4. Repeat from step 1 with ever-decreasing N_c and α (neighborhood and gain sequence).

With a small change in the update equation (step 3 in Algorithm 2) we can use the same framework for Learning Vector Quantization (LVQ), where the map functions as a clustering or classification algorithm.

3.4. Sparse Distributed Memory

SDM developed by Kanerva [58] may be regarded as a special form of a two-layer feedforward network, but is more often—and more conveniently—described as an associative memory. It is capable of storing and retrieving data at an address referred to as a "reference address." A major difference compared to conventional Random Access Memories (RAMs) is that, instead of having, e.g., 32-bit addresses, SDMs may have 1000-bit addresses. Since it is impossible to have a memory with 2^{1000} locations, SDMs must be sparse. Also, data are stored in counters instead of 1-bit cells as in RAMs

The SDM algorithm has a comparison phase, in which the sparsely distributed locations that are closest to the reference address are identified, and an update (write) or retrieval (read) phase, in which a counter value in each of these locations is used (see Fig. 5).

Algorithm 3. The SDM Algorithm

Training the network (i.e., writing to the memory):

1. The address register is compared to the location addresses and the distances are calculated.
2. The distances are compared to a threshold and those below the threshold are selected.
3. In the selected rows,
   where the data-in register is 1 the counter is incremented,
   where the data-in register is 0 the counter is decremented.

Recall from the network (i.e., reading from the memory):

1. The address register is compared to the location addresses and the distances are calculated.
2. The distances are compared to a threshold and those below the threshold are selected.

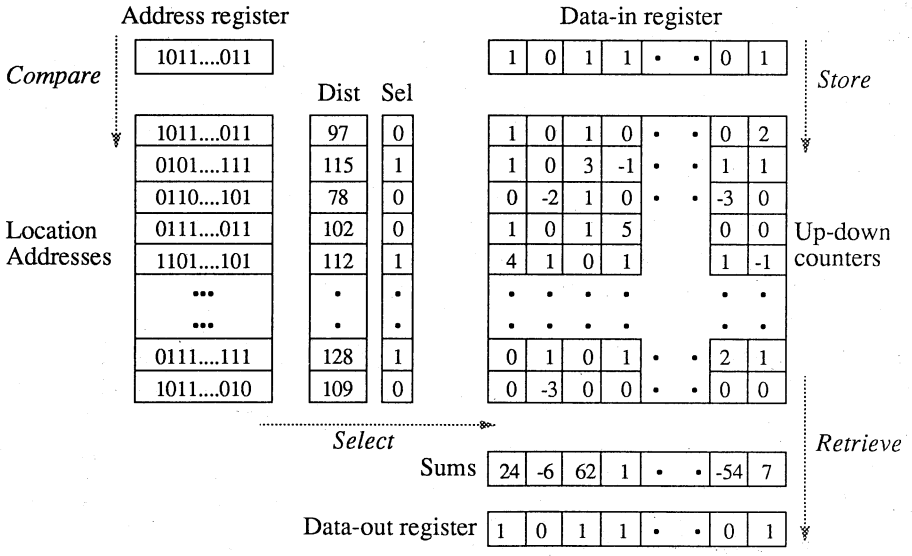

FIG. 5. The organization of a Sparse Distributed Memory as an array of addressable locations. Note that the address as well as the data can be of hundreds of bits in length, and yet there are only a small number (like thousands) of memory locations.

3. The selected rows are added columnwise.
   Where the sum is greater than zero the data-out register is set to one, else it is set to zero.

4.0. COMPUTATIONAL CONSIDERATIONS

The computations involved in neural network simulations show great similarities from one model to another. In this section we discuss some topics that are of general interest and not specific to one single model.

4.1. Basic Computations

For feedforward and feedback network algorithms the basic computation is a matrix-by-vector multiplication, where the matrices contain the connection weights and the vectors contain activation values or error values. Therefore, an architecture for ANN computations should have processing elements with good support for multiply, or even multiply-and-add, operations and a communication structure and memory system suitable for the access and alignment patterns of matrix-by-vector operations.

Assuming N units per layer, the matrix-by-vector multiplication contains N^2 scalar multiplications and N computations of sums of N numbers. The fastest possible way to compute this is to perform all N^2 multiplications in parallel, which requires N^2 processing elements (PEs) and unit time, and then form the sums by using trees of adders. The addition phase requires $N(N-1)$ adders and $O(\log N)$ time.

The above procedure means exploitation of both node and weight parallelism (defined later). For large ANNs this is unrealistic, depending on both the number of PEs required and the communication problems caused. Instead, most of the implementations that have been reported take the approach of basically having as many PEs as the number of neurons in a layer (node parallelism) and storing the connection weights in matrices, one for each layer. The PE with index j has access to row j of the matrix by accessing its own memory. Referring to Algorithm 1, a problem appears in step 4 relative to step 2. While step 2 corresponds to the matrix–vector operation WO, step 4 corresponds to $W^T\delta$. This means that we need to be able to access W^T as efficiently as we can access W. This introduces a mapping problem which we will return to in Section 6. Regardless of the mapping chosen, multiply-and-add is the basic operation in these calculations.

The first step of the SOM algorithm, using an inner-product as distance measure, can also be seen as a matrix-by-vector multiplication, where the matrix is composed of the weight vectors of the nodes and the vector is the training vector. Another distance measure used for the first step is a Euclidean metric which cannot be described as a matrix–vector operation. Still the basic operation in both metrics is multiply-and-add. Discussion on the two different distance measures can be found in [63, 88]. From the resulting vector a maximum or minimum must be found. The efficiency of this operation is strongly dependent on the communication topology, but may also depend on the characteristics of the PEs. In a later section we will demonstrate how bit-serial processors offer specific advantages. After a maximum (or minimum) node is found its neighbors are selected and updated. The selection time will depend on the communication topology, and the update time on the length of the training vectors.

Also in SDM the first step is a matrix-by-vector multiplication. But as both the matrix and the vector are binary valued the multiplications are actually replaced by exclusive-or and the summation by a count of ones. The counters are thereafter compared with a threshold. In all active positions the up-down counters are updated.

Thus, to be efficient for ANN computations computers need to have support for matrix-by-vector multiplications, maximum finding, spreading of activity, count of ones, and comparisons. In some of the implementations that we review these matters have been solved on existing parallel computers, in others new architectures have been devised, targeted at computations of this kind.

4.2. Numerical Precision

In order to optimize the utilization of the computing resources, the numerical precision and dynamic range in, e.g., the multiply-and-add operations should be studied with care.

With optical, analog, or bit-serial computers it is *not* very attractive to use 32- or 64-bit floating-point numbers for weights and activation values. The issue of weight sensitivity becomes important; how sensitive are the networks to weight errors? Unfortunately, one of the most used algorithms, back-propagation, is very sensitive to the precision and number range used [39]. This is due to the existence of large flat areas in the error surface, in which the BP algorithm may have difficulty in determining the direction in which to move the weights in order to reduce the error. To make progress from such areas high numerical precision is needed. If the neural network calculations are run on a computer which has advanced hardware support for floating-point calculations the required accuracy does not raise any problem. On the other hand, for many tasks of integer type, like low-level vision problems, the use of floating-point numbers will lead to a more complex architecture than necessary.

Using ordinary back-propagation with low precision without modifications will lead to instability and the net will often be unable to learn anything. There are modifications to back-propagation which seem to improve the situation somewhat [13, 21, 78, 116], and there are experiments in which different precision is used at different

stages of the algorithm [86]. By using high precision at the beginning of the training and lessening the precision as the network trains, the number of bits needed for weights in the fully trained network can be very low (a few bits). Without too large modifications, 8–16 bits per weight seems to be sufficient for most problems [11, 21, 85, 135]. More exact calculations of the sensitivity to weight errors and the precision needed can be found in [21, 121, 122, 123].

By some authors [39, 135] weights have been found to need a large dynamic range, implying floating-point representation. However, the use of weight saturation, i.e., limiting the weights to a certain limit, may remove the need for floating-point numbers [85].

Low precision is also attractive for ANNs as it makes the algorithms more biologically plausible [131]. An upper limit of the accuracy that the brain needs for its calculation could be estimated to 7–8 bits; i.e., neurons have a "dynamic range" of about 100 levels. The calculations are also fault tolerant. That is, if one neuron fails to fire, the computation is still carried out in the right fashion.

Finally, it should be noted that there is probably a trade-off between using few weights (nodes) with high precision and using many weights (nodes) with low precision.

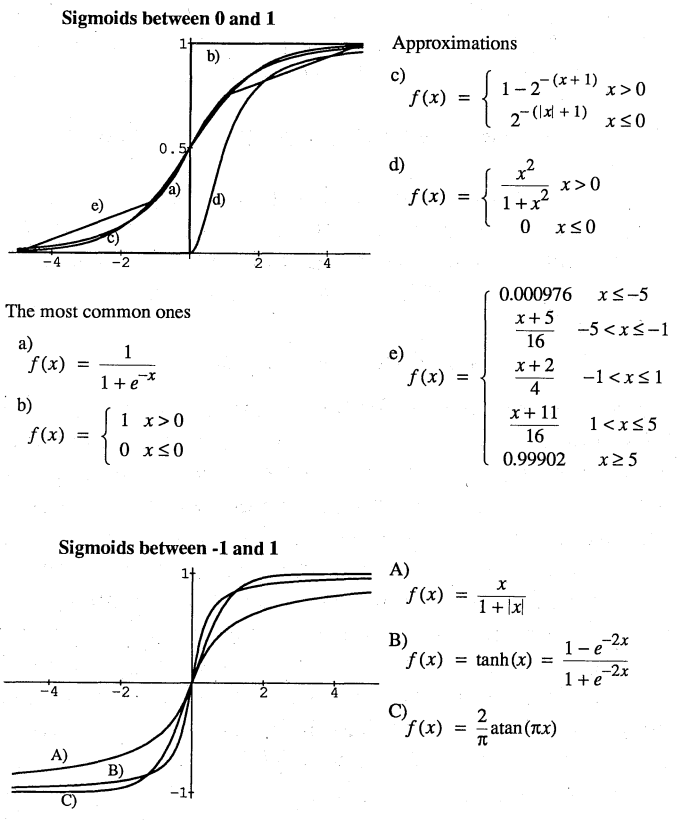

FIG. 6. Some of the used activation functions of sigmoid type.

4.3. Sigmoid

In many of the algorithms, e.g., BP and Hopfield networks, a sigmoid function like $f(x) = 1/(1 + e^{-x})$ needs to be calculated; see Fig. 6. To do this efficiently many implementations use a table lookup instead of direct calculation (typically with 8 bits precision). Others try to approximate the sigmoid with a piecewise linear function [11] like(e) in Fig. 6. Also, an approximation based on power of 2 calculations has been proposed, with digital computers in mind [94], (c) in Fig. 6.

A combination of table lookup and power of 2 calculation was tried in [136] for the GF11 computer, but in the end only table lookup and interpolation were used.

Table lookup on SIMD computers without local address modification seems difficult but is possible by a cyclic rotation and comparison. It takes n steps to do n table lookups using n PEs connected in a ring [132]. Other ways to distribute the lookup table have been discussed by Marchesi *et al.* [80].

In [115] e^x was calculated by means of range reduction techniques. The total number of operations required to calculate the sigmoid was five add/subtracts, one logical, two divisions, two shifts, and three multiplications.

In the backward phase the derivative $f(x)$ is to be calculated. The much used sigmoid $f(x) = 1/(1 + e^{-x})$ has the "nice" derivative given by $f'(x) = f(x)(1 - f(x))$.

Some networks and training situations have turned out to benefit from a sigmoid function between -1 and 1 instead of the usual 0 and 1. Some are given in Fig. 6. The function (A) has the useful property that is does not involve any transcendental functions.

4.4. Data Representation

When real and/or analog inputs to the ANN are used, the data representation must be studied carefully. For the binary code the problem is that Hamming distance (HD) is not a valid measure of similarity; see Table I. The so-called thermometer code, or the simple sum of ones, solves the problem with the HD but is wasteful of code bits, and thus stands in contrast to the requirement of using as few bits as possible.

Penz [93], referring to Willshaw *et al.* [134], has suggested a modification to the thermometer code, called the closeness code. This code has as few active positions as possible; i.e., it is optimally sparse, but still retains the HD as a similarity measure. The number of ones in an N-bit vector is $\log_2 N$.

There is a denser (with respect to code word length) version of a closeness code suggested by Jaeckel [56, 59] which could be called a band-pass code. The name reflects the fact that the thermometer code may be seen as a set of low-pass filters but Jaeckel's suggestion may be seen as a set of band-pass filters. Both a closeness and a

TABLE I
Different Ways to Encode Data: Binary, Thermometer, Closeness, and Band-Pass Codes

Decimal value	Binary		Thermometer		Closeness		Band Pass	
	Code	HD	Code	HD	Code	HD	Code	HD
0	000	0	0000000	0	11100000 00	0	11000	0
1	001	1	1000000	1	01110000 00	2	11100	1
2	010	1	1100000	2	00111000 00	4	01100	2
3	011	2	1110000	3	00011100 00	6	01110	3
4	100	1 !	1111000	4	00001110 00	6	00110	4
5	101	2	1111100	5	00000111 00	6	00111	5

6	110	2	1111110	6	x0000011 10	5/4	00011	4

7	111	3	1111111	7	xx000001 11	4/2	10011	3
							10001	2
							11001	1
							11000	0

Note. All but the binary codes have Hamming distance (HD) as the measure of similarity. Closeness and thermometer codes are wasteful of code bits, but the closeness code has fewer active bits. Band pass has good code density while keeping the HD as the measure of distance.

band-pass code can be extended to a circular code as shown below the dashed lines in Table I. A circular code is useful when coding things like angles.

When coding sets of mutually unrelated items, like letters in an alphabet, it is important *not* to introduce order or similarities that do not exist. Coding a, b, c, ... as 1, 2, 3, ... introduces a similarity between adjacent letters which has no relation to their use in language. Instead, 26 nodes of which only one is active, may be used.

4.5. Bit-Serial Calculations

Many of the massively parallel processors use bit-serial PEs. For the majority of operations, processing times on these computers grow linearly with the data length used. This may be regarded as a serious disadvantage (e.g., when using 32- or 64-bit floating-point numbers), or as an attractive feature (use of low-precision data speeds up the computations accordingly). In any case, bit-serial data paths simplify communication in massively parallel computers.

4.5.1. Multiplication Done Bit-Serially

In simple bit-serial processors the multiplication time grows quadratically with the data length. However, bit-serial multiplication can actually be performed in the time required to read the operands (bit by bit, of course) and store the result. The method, based on the carry-save adder technique, requires as many full adders as the length of one of the operands. Figure 7 shows the design of such a multiplier circuit.

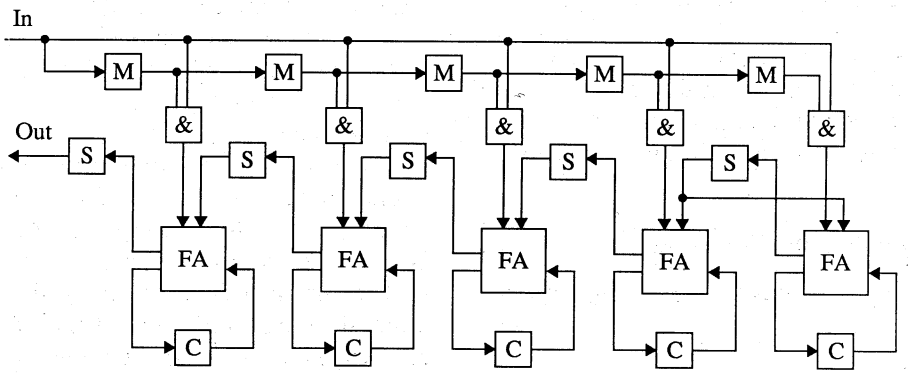

FIG. 7. Design of a two's-complement bit-serial multiplier. It is operated by first shifting in the multiplicand, most significant bit first, into the array of M flip-flops. The bits of the multiplier are then successively applied to the input, least significant bit first. The product bits appear at the output with least significant bit first.

This design was proposed but not implemented in the LUCAS project [27], and will be used in the continued project, REMAP[3] (Reconfigurable, Embedded, Massively Parallel Processor Project). A similar multiplier design has also been proposed for "Centipede," a further development of the AIS-5000 concept [135].

4.5.2. Floating-Point Calculations Done Bit-Serially

Floating-point calculations raise special problems on SIMD computers with bit-serial PEs. Additions and subtractions require the exponents to be equal before the operations are performed on the mantissas. This alignment process requires different PEs to take different actions, and this does not conform with the SIMD principle. The same problem appears in the normalization procedure.

However, these problems may also be solved by a fairly reasonable amount of extra hardware. Åhlander and Svensson [140] propose the addition of stacks in the PEs to hold the mantissas during alignments and normalizations. This arrangement allows floating-point operations to be performed as fast as data can be provided bit-serially from the memory.

4.5.3. Search for Maximum or Minimum Done Bit-Serially

Some operations benefit from the bit-serial working mode and can be implemented very efficiently. Search for maximum or minimum is such an operation. Assuming that one number is stored in the memory of each PE, the search for maximum starts by examining the most significant bit of each value. If anyone has a one, all PEs with a zero are discarded. The search goes on in the next position, and so on, until all bit positions have been treated. The time for this search is independent of the number of values compared; it depends only on the data length (provided that the number of PEs is large enough).

5.0. PARALLELISM IN ANN COMPUTATIONS

For implementation on a parallel computer, parts of the algorithm that can be run in parallel must be identified. Unfolding the computations into the smallest computational primitives reveals the different dimensions of parallelism.

5.1. Unfolding the Computations

A typical ANN algorithm has the following structure:

```
For each training session
  For each training example in the session
    For each layer (going Forward and Backward)
      For all neurons (nodes) in the layer
        For all synapses (weights) of the node
          For all bits of the weight value
```

This shows that there are (at least) six different ways of achieving parallelism:

```
Training session parallelism
  Training example parallelism
    Layer and Forward–Backward parallelism
      Node (neuron) parallelism
        Weight (synapse) parallelism
          Bit parallelism
```

5.2. The Dimensions of Parallelism

Here follows a discussion on each of the different ways of achieving parallelism. Which of the dimensions of parallelism are chosen in any particular implementation will depend on the constraints imposed by the hardware platform and by the constraints of the particular algorithm that is to be implemented.

5.2.1. Training Session Parallelism

Training session parallelism means starting different training sessions on different PEs. Different sessions may have different starting values for the weights, and also different learning rates. Using parallel machines with complex control makes it even possible to train networks of different sizes at the same time.

5.2.2. Training Example Parallelism

The number of training examples used is usually very large, typically much larger than the number of nodes in the network. The parallelism of the training set can be utilized by mapping different training examples to different PEs and letting each PE calculate the outputs for its training example. The weight changes are then summed.

Doing the weight update this way (batch or epoch updating) means that it is not done exactly as in the serial case. Back-propagation is known to perform gradient descent if update of weights takes place after processing of all the training data [110, 119]. Empirically it is found that updating after each pattern will save CPU cycles, at least on a sequential computer.

Training example parallelism is easy to utilize without communication overhead. Thus it gives an almost linear speedup with the number of PEs. However, a corresponding reduction in training time (time to reduce the total error to a specific level) should not be taken for granted. Even if this method gives a more accurate gradient it does not necessarily allow more weight updates to occur. Therefore there is a limit on the amount of actual training time speedup that is achievable using this method of parallelism. Parker [92] has shown that on a

30 × 30 × 10 network and 4156 training examples the optimal batch size was only 18. Beyond that level the refinement of gradient was wasted. This means that the use of extensive number crunching circuitry in order to utilize training example parallelism, although giving high CUPS (Connection Updates Per Second) performance, does not guarantee a corresponding reduction of training time.

Training example parallelism also demands that the PEs have enough memory to store the activation value of all the nodes in a network. With a 256-kbit memory per PE and 32-bit floating-point number representation, only about 8000 nodes can be simulated [119]. On the other hand, any sparsity of the network connections can be fully exploited with this type of parallelism.

5.2.3. Layer and Forward–Backward Parallelism

In a multilayer network the computations may be pipelined; i.e., more than one training pattern is going through the net at the same time. If the model has a backward pass, like BP, it is also possible to "fold" the pipeline back again.

5.2.4. Node (Neuron) Parallelism

The parallel processing performed by many nodes in each layer is perhaps the most obvious form of parallelism in an ANN. Each node computes a weighted sum of all its inputs. This form of parallelism corresponds to viewing the calculations as matrix operations and letting each row of the matrix map onto a processor.

Most of the layered models only send their activation values forward after all nodes have been calculated in a layer. This means that the maximum parallelism is available for the widest node layer (excluding the input layer). If this degree of parallelism is fully utilized, all layers with less nodes cannot fully utilize the computer.

5.2.5. Weight (Synapse) Parallelism

At each input to a neuron the arriving activation value is multiplied by the weight of the specific input. This can be done simultaneously at all inputs to the neuron. The subsequent summation of all the products may also be parallelized using a suitable communication structure.

5.2.6. Bit Parallelism

Utilizing the full degree of bit parallelism (i.e., treating all bits in a data item simultaneously) is often taken for granted. However, giving up this form of parallelism, and treating data bit-serially, increases the possibilities of using some of the other forms.

5.3. The Degrees of Parallelism

The typical degree of parallelism varies widely between the six different kinds, as the table below shows. As an illustration, the degrees of parallelism of the well-known NETtalk application (see section 7.2.1) have been given as well.

Parallelism	Typical range	NETtalk
Training session	$10–10^3$	100
Training example	$10–10^7$	5000
Layer and Forward–Backward	1–6	1
Node (neuron)	$100–10^6$	120
Weight (synapse)	$10–10^5$	203
Bit	1–64	32

The table gives an indication of what dimensions should be utilized in a massively parallel computer. Such a computer is capable of performing at least thousands of elementary operations simultaneously. Hence an ANN implementation that is to utilize the computing resources efficiently must utilize at least one of the following dimensions:

Training session parallelism
Training example parallelism
Node parallelism
Weight parallelism

The use of the two first-mentioned types is of interest only in batch processing situations in order to train a network. In real-time applications where the ANN is interacting with the outside world, training session and training example parallelism are not available. In those cases, node and/or weight parallelism must be chosen, maybe in combination with, e.g., bit and layer parallelism.

6.0. COMMUNICATION

A high degree of connectivity and large data flows are characteristic features of neural computing models. Hence the structure and bandwidth of internal and external (i.e., I/O) communication in the computer to be used are of great importance. Depending on what dimension of parallelism is chosen the demands on the communication will be different. We review the major communication principles and comment on their use in ANN implementations.

6.1. Communication by Broadcast

In most parallel computers the most efficient way to communicate is to broadcast, since so many destinations receive information simultaneously. In fact, in SIMD computers broadcast is also used to distribute control information. As shown above, training example parallel-

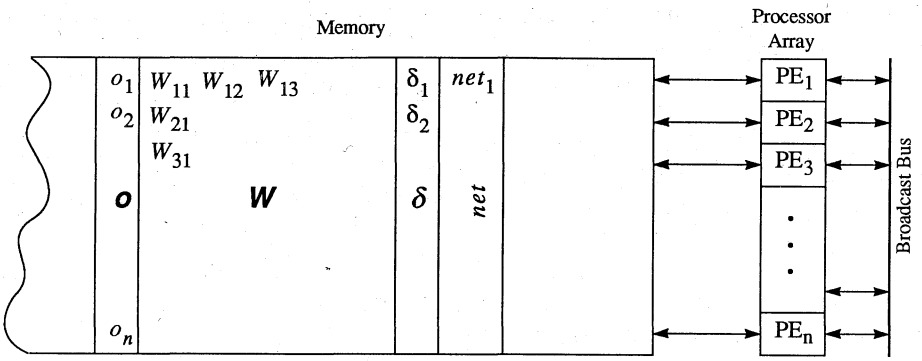

FIG. 8. Mapping of node parallelism into a processor array.

ism has a very large amount of possible parallelism. Since the basic communication needed with this form of parallelism is broadcasting it is a good choice if the maximum speed (CUPS) is to be obtained. If the batch or epoch type of weight updating cannot be used, node or weight parallelism must be used. With the required communication patterns in those forms of parallelism it is less obvious how broadcast can be used, and therefore it is harder to obtain maximum performance.

Using node parallelism, perhaps the most natural way to map the forward pass of a BP calculation on a processor array is to see it as a matrix–vector multiplication and map each row of W into one PE; see Fig. 8. In each step of the multiplication process, one PE broadcasts its node activation o_i to all other nodes. Each PE then multiplies the value with a weight value from the corresponding column of W and adds the result to a running sum net_i. After all activation values have been sent and multiplied, each PE holds one element of the resulting vector. In the backward phase, summation is required across the PEs instead of within each PE. This is slow unless another communication structure is added, e.g., an adder tree as proposed in [124]. Mapping in this way may also introduce inefficiency problems if the layers are of very different sizes. It is also difficult to utilize any sparsity in the connections with this method.

6.2. Grid-Based Communication

A natural way to arrange the communication required for weight parallelism is grid-based communication suitable for two-dimensional arrays of PEs. The PEs of the top edge, say, correspond to the source node layer, and the PEs of the right edge, say, correspond to the destination node layer. The weight matrix is distributed over the rest of the PEs. The input layer nodes at the top send their values down over the matrix by vertical broadcast. Then there is a horizontal summation phase. This is then repeated for the next layer, first horizontally and then vertically.

This scheme has been used or suggested by, e.g., Singer [120] and Fujimoto and Fukuda [33].

6.3. Communication by Circulation

One way to solve the communication when node parallelism is used is by a "systolic" ring structure [52, 70–72, 100]; see Fig. 9. In the forward phase (upper part of the figure) the activation values are shifted circularly along the row of PEs and multiplied with the corresponding weight values. Each PE accumulates the sum of the products it has produced. In the backward phase (lower part of the figure) the accumulated sum is shifted instead.

This scheme shares the possible inefficiency problems with the broadcast-based scheme (see end of Section 6.1).

6.4. Communication by General Routing

Some massively parallel computers, e.g., the Connection Machine, provide support for general routing by packet switching. Utilizing this facility is a straightforward method on these computers, but, since general routing normally is much slower than regular communication, the ANN computations will be communication bound, maybe with the exception of sparsely connected networks.

An implementation of directed graphs on computers lacking general routing capability has been suggested by Tomboulian [128]. It relies on SIMD computers with very modest communications facilities. Sending a value or a message between two nodes amounts to routing the message from PE to PE. A send is divided into time slots where each node knows if or where it should pass the current incoming message. There are means to extend the communication pattern dynamically which makes it very attractive for networks that use structural adaptation, like Fahlman and Lebiere's "Cascade Correlation" [24] or Kassebaum *et al.*'s [60] and Tenorio and Lee's self-organizing neural network (SONN) algorithm [125, 126].

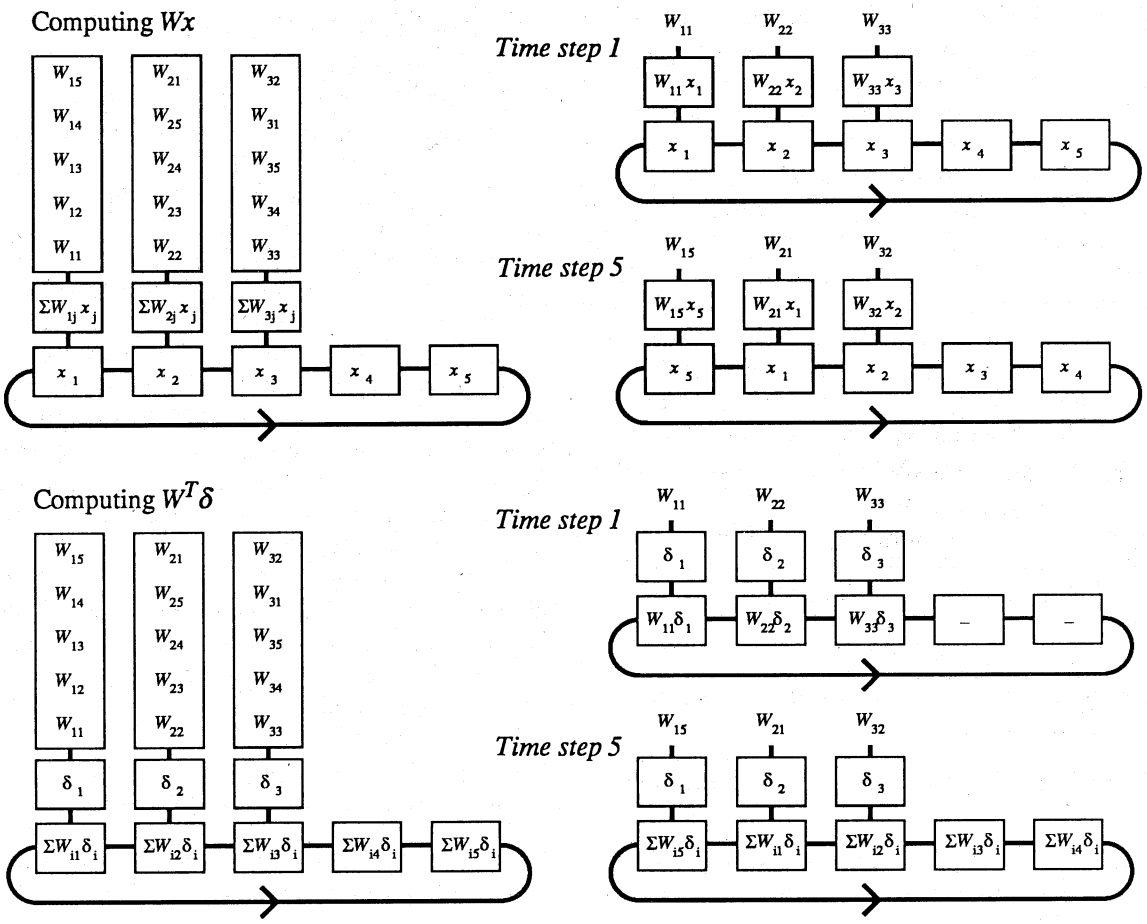

FIG. 9. Forward (top) and backward (bottom) phase of a BP calculation on a systolic ring processor.

Tomboulian's method has for dense networks been found to consume a great deal of PE memory for its tables [128]. For sparse networks the time and memory requirements are proportional to the product of the average number of connections per neuron and the diameter of the array.

6.5. Comments on Communication

For all communication strategies except the general routing methods, sparse connection matrices will lead to underutilization of the computer's PE resources. Lawson *et al.* have addressed this problem with their "SMART" (sparse matrix adaptive recursive transform) machine [73]. By using tables and special hardware they can make use of zero-valued connections to reduce simulation size and time. The communication is based on a ring structure together with a bus. However, Lawson's solution is not directly applicable on a massively parallel SIMD computer.

The massive flow of data into and out of the array, which will be the case in practical real-time applications and also in out of core calculation (very large simulations), places specific demands on the communication structure. In the design of a massively parallel computer for ANN, the PEs should be optimized for the typical operations, in order not to make the processing computation bound. Furthermore, the interconnection network should be optimized for the computational communication pattern, in order not to make the processing communication bound. Finally, the I/O system should be optimized to suit the needs of the application, in order not to make the system I/O bound. So far not very much attention has been paid to the latter problem complex, but it is certainly very much connected with the communication matters.

Training example parallelism is the form of parallelism that puts the weakest demands on communication. However, it is not interesting in real-time applications or in pure recall situations.

7.0. MEASURING THE SPEED OF ANN SIMULATIONS

In order to compare different implementations, some kind of standard measuring procedure is needed. The

number of multiply-and-add operations per second of which the computer is capable might be considered for such a measure, since this operation was identified as the most important one in most of the algorithms. However, it can serve only as an upper limit for performance; i.e., it marks the lowest speed that the computer manufacturer guarantees will never be exceeded. The utilization of this potential may be very different for the various ANN models and for different problem sizes.

Some of the commonly used indications of speed will be given below. They are of two kinds: first, there are general speed measurements, similar to the multiply-and-add performance measure. In order for these to be of any real value, the ANN model and problem size for which they were measured should always be given as additional information. Second, there are benchmarks, i.e., commonly implemented applications of specific size. The area is still too young (i.e., in the present "wave") for benchmarks to be very well developed.

7.1. Measurements

Some measurements commonly used in the ANN community are the following.

7.1.1. CPS or IPS—Connections (or Interconnections) per Second

Each time a new input is provided to an ANN, all connections in the network must be computed. The number of connections per second that the network can perform is often used as a measure of performance. When measuring CPS, computing the nonlinear thresholding function should be included. Comparing the CPS measure to the multiply-and-add measure roughly indicates how well the specific algorithm suits the architecture. When comparing CPS values for different implementations, the precision of calculation is important to consider. Other things which may influence the CPS measure are the problem size and the choice of nonlinear thresholding function.

7.1.2. CUPS or WUPS—Connection (or Weight) Updates per Second

The CPS number only measures how fast a system is able to perform mappings from input to output. To indicate the performance during training a measurement of how fast the connections can be updated is given as a CUPS figure. In a back-propagation network both the forward and the backward passes must be computed. Typically the CUPS number is 20–50% of the CPS number.

For self-organizing maps CUPS have been used as the number of connections in the network multiplied by the number of network updates per second, despite the fact that very few connections are actually updated when using small neighborhoods at the end of a training session.

7.1.3. Epochs

An epoch is defined as a single presentation of each of the examples in the training set, in either fixed or random order. The concept is well-defined only for problems with a fixed, finite set of training examples. Modification of weights may occur after every epoch, after every pattern presentation, or on some other schedule.

The epochs concept is sometimes used when measuring the number of times one must run through the training set (one run is one epoch) before some predefined condition is met. This is used to compare different versions of an algorithm or different algorithms. When doing this one should be aware of the fact that the computational effort to go through an epoch may vary considerably from one algorithm to another. Even if an algorithm learns in half the number of epochs it can still take longer time because the calculations are more complicated.

An epochs per second measure may also be used as an alternative to CUPS to indicate the speed of learning. It gives a number that is easier to grasp than the CUPS number. Of course the measure is strongly related to the problem and training set used.

For problems where an epoch is not well defined, learning time may instead be measured in terms of the number of individual pattern presentations.

7.1.4. CPSPW—Connections per Second per Weight (or SPR—Synaptic Processing Rate)

A measure that indicates the balance between processing power and network size (i.e., number of weights) has been introduced by Holler [46]. His argument for the importance of this measure is the following: Biological neurons fire approximately 100 times per second. This implies that each of the synapses processes signals at a rate of about 100 per second; hence the SPR (or, to use Holler's terminology, the CPSPW) is approximately 100. If we are satisfied with the performance of biological systems (in fact, we are even impressed by them) this number could be taken as a guide for ANN implementations. Many parallel implementations have SPR numbers which are orders of magnitude greater than 100, and hence have too much processing power per weight. A conventional sequential computer, on the other hand, has an SPR number of approximately 1 (if the network has about a million synapses); i.e. it is computationally underbalanced. It should be noted that Holler's argument is of course not applicable to batch processing training situations.

7.2. Benchmarks

There are some commonly used problems that may be considered benchmark problems. They are used in a number of different ways, e.g., to measure learning speed, quality of ultimate learning, ability to generalize, or combinations of these factors. Thus their use is not

restricted to speed comparisons. On the contrary, most of the benchmarks have been introduced to compare algorithms and not architectures. This means that most of the benchmarks should not be used to compare suitability of architectures for the simulation of ANNs. Among such benchmark problems are the XOR problem, the Parity problem, and the Two Spirals problem [22]. They are all small problems intended for qualitative comparison of algorithms.

7.2.1. NETtalk

A larger and more realistic application is known as NETtalk, a text to phoneme translation solved by a back-propagation network, described by Sejnowski and Rosenberg [114]. The task is to train a network to produce the proper phonemes, given a string of letters as input. This is an example of an input/output mapping task that exhibits strong global regularities, but also a large number of more specialized rules and exceptional cases. It is often used as a benchmark.

The experimental setup used by Sejnowski and Rosenberg [114] described by Fahlman in [22] was the following: The input to the network is a series of seven consecutive letters from the training text. The central letters in this sequence is the "current" one for which the phonemic output is to be produced. Three letters on either side of this central letter provide a context that helps to determine the pronunciation. Of course, there are a few words in English for which this local seven-letter window is not sufficient to determine the proper output. For the study using this "dictionary" corpus, individual words are moved through the window so that each letter in the word is seen in the central position. Blanks are added before and after the word as needed. Some words appear more than once in the dictionary, with different pronunciations in each case; only the first pronunciation given for each word was used in this experiment.

A unary encoding is used. For each of the seven letter positions of the input, the network has a set of 29 input units: one for each of the 26 letters in English, and three for punctuation characters. Thus, there are $29 \times 7 = 203$ input units in all. The output side of the network uses a distributed representation for the phonemes. There are 21 output units representing various articulatory features such as voicing and vowel height. Each phoneme is represented by a distinct binary vector over this set of 21 units. In addition, there are 5 output units that encode stress and syllable boundaries. Typically 60 to 120 hidden units have been used.

In the absence of very large benchmarks, we will sometimes use figures on NETtalk to compare architectures (if they have been reported). Otherwise we will report the implemented network structure and training method together with the performance measure given.

8.0. CHARACTERIZATION OF COMPUTER ARCHITECTURES

As the structure of ANN algorithms is naturally parallel it is relatively easy to make use of a very large number of PEs. Computers with a very large number of PEs are often called massively parallel. Being massive should mean that the structure gives an impression of being solid. That is, the number of units should be so high that is impossible to treat them individually; they must be treated *en masse*. By definition then, each PE must be told what to do without specifying it individually. This is actually the concept of the SIMD or SCMD computer (defined later).

The lower bound for massive parallelism will then be set by the largest computer in which each PE is treated as an individual with its own instruction flow (MIMD). We think that for the moment $2^{12} = 4096$ is a suitable limit.

An interesting extension to massively parallel is the concept of *continuously* parallel. This should mean the limit of massively parallel as the number of processing elements becomes infinite [77].

It is useful to have characterizations also of the lower degrees of parallelism. To get a reasonable "definition" with a nice symmetry we suggest a rough division between *highly* parallel, *moderately* parallel, and *barely* parallel. By defining the limits to 2^{12}, 2^8, 2^4, 2^0 we have an easy-to-remember scheme for the characterization. When appropriate, in the future the limits may be moved upward. Summarizing this we get the following "definitions" which will be used in this paper (N stands for the number of PEs):

Continuously parallel	$N \to \infty$
Massively parallel	$N \geq 2^{12}$
Highly parallel	$2^8 \leq N < 2^{12}$
Moderately parallel	$2^4 \leq N < 2^8$
Barely parallel	$2^0 < N < 2^4$

This characterization is completed with an "orthogonal" one describing the computational power of the PEs. The power or complexity can of course be measured in many ways but as a coarse measure we use the bit-length of the natural data type for the processing elements. These two characterizations result in the diagram shown in Fig. 10.

We will concentrate on the massively and highly parallel architectures in the next section. But we will also, for comparison, include some moderately and even barely parallel computers like Warp systems with more complex control. This is because the use of these computers has given interesting results with respect to algorithms and ways of mapping ANNs to a parallel computer. It should be noted that many of those algorithms do not use the powerful control, but instead use a SIMD or SCMD structure.

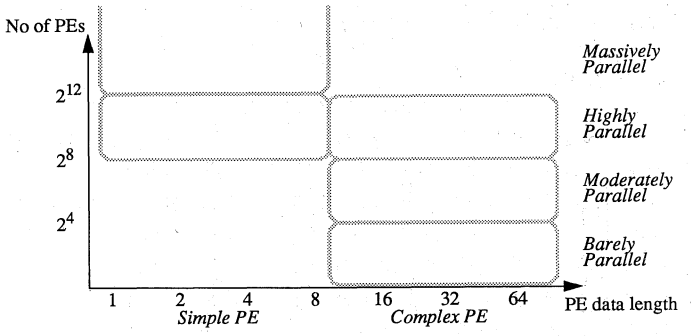

FIG. 10. Classifying architectures for ANN by dividing according to the number of and complexity of the processing elements that are used.

Equally important as the degree of parallelism is the organization of the computer. The much used characterization due to Flynn [28] divides the computers into groups according to the number of instruction streams and the number of data streams. Of interest for ANN computations are the groups with multiple data streams: SIMD (Single Instruction stream, Multiple Data streams) and MIMD (Multiple Instruction streams, Multiple Data streams).

To characterize a MIMD computer used as a SIMD architecture, SCMD (Same Code for Multiple Data streams) is suggested and used in this paper.

8.1. Division of SIMD

The SIMD category in itself shows great architectural variations. We briefly review some of the major groups.

8.1.1. Systolic Arrays

Systolic arrays represent a general methodology for mapping high-level computations into hardware structures. Developed at Carnegie Mellon by Kung and others [68], the concept relies on data from different directions arriving at cells/PEs at regular intervals and being combined. The number of cells in the array and the organization of the array are chosen to balance the I/O requirements.

Systolic architectures for ANN have been proposed by, among others, Hwang *et al.* [52] and Kung and Hwang [70, 71]. They have found that a ring structure or cascaded rings are sufficient (cf. Section 6.3 on communication by circulation).

8.1.2. Linear Processor Arrays

In this group are the processor arrays with the simplest structure, the linear (one-dimensional) arrays. The linear structure is often combined with the ring structure and the bus structure (i.e., broadcast). Actually, the arrays in this group are typically not massively parallel due to the limitations of the communication structure. Some of the arrays can be scaled to at least a few thousand processors. They can however be strung together with similar arrays and form a multiple SIMD array which can be much larger than any single SIMD module.

8.1.3. Mesh-Connected Processor Arrays

In silicon the planar structure of the connecting medium will tend to favor planar communication networks. Therefore, it is natural to build the communication on a mesh or a grid; i.e., each PE has four (or eight) neighbors. Even the computers with multidimensional communication have included mesh connections for faster local communication. Examples of mesh-connected arrays are DAP (Distributed Array Processor) [51], MPP [101], and BLITZEN [9].

8.1.4. Multidimensional Processor Arrays

Multidimensional architectures like hypercubes allow a more general communication pattern than any of the previous arrays. That is, no PE is further away than $\log_2 N$ steps, where N is the number of PEs.

It has been found that general communication is used mainly for transfers between "parameter spaces" (e.g., image → edges → corners). None of the efficient implementations of ANN algorithms on any of the studied architectures used/needed multidimensional communication.

9.0. PARALLEL COMPUTERS DESIGNED OR USED FOR ARTIFICIAL NEURAL NETWORKS

Placing some of the parallel computers used for ANN into the diagram of Fig. 10 results in the map shown in Fig. 11.

We will now give brief descriptions of these machines and review the reported ANN implementations. We certainly do not cover all the massively and highly parallel machines but our selection should be representative. The order of presentation is alphabetical within each group.

9.1. Massively Parallel Machines with Simple PEs

9.1.1. AAP-2

AAP-2 [132] is a two-dimensional (2D) mesh-connected computer enhanced with bypasses and ripple through operations. It contains 64K (256 by 256) bit-serial PEs featuring a simple 1-bit ALU and a 144-bit register file. There is also a 15-bit control register which can control each PE individually in a primitive way.

BP has been implemented in a node and weight paral-

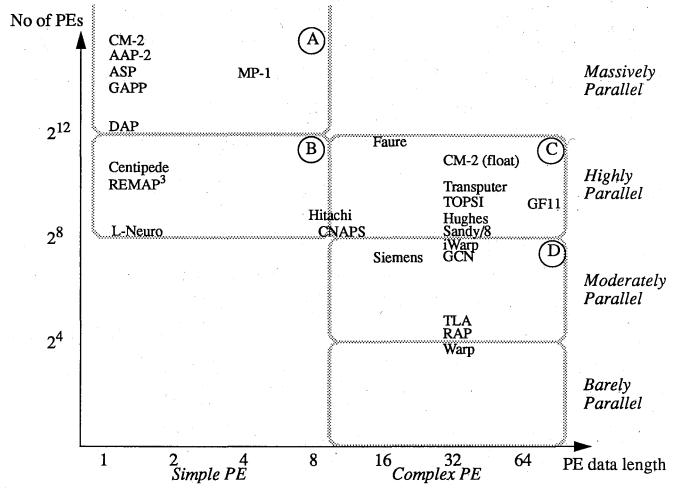

FIG. 11. A way of classifying architectures for ANN. A corresponds to massively parallel machines with simple PEs in Section 9.1, B to highly parallel machines with simple PEs in Section 9.2, C to highly parallel machines with complex PEs in Section 9.3, and D to moderately parallel machines with complex PEs in Section 9.4.

lel fashion using an interesting circular way to perform table lookup of the sigmoid function. Using 26 bits for weights and activations and a 256 × 256 × 256 network 18 MCUPS was achieved. When different sizes of the layers were used the efficiency decreased.

9.1.2. *Associative String Processor (ASP)*

The ASP is a computer designed and built by University of Brunel and Aspex Ltd. in England, with Lea as the principal investigator [74]. The computer consists of simple PEs (1-bit full adder, $n + a$-bit comparator) which are strung together into "strings" of more "complex" types such as 32-bit integers. Each processor has a 96-bit data register and a 5-bit activity register. Each string is linked by a communication network and there are data exchanges and buffers to support high-speed data I/O. The architecture is expandable up to 262,144 processors arranged as 512 strings of 512 processors each. A 16K machine has been built and a 64K machine is to be completed by the summer of 1992.

Krikelis and Grözinger [67] have implemented Hopfield net and BP on a simulator of the architecture. In the Hopfield net one neuron is mapped onto each PE as long as the internal memory is sufficient (1800 nodes). At this maximum, the real machine should be able to run at 1600 MCPS.

The BP network implementation also uses node and weight parallelism simultaneously. On a 63 × 45 × 27 network 4323 PEs are utilized, but in some time instances there are only 46 PEs actively taking part in the calculation. The weights are represented by 16-bit fixed-point numbers. A thresholded picture (1 bit deep) is used as input and the activation in the other nodes is represented by 12 bits. The sigmoid is approximated with a group of conditionals. The simulation indicates a performance of 12 MCUPS on the real machine.

9.1.3. *Connection Machine (CM, CM-2)*

The Connection Machine [40, 127] manufactured by Thinking Machines Corporation (TMC) is for the moment the most massively parallel machine built (from 8K up to 64K processing elements). In addition to its large number of processors, two of its strong points are its powerful hypercube connection for general communication and the multidimensional mesh connection for problems with regular array communication demands. In the CM-2 model TMC also added floating-point support, implemented as one floating-point unit per 32 PEs. This means 2048 floating-points units on a 64K machine, giving a peak performance of 10 GFlops.

CM-2 is one of the most popular parallel computers for implementing ANN algorithms. Most of the implementations so far concern BP, but within this model different forms of parallelism have been utilized in different applications.

Rosenberg and Blelloch [108] constructed an algorithm in which each neuron is mapped onto one PE and each weight (synapse) is mapped onto two PEs. This unusual mapping is chosen in order to use some special communication features of the Connection Machine. Their mapping could be seen as a general method for mapping directed graphs onto a computer with "copy scan," "send," and "plus-scan" operations. The resulting performance, limited solely by communication, was 2.8 MCUPS for NETtalk and around 13 MCUPS maximally. Forward–backward parallelism was mentioned but not implemented.

Brown [12] compared two different ways of parallelizing BP. One method used node parallelism with one PE per neuron and the other was the method (node plus weight parallelism) suggested by Rosenberg and Blelloch. Brown found that the latter had better performance.

Zhang *et al.* [139] combined training example and node parallelism plus good knowledge of the CM-2 communication and computational structure to achieve high performance on quite different sizes of ANNs. On NETtalk using 64K PEs they could get around 40 MCUPS (and 175 MCPS).

Using Zhang *et al.*'s approach on a much larger problem (280.5 megabyte of training examples) Diegert [19] reached 9.3 MCUPS on a 16K PE machine. With 64K PEs the estimated performance is 31 MCUPS. This is good in comparison with the NETtalk performance above, when considering that the training data are moved in and out of the secondary storage all the time.

Singer's implementation of BP [118, 119], which is the fastest implementation on the CM-2, uses training example parallelism. He reports a maximum of 1300 MCPS and 325 MCUPS on a 64K PE machine when using 64K training vectors.

Deprit [18] has compared two implementations of recurrent back-propagation (RBP) on the CM-2. First he used the same mapping as Rosenberg and Blelloch (R&B) with a minor modification to include the feedback connections. The second mapping was node parallelism with the communication method suggested by Tomboulian [128]; see Section 6.4. The basic finding was that the R&B method was clearly superior for densely connected networks, such as that used in NETtalk. Note that the R&B implementation is still communication bound, indicated by the fact that the simulation time changed almost imperceptibly when software floating point was used instead of the hardware units.

Obermayer *et al.* have implemented large SOM models on the CM-2 [91]. Node parallelism with up to 16K PEs (neurons) was used. The input vector length (input space dimension) was varied and lengths of up to 900 were tested. The same algorithm was also implemented on a self-built computer with 60 T800 (Transputer) nodes connected in a systolic ring. In addition to algorithmic analysis the two architectures were benchmarked. The conclusion was that the CM-2 (16K PEs) with floating-point support is equal to 510 Transputer nodes for the shortcut version of SOM. As a 16K CM-2 has 512 Weitek FPUs, each with approximately the same floating-point performance as one T800, it can be concluded that the shortcut method is basically computation bound. In a "high-communication" variant of SOM, a 30-node Transputer machine would run at one-third of the CM-2 speed.

Rogers [105] has used CM-2 as a workbench for exploring Kanerva's SDM model. Rowwise mapping (node parallelism) is used for the selection phase (steps 1 and 2 in Algorithm 3), but for the store/retrieve phase weight parallelism is used (as many PEs as there are counters). As the number of physical PEs in Rogers' implementation is of the same order as the number of counters in one column he actually uses node parallelism and rowwise mapping, letting the CM-2 sequencer take care of the looping over each column. Implementing this in *Lisp on an 8K CM-2 results in a performance of only approximately 3 iterations per second (256-bit address, 256-bit data, 8192 locations). Using a pure rowwise mapping in C* one of the present authors has been able to achieve between 30 and 70 iterations per second on a CM-2 of this size. The difference is probably due to some unnecessary but expensive communication needed in going from 1D to 2D representation.

The still relatively poor performance of SDM on CM-2 is found to depend on at least three factors: PEs are underutilized during the select/retrieve phase using node parallelism where as few as 0.1–1% of the total number of PEs are active; the natural rowwise mapping demands time-consuming sum-reduction across PEs; the "optimal" mixed mapping [89] (see Section 9.2.5) is hard to implement efficiently with the current sequencer.

9.1.4. DAP

The Distributed Array Processor—produced by ICL (International Computers Limited) and AMT (Active Memory Technology Ltd.) [51]—is a series of 2D SIMD computers with different sizes and different hosts. The sizes range from 1024 to 4096 PEs. The processing elements are simple and bit-serial. To overcome the shortcomings of the ordinary 2D array (its long distance communication performance) row and column "highways" have been included. A 4K DAP gives 32-48 Mflops and computes 8-bit multiplications at 250 Mops and 8-bit multiplications by scalar constant at 600 to 1200 Mops. In a forthcoming machine there will be some support for multiplication in each PE. It will then give 560 MFlops maximally for a 4K PE machine.

Forrest *et al.* [29, 30] report the use of the ICL DAP to implement Hopfield net, BP, Elastic net (applied to traveling salesman problems), etc. They also use Transputers for similar tasks. However, no performance figures are given; it is more of a feasibility study. The authors describe and use four of the various types of parallelism: node, weight, training example, and training session.

Núñez and Fortes [90] have used the AMT DAP to implement BP, recurrent BP, and mean field theory. The calculations are treated as matrix operations, which with our terminology results in a combination of weight and node parallelism. For activation distribution the DAP broadcast highways are utilized in a way similar to the grid-based communication method. On a 4K machine with 8 bits used for weight and activation the performance is 100–160 MCUPS for BP. With 16 bits used instead, the figures are 25–40 MCUPS. A NETtalk (203 $\times$ 64 $\times$ 32) implementation on the 1K DAP using 8 bits resulted in 50 MCUPS.

9.1.5. MasPar (MP-1)

MasPar MP-1 [8, 15, 87] is a SIMD machine with both mesh and global interconnection style of communication. It has floating-point support, both VAX and IEEE standards. The number of processing elements can vary between 1024 and 16,384. Each PE has forty 32-bit registers, a 4-bit integer ALU, floating-point "units" to hold Mantissa and Exponent, an addressing unit for local address modifications, and a 4-bit broadcast bus.

MP-1 has a peak performance, for a 16K PE machine, of 1500 MFlops single-precision [87]. It is programmed in parallel versions of Fortran (MasPar Fortran) or C (MasPar C) or a C-derived language (MasPar Application Language) for more direct contact with the hardware [15].

Chinn *et al.* [14] and Grajski *et al.* [35, 36] have implemented BP and SOM using floating-point arithmetic. Estimating the performance of a 16K PE machine from the figures of a 2K, 4K, and 8K machine gives approximately 10 MCUPS with BP on a 256 × 128 × 256 network, utilizing node and weight parallelism. The mapping of SOM into MP-1 is one PE to each SOM node. It was measured to give 18 MCUPS on a 4K machine when 32-dimensional input vectors (or larger) were used.

9.2. Highly Parallel Machines with Simple PEs

9.2.1. AIS-5000, Centipede

AIS-5000 [113] manufactured by Applied Intelligent Systems Corp. has up to 1024 bit-serial PEs arranged as a linear array. The PEs are similar to those of CM, DAP, and BLITZEN. The basic problem when using AIS-5000 for ANN is the lack of support for multiplication and the difficulties in performing the backward phase of back-propagation (BP). Both of these difficulties have been addressed in [124] for other linear processor arrays. In a new generation using the new Centipede chip [135] better support for multiply-and-add is incorporated and table lookup is also possible.

AIS-5000 is intended mainly for image processing applications like inspection, but Wilson [135] has shown a neural network implementation of feedback type (Hopfield). Despite the difficulties mentioned, between 19 and 131 MCPS (depending on the precision used) is achieved using node parallelism.

9.2.2. Connected Network of Adaptive ProcessorS (CNAPS)

CNAPS, manufactured by Adaptive Solutions, Inc., is one of the first architectures developed especially for ANN. Called X1 in the first description by Hammerstrom [37], it is a 256-PE SIMD machine with a broadcast interconnection scheme. Each PE has a multiply (9 × 16 bit)-and-add arithmetic unit and a logic-shifter unit. It has 32 general (16-bit) registers and a 4-kbyte weight memory. There are 64 PEs in one chip. The user may choose 1, 8, or 16 bits for weight values and 8 or 16 bits for activation values.

With 16 bits for weights and 8 bits for activation the peak performance is 5 GCPS for a feedforward execution, and 1 GCUPS using BP learning. For NETtalk 180 MCUPS is achieved using only one chip (64 PEs).

The performance of CNAPS on SOM was reported by Hammerstrom and Nguyen [38]. The figures are based on a 20-MHz version of CNAPS. With 512 nodes/neurons and a 256-dimensional input vector, best match using a Euclidean distance measure can be carried out in 215 μs, using 16-bit weights. Making their CUPS figure comparable to others, the performance is about 183 MCUPS.

9.2.3. Geometric Arithmetic Parallel Processor (GAPP)

GAPP is a mesh-connected SIMD systolic array. On one chip there are 72 PEs (6 × 12), each with 128 bits of RAM. Designed for image processing and working bit-serially, it runs at a clock speed of 10 MHz. The processing element is very simple, basically a full adder. The chip was developed by Holsztynski of Martin Marietta Aerospace Corp. and manufactured by NCR Corp [16]. It was the first commercially available systolic chip.

Brown *et al.* have used GAPP to implement BP [11]. As there is no floating-point support on GAPP they use fixed-point numbers. Ten bits precision is used for the activation values and 15 bits (4 + 11) for the weights. The sigmoid function is approximated by a stepwise linear function. Both weight and node parallelism are used. No performance figures are reported.

Barash and Eshera [5] have also used GAPP to implement feedforward networks with BP learning. Using a variation of communication by circulation described in Section 6.3 they combine weight and node parallelism. With a 40K GAPP and 8 bits for weight and activation values they estimate a performance of 300 MCUPS. The major bottleneck is reported to be the calculation of the sigmoid function.

9.2.4. L-Neuro

The Laboratoires d'Electronique Philips (LEP), Paris, have designed a VLSI chip called L-Neuro. It contains 16 processors working in SIMD fashion. In association with these chips Transputers are imagined as control and communication processors. Weights are represented by two's-complement numbers over 8 or 16 bits, and the states of the neurons are coded over 1 to 8 bits. The multiplication is done in parallel over 16 or 32 weights but bit-serially over the weight values, and as only one output node at the time is calculated (in one chip) it can be considered a weight parallel implementation. The node activation value must go outside the chip for distribution to other nodes, and no support for the calculation of the sigmoid function is needed/implemented inside the chip. Duranton and Sirat [20, 21] have described implementations of SOM, Hopfield, and BP networks using this chip as a building block.

9.2.5. REMAP³

REMAP³ is a cooperative research project between Luleå University of Technology and Halmstad University, both in Sweden. The project is aimed at obtaining a massively parallel computer architecture put together by modules in a way that allows the architecture to be adjusted to a specific application. This suggests that a certain architecture may be "compiled"; thus a modification of each module and adjustments of the connections be-

tween the modules are enforced. The intended application area in a broad sense is embedded industrial systems. A multimodule system is well suited for implementing a multiple-network artificial neural system.

A small prototype of a software configurable processor array module with bit-serial processors has been implemented [75] and a larger system consisting of several modules is in the process of being designed. Different variations can be realized by reprogramming. An architecture tuned for neural network computations, including a fast bit-serial multiplier, has been designed. Åhlander and Svensson [140] describe how support for floating-point calculations may be embedded in the bit-serial working mode if needed, resulting in impressive floating-point performance when implemented with VLSI.

The mapping and performance of some common ANN models (BP, Hopfield, SOM, SDM) have been reported [124, 88, 89]. As an example, a 4K PE machine reaches 953 MCPS or 413 MCUPS on BP when running at 10 MHz and using 8-bit data. A node parallel version of BP is used. For 16-bit data, 546 MCPS or 219 MCUPS is achieved. An SDM model running 30 iterations per second on a 8K CM-2 can run at speeds above 400 iterations per second on a 256-PE REMAP^3 (10 MHz) with counters. The implementation uses a "mixed mapping" of the SDM model: rowwise mapping for the selection phase and columnwise for the store/retrieve phase.

9.3. Highly Parallel Machines with Complex PEs

9.3.1. GF11

GF11 is an experimental SIMD computer built at IBM Research Laboratory [7]. It has 566 PEs running at 20 MHz and a peak performance of 11 GFlops (as the name implies). It is intended primarily for quantum chromodynamics (QCD) predictions. Each PE has two floating-point adders, two floating-point multipliers, and one fixed-point unit. Table lookup and selection are the only data-dependent operations available.

The memory is organized as a three-staged hierarchy of progressively larger and slower memories. The communication is done via a three-staged Benes network. The machine is programmed in conventional C with calls to special-purpose procedures. This generates, after an optimization step, large blocks of microcode. Using a very simple controller the code is sent to all processors (there is also an address generator/relocator).

Witbrock and Zagha have been able to use this computer, before it was completed, to run ANN algorithms [136]. They implemented BP and benchmarked it with the NETtalk text-to-speech benchmark, achieving a speed of 900 MCUPS on a 356-processor computer. Because of the memory hierarchy they needed a few tricks to obtain this high speed.

Witbrock and Zagha discuss various ways to parallelize BP and finally choose training example parallelism. When all the weight changes are added, $\log_2 N$ steps are required. They also discuss in detail how to compute the sigmoid function and how to implement Recurrent BP because this model is their main interest. They conclude that a sophisticated communication network is not necessary if processor-dependent addressing is available.

9.3.2. Hughes Systolic/Cellular Architecture

The Hughes machine is a mesh-connected SIMD architecture made for image formation of synthetic aperture radar (SAR) pictures. The prototype used has 256 PEs, each with seven function units working on 32-bit fixed-point data (two multipliers, two adders, a divider, a normalizer, and a sorter), 24 memory registers, and a small local memory.

On this computer Shams and Przytula have implemented BP [115]. Training example parallelism was used in one "dimension" of the mesh and node parallelism with communication by circulation in the other dimension. Benchmarking with NETtalk resulted in a performance of 100 MCUPS, including loading and unloading operations. For recall only the result was 240 MCPS.

9.3.3. UCL Neurocomputer

Treleaven *et al.* have set out to construct a general-purpose "neurocomputer" [129]. Each PE is a 16-bit RISC (16 instructions) containing an on-chip communication unit and on-chip local memory. For communication a ring structure and a broadcast bus are used. Each PE has a unique address to support a more general type of logical communication. The ALU can add and subtract in one clock cycle but it only supports multiplication with a multiply step. This means that the performance of each PE will be low on ANN problems that have many multiplications.

The 5-MHz, 1.5-μm CMOS chip which was ready in 1990 could contain only two PEs and have a maximum CPS rate of 156 kCPS/PE or 312 kCPS/chip. It looks like the complex control part of the chip made it difficult to include a more powerful ALU with a one-cycle multiply-and-add. The UCL approach should be compared with the CNAPS approach in which emphasis is placed on the basic computation of ANNs and the controller is shared between all the PEs (SIMD).

9.3.4. Transputers

The Transputer [54, 133] is a single-chip 32-bit microprocessor. It has support for concurrent processing in hardware which closely corresponds to the Occam [10, 53, 55] programming model. It contains on-chip RAM and four bidirectional 20 Mbits/s communication links.

By wiring these links together a number of topologies can be realized. Each Transputer of the T800 type is capable of 1.5 MFlops (20 MHz) and architectures with up to 4000 Transputers are being built [133].

The next generation of Transputers, called T9000, will provide around 10 sustained MFlops and have 16 kbyte of cache memory on chip. The communication has also been improved to support through-routing without processor involvement. This will be an even better building block for highly parallel computers for the nineties.

Back-propagation has been implemented by Hwang *et al.* [52, 71] and Petkov [98] using node parallelism and communication by circulation. Forrest *et al.* have implemented the Hopfield, BP, and Elastic net models on both DAP and Transputers [29, 30]. No figures of performance have been given, but Transputers with their relatively low communication bandwidth (relative to their computational capabilities) are more efficiently used if small-grain partitioning can be avoided. That is, node and weight parallelism should be avoided, especially if they are described as general graphs.

The SOM of Kohonen has been implemented by Hodges *et al.* [45] and Siemon and Ultsch [117]. Both implementations just distribute the nodes over the PEs and use ring communication to distribute the input vector and to find the global minimum. As long as the neighborhood is larger than the number of PEs this mapping is quite efficient. Good performance will be achieved also for high input dimension. Siemon and Ultsch state a performance of 2.7 MCUPS on a 16 Transputer machine applied to a network sized 128 × 128 using 17 input dimensions. Hodges *et al.* presented an equation for the performance but no concrete implementation.

A more general implementation framework, called CARELIA, has been developed by Koikkalainen and Oja [66]. The neural network models are specified in a CSP-like formalism [64–66]. The simulator is currently running on a network of Transputers and some of the models implemented are SOM, BP, and perceptrons. The performance of the simulator has not been reported.

The European PYGMALION project [130] is, like CARELIA, a general ANN programming environment which has Transputers as one of its target architectures. Its ANN programming languages are based on C++ and C; this together with a graphical software environment and algorithm libraries makes it a complete ANN workbench. Performance figures or an indication of how to use massively parallel computers as targets are unfortunately not given.

9.4. Moderately Parallel Machines with Complex PEs

9.4.1. DSP (Digital Signal Processor) Based

Because the current generation of DSPs and some of the RISC chips have outstanding multiply-and-add performance it is easy to conceive of them as building blocks for an ANN computer. There are at least five suggested architectures using i860, TMS320C40, or TMS320C30.

9.4.1.1. Sandy/8. Building Sandy/8, Kato *et al.* at Fujitsu [61] intend to use conventional processors or signal processors for the calculations and simple ring structures for communication. The system is projected to utilize 256 TMS320C30. The expected maximal performance using BP is above 500 MCUPS. Only 118 MCUPS is expected on NETtalk (203 × 60 × 26) as the mapping can utilize only 60 PEs.

9.4.1.2. Ring Array Processor (RAP). The RAP is a multi-DSP system for layered network calculations developed at the International Computer Science Institute, Berkeley, California [6, 83]. Each PE consists of a TMS320C30 connected to other PEs via a bus interface into a ring network. The 4-PE computer (one board) which has been implemented runs at maximally 13.2 MCUPS [82, 83]. A 16-PE system is estimated to run at 46 MCUPS.

9.4.1.3. GigaCoNnection (GCN). Hiraiwa *et al.* [43] at Sony Corp. are building an ANN computer called GCN in which each PE consists of a processor similar to the core of i860, 2 FIFOs, and 4 Mbyte RAM. Each PE will fit into a single chip (by 1992). The PEs are connected into a 2D mesh with wraparound. The ANN algorithm is mapped on the architecture using node parallelism in one direction (systolic ring) and training example parallelism in the other. The expected BP performance when running 128 PEs at 50 MHz will be above 1 GCUPS for a 256 × 80 × 32 network. The training data are then distributed into 32 groups.

9.4.1.4. TOPSI. TOPSI is a computer architecture built at Texas Instruments [31]. Each PE or module can be said to consist of a TMS320C40 and a 32-bit general processor like MC68040 together with support for communication. There are two complementary communication structures, one general reconfigurable inter-PE network and one hierarchical bus for broadcasting information. Only the latter is needed for the implementation of Hopfield and BP networks. A 100-module computer will run BP at a maximum speed of approximately 150 MCUPS, and a 1000 module computer at approximately 1.4 GCUPS.

9.4.1.5. PLANNS (Planar Lattice Architecture for Neural Network Simulations) and TLA (Toroidal Lattice Architecture. PLANNS is an improved version of the TLA, both suggested by Fujimoto and Fukuda [32–34]. They use node and weight parallelism with grid-based communication. Load balancing is achieved by first mapping the computations onto a virtual TLA and then splitting the work to suit a physical TLA. The physical

processor array must be able to support mesh communications.

A 16-PE Transputer array has been used as a prototype TLA resulting in 2 MCUPS on a feedforward network using BP. The authors claim an almost linear speedup with the number of PEs when their load balancing scheme is used. By using a more powerful PE like i860 and a larger number of nodes (some 30,000) they are planning to reach 10 GCUPS in a future implementation.

9.4.2. Warp

Warp is a one-dimensional array of 10 or more powerful processing elements developed at Carnegie Mellon in 1984–1987 [69]. Each cell/PE has a microprogrammable controller, a 5-MFlops floating-point multiplier, a 5-MFlops floating-point adder, and a local memory. Communication between adjacent cells can be conducted in parallel over two independent channels: a left-to-right X channel and a bidirectional Y channel. In 1991 Intel released a single-chip version of the Warp concept called iWarp [97]. Systems with up to 1024 iWarp chips can be built and can give a theoretical performance of 20 GFlops. Implementing a computer with these chips will at least double the performance figures given for Warp below.

Back-propagation was implemented by Pomerleau *et al.* [100] trying both node and training example parallelism. They found that with training example parallelism they could simulate much larger networks and/or at higher speeds. On NETtalk the 10-PE Warp reached 17 MCUPS.

An implementation of Kohonen SOM has been described by Mann and Haykin [79]. Using training example parallelism between 6 and 12 MCUPS was achieved. Some minor problems with the topology ordering process when using training example parallelism were reported. The authors suggest that either the network start at some order instead of at random state or the network be trained sequentially for the first 100–1000 steps, after which the training example parallelism is "turned on."

9.5. Other High-Performance Architectures

9.5.1. Vector (Super) Computers

For the sake of comparison with well-known powerful computers of a more conventional kind, some figures from implementations on a couple of, so-called, supercomputers are given.

9.5.1.1. CRAY X-MP. The performance on a Cray X-MP was given in the DARPA neural networks study [17] to be 50 MCPS. It can be compared to the theoretical maximal performance of 210 MFlops [44]. Even though the 50 MCPS performance is often cited, it is difficult to draw any conclusions from this, as the network size, the training algorithm, and even whether or not training is included are unknown.

9.5.1.2. NEC SX-2 and SX-X. NEC's SX-2 is a conventional supercomputer with four general vector pipelines giving a peak performance of 1.3 GFlops [44]. On ANN its performance is 72 MCUPS on NETtalk and its maximal performance on BP is 180 MCUPS [4] via [61].

9.5.2. VLSI Implementations

Even though the intention of this paper primarily is to review more complete computers, there are a few borderline cases like L-Neuro, UCL neurocomputer, and CNAPS. There are many other interesting suggestions and realizations of chips for ANN. More material and surveys can, for instance, be found in [17, 46, 84, 103, 112]. To review only the digital ones would lead to another paper of this length. Here we mention a few of the digital realizations:

9.5.2.1. Faure. Faure and Mazare [25, 26] have suggested an asynchronous cellular architecture which consists of a 2D mesh of size 65 × 65 for ANN. Each PE has a routing and a processing part running at 20 MHz. The routing is intended to control up to four message transfers in parallel. Using 16 bits for weights and an array configured for NETtalk the designers claim 51.4 MCUPS. Basically, node parallelism is used but each node is distributed over two PEs.

9.5.2.2. Hitachi. Masaki *et al.* and Yasunaga *et al.* at Hitachi have developed a wafer scale integration (WSI) neural network [81, 137, 138]. On one 5-inch silicon wafer they can have 540 neurons, each having up to 64 weights (the 64 largest values are chosen). Node parallelism is used, and the neurons communicate through a time-shared digital bus. Each PE has an 8 by 9-bit multiplier and a 16-bit accumulator. The measured time step is 464 ns. The performance of a wafer is then 138 MCPS. They intend to build a system out of 8 wafers and could then theoretically achieve 1100 MCPS. No on-chip learning is available.

9.5.2.3. SIEMENS. Ramacher and his colleagues at Siemens [102–104] have suggested and built parts of a 2D array composed of systolic neural signal processor modules. The basic components of their MA16 chip are pipelined 16-bit multipliers together with adders. In this respect the design is similar to CNAPS. However, the Siemens architecture does not use broadcast-based communication, but instead uses a systolic data flow (partitioned in groups of four). Each chip has a throughput on the order of 500 MCPS. To get a complete system, 256 MA16 chips are concatenated, which should give a maximum performance of 128 GCPS. No estimated learning rates have been given.

10.0. DISCUSSION AND CONCLUSIONS

10.1. ANN and Parallel Computers

Practically every powerful, parallel computer will do well on ANN computations that use training example parallelism. It is the kind of computation that even brings the performance close to the peak performance. This is of course interesting for people who do research on training algorithms or do application development where the training of the system can be done in batch. However, for training in real time, this form of parallelism cannot be applied; the user is obliged to use node or weight parallelism instead. This places other demands on the architecture, resulting in lower performance figures.

This is clearly illustrated by the various BP implementations made on the Connection Machine: Singer, relying entirely on training example parallelism, achieves 325 MCUPS. Zhang *et al.*, only partly utilizing training example parallelism, reach 40 MCUPS. Rosenberg and Blelloch, finally, who only use other forms of parallelism, end up with a maximum performance of 13 MCUPS. The latter implementation is so heavily communication bound that it does not even matter if bit parallelism is utilized or not!

However, if the architecture meets the communication demands, near peak performance can be reached also in real-time training. The vertical and horizontal highways of the DAP architecture seem to be the key to the good performance reported for this machine [90]. On a computer where the maximum number of 8-bit multiply-and-add operations per second is 450–700M, 160 MCUPS (8 bit) implies a very little amount of communication overhead.

A major conclusion from this survey is that the regularity of ANN computations suits SIMD architectures perfectly; in none of the implementations studied has a real MIMD division of the computational task been required.

The majority of ANN computations following the most popular models of today can be mapped rather efficiently onto existing architectures. However, some of the models, for example, SDM, require a highly or massively parallel computer capable of performing tailored, but simple, operations in parallel and maintaining a very large amount of storage.

10.2. Communication

Broadcast and ring communication can be very efficiently utilized in ANN computations. In highly parallel machines, broadcast or ring communication alone has proved to be sufficient. For massively parallel machines it is difficult to use only broadcast without using training example parallelism. This is due to the fact that the number of nodes in each layer, or the number of inputs to each node, seldom is as large as the number of PEs in the machine. On a two-dimensional mesh machine, broadcast in one direction at a time may be used and node and weight parallelism may be combined. Thus, broadcast is an extremely useful communication facility also in these machines. The "highways" of the DAP architecture serve this purpose.

10.3. Bit-Serial Processing

Bit-serial processor arrays are very promising host machines for ANN computations. Linear arrays, or arrays with broadcast, are suitable for utilizing node parallelism. In mesh-connected arrays node and weight parallelism may be used simultaneously, if desired. Multiplication is the single most important operation in ANN computations. Therefore, there is much to gain in the bit-serial architectures if support for fast multiplication is added, as shown in Centipede and the REMAP^3 project.

As an illustration of this we can compare the performance figures for the implementations of BP on AAP-2 and REMAP^3, respectively. On the 64K PE AAP-2 machine, which lacks support for multiplication, 18 MCUPS using 26 bits data are reported on a network that suits the machine perfectly. The same performance can be achieved on a 512-PE linear array REMAP^3 implementation, in which bit-serial multipliers are used. AAP-2 also lacks a fast broadcasting facility, but this is of minor importance compared to the slow multiply operations.

10.4. Designing a General ANN Computer

A fruitful approach when designing a massively or highly parallel computer for general ANN computations is to start with a careful analysis of the requirements that are set by the low-level arithmetic operations and design processing elements which meet these demands. Then an architecture is chosen that makes it possible to map the computations on the computational structure in a way that makes processing and communication balanced.

It seems that broadcast communication often is a key to success in this respect, since it is a way to time-share communication paths efficiently. The approach has been used in both the CNAPS and the REMAP^3 design processes, both resulting in "only" highly (not massively) parallel modules with broadcast, the former with bit-parallel processors, the latter with bit-serial ones. Neither design utilizes all the available parallelism; instead they leave weight parallelism to be serialized on the same processor. Both reach near peak performance on a variety of algorithms.

10.5. Implementing Artificial Neural Systems

The real challenge for computer architects in connection with the neural network area in the future lies in the implementation of Artificial Neural Systems, i.e., sys-

tems composed of a large number of cooperating modules of neural networks. Each of the modules should be allowed to implement a different network structure, and the modules must be able to interact in different ways and at high speed. This implies that heterogeneous systems composed of homogeneous processing arrays must be developed, and that special attention must be paid to the problem of interaction between modules and between peripheral modules and the environment. The role of MIMD architectures in neural processing probably lies in this area, actually meaning that MIMSIMD (Multiple Instruction streams for Multiple SIMD arrays) architectures will be seen.

These new computer architectures are sometimes referred to as "sixth-generation computers," or "action-oriented systems" [2, 3], since they are capable of interacting with the environment using visual, auditory, or tactile sensors, and advanced motor units.

So far these matters have not been addressed by very many computer architects (nor by artificial neural network researchers). We believe that flexible, massively (or maybe only highly) parallel modules are important tools in experimental work aimed at building such systems for qualified real-time pattern recognition tasks.

ACKNOWLEDGMENT

This work has been partly financed by the Swedish National Board for Technical Development (NUTEK) under contract no. 900-1583.

REFERENCES

1. Almeida, L. D. Backpropagation in perceptrons with feedback. In *NATO ASI Series: Neural Computers*. Neuss, Federal Republic of Germany, 1987.
2. Arbib, M. A. *Metaphorical Brain 2: An Introduction to Schema Theory and Neural Networks*. Wiley–Interscience, New York, 1989.
3. Arbib, M. A. Schemas and neural network for sixth generation computing. *J. Parallel Distrib. Comput.* **6,** 2 (1989), 185–216.
4. Asogawa, M., Nishi M., and Seo, Y. Network learning on the supercomputer. In *Proc. 36th IPCJ Meeting,* 1988, pp. 2321–2322. [In Japanese]
5. Barash, S. C., and Eshera, M. A. The systolic array neurocomputer: Fine-grained parallelism at the synaptic level. In *First International Joint Conference on Neural Networks,* 1989.
6. Beck, J. The ring array processor (RAP): Hardware (Tech. Rep. 90-048, International Computer Science Institute, Berkeley, CA, 1990.
7. Beetem, J., Denneau, M., and Weingarten, D. The GF11 parallel computer. In Dongarra (Ed.). *Experimental Parallel Computing Architectures*. North-Holland, Amsterdam, 1987.
8. Blank, T. The MasPar MP-1 Architecture. In *Proc. COMPCON Spring 90,* San Francisco, CA, 1990, pp. 20–24.
9. Blevins, D. W., *et al.* Blitzen: A highly integrated massively parallel machine. *J. Parallel Distrib. Comput.* (1990), 150–160.
10. Bowler, K. C., *et al. An Introduction to OCCAM 2 Programming*. Studertlitteratur, Lund, Sweden, 1987.
11. Brown, J. R., Garber, M. M. and Vanable, S. F. Artificial neural network on a SIMD architecture. In *Proc. 2nd Symposium on the Frontiers of Massively Parallel Computation*. Fairfax, VA, 1988, pp. 43–47.
12. Brown, N. H., Jr. Neural network implementation approaches for the Connection machine. In *Neural Information Processing Systems,* Denver, CO, 1987, pp. 127–136.
13. Caviglia, D. D., Valle, M., and Bisio, G. M. Effects of weight discretization on the back propagation learning method: Algorithm design and hardware realization. In *International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 631–637.
14. Chinn, G., *et al.* Systolic array implementations of neural nets on the MasPar MP-1 massively parallel processor. In *International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 169–173.
15. Christy, P. Software to support massively parallel computing on the MasPar MP-1. In *Proc. COMPCON Spring 90,* San Fransisco, CA, 1990, pp. 29–33.
16. Cloud, E., and Holsztynski, W. Higher efficiency for parallel processors. In *Proc. IEEE SOUTHCON,* 1984.
17. DARPA. *Neural Network Study*. AFCEA. Fairfax, VA, 1988.
18. Deprit, E. Implementing recurrent back-propagation on the Connection Machine. *Neural Networks* **2,** 4 (1989), 295–314.
19. Diegert, C. Out-of-core backpropagation. In *International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 97–103.
20. Duranton, M., and Sirat, J. A. Learning on VLSI: A general purpose digital neurochip. In *International Conference on Neural Networks,* Washington, DC, 1989.
21. Duranton, M., and Sirat, J. A. Learning on VLSI: A general-purpose digital neurochip. *Philips J. Res.* **45,** 1 (1990), 1–17.
22. Fahlman, S. Benchmarks for ANN. Information and data sets of common benchmarks have been collected by Scott Fahlman and are available by internet ftp. The machine for anonymous ftp are "pt.cs.cmu.edu" (128.2.254.155) below the directory "/afs/cs/project/connect/bench". 1990.
23. Fahlman, S. E. An empirical study of learning speed in back-propagation networks. Rep. No. CMU-CS-88-162, Carnegie Mellon, 1988.
24. Fahlman, S. E., and Lebiere, C. The cascade-correlation learning architecture. In *Neural Information Processing Systems 2,* Denver, CO, 1989, pp. 524–532.
25. Faure, B., and Mazare, G. *Microprocessing and Microprogramming, A Cellular Architecture Dedicated to Neural Net Emulation*. Vol. 30, North-Holland, Amsterdam, 1990.
26. Faure, B., and Mazare, G. Implementation of back-propagation on a VLSI asynchronous cellular architecture. In *International Neural Networks Conference,* Paris, 1990, Vol. 2, pp. 631–634.
27. Fernström, C., Kruzela, I., and Svensson, B. *LUCAS Associative Array Processor—Design, Programming and Application Studies*. Springer-Verlag, Berlin, 1986, *Lecture Notes in Computer Science,* Vol. 216.
28. Flynn, M. J. Some computer organizations and their effectiveness. *IEEE Trans. Comput.* **C-21** (1972), 948–960.
29. Forrest, B. M., *et al.* Implementing neural network models on parallel computers. *Comput. J.* **30,** 5 (1987), 413–419.
30. Forrest, B. M., *et al.* Neural network models. In *International Conference on Vector and Parallel Processors in Computational Science III,* Liverpool, UK, 1988, Vol. 8, pp. 71–83.
31. Frazier, G. TeraOPs and TeraBytes for neural networks research. *Texas Instruments Tech. J.* **7,** 6 (1990), 22–33.
32. Fujimoto, Y. An enhanced parallel planar lattice architecture for

large scale neural network simulations. In *International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 581–586.

33. Fujimoto, Y., and Fukuda, N. An enhanced parallel planar lattice architecture for large scale neural network simulations. In *International Joint Conference on Neural Networks,* Washington, DC, 1989, Vol. 2, pp. 581–586.
34. Fukuda, N., Fujimoto, Y., and Akabane, T. A transputer implementation of toroidal lattice architecture for parallel neurocomputing. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 43–46.
35. Grajski, K. A. Neurocomputing using the MasPar MP-1. In Przytula and Prasanna (Eds.), *Digital Parallel Implementations of Neural Networks*. Prentice–Hall, Englewood Cliffs, NJ, 1992.
36. Grajski, K. A., *et al.* Neural network simulation on the MasPar MP-1 massively parallel processor. In *The International Neural Network Conference,* Paris, France, 1990.
37. Hammerstrom, D. A VLSI architecture for high-performance, low-cost, on-chip learning. In *International Joint Conference on Neural Networks,* San Diego, 1990, Vol. 2, pp. 537–543.
38. Hammerstrom, D., and Nguyen, N. An implementation of Kohonen's self-organizing map on the Adaptive Solutions neurocomputer. In *International Conference on Artificial Neural Networks,* Helsinki, Finland, 1991, Vol. 1, pp. 715–720.
39. Hecht-Nielsen, R. Theory of the backpropagation neural networks. In *International Joint Conference on Neural Networks,* Washington, DC, 1989, Vol. 1, pp. 593–605.
40. Hillis, W. D., and Steel, G. L. J. Data parallel algorithms. *Comm. ACM.* **29,** 12 (1986), 1170–1183.
41. Hinton, G. E., and Sejnowski, T. J. Optimal perceptual inference. In *Proc. IEEE Computer Society Conference on Computer Vision & Pattern Recognition,* Washington, DC, 1983, pp. 448–453.
42. Hinton, G. E., and Sejnowski, T. J. Learning and relearning in Boltzmann machines. In Rumelhart and McClelland (Eds.). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* Vol. 2, *Psychological and Biological Models.* MIT Press, Cambridge, MA, 1986.
43. Hiraiwa, A., *et al.* A two level pipeline RISC processor array for ANN. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 137–140.
44. Hockney, R. W., and Jesshope, C. R. *Parallel Computer 2.* Adam Hilger, Bristol, United Kingdom, 1988.
45. Hodges, R. E., Wu, C.-H., and Wang, C.-J. Parallelizing the self-organizing feature map on multi-processor systems. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 141–144.
46. Holler, M. A. VLSI implementations of neural computation models: A review. Internal Report, Intel Corp., Santa Clara, CA, 1991.
47. Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proc. Nat. Acad. Sci. USA* **79** (1982), 2554–2558.
48. Hopfield, J. J. Neurons with graded response have collective computational properties like those of two-state neurons. *Proc. Nat. Acad. Sci. USA* **81** (1984), 3088–3092.
49. Hopfield, J. J., and Tank, D. Computing with neural circuits: A model. *Science* **233** (1986), 624–633.
50. Hubel, D. H. *Eye, Brain and Vision.* Scientific American Library, New York, 1988.
51. Hunt, D. J. AMT DAP—A processor array in a workstation environment. *Comput. Systems Sci. Engrg.* **4,** 2 (1989), 107–114.
52. Hwang, J.-N., Vlontzos, J. A., and Kung, S.-Y. A systolic neural network architecture for hidden markov models. *IEEE Trans. Acoustics Speech Signal Process.* **37,** 12 (1989), 1967–1979.
53. INMOS Limited. *Occam Programming Model.* Prentice–Hall, New York, 1984.
54. INMOS Limited. The Transputer family 1987, 1987.
55. INMOS Limited. *Occam 2 Reference Manual.* Prentice–Hall, London, 1988.
56. Jaeckel, L. A. Some methods of encoding simple visual images for use with a sparse distributed memory, with applications to character recognition. Tech. Rep. 89.29, RIACS, NASA Ames Research Center, Moffet Field, CA, 1989.
57. Kanerva, P. Adjusting to variations in tempo in sequence recognition. *Neural Networks Suppl. INNS Abstracts* **1** (1988), 106.
58. Kanerva, P. *Sparse Distributed Memory.* MIT press, Cambridge, MA, 1988.
59. Kanerva, P. Personal communication, 1990.
60. Kassebaum, J., Tenorio, M. F., and Schaefers, C. The cocktail party problem: Speech/data signal separation comparison between backpropagation and SONN. In *Neural Information Processing Systems 2,* Denver, CO, 1989, pp. 542–549.
61. Kato, H., *et al.* A parallel neurocomputer architecture towards billion connection updates per second. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 47–50.
62. Kohonen, T. *Self-Organization and Associative Memory.* Springer-Verlag. Berlin, 1988, 2nd ed.
63. Kohonen, T. The self-organizing map. Proc. IEEE **78,** 9 (1990), 1464–1480.
64. Koikkalainen, P. MIND: A specification formalism for neural networks. In *International Conference on Artificial Neural Networks,* Helsinki, Finland, 1991, Vol. 1, pp. 579–584.
65. Koikkalainen, P., and Oja, E. Specification and implementation environment for neural networks using communication sequential processes. In *International Conference on Neural Networks,* San Diego, CA, 1988.
66. Koikkalainen, P., and Oja, E. The CARELIA simulator: A development and specification environment for neural networks. Res. Rep. 15/1989, Lappeenranta University of Tech, Finland, 1989.
67. Krikelis, A., and Grözinger, M. Implementing neural networks with the associative string processor. In *International Workshop for Artificial Intelligence and Neural Networks,* Oxford, 1990.
68. Kung, H. T. Why systolic architectures? *IEEE Comput.* (Jan. 1982), 37–46.
69. Kung, H. T. The Warp computer: Architecture, implementation and performance. *IEEE Trans. Comput.* (Dec. 1987).
70. Kung, S. Y. Parallel architectures for artificial neural nets. In *International Conference on Systolic Arrays,* San Diego, CA, 1988, pp. 163–174.
71. Kung, S. Y., and Hwang, J. N. Parallel architectures for artificial neural nets. In *International Conference on Neural Networks,* San Diego, CA, 1988, Vol. 2, pp. 165–172.
72. Kung, S. Y., and Hwang, J. N. A unified systolic architecture for artificial neural networks. *J. Parallel Distrib. Comput.* (Apr. 1989).
73. Lawson, J.-C., Chams, A., and Herault, J. SMART: How to simulate huge networks. In *International Neural Network Conference,* Paris, France, 1990, Vol. 2, pp. 577–580.
74. Lea, R. M. ASP: A cost effective parallel microcomputer. *IEEE Micro.* (Oct. 1988), 10–29.
75. Linde, A., and Taveniku, M. LUPUS—A reconfigurable prototype for a modular massively parallel SIMD computing system."

Masters Thesis Rep. 1991:028E, University of Luleå, Sweden, 1991. [In Swedish]

76. Lippmann, R. P. An introduction to computing with neural nets. *IEEE Acoustics Speech Signal Process. Mag.* **4** (Apr. 1987), 4–22.
77. MacLennan, B. J. Continuous computation: Taking massive parallelism seriously. In *Los Alamos National Laboratory Center for Nonlinear Studies 9th Annual International Conference, Emergent Computation,* Los Alamos, NM, 1989. [Poster presentation]
78. Makram-Ebeid, S., Sirat, J. A., and Viala, J. R. A rationalized error back-propagation learning algorithm. In *International Joint Conference on Neural Networks,* Washington, DC, 1989.
79. Mann, R., and Haykin, S. A parallel implementation of Kohonen feature maps on the Warp systolic Computer. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 84–87.
80. Marchesi, M., *et al.* Multi-layer perceptrons with discrete weights. In *International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 623–630.
81. Masaki, A., Hirai, Y., and Yamada, M. Neural networks in CMOS: A case study. *Circuits and Devices* (July 1990), 12–17.
82. Morgan, N. The ring array processor (RAP): Algorithms and architecture. Tech. Rep. 90-047), International Computer Science Institute, Berkeley, CA, 1990.
83. Morgan, N., *et al.* The RAP: A ring array processor for layered network calculations. In *Proc. Conference on Application Specific Array Processors,* Princeton, NJ, 1990, pp. 296–308.
84. Murray, A. F. Silicon implementation of neural networks. *IEE Proc. F.* **138,** 1 (1991), 3–12.
85. Murray, A. F., Smith, A. V. W., and Butler, Z. F. Bit-serial neural networks. In *Neural Information Processing Systems,* Denver, CO 1987, pp. 573–583.
86. Nakayama, K., Inomata, S., and Takeuchi, Y. A digital multilayer neural network with limited binary expressions. In *International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 587–592.
87. Nickolls, J. R. The design of the MasPar MP-1: A cost effective massively parallel computer. In *Proc. COMPCON Spring 90,* San Fransisco, CA, 1990, pp. 25–28.
88. Nordström, T. Designing parallel computers for self organizing maps. Res. Rep. TULEA 1991:17), Luleå University of Technology, Sweden, 1991.
89. Nordström, T. Sparse distributed memory simulation on REMAP3. Res. Rep. TULEA 1991:16, Luleå University of Technology, Sweden, 1991.
90. Núñez, F. J., and Fortes, J. A. B. Performance of connectionist learning algorithms on 2-D SIMD processor arrays. In *Neural Information Processing Systems 2,* Denver, CO, 1989, pp. 810–817.
91. Obermayer, K., Ritter, H., and Schulten, K. Large-scale simulations of self-organizing neural networks on parallel computers: Application to biological modelling. *Parallel Comput.* **14,** 3 (1990), 381–404.
92. Parker, K. L. Parallelized back-propagation training and its effectiveness. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 179–182.
93. Penz, P. A. The closeness code: An integer to binary vector transformation suitable for neural network algorithms." In *International Conference on Neural Networks,* San Diego, CA, 1987, Vol. 3, pp. 515–522.
94. Pesulima, E. E., Panadya, A. S., and Shankar, R. Digital implementation issues of stochastic neural networks. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 187–190.
95. Peterson, C., and Andersson, J. A mean field theory learning algorithm for neural networks. *Complex Systems* **1** (1987), 995–1019.
96. Peterson, C., and Hartman, E. Explorations of mean field theory learning algorithm. *Neural Networks* **2,** 6 (1989), 475–494.
97. Peterson, C., Sutton, J., and Wiley, P. iWarp: A 100-MOPS, LIW microprocessor for multicomputers. *IEEE Micro.* (June 1991), 26–29, 81–87.
98. Petkov, N. Systolic simulation of multilayer, feedforward neural networks. *Parallel Process. Neural Systems Comput.* (1990), 303–306.
99. Pineda, F. J. Generalization of back-propagation to recurrent neural networks. *Phys. Rev. Lett.* **59,** 19 (1987), 2229–2232.
100. Pomerleau, D. A., *et al.* Neural network simulation at warp speed: How we got 17 million connections per second. In *Proc. IEEE International Conference on Neural Networks,* San Diego, CA, 1988.
101. Potter, J. L. *The Massively Parallel Processor.* MIT Press. Cambridge, MA, 1985.
102. Ramacher, U., and Beichter, J. Architecture of a systolic neuroemulator. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 59–63.
103. Ramacher, U., *et al.* Design of a 1st generation neurocomputer. In Ramacher, U., & Rückert, U. (Eds.), *VLSI Design of Neural Networks,* Kluwer-Academic Publishers, Dordrecht, The Netherlands, 1991.
104. Ramacher, U., and Wesseling, M. Systolic synthesis of neural networks. In *International Neural Network Conference,* Paris, France, 1990, Vol. 2, pp. 572–576.
105. Rogers, D. Kanerva's sparse distributed memory: An associative memory algorithm well-suited to the connection machine. In *Proc. Conference on Scientific Application of the Connection Machine,* Moffet Field, CA, 1988, Vol. 1, pp. 282–298.
106. Rogers, D. Kanerva's sparse distributed memory: An associative memory algorithm well-suited to the connection machine. Tech. Rep. 88.32, RIACS, NASA Ames Research Center, 1988.
107. Rogers, D. Statistical prediction with Kanerva's sparse distributed memory. In *Neural Information Processing Systems 1,* Denver, CO, 1988, pp. 586–593.
108. Rosenberg, C. R., and Blelloch, G. An implementation of network learning on the connection machine. In *Proc. 10th International Conference on AI,* Milan, Italy, 1987, pp. 329–340.
109. Rosenblatt, F. *Principles of Neurodynamics.* Spartan Books, New York, 1959.
110. Rumelhart, D. E., and McClelland, J. L. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* MIT Press, Cambridge, MA, 1986, Vols. I and II.
111. Rumelhart, D. E., and McClelland, J. L. *Explorations in Parallel Distributed Processing.* MIT Press, Cambridge, MA, 1988.
112. Sami, M., and Calzadilla-Daguerre, J. *Silicon Architectures for Neural Nets.* North-Holland, Amsterdam, 1991.
113. Schmitt, R. S., and Wilson, S. S. The AIS-5000 parallel processor. *IEEE Trans. Pattern Anal. Mach. Intell.* **10,** 3 (1988), 320–330.
114. Sejnowski, T. J., and Rosenberg, C. R. Parallel networks that learn to pronounce English. *Complex Systems* **1** (1987) 145–168.
115. Shams, S., and Przytula, K. W. Mapping of neural networks onto programmable parallel machines. In *Proc. IEEE International Symposium on Circuits and Systems,* New Orleans, LA, 1990.
116. Shoemaker, P. A., Carlin, M. J., and Shimabukuro, R. L. Backpropagation with coarse quantization of weight updates. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 1, pp. 573–576.
117. Siemon, H. P., and Ultsch, A. Kohonen networks on transputers:

Implementation and animation. In *International Neural Network Conference,* Paris, France, 1990, Vol. 2, pp. 643–646.

118. Singer, A. Exploiting the Inherent parallelism of artificial neural networks to achieve 1300 million interconnects per second. In *International Neural Network Conference,* Paris, France, 1990, Vol. 2, pp. 656–660.
119. Singer, A. Implementation of artificial neural networks on the Connection Machine Tech. Rep. RL90-2, Thinking Machine, Corp., 1990.
120. Singer, A. Implementations of artificial neural networks on the Connection Machine. *Parallel Comput.* **14,** 3 (1990), 305–316.
121. Sirat, J. A., *et al.* Unlimited accuracy in layered networks. In *First IEE Conference on Artificial Neural Networks,* London, 1989, pp. 181–185.
122. Stevenson, M., Winter, R., and Widrow, B. Sensitivity of layered neural networks to errors in the weights. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 1, pp. 337–340.
123. Stevensson, M., Winter, R., and Widrow, B. Sensitivity of feed-forward neural networks to weight errors. *IEEE Trans. Neural Networks* **1,** 1 (1990), 71–80.
124. Svensson, B., and Nordström, T. Execution of neural network algorithms on an array of bit-serial processors. In *10th International Conference on Pattern Recognition, Computer Architectures for Vision and Pattern Recognition,* Atlantic City, NJ, 1990, Vol. II, pp. 501–505.
125. Tenorio, M. F., and Lee, W.-T. Self-organizing network for optimum supervised learning. *IEEE Trans. Neural Networks* **1** (1990).
126. Tenorio, M. F. d. M. Topology synthesis networks: Self-organization of structure and weight adjustment as a learning paradigm. *Parallel Comput.* **14,** 3 (1990), 363–380.
127. Thinking Machines Corporation. Connection Machine, Model CM-2 Technical Summary. Version 5.1, TMC, Cambridge, MA, 1989.
128. Tomboulian, S. Introduction to a system for implementing neural net connections on SIMD architectures. In *Neural Information Processing Systems,* pp. 804–813, Denver, CO, 1987, pp. 804–813.
129. Treleaven, P., Pacheco, M., and Vellasco, M. VLSI architectures for neural networks. *IEEE Micro.* (Dec. 1989), 8–27.
130. Treleaven, P. C. PYGMALION neural network programming environment. In *International Conference on Artificial Neural Networks,* Helsinki, Finland, 1991, Vol. 1, pp. 569–578.
131. von Neumann, J. *The Computer and the Brain.* Yale Univ. Press, New Haven, CT, 1958.
132. Watanabe, T., *et al.* Neural network simulation on a massively parallel cellular array processor: AAP-2. In *International Joint Conference on Neural Networks,* Washington, DC, 1989, Vol. 2, pp. 155–161.
133. Whitby-Stevens, C. Transputers—Past, present, and future.'' *IEEE Micro.* (Dec. 1990), 16–82.
134. Willshaw, D. J., Buneman, O. P., and Longuet-Higgins, H. C. Non-holographic associative memory. *Nature* **222** (1969), 960–962.
135. Wilson, S. S. Neural computing on a one dimensional SIMD array. In *11th International Joint Conference on Artificial Intelligence,* Detroit, MI, 1989, pp. 206–211.
136. Witbrock, M., and Zagha, M. An implementation of back-propagation learning on GF11, a large SIMD parallel computer. Rep. No. CMU-CS-89-208, Computer Science, Carnegie Mellon, 1989.
137. Yasunaga, M., *et al.* A wafer scale integration neural network utilizing completely digital circuits. In *International Joint Conference on Neural Networks,* Washington, DC, 1989, Vol. 2, pp. 213–217.
138. Yasunaga, M., *et al.* Design, fabrication and evaluation of a 5-inch wafer scale neural network LSI composed of 576 digital neurons. In *Proc. International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 527–535.
139. Zhang, X., *et al.* An efficient implementation of the backpropagation algorithm on the Connection Machine CM-2. In *Neural Information Processing Systems 2,* Denver, CO, 1989, pp. 801–809.
140. Åhlander, A. and Svensson, B. Floating-point calculations in bit-serial *SIMD* computers. Res. Rep. CDv-9104, Halmstad University, Sweden, 1991.

Received August 19, 1991; revised October 1, 1991; accepted October 18, 1991

TOMAS NORDSTRÖM was born May 19, 1963, in Härnösand, Sweden. He holds an M.S. degree in computer science and engineering from Luleå University of Technology, Sweden. Since 1988 he has been a Ph.D. student at the same University, and since 1991 he has held a Licentiate degree in computer engineering. His research interests include parallel architectures and artificial neural networks.

BERTIL SVENSSON was born February 15, 1948, in Eldsberga, Sweden. He received his M.S. degree in electrical engineering in 1970 and his Ph.D. degree in computer engineering in 1983, both from the University of Lund, Sweden, where he also served as a researcher and assistant professor. In 1983 he joined Halmstad University College, Sweden, as assistant professor and became vice president the same year. Until recently he was a professor in computer engineering at Luleå University of Technology, Sweden, which he joined in 1989. Presently he is a professor in computer engineering at Chalmers University of Technology, Sweden. He is also head of the Centre for Computer Science, a research institution of Halmstad University. He has been working in the field of highly parallel SIMD computers and their applications, e.g., image processing, since 1978. Currently he is conducting research on the design of modular, reconfigurable, massively parallel computers for trainable, embedded systems.