A Computer Architecture to Support Neural Net Simulation

M. TUREGA

Department of Computation, University of Manchester Institute of Science and Technology, P.O. Box 88, Manchester M60 1QD

The Back Propagation Model for neural network simulation is a very simple and very popular model for solving real-world problems. It does, however, suffer from one major problem, that is, the time taken for the network to adjust its weights in order to response to an input. This paper proposes a computer architecture, specifically optimised for neural network simulation, which is capable of unlimited expansion without significant degradation in unit performance. It uses a high-speed conventional processor loosely coupled to a number of very simple processor and memory nodes, each with very limited functionality.

Received October 1990, accepted December 1991

1. INTRODUCTION

The Back Propagation Model for neural network simulation (Fig. 1) is a very simple and very popular model for solving real-world problems. It does however suffer from one major problem, that is, the time taken for the network to adjust its weights in order to respond to an input pattern in the desired way (Learning).

Several approaches have been adopted to circumvent this problem. These fall into two main categories, theoretical and hardware. The theoretical approach has been to try and understand the gradient descend approach to learning, in this case with a view to developing faster learning algorithms.[2,4,5] The hardware approach falls into two camps, those solutions supporting neural networks directly, usually using VLSI techniques,[6] and secondly those adopting and adapting digital architectures to simulate the process. There are two approaches to this, conventional machine architectures[7,10,14,21] and developing new digital architectures.[11,12,13,16,18,19]

The objective of the research project is to define a computer architecture specifically optimised for neural network simulation which is capable of unlimited expansion without significant degradation in unit performance.

This paper proposes an architecture which uses a powerful high-speed conventional processor loosely coupled to a number of very simple processor and memory nodes, each with very limited functionality. Whereas the architecture has only been simulated at present, a hardware implementation strategy for this architecture has been presented in a recent paper.[1]

The architecture has a number of properties which help it to achieve the objects, as follows.

- The processing node architecture is very simple with very limited functionality.
- The processing node has no stored program.
- The node has no need for data-dependent decision-making capability.
- Unlike a simple SIMD machine, the processor will repeatedly obey the latest instruction on new data until a new instruction is received.
- The connection mechanism chosen (a synchronous ring) allows for indefinite expansion without introducing the electrical or timing problems which would exist when using a bus.

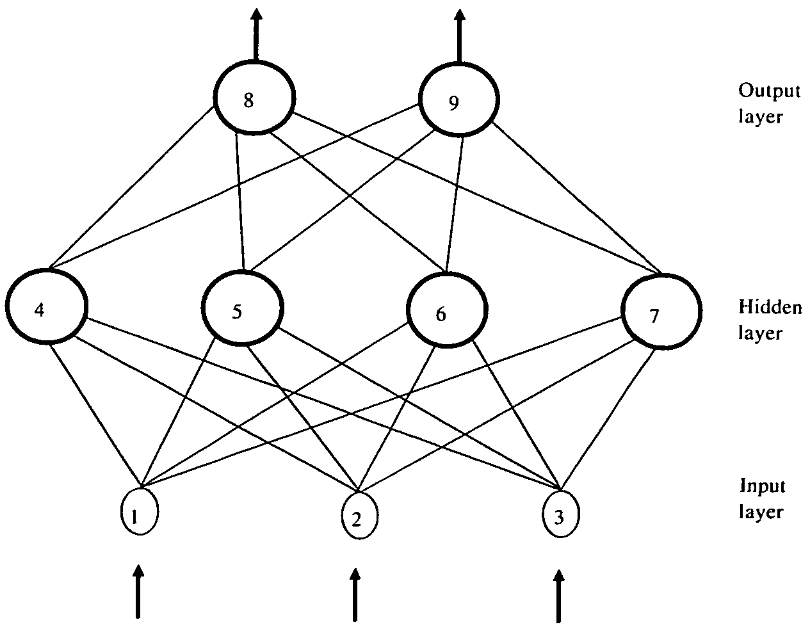


Figure 1. Example 3-4-2 network.

1.1 Learning in the back propagation (BackProp) model

Learning is an important aspect of any Parallel Distributed Processing (PDP) model and in particular the BackProp model. The simple 'delta' learning rule attributable to Widrow and Hoff[20] in 1960 has been the basis of developments over the years. Unfortunately, this rule requires that the output of a unit be compared with its expected value, giving an error factor. However, when a system has hidden units it becomes impossible to apply these rules, as there are no expected values for the output of the hidden units. For this reason a generalised delta rule has been developed by Rumelhart, Hinton and Williams,[15] which allows for hidden units in a feed-forward network (no feedback connection). This technique is called back propagation (BackProp) and is applied here to networks whose unit input and output values lie between 0 and 1.

The BackProp equations as defined by Rumelhart are reviewed here. The net input Net_j for node u_j is defined by Equation (1)

$$Net_j = \sum_{i=1}^{N} w_{ji} o_i, \tag{1}$$

where w_{ji} is the weight from node u_i to node u_j, o_i is the activation value of node u_i, N is the number of nodes connecting to node u_j.

So for an arbitrary node u_j

$$o_j = \frac{1}{(1+e^{-(Net_j+\theta_j)})}, \tag{2}$$

where o_j is the activation value of the node, θ_j is the bias* for the node.

The process of back-error propagation requires that an error factor be produced for each node in the network.

Because the desired output is known, the error for an output node can be determined simply as the difference between the desired output for a given exemplar (training pattern) and the actual activation value generated when that exemplar is applied as an input vector to the net. The error factor for output nodes is simple, and is defined in Equation (3) for an arbitrary output node u_j.

$$\delta_j = o_j(1-o_j)(t_j-o_j), \tag{3}$$

where t_j is the desired action value of node u_j for a given training pattern, o_j is the actual activation value of node u_j for a given training pattern and δ_j is the error factor for node u_j for a given training pattern. Unfortunately, because hidden nodes by definition do not have a 'visible' activation value, the desired activation value for a hidden node is not known. The BackProp algorithm therefore defines the error factor of a hidden node in terms of the nodes to which it is connected (the output layer nodes for the last hidden layer). This is expressed in Equation (4) for an arbitrary hidden unit u_j.

$$\delta_j = o_j(1-o_j) \sum_{k=1}^{N} \delta_k w_{kj}, \tag{4}$$

where δ_k is the error factor for node u_k, N is the number of nodes to which node u_j connects and w_{kj} is the weight from node u_j to node u_k. Once all the errors factors for nodes have been calculated, the weights can be changed. The change to any weight which takes place as a result of applying a training pattern to the input is defined in Equation (5).

$$\Delta w_{ji} = \eta \delta_j o_i, \tag{5}$$

where Δw_{ji} is the change to be made to w_{ji} as a result of applying a training pattern and η is the learning rate. The change to the weight Δw_{ji} can be applied to the weight after each training pattern (on-line training), after the whole set of training patterns (off-line or epoch training) or after a subset of the training patterns.

In order to cause learning to proceed as fast as possible, the learning rate η should be as large as possible; however, large values of η can cause instability in the algorithm. It is for this reason (and to filter out high-frequency oscillations) that Rumelhart has suggested adding a 'momentum' term to the rule as shown in Equation (6).

$$\Delta w_{ji_{\mathrm{NEW}}} = \eta \delta_j o_i + \alpha \Delta w_{ji\mathrm{OLD}}, \tag{6}$$

where α is a momentum term defining how much of the previous change to incorporate.

Finally the weights need updating. This is performed in line with the learning rule adopted (on-line or off-line) and is simply defined in Equation (7).

$$w_{ji} = w_{ji} + \Delta w_{ji}. \tag{7}$$

1.2 Existing solutions

The major problem with the BackProp algorithm is that the learning time increases $O(N^2)$, where N is the number of nodes. A number of solutions to this learning problem have been proposed and implemented; some of these are reviewed here before describing the solution proposed in this paper.

1.2.1 Systems using existing processors to exploit the concurrency in the algorithm

Morgan and others have described a neural network simulator based on commercial DSP devices connected in a ring (the RAP).[10] They chose the DSP devices because the basic time-consuming operation for neural network simulation is 'multiply and accumulate', a task for which they say DSPs are ideally suited. Their device is programmable and uses a stored program for control purposes. They deal with the problem of the weights being in the wrong place for the BackProp backward pass by performing the backward pass 'multiply accumulate' operations along the ring in a systolic manner, so that partial products are transferred from processor to processor. This technique does however assume that the processors must communicate with their neighbours in an intelligent manner.

Grajski and others[7] describe the use of the MasPar MP-1 parallel SIMD processor to implement a version of the BackProp algorithm. They used a version of the MasPar with 4096 processors, which as well as sharing a common control path are interconnected using an 'X-network' allowing each to communicate directly with each of 8 neighbours. They also assume a granularity of parallelism of a single weight within a node.

* Although the bias is dealt with as a separate entity here, in practice it can be treated exactly the same as a weight connected to an input that is always a constant (usually 1).

Witbrock and Zagha[21] have performed similar work to Grajski but using an experimental IBM SIMD processor, the GF11, with between 356 and 566 processors each with 512K of DRAM and 16K of fast SRAM. They have used the data partitioning approach utilising a separate processor for each training pattern in the training set. This would be of less value with on-line problems where no fixed training set exists.

Pomerleau and others[14] describe a system using a WARP, a ten-processor systolic array. The concurrent algorithm they have implemented is also based on the idea of data partitioning. Several processors use the same weight values to independently process several training set exemplars.

Whereas all these systems provide the ability to support neural networks concurrently with varying degrees of success, they are based upon general-purpose machines or processors, whose concurrent components are designed to perform a range of tasks. The complexity and functionality of these concurrent components is much greater than that required to support BackProp, so only a percentage of the potential computing power is used in the support of the algorithm.

1.2.2 Systems using dedicated digital hardware

There have been systems developed recently which provide support for neural network simulation using dedicated digital hardware.

A paper by Treleaven and Rocha,[18] describes two types of VLSI support device; the first is a programmable RISC-type general-purpose neural-processor device and the second is a generic neuron device.

Piazza and others[13] have described an architecture to support the BackProp algorithm with what they call a 'snake' of processing elements. These elements are connected together as an SIMD machine in a snake by three independent communication paths. Each of the processing elements has a stored program control – although all obey the same instruction sequence – and performs the whole of the BackProp algorithm as elements of data are passed along the communication channels in a systolic nature.

A paper by Sammut and Jones has proposed an architecture for the support of multiple learning and recall algorithms.[16] They have based the architecture on a pipeline of processing nodes, where data is moved from node to node in a systolic manner such that matrix operations may be performed. The processing nodes are intelligent with an RISC instruction set and are programmable. The activation function is implemented on the processing node using a least-squares approximation technique.

British Telecom have introduced their work on neural network processors in papers by Orrey, Myers and Vincent,[12] Vincent and Myers[19] and Myers and Brebner.[11] They describe their neural network chip 'Hannibal', which can be connected together in a linear array to provide parallel operations in support of the BackProp algorithm. The algorithm for BackProp is a function of the device, although they state that it is reconfigurable for Hopfield and other non-layered networks.[8]

When the existing parallel architectures are reviewed, it is apparent that although all the reviewed systems have positive attributes and some negative attributes, they generally suffer from one or more of three main drawbacks.

(i) They try to implement the whole of the neural network algorithm function in the processing elements, even though in most cases some of the functions performed are only undertaken once per learning cycle.

(ii) In order to control the operation of their processing elements, they either have a stored program or build the algorithm into the logic of the hardware.

(iii) Some of the architectures have complex communication systems.

This paper demonstrates that in order to develop a neural network simulator for the BackProp algorithm, whose performance increases in direct proportion to its size, with no scaling overheads: (*a*) it is not necessary to have any stored program control or BackProp algorithm-dependent logic in the processing elements; (*b*) it is not necessary for the processing elements to perform any of those functions whose total processing time only increases linearly with network size; (*c*) simple ring-based connection mechanism is sufficient.

1.3 An analysis of the processing requirements

An analysis of the activation, output and learning formulae for the BackProp model reveals that two types of calculation take place, those which occur only a few times during an evaluation or learning cycle and those which take place frequently. The first type are calculations like 'apply the activation function' (component of Equation (2)), which are only performed at the end of a cycle; the second type are the $\Sigma w_{ji} o_i$, $\Sigma w_{ji} \delta_i$ and the Δ product sums and weight-change operations (components of Equations (1) and (4)).

If we assume a network with N nodes per layer and that N is large (e.g. 1000), the calculations performed by the node will be predominantly the $\Sigma w_{ji} x_i$ and Δ type. The other types of calculation related to activation functions and learning are only performed relatively infrequently.

An analysis of the BackProp model shows that apart from the learning modifications (Δ operations), the only calculations which use the weight values are the Σ calculations.

Given that the Σ and Δ calculations dominate, a simulator where all the simulation except for the Σ and Δ operations are carried out on one conventional processor and only the Σ and Δ operations are carried out by the rest of the processors (the distributed processors) will have a performance roughly equivalent to a fully distributed and connected system where all the processors were equivalent.

As the Σ and Δ operations are the only ones which access the weight values, the weight memory is associated with and distributed with the distributed processors. This suggests a scenario of a system which consists of a central versatile processor to which are connected a large number of distributed memory units each with a very simple processor capable of performing the Σ and Δ operations on the memory.

The two main features of this architecture are its performance linearity when scaling the architecture (i.e. twice the size performs twice the work) and a novel very simple processor element on the processing node which is a special case of the SIMD model. It is this architecture which will be explored further in the rest of this paper.

2. AN ARCHITECTURAL PROPOSAL

The proposed architecture would be a hybrid based upon a conventional monoprocessor on which the BackProp neural net model would run. This processor would maintain the input, output and activation states for each of the units in the model; it would also perform all the calculations which only occur once every iteration of the model.

The conventional processor would be complemented by a collection of distributed processor nodes (DPNs). The simplest architectural model has one DPN for each node in the BackProp model being simulated. This turns out to be potentially inefficient and restrictive, therefore support for several BackProp nodes by one DPN is an important requirement of the architecture. For the purposes of this discussion, however, it is assumed that each node in the BackProp model would have its own DPN. Each DPN would be capable of simple limited operations and would also maintain the weight values for all the BackProp model connections impinging upon the BackProp node supported by the DPN.

To illustrate how the two parts would interact, consider a simple three-layered fully connected BackProp neural net simulation (Fig. 1) running on the conventional processor. At many points in the simulation it will be necessary for some product sums (i.e. $\Delta w_{ji} x_i \; i = 1..N$) for each of N units to be produced. On a conventional processor this would normally involve $O(N^2)$ product sum operations.

Using the DPNs, the conventional processor would broadcast each of the values of x_i (where x is o or δ) to all the DPNs. As each DPN received the value of x_i, it would use it to produce the product of x_i and w_{ji} (for unit j) and add this to the product sum. After the conventional processor had sent N values of x, each of the distributed processors would have a product sum ready for transmission back to the conventional processor. This would be achieved in N reads from the DPNs by the conventional processor.

Assuming that a DPN is capable of performing a product sum in the same time as the conventional processor (p), the time taken to produce the product sum conventionally on the conventional processor would be N^2p, and using the DPNs would be Np plus the time taken to return the product sums back to the conventional processor. When N is large the product sums are the dominant component in the simulation; this means that to a first-order approximation the simulation time is proportional to $O(N)$.

Experiments have shown that connecting processors in a ring structure is an efficient mechanism for neural network simulation using this approach. The neural network hybrid computer would therefore consist of a conventional processor performing the BackProp simulation. It would be connected synchronously to a ring of DPNs, such that the output from the conventional processor would be sent to the first DPN, which would pass its output to the next and so on. The last DPN would pass the results back to the conventional processor. All communication between DPNs would be synchronised to a common system clock.

2.1 Mapping the BackProp model to the hybrid architecture

The various functions of the BackProp model are distributed between the conventional processor and the DPNs in the following manner.

2.1.1 Weight storage

Each DPN has its own memory, which can be used for weight storage. This can only work if all the operations relating to the weight memory are performed locally by the DPNs. These operations will now be considered.

2.1.2 Direct weight manipulations

There are a number of operations which are performed on the weights directly and a number which are performed on the results of these direct weight operations (indirect operations). This section examines the direct operations.

There are three direct operations for the BackProp Model described below.

2.1.2.1 Forward pass product sum

The forward pass process requires that each node in the hidden or output layers should accumulate the weighted sum of all its inputs as part of that process.

If a DPN were to represent a BackProp model node and use its weight memory to represent the weights for the BackProp model node, then, assuming each of the input values was passed to the DPN in turn, the DPN would be capable of producing a product sum suitable for the forward pass process (Σ in equation (1)). It follows that if other DPNs are representing the rest of the BackProp nodes in a given layer, they would also be capable of producing their own product sums, assuming they had access to the same input values.

2.1.2.2 Back pass error sum

The error factors for output and hidden nodes (δ_j) in the BackProp model are determined in different ways.

For the output layer nodes, the error factor (Equation (3)) is determined simply from the actual value output from the node and the desired output (training value). It does not involve any direct use of the weights and would not need to be implemented in the DPN.

The hidden layers, however, have their error factors determined by a formula which has a component involving product sums of weights (Equation (4)). The proposal is that the product sum component be implemented in the DPN in the following way.

For a given hidden layer, each of the error factors from the following layer are propagated back to each of the DPNs representing the hidden layer in question. These are multiplied in turn by the weights in the DPNs representing the following layer (Σ in Equation (4)). This is a problem, as the required weights are not available at the DPNs doing the multiplying. The proposal is therefore that the weight associated with a connection in the BackProp model is maintained in both of the DPNs which represent the BackProp connection source and destination nodes (i.e. at both ends of the connection).

2.1.2.3 Weight change

The last consequence of maintaining weights in the weight memory of a DPN is the one of adjusting the weights. Weights need to be adjusted to correct errors in the way the net operates (Equations (5), (6) and (7)).

The need to store a weight in both the DPN which represents the source node of a connection and the DPN which represents the destination node of a connection presents a problem, but fortunately not an insoluble one. Two slightly different techniques are needed to ensure that the changes made to a weight at the source of a connection are the same as those made at the destination.

For the destination node of a connection in the BackProp model, the formula demands that the change in weight is a function of the previous change and the product of the error factor of the BackProp node multiplied by the output value of the source node associated with that weight (Equation (4)). Assuming the DPN representing the BackProp node 'knows' the error factor for that node (δ_j), then by presenting it with each of the output values from the previous layer in turn (o_i) it can adjust each of the weights in turn. In a similar manner to the Forward Pass Product sum, all DPNs can potentially operate concurrently on their own weight memory.

The copy of a weight stored at the DPN representing the source of the connection in the BackProp model has to be modified by the same amount as the weight at the DPN representing the destination of the connection. The DPN, 'knows' its own output value (o_i) and needs to be presented with each of the following layer's node error factor values (δ_j) in turn, but apart from that the process is similar to the above.

2.1.3 Indirect weight manipulations

All the other operations of the BackProp model such as applying the squashing function (component of Equation (2)) to the product sum of weighted inputs would be undertaken by the central processor. The benefits of implementing these operations (which only occur once per learning cycle per node) in the DPN would become insignificant for larger BackProp models where the product sum operations dominate.

2.2 The DPN architecture

The DPN is very simple in its operation. It is effectively a state machine which processes a data stream depending upon the current state (or instruction). Changes to the state of the DPN (instructions) are embedded in the data stream and are distinguishable from data elements. Because of this mode of operation, there is no need for any stored program mechanism.

The architecture of the DPN will be discussed in four parts, an overview of the DPN functionality, the register/memory structure, the instruction/data format and a description of the operations during each of the states.

2.2.1 Overview of the DPN functionality/architecture

The DPN architecture is intended to be as simple as possible. Any function which can be performed by the conventional processor without destroying the basic linear scaleability of the whole system will be performed by the conventional processor. For example, the process of setting the weight memories to 'small random' values is left to the conventional processor, which 'writes' the respective values into each of the DPNs via the ring.

Control of the DPNs is a very important aspect of the implementation of the architecture. The algorithm for performing an iteration of a BackProp simulation is fixed for a given size of neural network, hence the tasks to be performed by the DPNs are fixed and deterministic. The problem of control within a DPN has been considered with this deterministic nature in mind.

The control function is provided by interspersing instructions in the middle of the data stream. When a DPN receives a new instruction it stores it and then repeatedly obeys that instruction on each new data item it receives until another instruction is received. This means for example that the DPN does not need to know how many data items are required to be processed; this is determined by the conventional processor, which sends a new instruction at the correct time.

In order to simulate the BackProp algorithm, it is not always desirable to have every node obey every instruction (for example when asking a node to return a value). The mechanism to achieve this is to give each DPN a unique identity. The identity will not be a fixed attribute of the DPN but will be assigned at the start of the simulation by the use of a 'set identity' (RSET) instruction. This also serves the purpose of informing the conventional processor of the number of DPNs currently configured in the ring.

Each instruction sent to the DPNs will be tagged by either a global tag, which identifies the instruction as applying to all DPNs, or an identity tag, which will identify the DPN to which the instruction is to apply (all other DPNs will interpret the instruction as 'IDLE', the 'do nothing' instruction).

2.2.2 Register/memory structure

The following registers define the complete 'state' of the DPN.

Register bits	Description
INPUT 25	Register into which instructions or data are received at the start of a processing cycle.
OUTPUT 25	Register used to output the results of processing at the end of a processing cycle.
SELF 14	Register used to hold a unique identity for each DPN.
ACC 24	Floating-point storage register.
MA 14	Register used to identify which memory location to use.
MB 14	Register used to identify which memory location to use.
INSTR 5	Current instruction in use by the DPN.
MEM[ ] 24	An array of memory used to store weight and related values.

2.2.3 Instruction/data format

The instruction/data format is illustrated in Fig. 2.

Each 25-bit value received via the INPUT register can be data (24-bit floating-point value or 14-bit integer value) or an instruction. One bit of the input is defined as an instruction/data tag (InstrTag).

Data. Data items are tagged by a '0' in InstrTag.

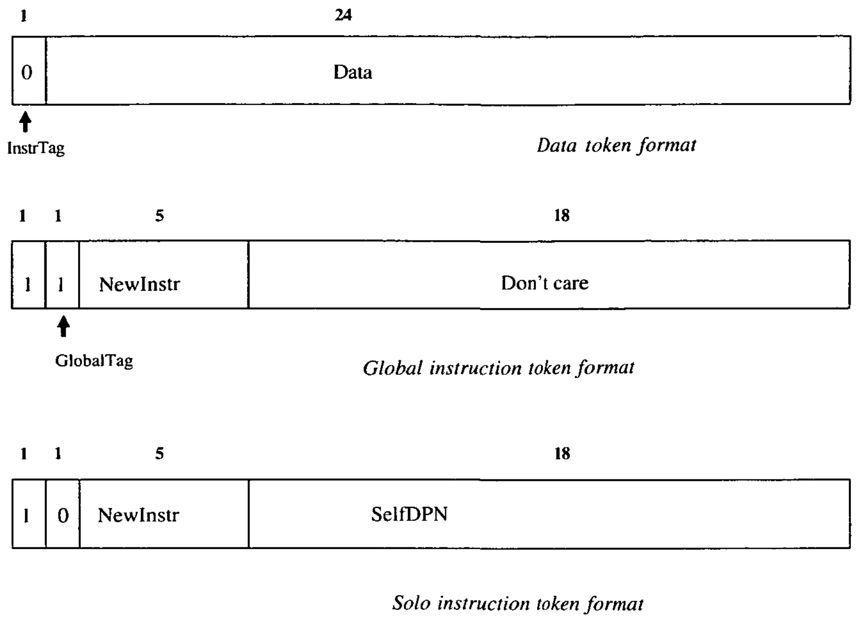


Figure 2. Instruction/data format.

Instructions. Instructions are tagged by a '1' in InstrTag and have a fixed format of fields defined as follows.

GlobalTag. A tag to indicate that all DPNs should respond to the instruction.

SelfDPN. This identifies the identity number (SELF) of the DPN which should respond to the instruction.

NewInstr. This identifies a new instruction (or state) for all DPNs which should respond to the instruction. For those DPNs which should not respond, the instruction is defined to be 'IDLE'.

2.2.4 Operations

The DPN receives a 25-bit value in the register INPUT at the start of each machine cycle. This value can be an instruction or a data item. If it is an instruction, the new value of the instruction 'NewInstr' (or 'IDLE' if the instruction is not directed at that DPN) is loaded into the instruction register 'INSTR' (note: it is not obeyed at this time). If however the value in INPUT is a data item then the current instruction defined by register 'INSTR' is obeyed using the data received, the register 'INSTR' remaining unaltered.

The different instructions are:

IDLE, RSET, SACC, SRMA, SRMB, RACC, WMEM, RMEM, CSUM, CERR, UPDM, MODM

and their functionality is best described by considering the following Pascal-like code; the operation of the node in a Pascal-like form is shown here.

For each DPN processor cycle

```
if (InstrTag = 1) then
begin
  if ((SelfDPN = SELF) or (GlobalTag = 1)) then
    INSTR = NewInstr
  else
    INSTR = 'IDLE';
  OUTPUT = INPUT;
end
else
case (INSTR)
  'IDLE':  OUTPUT: = INPUT
  'RSET':  SELF: = INPUT;
           OUTPUT: = INPUT+1
  'SACC':  ACC: = INPUT; OUTPUT: = INPUT
  'SRMA':  MA: = INPUT; OUTPUT: = INPUT
  'SRMB':  MB: = INPUT; OUTPUT: = INPUT
  'RACC':  OUTPUT: = ACC
  'WMEM':  MEM[MA]: = INPUT; MA: = MA+1;
           OUTPUT: = INPUT
  'RMEM':  OUTPUT: = MEM[MA];
           MA: = MA+1
  'CSUM':  ACC: = ACC+(INPUT*MEM[MA]);
           MA: = MA+1; OUTPUT: = INPUT
  'CERR':  MEM[MA]: = MEM[MA]+
           (INPUT* ACC);
           MA: = MA+1;
           OUTPUT: = INPUT
  'UPDM':  MEM[MA]: =
           MEM[MA]+MEM[MB];
           MA: = MA+1; MB: = MB+1;
           OUTPUT: = INPUT
  'MODM':  MEM[MA]: = MEM[MA]*ACC;
           MA: = MA+1; OUTPUT: = INPUT
endcase
```

2.3 The conventional processor

The conventional processor component of the simulator consists of a 'compiler' and run-time system running on a conventional machine with access to the DPN ring.

The function of the compiler is to allocate memory within the DPNs and then to generate the necessary instructions for use by the run-time system for a generic BackProp simulation of a particular size and for a particular number of DPNs. The compiler produces three distinct instruction sequences, an initialisation sequence, an iteration sequence and a conclusion sequence.

2.3.1 Initialisation sequence

The run-time system first uses the initialise sequence to initialise the DPNs and to write the initial weight memory to the DPNs.

2.3.2 Iteration sequence

The run time system applies the iteration sequence repeatedly until the BackProp simulation termination conditions are met (as determined by the run time system). It is during the iterative cycle that the run time system uses returned values to compute nodal output values, network output errors and node error factors. It is only during this phase that the application problem specific training values are used.

2.3.3 Conclusion sequence

When the run time system has determined that the termination conditions have been met, then the conclusion sequence is applied. The main purpose of this is to recover the weight and bias values from the DPNs for storage and possible further use.

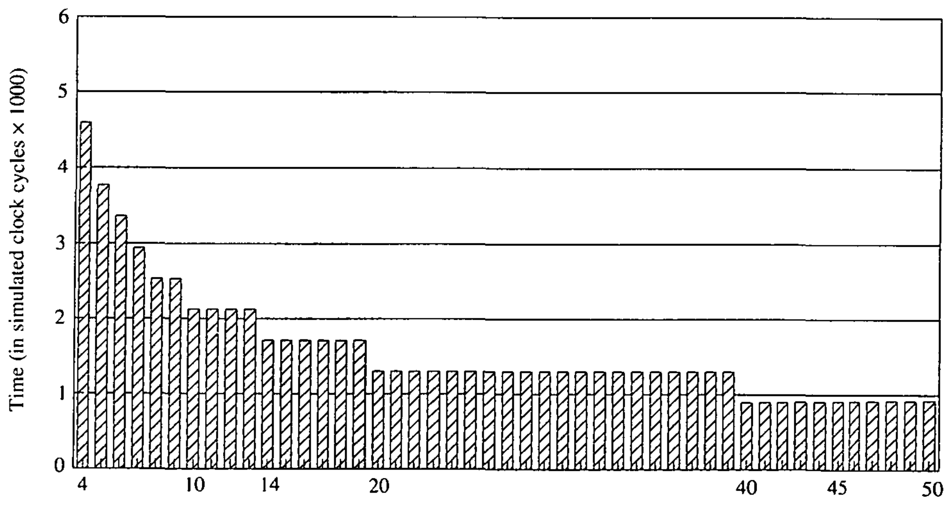
3. EVALUATION OF THE ARCHITECTURE

The BackProp neural network software was written to utilise a varying number of the DPNs. The software was run on a number of basic neural problems with a varying number of emulated DPNs. Simulation had demonstrated that the basic ideas are sound.

In all cases, the results of a BackProp simulation obtained for a particular application problem were identical irrespective of the number of emulated DPNs utilised. For example, the results of the 4-2-4 problem were identical when run on a ring of 1, 2, 3, 4 or even 6 emulated DPNs.

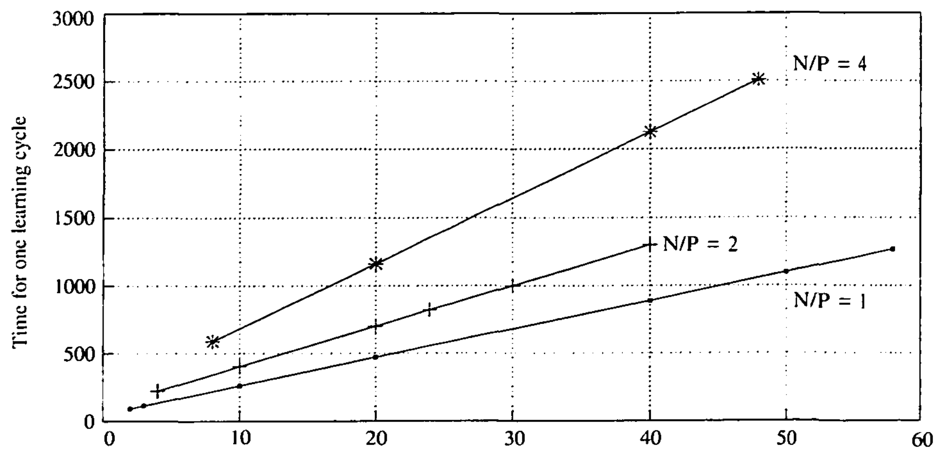
For simplicity, only networks where the numbers of input, hidden and output nodes are the same have been presented. The relationship between a particular network of N nodes in each layer and P DPNs is that the time taken to perform one forward and back pass of the BackProp algorithm decreases non-linearly as P increases. This continues until the time is at a minimum when $P = N$. Where P exceeds N there is a slight increase in time. This is illustrated in Fig. 3 for a 40-node-per-layer network.

One of the original goals of the research was to develop an architecture whose efficiency was not reduced



Number of processing elements

Figure 3. Processing time against the number of DPNs (40 node per layer network).



Number of nodes per layer

Figure 4. Time compared with the ratio of N/P. N, number of nodes per layer; P, number of processing elements (DPNs).

as it was scaled up. Experiments on the simulation have confirmed the theoretical basis of the architecture by showing a strict linear relationship between the number of nodes in each layer of the BackProp model (N), the number of DPNs (P) and the time (t). This is:

$$t = aP + c$$

for the case where N is a simple multiple of P (where a and c are DPN/BackProp algorithm-dependent constants).

Fig. 4 illustrates this for the 3 cases:

$$N = P,\ N = 2P,\ N = 4P.$$

As the size of the model (N) and the size of the DPN ring (P) increases, c becomes increasingly insignificant. Therefore for the large simulation it is reasonable to ignore the overhead c. This means the amount of useful work performed by each DPN in a large system tends to a constant.

4. CONCLUSION

This paper has demonstrated that as far as large-model BackProp neural network simulation is concerned, a computer architecture based upon a conventional processor and P distributed processor nodes (with memory) can achieve or better the performance of a system consisting of 'P+1' conventional processors (with memory) in a multiprocessor environment (assuming that the speed of performing a particular function on a DPN is less than or the same as the speed on the conventional processor).

An objective of this research project is to produce an architecture which is capable of indefinite expansion, where the amount of processing performed is proportional to the number of DPNs. The proposed DPN architecture together with the use of a ring communication mechanism serves to achieve this objective.

Furthermore, it has been demonstrated that the architecture of the DPN can be very simple, with no stored program control needed, but with the current processing requirement defined by the INSTR register.

Acknowledgements

The author would like to acknowledge the ideas, and the help in preparing this paper, of Professor D. Aspinall and Dr G. Brebner.

REFERENCES

1. D. Aspinall, G. Brebner and M. A. Turega, An artificial net based on a ring architecture, silicon architectures for neural nets. In *Proceedings of the IFIP WG10.5 Workshop on Silicon Architectures for Neural Nets*, edited M. Sami and J. Calzadilla-Daguerre, pp. 89–100. *St Paul de Vence, France 28–30 November 1990*, North-Holland, Amsterdam (1991).
2. J. P. Carter, Successfully using peak learning rates of 10 (and greater) in Back Propagation networks with the Heuristic Learning Algorithm. *First International Conference on Neural Networks, San Diego, 1987* II, 645–651.
3. M. Duranton and N. Maudit, A general purpose digital architecture for neural network simulations. *Proceedings of the First IEE International Conference on Artificial Neural Networks, London, October 1989*, pp. 62–66.
4. S. E. Fahlmann, Faster learning variations on back propagation: an empirical study. *Proceedings of the 1988 Connectionist Models Summer School*, pp. 38–51. Morgan Kaufman, Los Altos, (1988).
5. S. E. Fahlmann and C. Lebiere, *The Cascade-Correlation Learning Architecture*. Report CMU-CS-90-100, Carnegie Mellon University (1990).
6. H. Graf and P. de Verger, A CMOS implementation of a neural network. In *Advanced Research in VLSI*, edited E. Losleber, pp. 351–367. MIT Press, Stanford (1987).
7. K. Grajski *et al.* Neural network simulation on the MasPar MP-1 massively parallel processor. *Proceedings of International Neural Network Conference, Paris, 1990*, pp. 673–676, vol. 2, Kluwer, Amsterdam (1990).
8. J. J. Hopfield and D. W. Tank, Computing with neural circuits: a model. *Science* **233**, 625–633.
9. J. J. Hopfield, Artificial neural networks. *IEEE Circuits and Devices Magazine*, pp. 3–10 (September 1988).
10. N. Morgan *et al.* The RAP: a ring array processor for layered network calculations. *Proc. Conf. on Applications of Specific Array Processors, Princeton, NJ, Sept. 5–7, 1990*, pp. 296–308.
11. D. J. Myers and G. E. Brebner, The implementation of hardware neural net systems. *IEE 1st Int. Conf. on Artificial Neural Networks, 16–18 Oct. 1989, Conf. pub. 313*, pp. 57–61.
12. D. A. Orrey, D. J. Myers and J. M. Vincent, A high performance digital processor for implementing large artificial neural networks, *Proceedings of the Custom Integrated Circuits Conference, San Diego, May 1991.*
13. F. Piazza, M. Marchesi and G. Orlandi, A digital 'SNAKE' implementation of the back propagation neural network. *Proceedings of the International Symposium on Circuits and Systems, Portland OR, 1989* **3**, 2185–2188. IEEE, New York.
14. D. A. Pomerleau, G. L. Gusciora, D. S. Touretzsky and H. T. Kung, Neural network simulation at Warp speed: how we got 17 million connections per second. *Proceedings of the IEEE International Conference on Neural Networks, San Diego 1988* **2**, 143–150. IEEE, New York.
15. D. E. Rumelhart, G. E. Hinton and R. J. Williams, Learninginternal representations byerror propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Vol. 1: *Foundations*, edited D. E. Rumelhart and J. L. McClelland, pp. 318–362. 318–362, MIT Press, Cambridge, MA (1986).
16. K. Sammut and S. Jones, The toroidal neural net: an architecture for the support of multiple learning and recall algorithms. *Proceedings of the Second Australian Conference on Neural Networks 1991.*
17. J. Schmidhuber, Accelerated learning in back propagation nets, In *Connectionism in Perspective*, edited R. Pfeifer *et al.* Elsevier Science–North-Holland, Amsterdam (1989).
18. P. Treleaven and P. V. Rocha, Towards a general-purpose neurocomputing system. *Proceedings of the IFIP WG10.5 Workshop on Silicon Architectures for Neural Nets, St Paul de Vence, France, 28–30 November 1990*, edited M. Sami and J. Calzadilla-Daguerre. North-Holland, Amsterdam (1991).
19. J. M. Vincent and D. J. Myers, Parameter selection for digital realisations of neural networks. *Proceedings IEE Electronics Division Colloquium on Neural Networks: Design Techniques and Tools, London, March 1991.*
20. G. Widrow and M. E. Hoff, Adaptive switching circuits. *Proceedings, Institute of Radio Engineers, Western Electronic Show and Convention*, part 4, pp. 96–104.
21. M. Witbrock and M. Zagha, *An implementation of Back-Propagation Learning on GF11, a Large SIMD Parallel Computer*. Report CMU-CS-89-208, Carnegie Mellon University (1989).