ELSEVIER

Microprocessing and Microprogramming 41 (1996) 757–769

Microprocessing and Microprogramming

Mixing floating- and fixed-point formats for neural network learning on neuroprocessors

Davide Anguita [a,*], Benedict A. Gomes [b,1]

[a] *Univ. of Genova, D.I.B.E., Via Opera Pia 11a, 16145 Genova, Italy*

[b] *Int. Comp. Science Inst., 1947 Center St., Berkeley, CA, USA*

Received 1 March 1995; revised 11 October 1995; accepted 18 January 1996

Abstract

We examine the efficient implementation of back-propagation (BP) type algorithms on T0 [3], a vector processor with a fixed-point engine, designed for neural network simulation. Using Matrix Back Propagation (MBP) [2] we achieve an asymptotically optimal performance on T0 (about 0.8 GOPS) for both forward and backward phases, which is not possible with the standard on-line BP algorithm. We use a mixture of fixed- and floating-point operations in order to guarantee both high efficiency and fast convergence. Though the most expensive computations are implemented in fixed-point, we achieve a rate of convergence that is comparable to the floating-point version. The time taken for conversion between fixed- and floating-point is also shown to be reasonably low.

Keywords: Neural networks; Neuroprocessors; Fixed-point format

1. Introduction

Among the large number of dedicated VLSI architectures for neural networks developed in recent years, several of the most successful proposals have regarded digital implementations. Most of these dedicated processors are oriented toward the efficient execution of various learning algorithms with a strong accent on back-propagation (BP). Some well-known examples in this field are CNAPS [13], Lneuro [18], MA-16 [20], and SPERT [28]: they are the building blocks for larger systems that exploit massive parallelism to achieve performances orders of magnitude greater than conventional workstations [21,4]. The common characteristic of these processors is the use of a fixed-point engine, typically 16 bits wide or less, for fast computation.

The drawback for the final user who wants to implement an algorithm for neural network learning on

* Corresponding author. Email: anguita@dibe.unige.it.

[1] Email: gomes@icsi.berkeley.edu

PII S0165-6074(96)00012-9

Table 1
The MBP algorithm

Pseudo-code		# of operations	Point
/* *Feed-forward* */			
for $l := 1$ **to** L			
$\mathbf{S}_l := \mathbf{S}_{l-1} \cdot \mathbf{W}_l$	(1.1)	$2N_P N_l N_{l-1}$	fixed
$\mathbf{S}_l := \mathbf{S}_l + \mathbf{b}_l \cdot \mathbf{1}^T$	(1.2)	$N_P N_l$	fixed
$\mathbf{S}_l := f\{\mathbf{S}_l\}$	(1.3)	$N_P N_l k_1$	fixed
/* *Error back-prop* */			
$\Delta_L := \mathbf{T} - \mathbf{S}_L$	(2.1)	$N_P N_L$	floating
$\Delta_L := \Delta_L \times g\{\mathbf{S}_L\}$	(2.2)	$N_P N_L(1 + k_2)$	floating
for $l := L - 1$ **to** 1			
$\Delta_l := \Delta_{l+1} \cdot \mathbf{W}_{l+1}^T$	(2.3)	$2N_P N_{l+1} N_l$	fixed
$\Delta_l := \Delta_l \times g\{\mathbf{S}_l\}$	(2.4)	$N_P N_l(1 + k_2)$	fixed
/* *Weight variation* */			
for $l := 1$ **to** L			
$\Delta\mathbf{W}_l^{new} := \mathbf{S}_{l-1}^T \cdot \Delta_l$	(3.1)	$2N_P N_l N_{l-1}$	fixed
$\Delta\mathbf{b}_l^{new} := \Delta_l \cdot \mathbf{1}$	(3.2)	$N_P N_l$	fixed
$\Delta\mathbf{W}_l^{new} := \eta\Delta\mathbf{W}_l^{new} + \alpha\Delta\mathbf{W}_l^{old}$	(3.3)	$3N_l N_{l-1}$	floating
$\Delta\mathbf{b}_l^{new} := \eta\Delta\mathbf{b}_l^{new} + \alpha\Delta\mathbf{b}_l^{old}$	(3.4)	$3N_l$	floating
/* *Weight update* */			
for $l := 1$ **to** L			
$\mathbf{W}_l := \mathbf{W}_l + \Delta\mathbf{W}_l^{new}$	(4.1)	$N_l N_{l-1}$	floating
$\mathbf{b}_l := \mathbf{b}_l + \Delta\mathbf{b}_l^{new}$	(4.2)	N_l	floating

this kind of processors is the fixed-point format that requires greater attention during the implementation, compared with a conventional floating-point format. This is not a new problem, in fact both analog and digital implementations of neural networks suffer from some constraint due to physical limitations. For this reason, the effect of discretization on feed-forward networks and back-propagation learning received some attention shortly after the introduction of the algorithm [10,5,15]. Most of the results indicate that a representation of 16 bits for the fixed-point format is reliable enough to obtain reasonable results with on-line backpropagation. On the other hand, despite this general agreement, there has been some effort to reduce the precision needed during the computation [14,22], mainly because the effect of the discretization during learning is not completely understood and it seems to be both problem and algorithm dependent. In fact, there are many variations of the BP algorithm and each of them can show different sensitivity to the approximations caused by the fixed-point arithmetic, leading to different convergence problems. Some theoretical results on the precision issue have been found [1,23], but often they rely on difficulty to predict parameters (e.g. the number of iterations to convergence).

One solution to overcome these limitations is to mix conventional floating-point operations with fixed-point operations when required. An example of this approach is [12] where the feed-forward and the backward phase of the algorithm are computed in fixed- and floating-point format respectively. However, this solution does not address the efficiency issue because the most computationally expensive part of the algorithm (the backward phase) is still performed in floating-point format, losing all the advantages of a fast fixed-point engine.

We show here a mixed floating/fixed-point implementation of Matrix Back Propagation (MBP) [2] that isolates the most computationally expensive steps of the algorithm and implements them efficiently in fixed-point format. Other parts of the algorithm with less demand in terms of computational power but with more critical needs in terms of accuracy are implemented in conventional floating-point format. The target architecture is the neuroprocessor T0, but the method is of general validity.

Despite the need for conversions between the two formats and the simulation of the floating-point operations in software, good performances are obtainable with reasonably large networks, showing a high efficiency in exploiting the T0 hardware.

The following section describes the learning algorithm implemented. Section 3 describes the mixed floating/fixed-point approach. Section 4 summarizes the main characteristics of T0. Section 5 shows the implementation details and performance evaluation and Section 6 compares the effect of the mixed approach with the standard algorithm.

2. Matrix back propagation

In Table 1 the MBP algorithm is summarized. It can be used to represent several BP learning algorithms with adaptive step and momentum [26,27]. The second column of the table contains the number of operations needed by each step. The third column indicates if the computation for each step is performed in fixed- or floating-point format (this choice will be explained in the following section). Bold letters indicate vectors or matrices.

We assume that our feed-forward network is composed of L layers of N_l neurons, with $0 \leq l \leq L$. The weights for each layer are stored in matrices $\mathbf{W}_l$ of size $N_l \times N_{l-1}$ and the biases in vectors $\mathbf{b}_l$ of size N_l.

The learning set consist of N_P patterns. Input patterns are stored in matrix $\mathbf{S}_0$ in row order and target patterns similarly in matrix $\mathbf{T}$. The order of storing is particularly important for the efficiency of the implementation: if the patterns are stored in row order, the elements of each pattern lie in consecutive memory locations and can be accessed with no performance penalty on the vast majority of current processor architectures including T0. Matrices $\mathbf{S}_1, \ldots, \mathbf{S}_L$ contain the output of the corresponding layer when $\mathbf{S}_0$ is applied to the input of the network. The size of $\mathbf{S}_l$ is $N_P \times N_l$ and the size of $\mathbf{T}$ is $N_P \times N_L$.

The back-propagated error is stored in matrices Δ_l of size $N_P \times N_l$ and the variations of weights and biases computed at each step are stored respectively in matrices $\Delta\mathbf{W}_l$ of size $N_l \times N_{l-1}$ and vectors $\Delta\mathbf{b}_l$ of size N_l. For simplicity, connections between non-consecutive layers are not considered.

The total number of operations of MBP is

$$n_{op}^{MBP} = 2N_P\left(3\sum_{l=1}^{L} N_l N_{l-1} - N_1 N_0\right) \tag{5}$$

$$+(3+k_1+k_2)N_P\sum_{l=1}^{L} N_l + 4\sum_{l=1}^{L} N_l N_{l-1} - N_P N_L \tag{6}$$

$$+4\sum_{l=1}^{L} N_l, \tag{7}$$

where k_1 and k_2 are respectively the number of operations needed for the computation of the activation function of the neurons and its derivative. If the activation function is the usual sigmoid, then $k_2 = 2$.

On a conventional RISC, if each operation is completed in a single cycle, the total computational time is $T \propto n_{cycles} = n_{op}$. On vector or multi-ALU processors like T0, the expected time is $T \propto n_{cycles} = n_{op}/P$, where P is the number of ALUs. Obviously the implicit assumptions are: (a) there is no additional cost to load or store the data in memory, (b) one instruction can be issued every cycle, and (c) the order in which the operations are issued allows a complete exploitation of the ALUs. It has already been shown [2] that with a relatively small effort these constraints can be satisfied reasonably well on some RISCs. In Section 4 we will address this problem for T0.

3. The neuroprocessor T0

T0 belongs to the family of neuroprocessors with fast fixed-point capabilities and it will be the first implementation of the Torrent architecture [3]. It is tailored for neural-networks calculations and inherits some of the features of a previous neuro-processor [28]. The next implementation (T1) will be the building block for a massively parallel neuro-computer [4].

In particular, T0 is composed of a standard MIPS-II RISC engine [17] with no floating-point unit but with a fixed-point vector unit that can execute up to two operations per cycle on 8-word vectors, or, in other words, compute 16 results in a single cycle. This translates to a peak performance of 0.8 GOPS (Giga Operations per Second) if the processor is clocked at 50MHz, or approximately 0.2 GCUPS (Giga Connection Updates per Second) for one hidden layer networks, a result comparable to supercomputer implementations. Fig. 1 summarizes the architecture of the vector unit. The two 8-word ALUs are VP0 and VP1 connected to the 32-bit vector register bank. Each vector register contains 32 elements, therefore each ALU can execute an operation on a complete vector in 4 cycles. The data path to/from the memory is 128 bits wide allowing the loading/storing of eight 16-bit words in a single cycle.

4. The mixed format algorithm

We will explain here in detail the choice of the format for each step of the algorithm. The main idea is to perform the most computational expensive part of the algorithm in fixed-point and resort to floating-point only where the computation must be particularly accurate.

Using Table 1 we can observe that the more expensive steps are (1.1), (2.3) and (3.1). They require $O(n^3)$ operations (where n is in general the size of the problem), therefore they will be performed in fixed-point. Note that matrix $\mathbf{S}_0$ that contains the input patterns is likely to be already in fixed-point format in real-world applications, deriving, for example, from an A/D conversion. Step (1.2) can be easily computed in the same way.

Step (1.3) requires a function computation. With the use of the fixed-point format, this can be substituted with an indexed load from a table where the values of the functions are pre-stored.

Before starting the error-back propagation, we can translate the output of the network to floating-point in order to have an accurate computation of the error (2.1) and its derivative (2.2). The interesting side-effect of performing these operations in floating-point is that we know (after step (2.2)) the numeric range of the error, therefore it is possible to choose a good fixed-point representation for the subsequent steps.

The next conversion is performed before step (3.3) and (3.4) in order to compute with great accuracy the variation of the weights and biases of the network. Note that both η (the learning step) and α (the momentum term) are in general floating-point variables.

To summarize the algorithm: the conversion from fixed- to floating-point format must be performed at the end of the forward phase on matrix $\mathbf{S}_L$ and at the end of the backward phase on $\Delta\mathbf{W}_l$ and $\Delta\mathbf{b}_l$. The conversion from floating- to fixed-point format must be performed at the beginning of the forward phase on each $\mathbf{W}_l$ and $\mathbf{b}_l$ and at the beginning of the backward phase on Δ_L.

5. Optimal implementation on the T0 neuroprocessor

If the implementation of an algorithm on T0 is optimal, in the sense that it can completely exploit its hardware, we can expect to have $n_{cycles} = n_{op}/16$. For

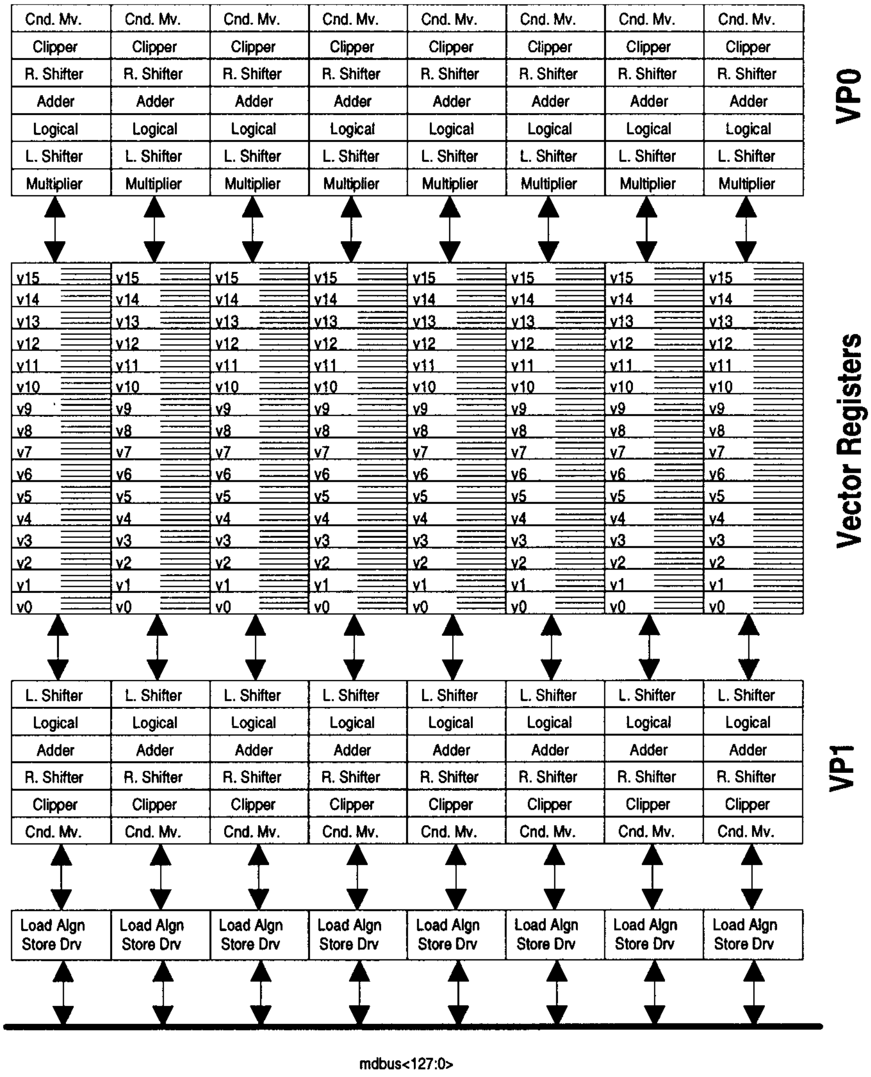


Fig. 1. Simplified architecture of the Vector Unit of T0.

this reason, we will refer to an algorithm as *asymptotically optimal* for T0 (or simply *optimal*) if the efficiency ϵ of its implementation goes to 1 as the size of the problem (N_P,N_l) grows. In other words: $\epsilon = n_{op}/16n_{cycles} \rightarrow 1$.

Our purpose is to show that MBP can be implemented optimally in this sense, even though some of the computations are done in floating-point and must be simulated in software.

As mentioned before, the computational load be-

Table 2
Scalar and vectorized matrix products

	Scalar matrix products	Vectorized matrix products
$\mathbf{S}_l = \mathbf{S}_{l-1} \cdot \mathbf{W}_l$	**for** $i := 0$ **to** $N_P - 1$ $\quad$ **for** $j := 0$ **to** $N_{l-1} - 1$ $\quad\quad$ **for** $k := 0$ **to** $N_l - 1$ $\quad\quad\quad s^l_{i,j} \mathrel{+}= s^{l-1}_{i,k} * w^l_{k,j}$	**for** $j := 0$ **to** $N_{l-1} - 1$ **step** V_L $\quad$ **for** $i := 0$ **to** $N_P - 1$ **step** U $\quad\quad$ **for** $k := 0$ **to** $N_l - 1$ $\quad\quad\quad s^l_{i,[j,j+V_L]} \mathrel{+}= s^{l-1}_{i,k} * w^l_{k,[j,j+V_L]}$ $\quad\quad\quad s^l_{i+1,[j,j+V_L]} \mathrel{+}= s^{l-1}_{i+1,k} * w^l_{k,[j,j+V_L]}$ $\quad\quad\quad \vdots$ $\quad\quad\quad s^l_{i+U-1,[j,j+V_L]} \mathrel{+}= s^{l-1}_{i+U-1,k} * w^l_{k,[j,j+V_L]}$
$\Delta_l = \Delta_{l+1} \cdot \mathbf{W}^T_{l+1}$	**for** $i := 0$ **to** $N_P - 1$ $\quad$ **for** $j := 0$ **to** $N_{l+1} - 1$ $\quad\quad$ **for** $k := 0$ **to** $N_l - 1$ $\quad\quad\quad \delta^l_{i,j} \mathrel{+}= \delta^{l+1}_{i,k} * w^{l+1}_{j,k}$	**for** $i := 0$ **to** $N_P - 1$ **step** V $\quad$ **for** $j := 0$ **to** $N_{l+1} - 1$ **step** V $\quad\quad$ **for** $k := 0$ **to** $N_l - 1$ **step** V_L $\quad\quad\quad \delta^l_{i,j} \mathrel{+}= \delta^{l+1}_{i,[k,k+V_L]} * w^{l+1}_{j,[k,k+V_L]}$ $\quad\quad\quad \vdots$ $\quad\quad\quad \delta^l_{i+V,j+V} \mathrel{+}= \delta^{l+1}_{i+V-1,[k,k+V_L]} * w^{l+1}_{j+V-1,[k,k+V_L]}$
$\Delta\mathbf{W}_l = \mathbf{S}^T_{l-1} \cdot \Delta_l$	**for** $i := 0$ **to** $N_{l-1} - 1$ $\quad$ **for** $j := 0$ **to** $N_l - 1$ $\quad\quad$ **for** $k := 0$ **to** $N_P - 1$ $\quad\quad\quad \Delta w^l_{i,j} \mathrel{+}= s^{l-1}_{k,i} * \delta^l_{k,j}$	**for** $j := 0$ **to** $N_l - 1$ **step** V_L $\quad$ **for** $i := 0$ **to** $N_{l-1} - 1$ **step** U $\quad\quad$ **for** $k := 0$ **to** $N_P - 1$ $\quad\quad\quad \Delta w^l_{i,[j,j+V_L]} \mathrel{+}= s^{l-1}_{k,i} * \delta^l_{k,[j,j+V_L]}$ $\quad\quad\quad \Delta w^l_{i+1,[j,j+V_L]} \mathrel{+}= s^{l-1}_{k,i+1} * \delta^l_{k,[j,j+V_L]}$ $\quad\quad\quad \vdots$ $\quad\quad\quad \Delta w^l_{i+U-1,[j,j+V_L]} \mathrel{+}= s^{l-1}_{k,i+U-1} * \delta^l_{k,[j,j+V_L]}$

longs to steps (1.1), (2.3) and (3.1). To compute these steps, three matrix multiplications must be performed: (1.1) is a conventional matrix product, (2.3) is a matrix product with the second matrix transposed and (3.1) is a matrix product with the first matrix transposed.

The three operations are shown in pseudo-code in the second column of Table 2. The third column shows the vectorized versions. V_L is the vector register length (32 in the current implementation of T0) and U, V are the unrolling depth needed to fill the processor pipelines.

The increase of the unrolling depth shifts the balance of the loop from memory-bound to CPU-bound, therefore extra cycles are available for the memory port to load (store) the operands while the processor is computing the arithmetic operations. The unrolling depth is limited by the number of registers available for storing intermediate results: in our case $U = 8$ and $V = 2$.

As can be easily noted, the vectorized version performs its vector references to each matrix in row order to exploit the memory bandwith of T0. In fact, the use of stride-1 access to the memory allows the processor to load an entire 8-word vector (of 16 bits) in a single cycle, while a generic stride-n access to memory ($n > 1$) requires one cycle per element.

We will assume in the following text that all the matrix dimensions are a multiple of V_L. If this is not the case, there is some overhead due to an underutilization of the vector unit, but it does not affect the asymptotical behavior of the implementation. For an

Table 3
Number of cycles for MBP on T0 in the general case

Step	n_{cycles}
(1.1)	$(4N_lU + 4U)\lceil N_P/U\rceil \lceil N_{l-1}/V_L\rceil$
(1.2)	$\left(8\lceil N_l/V_L\rceil + 1\right) N_P$
(1.3)	$1.5N_P\lceil N_l/V_L\rceil V_L$
(2.1)	$k_fN_P\lceil N_L/V_L\rceil V_L$
(2.2)	$3k_fN_P\lceil N_L/V_L\rceil V_L$
(2.3)	$\left(4\lceil N_l/V_L\rceil V^2 + 20 + V^2\right)\lceil N_P/V\rceil \lceil N_{l+1}/V\rceil$
(2.4)	$12\lceil N_l/V_L\rceil N_P$
(3.1)	$(4N_PU + 4U)\lceil N_l/V_L\rceil \lceil N_{l-1}/U\rceil$
(3.2)	$(4N_P + 4)\lceil N_l/V_L\rceil$
(3.3)	$3k_fN_l\lceil N_{l-1}/V_L\rceil V_L$
(3.4)	$3k_f\lceil N_l/V_L\rceil V_L$
(4.1)	$k_fN_l\lceil N_{l-1}/V_L\rceil V_L$
(4.2)	$k_f\lceil N_l/V_L\rceil V_L$

Table 4
Number of cycles for optimized matrix multiplications

Step	n_{cycles}
(1.1)	$\frac{1}{8}N_PN_lN_{l-1} + \frac{1}{8}N_PN_{l-1}$
(2.3)	$\frac{1}{8}N_PN_lN_{l+1} + 6N_PN_{l+1}$
(3.1)	$\frac{1}{8}N_PN_lN_{l-1} + \frac{1}{8}N_lN_{l-1}$

exact computation of the number of cycles in the general case, the reader can refer to Table 3.

Table 4 shows the number of cycles needed by T0 to perform the optimized matrix multiplications.

Step (1.1) requires four cycles in the inner loop to compute a single vector multiplification/addition and four cycles to store each result back in memory at the end of the loop. The load of element $s_{i,k}$ can be overlapped with the computation thanks to the unrolling of the external loop. It is easy to prove the optimality of (1.1):

$$\epsilon^{(1.1)} = \frac{n_{op}^{(1.1)}}{16n_{cycles}^{(1.1)}} = \frac{2N_PN_lN_{l-1}}{16\left(\frac{1}{8}N_PN_lN_{l-1} + \frac{1}{8}N_PN_{l-1}\right)}$$
$$= \frac{1}{1 + 1/N_l} \rightarrow 1. \qquad (8)$$

The second product (2.3) can be seen as a sequence of dot-products. This operation is not directly implemented on T0 and needs ~ 20 cycles for a vector of $V_L = 32$ words. This problem is known and could be eventually solved in future releases of the processor [3]. In any case, the overhead due to the absence of the dot-product is not particularly annoying when dealing with matrix products: in fact, partial dot-products of length V_L can be kept in vector registers and the final result can be computed at the end of the inner loop. Note that matrix–vector products (used in the standard BP algorithm) would suffer from a bigger overhead, in this case the absence of an implemented dot-product appears only in the second-order term and becomes negligible for large problems.

The third product (3.1) is similar to (1.1), but with a different order of the loops.

Other steps performed in fixed-point format are: the computation of the output of each neuron through its activation function (1.3), the computation of its derivative in the internal layers (2.4), the bias addition in the feed-forward phase (1.2) and the bias computation in the backward phase (3.2).

The computation of the activation function is quite expensive if it is done using a floating-point math library [11], and it would cause a large penalty on T0 due to the absence of a floating-point unit. Yet, if (1.3) is performed in fixed-point format, the activation function can be easily computed using a look-up table of size 2^B where B is the number of bits of the fixed-point format [6]. The vector unit of T0 is provided with a vector instruction to perform indexed load, so the number of cycles needed to compute the value using the table is only ~ 1.5/element.

The pseudo-code for steps (1.2), (2.4) and (3.2) is shown in Table 5. The three loops are memory-bounded, therefore the number of cycles is easy to compute (assuming sufficient unrolling). All the other steps are done in floating-point format.

Let us consider now the overhead due to the conversion of the matrices from floating- to fixed-point format and vice versa. The scalar conversion takes ~ 46 cycles/element on T0, but it is possible to lower this

Table 5
Other vectorized operations

Step	Pseudo-code	n_{cycles}

(1.2)	**for** $i := 0$ **to** $N_P - 1$
	  for $j := 0$ **to** N_l **step** V_L
	    $s^l_{i,[j,j+V_L]} += b^l_i$
		$\frac{1}{4}N_l N_P + N_P$

(2.4)	**for** $i := 0$ **to** $N_P - 1$
	  for $j := 0$ **to** $N_l - 1$ **step** V_L
	    $\delta^l_{i,[j,j+V_L]} = \delta^l_{i,[j,j+V_L]} * (1 - s^l_{i,[j,j+V_L]} * s^l_{i,[j,j+V_L]})$
		$\frac{3}{8}N_l N_P$

(3.2)	**for** $j := 0$ **to** N_l **step** V_L
	  for $i := 0$ **to** N_P
	    $b^l_{[j,j+V_L]} += \delta^l_{i,[j,j+V_L]}$
		$\frac{1}{8}N_P N_l + \frac{1}{4}N_l$

number using the vector unit. For vectors between 100 and 1000 elements, the translation from floating-point to fixed-point format requires only $k_{fx} = 2.6 \leftrightarrow 1.8$ cycles/element and $k_{xf} = 3.6 \leftrightarrow 2.5$ cycles/element for the inverse conversion (these figures have been measured experimentally).

The total number of cycles needed for the conversions is

$$n^{conv}_{cycles} = (k_{xf} + k_{fx}) \left[\sum_{l=1}^{L} N_l(N_{l-1} + 1) + N_P N_L \right]. \tag{9}$$

T0 does not implement the floating-point unit of the MIPS architecture, so the floating-point operation must be simulated in software. Currently the RISC core is used to perform the simulation, but an IEEE compatible floating-point library that uses the vector unit is under development and the expected performance will be in the range of 10 ↔ 50 cycles/element. Then the number of cycles for the floating-point steps of the algorithm will be $n^{flp}_{cycles} = k_f n^{flp}_{op}$, with $k_f \in [10, 50]$.

We now have all the elements to compute the number of cycles needed by T0 to execute MBP,

$$n^{MBP}_{cycles} = \frac{N_P}{8}\left(3\sum_{l=1}^{L} N_l N_{l-1} - N_1 N_0\right) \tag{10}$$

$$+\frac{67}{8}N_P \sum_{l=1}^{L} N_l + \left(4k_f + k_{fx} + k_{xf} + \frac{1}{8}\right) \sum_{l=1}^{L} N_l N_{l-1} \tag{11}$$

$$+ \left(4k_f + k_{fx} + k_{xf} - \frac{1}{2}\right) N_P N_L - 6 N_P N_1 + \frac{N_P N_0}{8} \tag{12}$$

$$+L N_P + \left(4k_f + k_{fx} + k_{xf} + \frac{1}{8}\right) \sum_{l=1}^{L} N_l. \tag{13}$$

If we compare the $O(n^3)$ term (10) with the corresponding term for n^{MBP}_{op}, we can deduce easily the optimality of this implementation of MBP.

Obviously, the asymptotical behavior of MBP on T0 is not of primary importance when dealing with real-world applications. It is interesting therefore to analyze the second- (11), (12) and first-order (13) terms of the above expression.

First of all, we note that the overhead due to the conversions from fixed- to floating-point and vice-versa depends mainly on the size of the network and only marginally on the dimension of the training set, as can be seen from the second term of (11) and the first term of (12). The dependence from the size of the training set is controlled by the number of neurons of the output layer (N_L), so we expect better performance when dealing with networks with a small number of outputs (e.g. classification problems, as opposed to encoding problems [8]). If this is not the case, some techniques to reduce the number of output neurons in

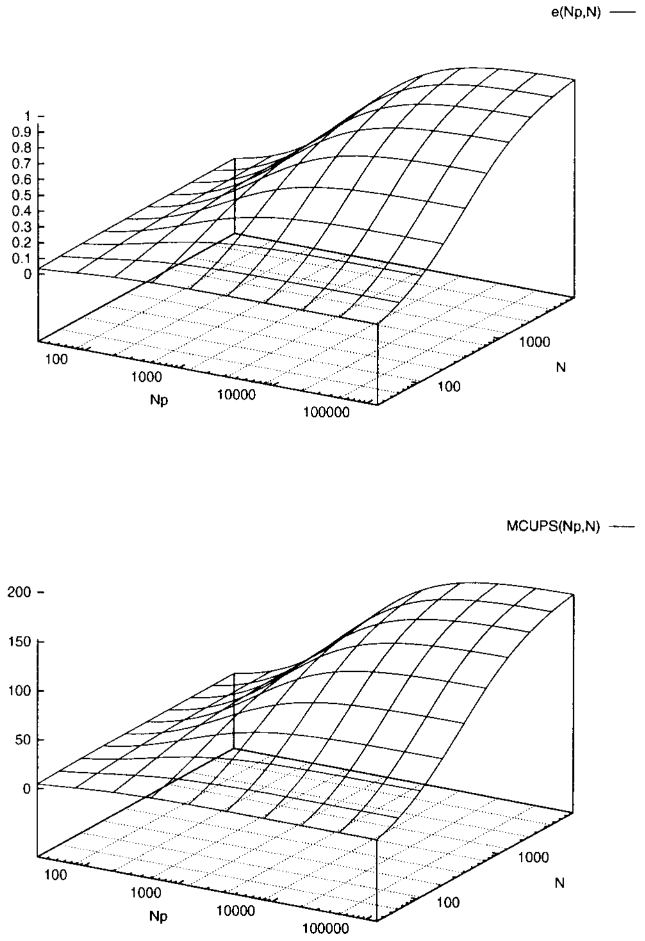


Fig. 2. Efficiency and performance (MCUPS) of MBP on T0.

classification problems can be applied [19].

There is also an explicit dependence in the first-order term (13) on the number of layers of the network (L). This term is of small importance being of first order, but we can expect an increase of overhead in networks with a very large number of layers. However, this is not a common case, as a large number of layers is not theoretically justified [9] and practical applications seldom require more than four layers (see, for example, [16] for a real problem that requires such an architecture).

To sketch the behavior of MBP on T0, we can simplify both the expressions for n_{op} and n_{cycles} assuming $\forall l \; N_l \approx N$ and plot the efficiency and the performance

Table 6
Some real-world applications

Name	Network size				Description
	N_0	N_1	N_2	N_3	
NETtalk [24]	203	80	26	–	Pronunciation of text
Neurogammon [25]	459	24	24	1	Backgammon player
Speech [7]	234	1000	69	–	Speech recognition

in MCUPS (Fig. 2) as functions of the size of the training set (N_P) and the network (N).

We assume $k_1 = 6$ to compute n_{op} (as suggested in [11]) and the worst case for floating-point and conversion routines on T0 ($k_f = 50$, $k_{fx} = 4$, $k_{xf} = 3$) to compute n_{cycles}. The asymptotic performance is 160 MCUPS; obviously, the asymptotic performance of a generic RISC processor with the same clock and only one FPU would be 10 MCUPS.

Fig. 2 allows us to easily understand the behavior of the implementation, but it is of little practical use due to the peculiar network architecture. For this reason we show here the performance of MBP on T0 with networks that have been used in some real-world applications (Table 6).

Fig. 3 summarizes the performance for the applications mentioned above. It is interesting to note that, for all problems, the number of patterns for which half of the peak performance is attained ($n_{1/2}$) is reasonably small ($N_P \sim 500$).

6. Learning with the mixed format algorithm

To test the effectiveness of the mixed format algorithm we chose the speech recognition problem described in the previous section.

Fig. 4 shows the learning on a subset of the speech database with different ranges of the fixed-point variables. In particular, E is the exponent of the most significant digit of the fixed-point format. With 16-bit words we can represent values in the range $[-2^E, 2^E - 2^{E-15}]$.

It is clear that the error back propagation is quite sensitive to the range of the fixed-point format. If the fixed-point representation is too coarse (e.g. $E = 2$), the algorithm tends to get stuck due to the underflow of the back-propagated error. However, thanks to the use of the mixed format, it is possible to choose a good range for the fixed-point variables before starting the error back propagation, because the error computation in the last layer is done in floating-point format. The choice of the correct range can easily be done looking at the largest floating-point value. In this case, the learning with mixed format is comparable to the learning in floating-point format in terms of number of learning steps but, of course, far more efficient from a computational point of view.

7. Conclusions

We have detailed here an efficient implementation of a back-propagation algorithm on T0. The use of the mixed fixed/floating-point mode in the implementation shows good performance with real-world networks, both in terms of the efficiency of computation and in terms of the convergence rate. The limited precision supported by the hardware is not a problem provided the range is appropriately chosen. The mixed model computes the output layer's error using floating-point, and uses the floating-point values to determine an appropriate range for the following fixed-point.

This work shows that digital neuroprocessors and particularly T0 can be efficient test beds for various BP-type algorithms, even when limited by fixed-point formats.

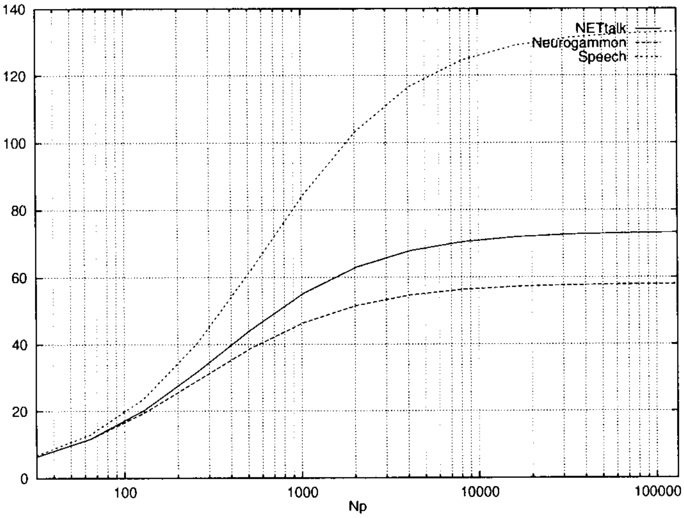


Fig. 3. Performance for some real-world applications (in MCUPS).

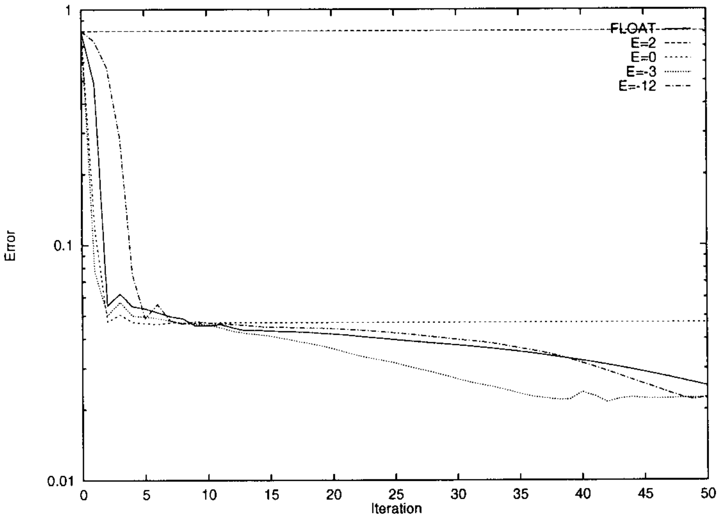


Fig. 4. Learning behavior for different fixed-point ranges.

Acknowledgements

Thanks to David Johnson for providing the emulation routines for fixed- and floating-point math and for several interesting discussions on T0, Naghmeh Nikki Mirghafori for providing the speech database, and Professor Nelson Morgan for suggestions on the learning algorithm. We would also like to thank two anonymous reviewers for their suggestions on how to improve this paper.

This work was developed while D. Anguita was visiting researcher at ICSI, Berkeley, USA, under a grant of "CNR – Consiglio Nazionale Ricerche", Italy.

References

[1] C. Alippi and M.E. Negri, Hardware requirements for digital VLSI implementations of neural networks, *Int. Joint Conf. on Neural Networks*, Singapore (1991) pp. 1873–1878.

[2] D. Anguita, G. Parodi and R. Zunino, An efficient implementation of BP on RISC-based workstations, *Neurocomputing* 6 (1994) 57–65.

[3] K. Asanović, J. Beck, B. Irissou, D. Kingsbury, N. Morgan and J. Wawrzynek, The T0 vector microprocessor, *Hot Chips VII Symposium*, Stanford Univ. (13–15 Aug. 1995).

[4] K. Asanović, J. Beck, J. Feldman, N. Morgan and J. Wawrzynek, Designing a connectionist network supercomputer, *Int. J. Neural Systems* 4(4) (Dec. 1993) 317–326.

[5] K. Asanović and N. Morgan, Experimental determination of precision requirements for back-propagation training of artificial neural networks, in *Proc. of 2nd Int. Conf. on Microelectronics for Neural Networks*, Münich, Germany (16–18 Oct. 1991) pp. 9–15.

[6] V. Bochev, Distributed arithmetic implementation of artificial neural networks, *IEEE Trans. on Signal Processing* 41(5) (May 1993).

[7] H. Bourlard and N. Morgan, Continuous speech recognition by connectionist statistical methods, *IEEE Trans. on Neural Networks* 4(6) (Nov. 1993) 893–909.

[8] S. Carrato, A. Premoli and G.L. Sicuranza, Linear and nonlinear neural networks for image compression, in *Digital Signal Processing*, V. Cappellini and A.G. Constantinides, eds. (Elsevier, Amsterdam, 1991) pp. 526–531.

[9] G. Cybenko, Approximation by superposition of a sigmoidal function, *Math of Control, Signal, and Systems* 2 (1989) 303–314.

[10] D.D. Caviglia, M. Valle and G.M. Bisio, Effect of weight discretization on the back propagation learning method: Algorithm design and hardware realization, *Proc. of IJCNN '90*, San Diego, USA (17–21 June 1990) pp. 631–637.

[11] A. Corana, C. Rolando and S. Ridella, A highly efficient implementation of back-propagation algorithm on SIMD computers, in *High Performance Computing, Proc. of the Int. Symp.*, Montpellier, France (22–24 March 1989) J.-L. Delhaye and E. Gelenbe, eds. (Elsevier, Amsterdam, 1989) pp. 181–190.

[12] E. Fiesler, A. Choudry and H.J. Caulfield, A universal weight discretization method for multi-layer neural networks, *IEEE Trans. on SMC*, to appear.

[13] D. Hammerstrom, A VLSI architecture for high-performance, low-cost, on-chip learning, *Proc. of the IJCNN '90*, San Diego, USA (17–21 June 1990) pp. 537–544.

[14] M. Hoehfeld and S.E. Fahlman, Learning with numerical precision using the cascade-correlation algorithm, *IEEE Trans. on Neural Networks* 3(4) (July 1992) 602–611.

[15] P.W. Hollis, J.S. Harper and J.J. Paulos, The effect of precision constraints in a backpropagation learning network, *Neural Computation* 2(3) 1990.

[16] M.A. Kramer, Nonlinear principal component analysis using autoassociative neural networks, *AIChE J.* 37(2) (Feb. 1991) 233–243.

[17] G. Kane and J. Heinrich, *MIPS RISC Architecture* (Prentice Hall, Englewoods Cliffs, NJ, 1992).

[18] N. Maudit, M. Duranton, J. Gobert and J.A. Sirat, Lneuro 1.0: a piece of hardware LEGO for building neural network systems, *IEEE Trans. on Neural Networks* 3(3) (May 1992) 414–421.

[19] N. Morgan and H. Bourlard, Factoring networks by a statistical method, *Neural Computation* 4(6) (Nov. 1992) 835–838.

[20] U. Ramacher et al., eds., *VLSI Design of Neural Networks* (Kluwer Academic, Dordrecht, 1991).

[21] U. Ramacher et al., SYNAPSE-X: a general-purpose neurocomputer, *Proc. of the 2nd Int. Conf. on Microelectronics for Neural Networks*, Münich, Germany (Oct. 1991) pp. 401–409.

[22] S. Sakaue, T. Kohda, H. Yamamoto, S. Maruno and Y. Shimeki, Reduction of required precision bits for backpropagation applied to pattern recognition, *IEEE Trans. on Neural Networks* 4(2) (March 1993) 270–275.

[23] J.A. Sirat, S. Makram-Ebeid, J.L. Zorer and J.P. Nadal, Unlimited accuracy in layered networks, *IEEE Int. Conf. on Artificial Neural Networks*, London (1989) pp. 181–185.

[24] T.J. Sejnowsky and C.R. Rosenberg, Parallel networks that learn to pronounce English text, *Complex Systems* 1 (1987) 145–168.

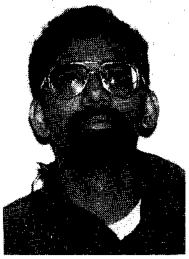
[25] G. Tesauro and T.J. Sejnowsky, A neural network that learns to play backgammon, in *Neural Information Processing Systems*, D.Z. Anderson, ed. (1987) pp. 442–456.

[26] T. Tollenaere, SuperSAB: fast adaptive back propagation with good scaling properties, *Neural Networks* 3(5) (1990) 561–573.

[27] T.P. Vogl, J.K. Mangis, A.K. Rigler, W.T. Zink and D.L. Alkon, Accelerating the covergence of the back-propagation method, *Biological Cybernetics* 59 (1989) 257–263.

[28] J. Wawrzynek, K. Asanović and N. Morgan, The design of a neuro-microprocessor, *IEEE Trans. on Neural Networks* 4(3) (May 1993) 394-399.

Davide Anguita obtained the "laurea" degree in Electronic Engineering from Genoa University in 1989. He worked at Bailey-Esacontrol in the field of wide-area distributed control systems, then he joined the Department of Biophysical and Electronic Engineering (DIBE) of Genoa University, where he received the Doctorate in Computer Science and Electronic Engineering. After a one-year visit to the International Computer Science Institute, Berkeley, CA, he is currently a postdoc research assistant at DIBE. His research activities cover neurocomputing and parallel architectures, including applications and implementation of artificial neural networks and the design of parallel and distributed systems.



Benedict Gomes received the B.S. degree in Computer Engineering from Case Western Reserve University, Cleveland, OH, and his M.A. in Computer Science from U.C. Berkeley. He is currently working on his PhD at U.C. Berkeley. His research centers around mapping structured connectionist networks onto general purpose parallel machines.