Svensson, B. and T. Nordström, "Execution of neural network algorithms on an array of bit-serial processors," in 10*th International Conference on Pattern Recognition, Computer Architectures for Vision and Pattern Recognition,* Atlantic City, NJ, USA, 1990, vol. II, pp. 501-505, ISBN 0-8186-2063-3.

EXECUTION OF NEURAL NETWORK ALGORITHMS ON AN ARRAY OF BIT-SERIAL P ROCESSORS

B. Svensson T. Nordström

Division of Computer Engineering
Department of Systems Engineering and Mathematics
Luleå University of Technology, Sweden

E-mail: bertil@sm.luth.se or tono@sm.luth.se

ABSTRACT

Large processor arrays are candidates for performing computations of neural network models at speeds required for real time applications, e.g. in pattern recognition. The paper gives a general model of an array of bit-serial processors and demonstrates the mapping of neural net models on such an array.

The approach maps a neuron on each processing element and makes communication all-to-all available by connection weight matrices. The required communication structure is very simple.

The bit-serial approach allows trade-offs between speed and precision, even dynamically. Performance figures are given. A bit-serial multiplier is an important part of the design. Implementation aspects are discussed and it is shown that a one-board realization of a 1024 processor system is feasible with current, commonly available, technology.

INTRODUCTION

Recent years have seen an enormous increase of interest in neural networks. It has been realized that massive parallelism is required for human-like performance in pattern recognition. Neural networks provide one technique to do this. Processor arrays are candidates for performing the computations efficiently. The subject of this paper is to study the mapping of neural network computations on a regular array of a large number of simple processors. The computations are uniform and arithmetically simple. This suggests that simple processing elements are sufficient and that the SIMD type of architecture is appropriate. The number of interconnections in a neural network is often orders of magnitude greater than the number of processing units. This suggests that connectivity be stored partly in matrices.

We study the mapping of both feedforward (with back-propagation) and feedback neural nets. The characteristics of these models will be briefly outlined. Before that we introduce a generic architecture for a bit-serial array processor (BAP). We describe the algorithms in a parallel language (Pascal/L) which includes constructs directly implementable as elementary operations of a BAP. The computations are analysed, performance figures are given, and system implementation is discussed.

BIT-SERIAL ARRAY PROCESSORS

A Bit-serial Array Processor (BAP) is characterized by the following properties:

- It is organized as an SIMD processor, i.e. it consists of many processing elements (PEs) and one common control unit.
- The PEs treat data bit-serially and the data paths to and from each PE are only one bit wide.
- Activation of the PEs may be data driven ("associative process"), which means it is not the location or address of a PE that decides its action on an instruction from the control unit, but some property of the data in the memory or registers of that PE.
- An interconnection network defines a topological relationship between PEs.

A commonly used organization of the BAP is to place the interconnection network between the memory part and the logic part of the PE as shown in Figure 1. This does not mean that memory and logic are physically apart - on the contrary, they are seen as a whole and should preferably be put on the same chip.

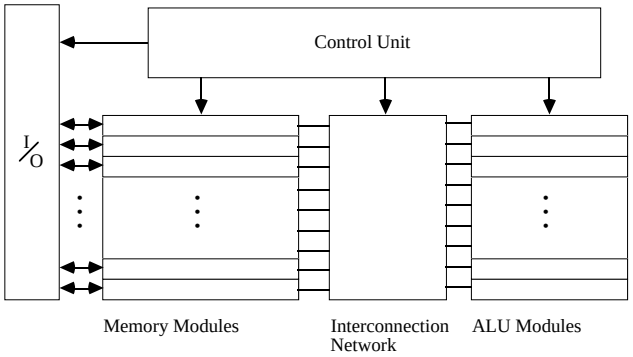

Figure 1. Organization of a Bit-serial Array Processor.

A BAP is defined by the characteristics of five parts: data storage, processing, data alignment, input/output, and control.

Data storage is organized as Memory Modules (MMs). One bit from each MM is accessible at a time. A number of such *bit-slices*, normally consecutive, form a *field*.

The *Processing part* is an ensemble of Arithmetic and Logic Units (ALUs) which implement functions on a set of one-bit arguments. The complexity of the ALU may vary from boolean functions of two variables through bit-serial multipliers to full bit-serial floating point units.

One of the registers in the ALU is the one-bit *Activity Register*, the contents of which determines whether or not the ALU takes part in the specified operation. To choose only one ALU for activity, a *select first* facility is included. In more elaborate models multi-bit registers may be used to determine one out of a set of actions to be performed.

The *Data Alignment part* consists of an interconnection structure that allows each ALU to receive data also from "neighbouring" modules. Common structures are the square grid, the linear array, the n-cube and the shuffle-exchange. Many separate structures may be implemented on the same processor.

The structure of the *Input/Output part* design is strongly dependent on the demands of the application and may be varied in several ways. For example, in some cases a direct bit-slice wide interface to the data source may be motivated.

CH2898-5/90/0000/0501$01.00 © 1990 IEEE

The total activity is mastered by the *Control Unit*, which takes instructions from an ordinary sequential processor. The most obvious task for the Control Unit is to translate operations on data items (e.g. vectors and matrices) to sequences of bit operations. This should be performed without any overhead.

Based on the types of operands and results six basic types of instructions to manipulate data in the array can be identified:

Instruction type	*Example*
field —> field	Increment field, Permute field
field —> selector	Max/min value of a field
field,field —> field	Multiply fields, Pairwise max
field,field —> selector	Pairwise equality
constant,field —> field	Multiply by constant
constant,field —> selector	Closest match, Greater than

Multiplication is a frequent operation in many application areas. Using ALUs with a complexity comparable to a full adder only, the multiplication time grows quadratically with the data length. Ohlsson [1] suggested a bit-serial multiplier in each ALU, giving a multiplication time that is no longer than the time required to read the operands and store the result.

Figure 2 shows the design for multiplication of two 2's complement integers using a series of full adders (FA). The multiplicand is first shifted in, most significant bit first, into the array of M flip-flops. The multiplier is then applied to the input, least significant bit first, and the product bits appear at the output, least significant bit first. The S flip-flops store the accumulated sum. A more detailed description is given in [1] and [2].

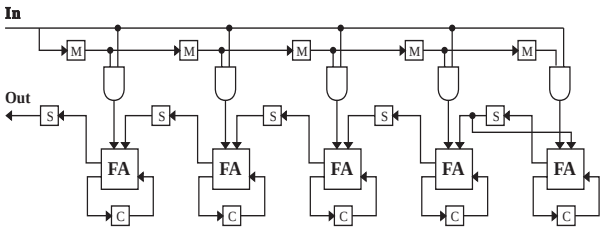

Figure 2. A 5 bit wide bit-serial multiplier using carry-save technique. M, S and C are flip-flops and FA are full adders.

NEURAL NETWORK ALGORITHMS

Several neural net models have been proposed. They are characterized by network topology, node characteristics, and training rules. Frequently used and discussed models are the *multilayer feedforward networks* with supervised learning by error back-propagation [3] and the *feedback networks* , either with symmetric connectivity and stochastic nodes (Boltzmann machines [4, 5]), symmetric connectivity and deterministic nodes (Hopfield net [6, 7, 8]), or nonsymmetric connectivity and deterministic nodes[9, 10].

In order to be as general as possible in the implementation studies we use a feedback algorithm without any assumption on symmetry of the weight matrix. Thus, for the Hopfield model and the Boltzmann machine shorter execution times than those reported below can be expected (both use symmetric matrices).

The back-propagation model is used as a pattern classifier or feature detector. The feedback models are used as auto-associative memories for tasks like pattern completion.

Feedforward networks with error back-propagation

A feedforward net (ff net) with four layers is shown in Figure 3. Each node (neuron) in a layer receives input from every node in the previous layer. 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 of the neuron. A sigmoid function, with a smooth threshold like curve, is the most frequently used activation function in feedforward networks.

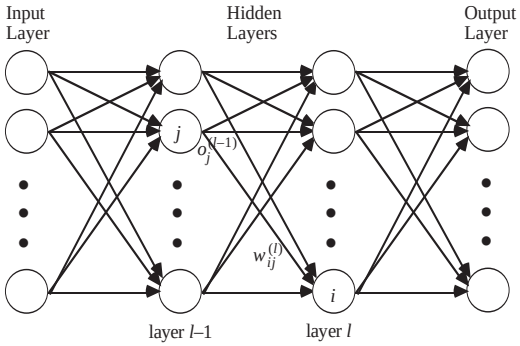

Figure 3. A four-layer feedforward network.

The back-propagation algorithm (also known as the generalized delta rule) [3] is used to train the network in our examples.

In the first phase the input to the network is provided and values propagate through the network to compute the output vector *O*. *O* is then compared with a target vector *T* provided by a teacher, resulting in an error vector *E*. In the second phase the values of the error vector are propagated back. The error signals for hidden units are thereby determined recursively: Error values for layer l are determined from a weighted sum of the values of layer l+1, again using the connection weights – now "backwards". The weighted sum is multiplied by the derivative of the activation function to give the error value δ.

Finally, appropriate changes of weights and thresholds are made. The weight change 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 and the error value δ: $\Delta w_{ij}^{(l)} = \eta \delta_i^{(l)} o_j^{(l-1)}$. The bias (threshold) value may be seen as the weight from a unit that is always on. The algorithm is summarized below.

1. Apply input
2. Compute output $o_j^{(l)} = f\left(net_j^{(l)} + b_j^{(l)}\right)$

   where $net_j^{(l)} = \sum_i w_{ij}^{(l)} o_i^{(l-1)}$ for each layer.
3. Determine error vector $E = T - O$
4. Propagate error backwards.

   If node j is an output node then the elements of the error value vector D are

   $\delta_j^{(l)} = o_j^{(l)}(1 - o_j^{(l)})(t_j^{(l)} - o_j^{(l)}) = o_j^{(l)}(1 - o_j^{(l)})e_j^{(l)}$

   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 $f(x) = \frac{1}{1+e^{-x}}$ 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.

Algorithm 1. Back-propagation training algorithm

Feedback networks

A feedback network has a single set of completely interconnected nodes, see Figure 4. All nodes are both input and output nodes. Each node computes a weighted sum of all its inputs and applies a nonlinear activation function to the sum. The resulting value is treated as input to the network in the next 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.

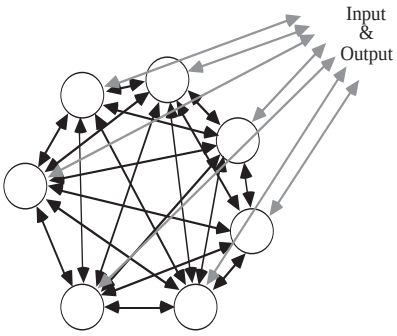

Figure 4. A seven node feedback network.

Training or learning can be done in supervised mode with the delta rule [11] or back propagation [10], or unsupervised by a Hebbian rule [11]. The delta rule is more powerful than the Hebb rule, and more commonly used than back-propagation (for feedback nets). We only analyse the delta rule algorithm.

In the first phase of training the pattern is imposed on the net at time zero by forcing the output from the net to match the pattern. Following this initiation, the net iterates in discrete time steps using the given formula. When the net has converged the activation a_i is compared to the target t_i and the error is calculated as $e_i = t_i - a_i$. The weights are changed in proportion to the product of the activation a_j and the error e_i, i.e. $\Delta w_{ij} = \eta e_i a_j$

1. Set activation values to external input values
2. Calculate new activation values $a_j = f(net_j + b_j)$ where $net_j = \sum_i w_{ij} a_i$, until the network is stable
3. Determine error vector $E = T - A$
4. Adjust the weights $\Delta w_{ij} = \eta e_i a_j$ and bias $\Delta b_i = \eta e_i$.
5. Repeat from 1.

Algorithm 2. Delta learning algorithm for feedback networks

MAPPING NEURAL NETWORKS ON A BAP

It should be clear from the above descriptions of networks, that the computations of both a feedforward network with error back-propagation and a feedback network involve mainly matrix-by-vector multiplications, where the matrices contain the connection weights and the vectors contain activation values or error values. Such a 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 PEs and unit time, and then form the sums by using trees of adders. The addition requires $N(N\text{-}1)$ adders and $\boldsymbol{O}(\log N)$ time. This is, however, an unrealistic method depending on both the number of PEs required and the communication problems caused. Instead we take the approach of having as many PEs as neurons in a layer, N, and storing the connection weights in matrices, sized N by N, one for each layer. The PE with index j has access to row j of the matrix by accessing its own memory word.

Feedforward net with error back-propagation

Figure 5 shows the data storage for each layer. In the forward pass the *net* vector is computed by N successive, parallel multiply-and-add operations, each requiring access to a different output value from the previous layer. Thus the PEs must, one after the other, broadcast their output values to all PEs. The O-vector is computed by a parallel application of the activation function.

$$\begin{array}{cccc} \mathbf{W}^{(l)} & & B^{(l)} \; O^{(l)} \; net^{(l)} \; E^{(l)} & \\ w_{00}^{(l)} \; w_{01}^{(l)} \cdots w_{0,N-1}^{(l)} & & b_0^{(l)} \; o_0^{(l)} \; net_0^{(l)} \; e_0^{(l)} & \Longleftrightarrow \mathrm{PE}_0 \\ w_{10}^{(l)} \; w_{11}^{(l)} \quad\; w_{1,N-1}^{(l)} & & b_1^{(l)} \; o_1^{(l)} \; net_1^{(l)} \; e_1^{(l)} & \Longleftrightarrow \mathrm{PE}_1 \\ \vdots \qquad \ddots \quad \vdots & & \vdots \quad \vdots \quad \vdots \quad \vdots & \vdots \\ w_{N-1,0}^{(l)} \; w_{N-1,1}^{(l)} \cdots w_{N-1,N-1}^{(l)} & & b_{N-1}^{(l)} \; o_{N-1}^{(l)} \; net_{N-1}^{(l)} \; e_{N-1}^{(l)} & \Longleftrightarrow \mathrm{PE}_{N-1} \end{array}$$

Figure 5. Data storage for layer l.

In the backward pass the computation of the error vector for each layer requires vertical addition. We suggest a bit-serial adder tree. The addition of each column can be overlapped with the multiplications for the next. On completion of this phase, weight changes are calculated. The O-vector of layer l–1 is first multiplied by a constant, η. Then, for each j, the j:th value of the result is broadcast to all other PEs, the E-vector is multiplied by this value and the result is added to the j:th column of the weight matrix. The threshold vector is changed in a similar way.

The Pascal/L language

Pascal/L is an extension of Pascal for parallel processing, developed in the LUCAS project [2]. In Pascal/L the parallelism of the architecture has a correspondence in the syntax of the language. Thus, constructs in the language are directly implementable as elementary operations of a BAP.

A **selector** defines a boolean vector over the MMs and is used to control the parallelism of operations. A **parallel array** has a fixed number of components, all of the same type and located in the MMs. An indexing scheme allows simultaneous access to a column or a subset of the column components of a two-dimensional array. For example: **W**[*,5] selects column 5 of **W**, **W**[SEL,5] selects a subset of column 5. A parallel array may be used without any index at all (and no brackets), in which case all components of the array are referenced.

To support data-driven processing a number of standard functions and procedures can be applied to selectors. The **first** function finds the first component of a selector with the value *true* and returns a new selector with only this element true. The **next**-procedure assigns *false* to the first true element of the selector. This is useful when elements are to be processed sequentially. The **some** function returns *true* if there is at least one true element of the selector, otherwise it returns *false*.

The Pascal/L program

For the feedforward net, the following declarations are needed (For simplicity of notation we consider only one layer):

```
var  W  : parallel array [0..N–1,0..N–1] of integer(b);
     net : parallel array [0..N–1] of integer(b + logN);
     B,O,O⁻,E,E⁺:parallel array[0..N–1] of integer(b);
{ O⁻ is the output of the previous layer, E⁺ is the error of the
following layer }
     sel : selector [0..N–1];
     j   : integer;
```

The following program parts implement the algorithm:

```
Forward pass:
      j := 0; net := 0; sel := TRUE;
      while some(sel) do begin
          net := net + W[*,j] * O-[first(sel)]; next(sel); j := j+1;
      end;
      O := f(net+B);
Backward pass:
 Computation of error vectors:
      j := 0; E := 0; sel := TRUE;
      while some(sel) do begin
          E:= E + E+[first(sel)] * W[*,j]; next(sel); j:=j+1;
      end;
      E := O*(1–O)*E;
Computation of new weights:
      j := 0; sel := TRUE;
      while some(sel) do begin
          W[*,j] := W[*,j] + η*O-[first(sel)]*E; next(sel); j:=j+1;
      end;
      B := B + η*E;
```

General feedback algorithm

The data storage required for a general feedback network is approximately the same as for one layer of the ff network. The computations of the forward pass are the same as in the ff net (one layer), but are repeated until no more changes occur, or for a fixed number of times. To update the weights, the error vector is first calculated in parallel and then multiplied by η, also in parallel. The values of the activation vector are then broadcast one by one to all other PEs. Each PE multiplies the value with its (η*error)-value and adds the result to the corresponding weight.

The Pascal/L program

```
var W   : parallel array [0..N-1,0..N-1] of integer(b);
    net : parallel array [0..N-1] of integer(b + logN);
    B,A,Aold,E : parallel array [0..N-1] of integer(b);
    ExtInputs  : parallel array [0..N-1] of integer(b);
    sel : selector [0..N-1];
    j,m : integer;
{Initialize}
    rand(W);            {Initialize the weights}
    A := ExtInputs;     {Set activation values}
    Aold := A+2*e;      {Get past first while test}
{Train the net one pass with one pattern
Calculate new activation values during m cycles or to convergence}
    m := 0;
    while some(abs(Aold - A) > e) and (m < mmax) do begin
        j:= 0; net := 0; sel := TRUE; Aold := A;
        while some(sel) do begin
          net := net + W[*,j]*A[first(sel)]; next(sel); j := j+1;
        end;
        A := f(net+B); m := m+1;
    end;
{Update weights}
    E := η*(ExtInputs - A); j := 0; sel := TRUE;
    while some(sel) do begin
        W[*,j] := W[*,j] + A[first(sel)]*E; next(sel); j:=j+1;
    end;
    B:= B + η*E;
```

COMPUTATION TIME

Feedforward networks with error back-propagation

The computations for one layer of the feedforward pass contain N operations of type multiply(by constant)-and-add followed by a few (maximum ten) multiply-and-add operations to compute the activity function (e.g. by a piecewise linear activation function which approximates the sigmoid function). Since N is large in the applications we consider we can leave the latter operations out when we estimate the computation time.

During accumulation the sum will grow to a maximal length of b+log$N$ bits. On the average the number of cycles for multiply-and-add will be 4b+log$N$–1, using the bit-serial multiplier of Figure 2.

In the backward error computation phase the summation is made over the adder tree in b+log$N$ cycles. A multiplication and a tree addition can be made simultaneously. The weight changing phase, finally, takes 4b cycles per column.

In total the computations for one layer consume: [8b + log$N$–1 + max(3b, b+log$N$)]N cycles during training and (4b + log$N$–1)N cycles during recall.

Assuming a clock frequency of 10 MHz (which is fairly conservative) execution times shown in Table 1 are derived.

A common measure of the performance of neural net hardware is the number of "CPS" (Connections per second). In a net with N neurons per layer, N^2 connections are used and/or updated in each layer. The MegaCPS figures for BAPs of different sizes are given in Table 2.

		N		
		256	1024	4096
b	8	2.4	9.9	40.6
	12	3.6	14.4	58.6
	16	4.7	18.9	76.6

a)

		N		
		256	1024	4096
b	8	1.0	4.2	17.6
	12	1.4	5.8	24.2
	16	1.8	7.5	30.7

b)

Table 1. a) Training time per layer (ms). b) Recall time per layer (ms). 10 MHz clock frequency is assumed.

		N		
		256	1024	4096
b	8	27	106	413
	12	18	73	286
	16	14	55	219

Training

		N		
		256	1024	4096
b	8	66	250	953
	12	47	180	694
	16	36	140	546

Recall

Table 2. Number of MegaCPS (Million Connections Per Second) for different network sizes and data precisions.

Feedback networks with delta rule.

The computations of one iteration contain N multiply-and-add operations. We do these computations in (4b+log$N$–1)N cycles. Thus, the figures of Table 1b apply. The weight changing phase takes 4bN cycles. Executing m iterations and one weight update then takes $m(4b + logN–1)N + 4bN$ cycles.

Table 3 shows the training times for different network sizes and precisions. The times for recall equal those of Table 1b.

MEMORY REQUIREMENTS AND IMPLEMENTATION ASPECTS

The amount of storage per word required for the feedback network, or for each layer in the ff net, is approximately bN. See Table 4. An m layer network requires m–1 times as much memory

		N		
		256	1024	4096
b	8	1.8	7.4	30.7
	12	2.6	10.8	43.8
	16	3.5	14.0	56.9

a)

		N		
		256	1024	4096
b	8	101	423	1770
	12	142	586	2440
	16	183	754	3100

b)

Table 3. Training times (ms) for feedback network. a) with 1 iteration, b) with 100 iterations. 10 MHz clock frequency is assumed.

(with a m–1 connection weight matrices stored). It should also be mentioned that there are methods proposed that include a *momentum* term α in the weight changing rule for the ff networks (refer to Algorithm 1):

$$\Delta w_{ij}^{(l)}(t) = \eta \delta_i^{(l)} o_j^{(l-1)} + \alpha \, \Delta w_{ij}^{(l)}(t\text{-}1)$$

Thus the past weight change affects the current direction to an amount determined by the constant α. This is considered to allow high learning rate without leading to oscillations. However, it requires that the weight changes be stored as well, which doubles the required memory space.

		N		
		256	1024	4096
b	8	2	8	32
	12	3	12	48
	16	4	16	64

Table 4. Memory requirements (kilobits) per processor for one layer (approximate figures).

Using commercially available RAM chips for the large amount of memory needed offers obvious advantages. On the other hand, memory on the same chip as the PEs can increase processing speed significantly.

If external RAM is used 64 PEs of the complexity we discuss can easily be integrated on one VLSI chip. A 1024 PE array will have 16 such chips, each with approximately 100 pins. Memory can be implemented using chips with 64k x 4 bits, giving a chip count of 256. With appropriate mounting technology such a network may be implemented on one board. It would run a four-layered feedforward network with 1024 neurons per layer at the speed of 34 training examples or 80 recall examples per second.

An implementation of a prototype BAP is currently being made as a joint project between Luleå University of Technology and Centre for Computer Science at Halmstad University College [12]. The implementation is software configurable to allow for "compilation" of a certain architecture to suit a specific application. Neural network computations constitute one such application area.

CONCLUSIONS

We have given a general model of a bit-serial array processor (BAP) and have shown how the computations of different neural network models can be performed on such a processor. A major advantage of the bit-serial working mode is that precision can be traded for speed. We have calculated execution times and memory requirements for feedforward and feedback networks of different sizes and with different numerical precision. Results show a large speed advantage over commercial neural net simulators and form the basis for the outline of a one-board implementation comprising 1024 processing elements.

An interesting result is that the computations do not require the processor array to have a very rich communication structure. The facilities needed are the ability to broadcast a single bit from any processor to all others, a means for selecting processors in order, one by one (a select first chain), and a bit-serial adder tree to add the values of a field.

The approach taken is to map one neuron on each processor — in the case of multilayer networks the same processors are used for all layers. If more processors are available, or if the processors are fewer than the neurons, the programs presented must be slightly changed. The speed will increase or degrade accordingly. Thus, the speed of a certain network can be adjusted by choosing the number of processors.

A critical operation in the computations is multiplication. We have shown how a very simple bit-serial multiplier structure using carry-save technique can equalize multiplication time relative to addition time.

It is seen that the different net models that we have studied put the same demands on the processing array. These models are representative for the neural networksarea, implying that efficient execution of most kinds of neural networks on a BAP can be expected.

Acknowledgment: Part of this research is financed by Halmstad University College and STU under contract no. 88-03901P

REFERENCES

[1] Ohlsson, L. "An improved LUCAS architecture for signal processing." (Technical report), Dept. of Comp.Eng., Univ. of Lund, 1984.

[2] Fernström, C., I. Kruzela and B. Svensson. *LUCAS Assosiative Array Processor - Design, Programming and Application Studies*. Vol 216 of *Lecture Notes in Computer Science*. Springer Verlag. Berlin. 1986.

[3] Rumelhart, D. E. and J. L. McClelland. *Parallel Distributed Processing; Explorations in the Microstructure of Cognition*. Vol I and II, MIT Press. Cambridge. 1986.

[4] Hinton, G. E. and T. J. Sejnowski. "Optimal perceptual inference." In *Proceedings of the IEEE Computer Soci ety Conference on Computer Vision & Pattern Recognition*, pp. 448-453, Washington, D.C., 1983.

[5] Hinton, G. E. and T. J. Sejnowski. "Learning and relearning in Boltzmann machines." In vol. II of [3].

[6] Hopfield, J. J. "Neural networks and physical systems with emergent collective computational abilities". *Proceedings of the National Academy of Science USA*. 79: pp. 2554-2558, 1982.

[7] Hopfield, J. J. "Neurons with graded response have collective computational properties like those of two-state neurons". *Proceedings of the National Academy of Science USA*. 81: pp. 3088-3092, 1984.

[8] Hopfield, J. J. and D. Tank. "Computing with neural circuits: A model." *Science*. Vol. 233: pp. 624-633, 1986.

[9] Pineda, F. J. "Generalization of Back-Propagation to Recurrent Neural Networks." *Phys. Review Letters*. Vol. 59(19): pp. 2229-2232, 1987.

[10] Almeida, L. D. "Backpropagation in perceptrons with feedback." In *NATO ASI series: Neural Computers*, Neuss, F.R. Germany, 1987.

[11] Rumelhart, D. E. and J. L. McClelland. *Explorations in Parallel Distributed Processing*. MIT Press. Cambridge, Massachusetts. 1988

[12] Svensson, B. "Implementation and application of a software configurable massively parallel computer." *Second Swedish Workshop on Computer Systems Architecture*, Bålsta, Sweden, 1989.