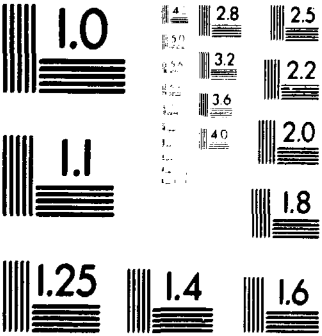
1.0
2.8
2.5
3.2
2.2
3.6
2.0
4.0
1.1
1.8
1.25
1.4
1.6

95001?7

Dipartimento Innovazione

AN EFFICIENT IMPLEMENTATION OF A BACKPROPAGATION LEARNING ALGORITHM ON A QUADRICS PARALLEL SUPERCOMPUTER

SERGIO TARAGLIO
Centro Ricerche della Casaccia, Roma

FEDERICO MASSAIOLI
CASPUR, c/o CICS Università degli Studi di Roma "La Sapienza", Roma

Work presented at the HPCN '95 - EUROPE
Milano, maggio 1995

RT/INN/95/12

VOL

ENEA
ENTE PER LE NUOVE TECNOLOGIE,
L'ENERGIA E L'AMBIENTE

Dipartimento Innovazione

AN EFFICIENT IMPLEMENTATION OF A BACKPROPAGATION LEARNING ALGORITHM ON A QUADRICS PARALLEL SUPERCOMPUTER

SERGIO TARAGLIO
Centro Ricerche della Casaccia, Roma

FEDERICO MASSAIOLI
CASPUR, c/o CICS Università degli Studi di Roma "La Sapienza", Roma

Work presented at the HPCN '95 - EUROPE
Milano, maggio 1995

RT/INN/95/12

Testo pervenuto nel giugno 1995

SOMMARIO

Si presenta una libreria software per simulare ed addestrare Percettroni Multi Strato tramite l'algoritmo di Back Propagation (Retro Propagazione dell'Errore). La macchina verso cui questa libreria è indirizzata è il super computer massivamente parallelo Quadrics. Si mostrano delle misure di prestazioni su tre diverse macchine con diversi numeri di processori, per due reti neurali di esempio. Viene presentato un prototipo di codice sorgente, quale esempio.

SUMMARY

A parallel implementation of a library to build and train Multi Layer Perceptrons via the Back Propagation algorithm is presented. The target machine is the SIMD massively parallel supercomputer Quadrics. Performance measures are provided on three different machines with different number of processors, for two network examples. A sample source code is given.

An efficient implementation of a Backpropagation learning algorithm on a Quadrics parallel supercomputer

Sergio Taraglio[1]
Federico Massaioli[2]

[1] ENEA - C.R. Casaccia, Via Anguillarese 301, 00060 Roma
[2] CASPUR, c/o CICS Università "La Sapienza", P.le Aldo Moro 5, 00185 Roma

Introduction

In the beginning of their history, artificial neural networks architectures were employed to mimic in an extremely simplified way the nervous system. Afterwards a more engineering point of view entered the scene. Nowadays the use of such computing technique is spread among a huge and increasing variety of fields of application.

We present here the implementation of the most classical learning algorithm for the most popular neural architecture: i.e. the backpropagation used to train a multi layer perceptron.

The target computer used for this aim is the Quadrics super computer, a SIMD massively parallel machine derived from the APE 100 project of the Italian National Institute for Nuclear Physics (INFN).

The parallel algorithm is here presented together with the performance evaluation on this machine and an analysis of its performance trend on larger computers of the same breed with a larger number of processors is shown.

The Quadrics Architecture

The Quadrics parallel supercomputer series derive from the INFN project APE 100, which was born to solve the need for highly intensive computing in the field of particle physics, namely Quantum Chromo Dynamics (QCD) see [1] and [2].

It is a SIMD (Single Instruction Multiple Data) parallel machine, attached to a host computer which is responsible for the uploading/downloading of programs and data and the managing of I/O operations.

Quadrics is composed of an integer CPU (Z-CPU) which controls the flux of the program and the integer arithmetic and of an array of custom floating point units (MAD, Multiply and Add Device), each one equipped with its own memory. The nodes are logically arranged on the vertexes of a 3D cubic mesh and are capable of synchronous, parallel first neighbours communications.

In the Quadrics architecture, each processor board contains eight floating point units. The different implementations may span from eight to 2048 processors, with peak computing power ranging from 400 MFlops to 100 GFlops.

This work has been presented at the HPCN '95 - EUROPE conference and published in the proceedings by Springer-Verlag.

The floating point processors are VLIW pipelined devices optimised to deliver the result of a so called "normal" operation, i.e.

$$A = B*C+D \tag{1}$$

once at each clock time.

The Quadrics system can be programmed in TAO, a FORTRAN-like data-parallel language. This language can be modified and extended through the dynamical ZZ parser on which the compiler is grounded. In this way the user can create a custom meta-language which allows a much more easy and "natural" programming. For more information concerning the Quadrics architecture see [1] and [2].

Neural Networks and Back Propagation

An artificial neural network is composed by a big array of simple non linear elements (the so called neurons), highly interconnected by links of different weight. These elements decide their own state according to the weighted sum of all the signals incoming from the other neurons. It is evident the similarity to the activity of the neurons in the nervous system.

The y_i state of the i neuron is computed by the:

$$y_i = g\left(\sum_{j=1}^{N} w_{ij}x_j - \theta_i\right) \tag{2}$$

where w_{ij} is the weight of the connection from element i to element j, x_j is the value of activation of neuron j and θ_i is a threshold value for the activation of neuron i.

The function $g()$ is the:

$$g(x) = \frac{1}{1+e^{-x}} \tag{3}$$

Out of the high number of connections and elements and their non linearity it is possible to obtain any desired value as output while presenting a given input. In other words, changing the values of the connections it is possible to compute any function of the input. For greater details see e.g. [3] or [5].

The backpropagation is a training algorithm for a huge class of neural networks: the multi layer perceptrons. This architecture is one of the most versatile, allowing the resolution of non linearly separable problems. Through this procedure it is possible to train a network to give the correct association between input and output pairs.

To describe the algorithm let us assume a three layer network, composed of an input layer (i), a hidden layer (j) and an output one (k). We desire the target pattern t_k as output whenever we present as input the pattern p_i.

Through equations 2 and 3 we compute the activations of the hidden layer h_j, then the values for the output one o_k.

It is now possible to compute the error made by the network on pattern p:

$$E_p = \sum_k (t_k - o_k)^2 \tag{4}$$

In order to lower this error, it is necessary to change the connections between the output and hidden layer by the:

$$\begin{gathered} \delta_k = (t_k - o_k) o_k (1 - o_k) \\ \Delta w_{jk} = \eta \delta_k h_j \end{gathered} \tag{5}$$

while to modify the connections between input and hidden layer we use the:

$$\begin{gathered} \delta_j = h_j (1 - h_j) \sum_k (\delta_k w_{kj}) \\ \Delta w_{ij} = \eta \delta_j p_i \end{gathered} \tag{6}$$

The summation in equation 6 represents the actual back propagation of the error signal from the above layer, from which this algorithm takes its name. Here η is a "learning rate" of the system.

This procedure is repeated for each pattern in the learning set and repeated as well until the total error over all the patterns is small enough. At this point we possess an artificial neural network capable of correctly classify the training patterns.

It is possible to show that this algorithm performs a steepest descent optimisation over the total error function:

$$E = \frac{1}{2} \sum_p E_p = \frac{1}{2} \sum_p \sum_k (t_k - o_k)^2 \tag{7}$$

In order to have a faster convergence of the learning algorithm, we have also implemented the so called momentum term in the update of the weights:

$$\Delta w_{ij}^{new} = \Delta w_{ij} + \beta \Delta w_{ij}^{old} \tag{8}$$

this means that the current update will remember the direction of the previous one as scaled by the momentum rate $\beta < 1$.

The Parallel Algorithm

The parallel implementation of a backpropagation learning algorithm could be done in two ways:

1. one can parallelise on the neurons, giving subsections of the network to each processor, keeping the presentation of the training patterns serial,

2. one can create N clones of the network, one per processor, keeping them serial and parallelise on the presentation of the training patterns.

We have decided to implement the latter approach, see also [4], the main reasons for this being the following. First of all our typical target system is going to be an embedded one of small to medium size (i.e. in the hundreds of neurons) for real time operations on sensory input data (e.g. the vision sub system of a robotised vehicle). Hence the memory needs could be granted by such an approach having at our disposition a sufficiently large quantity of memory per processor. Secondly this approach represents the real backpropagation as originally introduced, while the serial presentation of the training patterns is an approximation introduced out of lack of parallel hardware (see [5]) and last but not least one of us already investigated it, see [7].

We are well aware, though, that this approach is memory limited, if one needs to implement a very large network it is possible to run out of local memory, since the number of connections grows approximately with the square of the number of neurons. But for our purposes, as above said, the hardware memory characteristics of Quadrics (1 Mword) is more than sufficient.

Implementation of a Back Propagation Learning Algorithm

We have exploited the features of the dynamic parser ZZ in order to create some new statements to automatically manage the declaration of and gain transparent access to the many arrays and variables needed to represent and train the neural network (e.g. connection weights, state values, etc.). In this way it has been possible to perform a linear and "clean" programming of the source code.

Presently, the TAO Neural Network Library is directed towards parallel training of Multi Layer Perceptrons via the Back Propagation learning algorithm.

This library is composed by a set of statements which can be accessed by the end user to create and manage the data structures of the neural network.

In order to get a flavour of the use of such a library, please have a look at figure 1, where a sample source code is presented.

In figure 1 is presented the code needed to program a three layer network composed of 20 input neurons, 20 hidden ones and 10 output ones, trained by ten input-output training pairs per processor. This network undergoes a full training session which spans for 1000 iterations. It has to be noted that there are no limits on the number of layers in the network.

The statement `network backprop` *`name (list of layer widths)`* `endnetwork` automatically creates, on each node of the machine, all the variables needed to encode the network state and to perform the training phase. All those variables are arrays whose contents are directly available to the end user through the syntax *`name`*`[`*`layer`*`].`*`object`*`[`*`index`*`]` where *`name`* is the network name, *`layer`* is the (zero based) layer number and *`object`* could be `weights`, `bias`, etc. The network output is available through the syntax *`name`*`[out][`*`index`*`]`.

```
/include "neural_network.hzt"
network backprop sergio
    20
    20
    10
endnetwork

real pattern[10,20]
real target[10,10]

!! read in training pairs
...

initialize sergio

do iter=1, 1000
  do npat=1, 10
    forward pass sergio from pattern[npat,0]
    backprop pass sergio against target[npat,0]
  enddo
  update sergio
enddo
```

Fig. 1. The source code for a 20-20-10 Multi Layer Perceptron with ten training pairs.

A set of statements allows the user to perform operations on the network. Those statements are automatically expanded by the parser in the most computationally efficient way, thanks to the topology information collected in the declaration phase. In particular, aggressive loop unrolling and code inlining is performed automatically, in a manner suitable to the peculiarities of the machine architecture. As a consequence, the user is relieved of the burden of code optimisation.

A very high level statement, `forward pass` *`network`* `from` *`input vector`*, expands incline all the above referred statements to perform a single classification pass through a network. The statement `backward pass` *`network`* `against` *`target vector`*, performs a single back propagation step, accumulating for every layer the corrections to weights and biases.

All the above operations are completely local to the Quadrics nodes, allowing parallel classification and training of a network on different input vectors. On the contrary, the update phase performs the global sum of the corrections computed on all the nodes, and applies them to each network replica. This step is the only one requiring communications, which are performed, in the most effective way, via three loops along the axes of the nodes grid.

Finally, the name of a network can be directly used in the standard TAO I/O statements, to load or save the full network state.

Experimental Measures

In order to experimentally measure the performances of our library, we had at our disposition three different machines of the Quadrics series, a Q1, a Q4 and a QH4, respectively composed of 8 (2 by 2 by 2), 32 (2 by 2 by 8) and 512 (8 by 8 by 8) processors.

To better test our implementation, we have made a theoretical model. The algorithm is composed of two distinct phases. In the first one all the forward pass and backward pass computations are performed locally by each clone of the network inside each processor. In the second all the computed corrections on all the relevant quantities are broadcasted among all the processors and the update is performed.

In other words we expect that the computational time needed to perform a training session is composed of two different terms, a term τ_{fb} for the forward and backward passes on a set of training examples and a term τ_u for the update step, i.e.

$$T = \tau_{fb} + \tau_u \tag{9}$$

For the chosen implementation, both terms must depend on the network structure. However, while the first term must depend linearly on the number of training examples, the second one must depend on the machine topology.

The update step adds up contributions from all the machine nodes by three loops along the axes spanning the processor grid, and then corrects the network weights and biases. On a $N_x \times N_y \times N_z$ machine, the number of communication substeps needed to collect all the corrections along one machine direction is $N_\alpha - 1$. Hence the update term becomes:

$$\tau_u = (N_x + N_y + N_z - 3)k - \tau_o \tag{10}$$

where τ_0 is the time needed for the actual update of the weights plus some needed bookkeeping. τ_0 and k must depend only on the network structure.

All of the following is based on the timing measures made on a first test network composed of 20 input neurons, 20 hidden ones and 10 output neurons (a total of 630 connections) and on a second 20-20-20-10 more complex one (with a total of 1050 weights).

As a first measure we present the training time performances of this network as a function of the number of training patterns per processor. In figure 2 are shown the time needed to train for 1000 iteration a 20-20-10 network as a function of the training patterns on the three different machines.

The linear behaviour is evident. If we fit the data with a linear regression, we obtain different intercepts but equal slopes for the three different plots, with an extremely high correlation term. These results show us that τ_{fb} really depends linearly only on the number of training examples, testifying that we are actually running a scaling code on a SIMD parallel computer, while the term τ_u is represented by the different intersections. For the details see table 1.

If we fit the three different intersections as a function of the different number of communication substeps due to different number of processors, as in equation 10, we find that there is a linear relation with high correlation, confirming indeed our theoretical model. We obtain a $\tau_0 = 379 \pm 2$ and a $k = 980 \pm 30$.

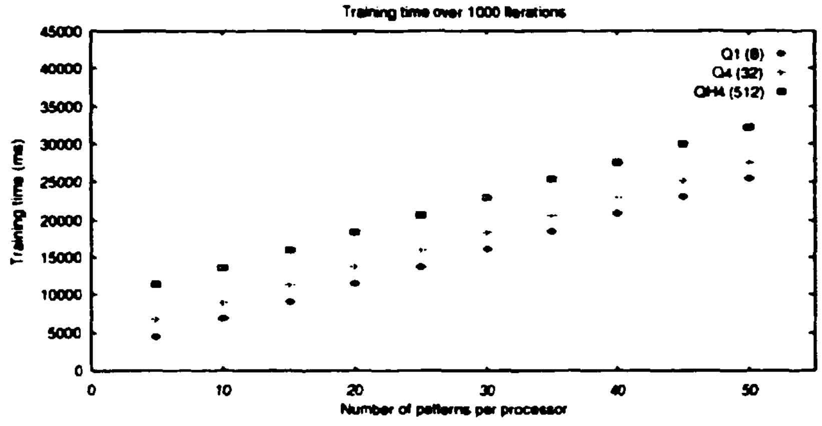

Fig. 2. The total training time for 1000 iterations of the Back Propagation algorithm as a function of the number of patterns per processor on the 20-20-10 network.

HW (Processors)	Slope	Intercept
Q1 (8)	467±1	2135±1
Q4 (32)	464±1	4373±1
QH4 (512)	467±1	8958±1

Table 1. The results of the linear regression on the data plotted in figure 1.

This set of measures has been repeated with the second network, on the Q1, Q4 and the QH4 machines, yielding the same behaviour but, naturally, different figures, since the network structure has changed. The results are presented in figure 3 and table 2.

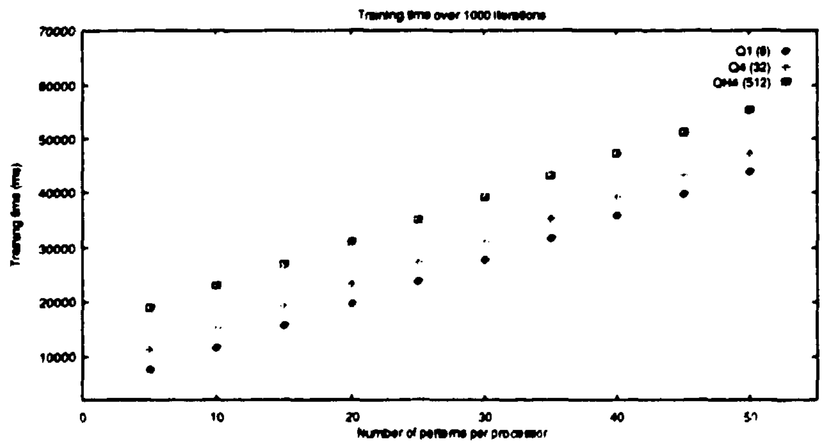

Fig. 3. The total training time for 1000 iterations of the Back Propagation algorithm as a function of the number of patterns per processor on the 20-20-20-10 network.

HW (Processors)	Slope	Intercept
Q1 (8)	807±1	3548±1
Q4 (32)	802±1	7260±1
QH4 (512)	807±1	14887±1

Table 3. The results of the linear regression on the data plotted in figure 2.

Proceeding as above, for this second network the fit to equation 10 yields $\tau_0=630 \pm 3$ and a $k = 1660 \pm 30$.

In the literature several different neural network implementations can be found. These systems have performances that are usually measured in terms of connection updates per second (*cups*) which, in the training mode, span in the range from 0.016 to 16 Mcups [6] and [8] apart from an isolated value of 90 Mcups for the *dedicated* machine NETSIM of the Texas Instruments and Cambridge University [6].

The performance reached through our TAO Neural Network Library greatly depends on the network structure, on the machine used and on the number of training examples per processor. In any case we can provide a range of performance for our implementation.

The lower end of this range is represented by a figure of *1.94 Mcups*, obtained on a Q1 machine (8 processors) with only one training example per processor on a 20-20-10 network. The upper end on the same network structure is represented by a figure of *499.09 Mcups*, obtained on a QH4 machine (512 processors) with 50 examples per processor. Using the second, more complex, network we obtained respectively *1.92 Mcups* on a Q1 and *486.62 Mcups* on a QH4 machine.

Conclusions

The results show that the discussed implementation of the Multi Layer Perceptron trained by Backpropagation on the Quadrics parallel supercomputers attains extremely interesting performance. As a consequence, the well known burden of the accurate tuning of parameters and topology of a neural network can be greatly reduced.

The TAO Neural Network Library will be extended in the near future.

First of all, the additional storage for the support to the training phase will be declared separately from the bare network, to avoid unnecessary memory overheads in the use of previously trained networks in a Quadrics application.

Secondly, extension of the syntax to other neural network architectures is envisaged to widen the scope of the present library.

The capability of independent parallel training of the same network with different learning rates or training sets will also be considered, together with other minor adjustments to the whole library.

Acknowledgements

Authors are indebted to Dr. Gianmarco Todesco of the Rome INFN section, for precious suggestions and insights in the ZZ parser, through which the library has been developed and to Dr. Raffaele Tripiccione of the Pisa INFN section for useful discussions.

References

[1] A. Bartoloni et al., "A hardware implementation of the APE100 architecture", 1993, *International Journal of Modern Physics C*, **4**, No. 5, 969-976.

[2] A. Bartoloni et al., "The software of the APE100 processor", 1993, *International Journal of Modern Physics C*, **4**, No. 5, 955-967.

[3] J. L. Mc Clelland, D. E. Rumelhart, "Parallel Distributed Processing", 1987, MIT Press: Cambridge, Massachussets, U.S.A..

[4] S. L. Hung, H. Adeli, "Parallel backpropagation learning algorithms on CRAY Y-MP8/864 supercomputer", 1993, *Neurocomputing*, **5**, 287.

[5] S. I. Gallant, "Neural Network Learning and Expert Systems", 1993, MIT Press: Cambridge, Massachussets, U.S.A..

[6] E. J. H. Kerckhoffs, F.W. Wedman, E.E.E. Frietman, "Speeding up backpropagation training on a hypercube computer", 1992, *Neurocomputing*, **4**, 43.

[7] S. Taraglio, "Note sui primi esperimenti di supercalcolo parallelo in ENEA: le Reti Neurali", 1994, ENEA Technical Report, RT/INN/94/41.

[8] D.A. Pomerlau, G.L. Gusciora, D.S. Touretzky, H.T. Kung, "Neural network simulation at Warp speed: How we got 17 million connections per second", 1988, *Proceedings of IEEE International Joint Conference on Neural Networks*, San Diego, CA.

Edito dall'Enea, Direzione Relazioni Esterne
Viale Regina Margherita, 125 - Roma
Finito di stampare nel mese di agosto 1995
presso il Laboratorio Tecnografico

L T347

95 12 14