INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

MBP on T0: mixing floating- and fixed-point formats in BP learning

Davide Anguita * † Benedict A. Gomes * ‡

TR-94-038

August 1994

Abstract

We examine the efficient implementation of back prop type algorithms on T0 [4], a vector processor with a fixed point engine, designed for neural network simulation. A matrix formulation of back prop, Matrix Back Prop [1], has been shown to be very efficient on some RISCs [2]. Using Matrix Back Prop, 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 method. Since high efficiency is futile if convergence is poor (due to the use of fixed point arithmetic), we use a mixture of fixed and floating point operations. The key observation is that the precision of fixed point is sufficient for good convergence, *if* the range is appropriately chosen. 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 reasonable.

*International Computer Science Institute (ICSI), Berkeley, USA

†Department of Biophysical and Electronic Engineering, University of Genova, Italy

‡Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, USA

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 to the efficient execution of various learning algorithms with a strong accent on back-propagation. Some of the best-known processors in this field are CNAPS [13], GENES-IV [18] and MA-16 [21]: they are the building blocks for larger system that exploit massive parallelism to achieve performances orders of magnitude greater than conventional workstations [22, 28]. T0 belongs to this family of processors and it will be the first implementation of the Torrent architecture [5]. It is tailored for neural-networks calculations and inherits some of the features of a previous neuro-processor [30]. The next implementation (T1) will be the building block for a massively parallel computer [6]. The common characteristic of these processors is the use of a fixed-point engine, typically 16-bits wide or less, for fast computation. In particular, T0 is composed of a standard MIPS RISC engine [17] with no floating-point unit and 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 figure translates in a peak performance of 0.8 GOPS if the processor is clocked at 50MHz or ~ 0.2 GCUPS for 2-layer networks (a result comparable to supercomputer implementations).

The drawback for the final user who wants to implement an algorithm for neural network learning 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 [9, 7, 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, 24], mainly because the effect of the discretization during the learning is not completely understood and it seems to be both problem and algorithm dependent.

One solution to overcome the limitations of a BP implementation 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 are done in fixed- and floating-point format respectively. On the other hand, this solution does not address any efficiency issue because the most computationally expensive part of the algorithm (the backward phase) is still performed on conventional hardware, losing all the advantages of a fast fixed-point engine.

We show here a mixed floating- fixed-point implementation of Matrix Back Propagation (MBP) [1] on T0 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. 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 exploting the T0 hardware.

The following section describes the learning algorithm implemented. Section 3 shows the implementation details and performance evaluation. Section 4 describes some experiments with the mixed model approach. In Section 5 some comments on the algorithm are presented.

2 Matrix Back Propagation

In Table 1 the MBP algorithm is summarized. It can be used to represent several batch/block learning algorithms with adaptive step and momentum [29, 3, 27]. We are interested in finding the most computational expensive part of the algorithm, so we will use the left column of Table 1 that shows the number of operations needed by each step of MBP.

Table 1: The MBP algorithm

Step	Description	# of operations
(1a) Feed-Forward	$[S]_l = [S]_{l-1} \cdot [W]_l$	$2N_P N_l N_{l-1}$
(1b)	$[S]_l \leftarrow [S]_l + \mathbf{b}_l \cdot \mathbf{1}^t$	$N_P N_l$
(1c)	$[S]_l \leftarrow f\{[S]_l\}$	$N_P N_l k_1$
(2a) Error back-prop	$[\Delta]_L = [T] - [S]_L$	$N_P N_L$
(2b)	$[\Delta]_L \leftarrow [\Delta]_L \times g\{[S]_L\}$	$N_P N_L(1+k_2)$
(2c)	$[\Delta]_l = [\Delta]_{l+1} \cdot [W]^t_{l+1}$	$2N_P N_{l+1} N_l$
(2d)	$[\Delta]_l \leftarrow [\Delta]_l \times g\{[S]_l\}$	$N_P N_l(1+k_2)$
(3a) Weight variation	$[\Delta W]^{new}_l = [S]^t_{l-1} \cdot [\Delta]_l$	$2N_P N_l N_{l-1}$
(3b)	$\mathbf{\Delta b}^{new}_l = [\Delta]^t_l \cdot \mathbf{1}$	$N_P N_l$
(3c)	$[\Delta W]^{new}_l \leftarrow \eta[\Delta W]^{new}_l + \alpha[\Delta W]^{old}_l$	$3N_l N_{l-1}$
(3d)	$\mathbf{\Delta b}^{new}_l \leftarrow \eta \mathbf{\Delta b}^{new}_l + \alpha \mathbf{\Delta b}^{old}_l$	$3N_l$
(4a) Weight update	$[W]_l \leftarrow [W]_l + [\Delta W]^{new}_l$	$N_l N_{l-1}$
(4b)	$\mathbf{b}_l \leftarrow \mathbf{b}_l + \mathbf{\Delta b}^{new}_l$	N_l

We assume that our feed-forward network is composed of L layers of N_l neurons, with $0 \leq l \leq L$. The weights are stored in matrices $[W]_l$ 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 $[S]_0$ in row order and target patterns similarly in matrix $[T]$. The order of storing is particularly important for the efficiency of the implementation: if the patterns are stored in row-order elements of a pattern are stored in consecutive memory location and can be accessed with no performance penalty on the vast majority of current processor architectures including T0. Matrices $[S]_1 \cdots [S]_L$ contain the output of the corresponding layer when $[S]_0$ is applied to the input of the network. The size of $[S]_l$ is $N_P \times N_l$ and the size of $[T]$ is $N_P \times N_L$.

The back-propagated error is stored in matrices $[\Delta]_l$ of size $N_P \times N_l$ and the variations of weights and biases computed at each step are stored respectively in matrices $[\Delta W]_l$ of size $N_l \times N_{l-1}$ and vectors $\mathbf{\Delta b}_l$ 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 a sigmoid or hyperbolic tangent then $k_2 = 2$.

On a conventional RISC, each operation can be completed in a single cycle, therefore the total computational time is $T \propto n_{cycles} = n_{op}$. On vector machines or processors with several ALUs the peak performance is $n_{cycles} = n_{op}/P$ where P is the number of ALUs. Obviously, these are the theoretical limits of performance: the implicit assumption is that (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 [1, 2] that with a relative small effort these constraints can be satisfied reasonably well on some RISCs. In the following section we will address this problem for T0.

3 MBP on T0

If the implementation of an algorithm is optimal, in the sense that it can completely exploit the T0 hardware, we can expect to have $n_{cycles} = n_{op}/16$. For this reason, we will refer to an operation as *asymptotically optimal* for T0 (or simply *optimal*) if the efficiency ϵ of its implementation goes to 1 as the size of the problem grows. In other words: $\epsilon = \frac{n_{op}}{16 n_{cycles}} \rightarrow 1$.

We want to show here that MBP is optimal in this sense, even though some of the computations are done in floating-point and therefore must be simulated in software.

It is clear that the computational load belongs to steps (1a),(2c) and (3a) that show $O(n^3)$ complexity, where n is the size of the problem. To compute these steps, three matrix multiplications must be performed: (1a) is a conventional matrix product, (2c) is a matrix product with the second matrix transposed and (3a) is a matrix product with the first matrix transposed.

In the first column of Table 2 the three operations are showed in pseudo-code. In the second column of Table 2, the vectorized pseudo-code is shown, where V_L is the vector length (32 in the current implementation of T0) and U, V are the unrolling depth needed to fully exploit the processor pipelines.

Table 2: Vectorized matrix products

$[S]_l = [S]_{l-1} \cdot [W]_l$	Vectorized (for T0)

for $i := 0$ to $N_P - 1$
  for $j := 0$ to $N_{l-1} - 1$
  for $k := 0$ to $N_l - 1$
  $s^l_{i,j} \mathrel{+}= s^{l-1}_{i,k} * w^l_{k,j}$
	for $j := 0$ to $N_{l-1} - 1$ step V_L
	  for $i := 0$ to $N_P - 1$ step U
	  for $k := 0$ to $N_l - 1$
	  $s^l_{i,[j,j+V_L]} \mathrel{+}= s^{l-1}_{i,k} * w^l_{k,[j,j+V_L]}$
	  $s^l_{i+1,[j,j+V_L]} \mathrel{+}= s^{l-1}_{i+1,k} * w^l_{k,[j,j+V_L]}$
	  $\vdots$
	  $s^l_{i+U-1,[j,j+V_L]} \mathrel{+}= s^{l-1}_{i+U-1,k} * w^l_{k,[j,j+V_L]}$

$[\Delta]_l = [\Delta]_{l+1} \cdot [W]^t_{l+1}$	Vectorized (for T0)

for $i := 0$ to $N_P - 1$
  for $j := 0$ to $N_{l+1} - 1$
  for $k := 0$ to $N_l - 1$
  $\delta^l_{i,j} \mathrel{+}= \delta^{l+1}_{i,k} * w^{l+1}_{j,k}$
	for $i := 0$ to $N_P - 1$ step V
	  for $j := 0$ to $N_{l+1} - 1$ step V
	  for $k := 0$ to $N_l - 1$ step V_L
	  $\delta^l_{i,j} \mathrel{+}= \delta^{l+1}_{i,[k,k+V_L]} * w^{l+1}_{j,k+V_L}$
	  $\vdots$
	  $\delta^l_{i+V,j+V} \mathrel{+}= \delta^{l+1}_{i+V-1,[k,k+V_L} * w^{l+1}_{j+V-1,[k,k+V_L]}$

$[\Delta W]_l = [S]^t_{l-1} \cdot [\Delta]_l$	Vectorized (for T0)

for $i := 0$ to $N_{l-1} - 1$
  for $j := 0$ to $N_l - 1$
  for $k := 0$ to $N_P - 1$
  $\Delta w^l_{i,j} \mathrel{+}= s^{l-1}_{k,i} * \delta^l_{k,j}$
	for $j := 0$ to $N_l - 1$ step V_L
	  for $i := 0$ to $N_{l-1} - 1$ step U
	  for $k := 0$ to $N_P - 1$
	  $\Delta w^l_{i,[j,j+V_L]} \mathrel{+}= s^{l-1}_{k,i} * \delta^l_{k,[j,j+V_L]}$
	  $\Delta w^l_{i+1,[j,j+V_L]} \mathrel{+}= s^{l-1}_{k,i+1} * \delta^l_{k,[j,j+V_L]}$
	  $\vdots$
	  $\Delta w^l_{i+U-1,[j,j+V_L]} \mathrel{+}= s^{l-1}_{k,i+U-1} * \delta^l_{k,[j,j+V_L]}$

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 register available for storing intermediate results: in the current implementation $U = 8$ 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 exact computation of the number of cycles in the general case, the reader can refer to Table 6 in

the Appendix.

The first product (1a) shown in Table 2 can be performed on T0 in

$$n_{cycles}^{(1a)} = (4N_l U + 4U)\,\frac{N_P N_{l-1}}{U V_L} \tag{8}$$

In fact, four cycles are needed to compute a single vector multiply/add in the inner loop and four cycles are needed 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. Equation (8) can be rewritten to show that this implementation of (1a) is optimal:

$$n_{cycles}^{(1a)} = \frac{N_P N_l N_{l-1}}{8} + \frac{N_P N_{l-1}}{8} \tag{9}$$

therefore

$$\epsilon^{(1a)} = \frac{n_{op}^{(1a)}}{16 n_{cycles}^{(1a)}} = \frac{2N_P N_l N_{l-1}}{16\left(\frac{N_P N_l N_{l-1}}{8} + \frac{N_P N_{l-1}}{8}\right)} = \frac{1}{1+\frac{1}{N_l}} \to 1 \tag{10}$$

The second product (2c) 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 [4]. 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. This is not true for matrix-vector or vector-vector products that suffer from a slightly bigger overhead. The total number of cycles for this operation is

$$n_{cycles}^{(2c)} = \left[4\frac{N_l V^2}{V_L} + 20 + V^2\right]\frac{N_P N_{l+1}}{V^2} = \frac{N_P N_l N_{l+1}}{8} + 6N_P N_{l+1} \tag{11}$$

As can be seen, the absence of an implemented dot-product appears only in the second-order term and becomes neglegible for large problems.

The third product (3a) is similar to (1a), but with a different order of the loops. The number of cycles in this case is

$$n_{cycles}^{(3a)} = (4N_P U + 4U)\,\frac{N_l N_{l-1}}{U V_L} = \frac{N_P N_l N_{l-1}}{8} + \frac{N_l N_{l-1}}{8} \tag{12}$$

Before converting the result of the matrix products to floating-point format, it is advisable to perform some other operations in fixed-point format. In particular: the computation of the output of each neuron through its activation function (1c), the computation of its derivative in the internal layers (2d), the bias addition in the feed-forward phase (1b) and the bias computation in the backward phase (3b).

The computation of the activation function is quite expensive if it is done using the floating-point math library [10], and it would cause a large penalty on T0 due to the absence of a FPU. Yet, if (1c) 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. The vector unit of T0 is provided with a vector instruction to perform

Table 3: Other vectorized operations

Step	Pseudo-code

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

(2d)	for $i := 0$ to $N_P - 1$
	  for $j := 0$ to $N_l - 1$ step V_L
	  $\delta^l_{i,[j,j+V_L]} = \delta^l_{i,[j,j+V_L]} * (1 - s^l_{i,[j,j+V_L]} * s^l_{i,[j,j+V_L]})$

(3b)	for $j := 0$ to N_l step V_L
	  for $i := 0$ to N_P
	  $b^l_{[j,j+V_L]} \mathrel{+}= \delta^l_{i,[j,j+V_L]}$

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 (1b), (2d) and (3b) is showed in Table 3.

All the three loops are memory-bounded and require respectively:

$$n^{(1b)}_{cycles} = \left(\frac{8N_l}{V_L} + 1\right) N_P = \frac{N_l N_P}{4} + N_P \tag{13}$$

$$n^{(2d)}_{cycles} = 12\frac{N_l N_P}{V_L} = \frac{3}{8} N_l N_P \tag{14}$$

$$n^{(3b)}_{cycles} = (4N_P + 4)\frac{N_l}{V_L} = \frac{N_P N_l}{8} + \frac{N_l}{4} \tag{15}$$

assuming sufficient unrolling.

To complete the fixed-point part, we must consider 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 number using the vector unit. In this case, the translation from floating-point to fixed-point format requires only $k_{fx} = 2.6 \leftrightarrow 1.8$ cycles/element (for vector sizes between 100 and 1000) and $k_{xf} = 3.6 \leftrightarrow 2.5$ cycles/element for the inverse conversion.

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

The total number of cycles needed for the conversions is:

$$n^{conv}_{cycles} = (k_{xf} + k_{fx}) \left[\sum_{l=1}^{L} N_l (N_{l-1} + 1) + N_P N_L\right] \tag{16}$$

All the other computations are done in floating-point format. This allows us to control with greater accuracy the exact range of the back-propagated error (2a), (2b) and the

behavior of the update algorithm (3c), (3d), (4a), (4b). In summary, we implement the computation of the error at the output layer and the weight changes in floating point so that the range of the output error may be correctly determined. T0 doesn't 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 $10 \leftrightarrow 50$ cycles/element. Then the number of cycles for these steps of the algorithm will be $n_{cycles}^{flp} = k_f n_{op}^{flp}$ with $k_f \in [10, 50]$.

We have now all the element to compute the number of cycles needed by T0 to execute MBP.

$$
\begin{aligned}
n_{cycles}^{MBP} &= \frac{N_P}{8}\left(3\sum_{l=1}^{L} N_l N_{l-1} - N_1 N_0\right) + && (17)\\
&+\frac{67}{8} N_P \sum_{l=1}^{L} N_l + \left(4k_f + k_{fx} + k_{xf} + \frac{1}{8}\right)\sum_{l=1}^{L} N_l N_{l-1} + && (18)\\
&+\left(4k_f + k_{fx} + k_{xf} - \frac{1}{2}\right) N_P N_L - 6 N_P N_1 + \frac{N_P N_0}{8} + && (19)\\
&+LN_P + \left(4k_f + k_{fx} + k_{xf} + \frac{1}{8}\right)\sum_{l=1}^{L} N_l && (20)
\end{aligned}
$$

If we compare the $O(n^3)$ term (17) with the corresponding term for n_{op}^{MBP} 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- (18,19) and first-order (20) terms of the above expression.

First of all, we can 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 (18) and the first term of (19). The dependency from size of the training set is controlled by the number of neurons of the output layer (N_L), so we expect better performances when dealing with networks with a small number of outputs (e.g. classification problems, as opposed to encoding problems [8]). Furthermore, some techniques to reduce the number of output neurons in classification problems are available in the literature [20].

There is also an explicit dependency in the first order term (20) 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 the number of layers is seldom greater than four in practical applications (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 $N_l \approx N \; \forall l$ and plot the efficiency and the performance in MCUPS (Figure 1) as functions of the size of the training set (N_P) and the network (N).

We assume $k_1 = 6$ [10] to compute n_{op} 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

Figure 1: Efficiency and performance (MCUPS) of MBP on T0

performance is 160 MCUPS; obviously, the asymptotic performance of a generic RISC processor with the same clock and one FPU would be 10 MCUPS.

Figure 1 allows us to easily understand the behavior of the implementation, but it is of almost no practical use due to the 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 4).

Table 4: Some real-world applications.

Name	Network Size				Description
	N_0	N_1	N_2	N_3	
NETtalk [25]	203	80	26	-	Pronunciation of text.
Neurogammon [26]	459	24	24	1	Backgammon player.
Speech [23]	234	1000	69	-	Speech recognition.

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

Figure 3 shows the learning on a subset of the database for the speech recognition problem with different ranges of the fixed-point variables. In particular, EXPbp is the exponent of the most significant digit of the fixed-point variables in the backward phase. The 16-bit format allows us to represent values in the range $[-2^{EXPbp}, 2^{EXPbp} - 2^{EXPbp-15}]$.

It is clear that the algorithm is quite sensitive to the range of the fixed-point format. If the fixed-point representation is too coarse, 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 (steps 2a, 2b) is done in floating-point format. In this case, the learning with mixed format is not inferior to the learning in floating-point format.

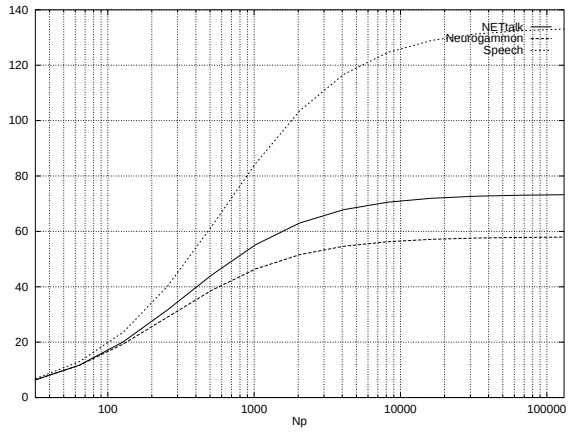

Figure 2: Performance for some real-world applications

4 Comments on batch/block learning.

We want to add here some comments on the learning method of the back propagation algorithm. In the following text we will refer to a method that updates the weights of the network after each pattern as *by-pattern* (bP). Differently, if a method updates the weights after the presentation of the whole training set, it will be referred as *by-epoch* (bE). There is an intermediate option if the updating is done every N_P^* patterns (with $1 \ll N_P^* \ll N_P$): in this case the method will be indicated as *by-batch* or *by-block* (bB). Obviously, bP and bE can be seen as special cases of bB.

MBP is a straightforward implementation of a bE method but can implement a bB provided that $N_P^* \gg 1$, otherwise some of the matrices degenerates in vectors. We will show here that dealing with matrices is, in general, more efficient from the computational point of view on the vast majority of architectures [11]. On the other hand, it has the disadvantage of requiring more memory.

If we examine the time required to compute matrix-matrix and matrix-vector products on a generic processor, we must consider two factors: one is the time needed by the processor to perform the arithmetic operation, the other is the time needed to retrieve or store the appropriate data in memory. The computational power of a processor is exploited only if the ratio between those two operations is greater than one, in other words if the CPU does not wait for the operands to be moved in or from the memory. Unfortunately this is not the case for the vast majority of the architecture: in fact, most of them rely on a hierarchical memory structure. Only registers and eventually the first order cache are able to provide data at the maximum speed; if the data is located more deeply in the memory hierarchy (or in the case of parallel architectures on a non-local memory) most of the time is wasted not on effective computation but in data movement[1].

Let's consider a matrix product $[C] = [A] \cdot [B]$ with, for simplicity, $dim(A) = dim(B) =$

[1] T0, for example, can access the memory in a single cycle in every circumstance except for a stride-n access to a vector that requires one cycle per element.

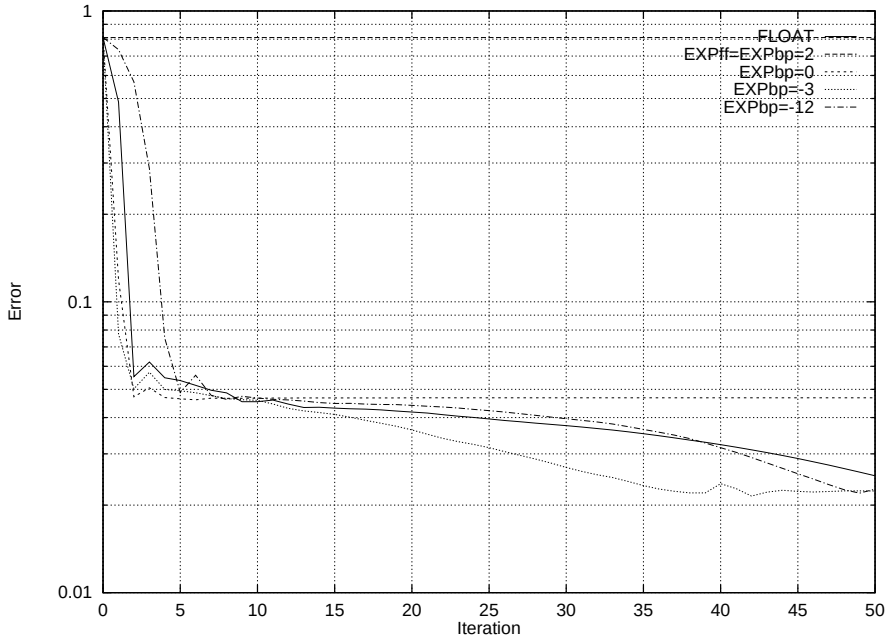

Figure 3: Learning behavior for different fixed-point ranges

$dim(C) = N \times N$: this is the core operation for the bE method. The bP method uses a series of matrix-vector products instead: $\mathbf{c} = [A] \cdot \mathbf{b}$. To overcome the problem of time-consuming access to the memory, the obvious solution is to keep part of the data in fast-accessible memory and perform most of the computation before discarding it. This can be accomplished by dividing the matrices into several sub-matrices, and the vectors into sub-vectors. To compute the products, we need three sub-matrices in the bE case and one sub-matrix and two sub-vectors in the bP (see Table 5).

Table 5: Memory efficient products.

Matrix–Matrix product	Matrix–Vector product

for $i := 1$ to N/B_{bE}
  for $j := 1$ to N/B_{bE}
  for $k := 1$ to N/B_{bE}
  $[C]_{i,j} += [A]_{i,k} * [B]_{k,j}$
	for $i := 1$ to N/B_{bP}
	  for $j := 0$ to N/B_{bP}
	  $\bar{c}_i += [A]_{i,j} * \bar{b}_j$

If we assume that the amount of fast memory is M, we must ensure that the amount of data needed for the computation never exceeds this value:

$$3B_{bE}^2 \leq M \qquad \text{and} \qquad B_{bP}^2 + 2B_{bP} \leq M \tag{21}$$

where B_{bE} and B_{bP} are the dimension of the corresponding sub-matrices and sub-vectors.

We can now write the ratio between the number of operations and the number of references to the memory for the bE case

$$R_{bE} = \frac{n_{op}^{bE}}{n_{ref}^{bE}} = \frac{2N^3}{\left(2B_{bE}^2 \frac{N}{B_{bE}} + B_{bE}^2\right) \frac{N^2}{B_{bE}^2}} \leq \frac{\sqrt{M}}{2\sqrt{3}} \tag{22}$$

and for the bP case

$$R_{bP} = \frac{n_{op}^{bP}}{n_{ref}^{bP}} = \frac{2N^2}{\left[\left(B_{bP}^2 + B_{bP}\right) \frac{N}{B_{bP}} + B_{bP}\right] \frac{N}{B_{bP}}} \leq 2 \tag{23}$$

assuming that $N \gg M$.

Equation (23) shows that the ratio between number of operations and number of memory references is always less than or equal to 2 *independent* of the size of the fast memory. This means that if the memory is not capable of providing at least one operand in the same amount of time needed to perform two arithmetic operations, the ALU will be unable to perform at maximum speed. For many processors, this constraint is even more strict due to the presence of a multiply/add instruction that can perform two arithmetic operations in a single cycle.

On the other hand, the ratio R_{bE} can be increased as needed by adding more fast memory. If the size of the fast memory is made large enough, the overhead due to the memory references can be completely hidden by the arithmetic operations.

The drawback of the bE method is the memory requirement. In fact, it is approximately $2L$ times the requirement of the bP method (where L is the number of layers of the network).

There is another concern about bE methods that regards the speed of convergence. It has been shown that for small problems bE methods are very effective. This is, in general, not true for medium and large problems, where, in some cases, bP methods are much faster. This is an undesirable effect, because all the advantages of a bE method in terms of computational speed can be wasted by a slow converging algorithm.

It has been suggested that the possible poor performance of bE methods is directly related to the redundancy of the data in the training set [19], therefore it can be worthwhile to start the learning with a subset of the training set and increase its dimension as the learning proceed. Using this technique, a bE method is usually faster in convergence than a bP method (see [19] for more details). This technique can be seen as a bB method that can be implemented through MBP; the computational efficiency will grow as the learning proceed due to the increase of the learning set dimension.

To confirm the hypotesis that a bB method is effective regarding learning speed, we performed some experiment using different block sizes with the speech database mentioned before. The results are summarized in Figure 4. In particular, the block size was varied between 1/2 and 1/32 of the entire database (or from 10,000 to 625 patterns per block). For simplicity, this size was kept fixed during the whole learning.

The database consists of 20,000 patterns and the dimension of the network is $130 \times 128 \times 69$. The performance of the network during the learning is measured on a test set of 1,000 patterns and is simply the percentage of misclassifications. The slope of the error on the learning set is similar.

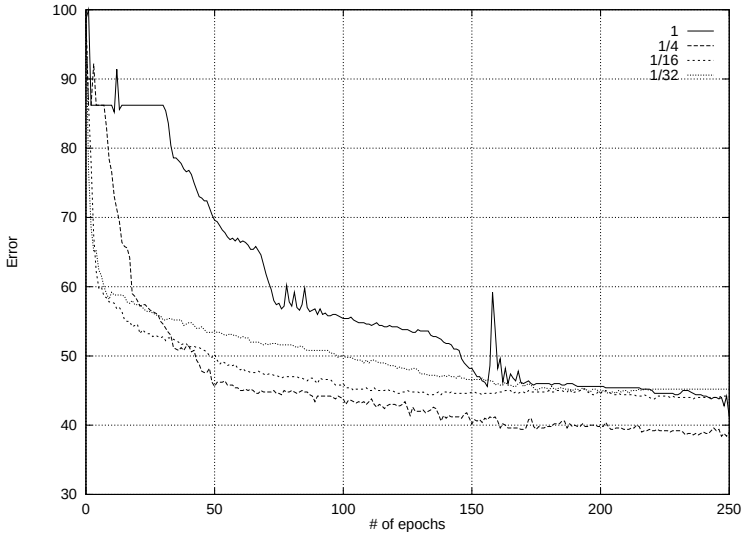

Figure 4: Error during learning for different block sizes.

As can be easily seen, the size of the block affects sensibly the behavior of the learning, expecially in the first iterations. The learning speed increases as the block size decreases, until a critical size is reached. At this point, decreasing the size of the block is more harmful than beneficial. An intuitive explanation of this phenomenon could be that the amount of information in each block of patterns is no longer sufficient to generate a meaningful gradient direction.

We can define a measure of the effectiveness of the block size counting the number of iterations "wasted" by the network before providing a useful result; in other words, the number of epochs needed by the network to cross the 50% error figure on the test set. This is shown in Figure 5.

5 Conclusions.

We have detailed here an efficient implementation of a back-propagation algorithm on T0. Despite the use of the mixed fixed- floating-point mode the implementation shows good performance with real-world networks, both in terms of the 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. The accumulated errors are also stored using floating point, to permit accumulation of error values from different iterations with different exponent ranges. Even though more work must be done in the understanding of the behavior of bE and bP techniques on large databases, we have shown that T0 can be an efficient test bed for batch/block learning algorithms.

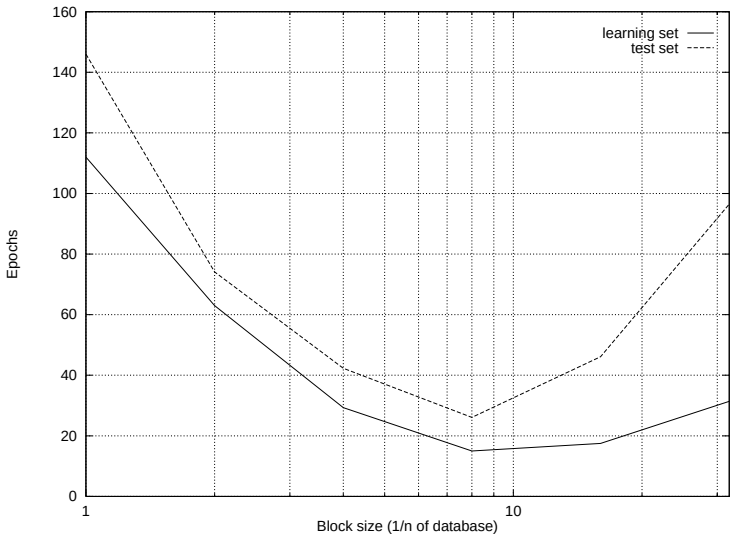

Figure 5: Number of *wasted* epochs for different block sizes.

6 Acknowledgments.

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 Nelson Morgan for suggestions on the bE/bP problem.

7 Appendix.

In Table 6 the exact expressions of the number of cycles for each step of MBP is reported, when the dimension of the matrices is not a multiple of V_L, U or V.

Table 6: Number of cycles for MBP on T0.

Step	n_{cycles}
(1a)	$(4N_lU + 4U) \left\lceil \frac{N_P}{U} \right\rceil \left\lceil \frac{N_{l-1}}{V_L} \right\rceil$
(1b)	$\left(8 \left\lceil \frac{N_l}{V_L} \right\rceil + 1\right) N_P$
(1c)	$1.5 N_P \left\lceil \frac{N_l}{V_L} \right\rceil V_L$
(2a)	$k_f N_P \left\lceil \frac{N_L}{V_L} \right\rceil V_L$
(2b)	$3k_f N_P \left\lceil \frac{N_L}{V_L} \right\rceil V_L$
(2c)	$\left(4 \left\lceil \frac{N_l}{V_L} \right\rceil V^2 + 20 + V^2\right) \left\lceil \frac{N_P}{V} \right\rceil \left\lceil \frac{N_{l+1}}{V} \right\rceil$
(2d)	$12 \left\lceil \frac{N_l}{V_L} \right\rceil N_P$
(3a)	$(4N_PU + 4U) \left\lceil \frac{N_l}{V_L} \right\rceil \left\lceil \frac{N_{l-1}}{U} \right\rceil$
(3b)	$(4N_P + 4) \left\lceil \frac{N_l}{V_L} \right\rceil$
(3c)	$3k_f N_l \left\lceil \frac{N_{l-1}}{V_L} \right\rceil V_L$
(3d)	$3k_f \left\lceil \frac{N_l}{V_L} \right\rceil V_L$
(4a)	$k_f N_l \left\lceil \frac{N_{l-1}}{V_L} \right\rceil V_L$
(4b)	$k_f \left\lceil \frac{N_l}{V_L} \right\rceil V_L$

These values have been used in the computation of the performances of MBP on T0 in the general case.

References

[1] D.Anguita, G.Parodi, and R.Zunino. *An Efficient Implementation of BP on RISC-based Workstations.* Neurocomputing 6, 1994.

[2] D.Anguita. *Matrix Back Propagation.* Technical Report, DIBE, University of Genova, Italy, November 1993.

[3] D.Anguita, M.Pampolini, G.Parodi, and R.Zunino. *YPROP: Yet Another Accelerating Technique for the Back-Propagation.* ICANN '93, Amsterdam, The Netherlands, 1993.

[4] K.Asanović. *T0 Reference Manual.* Internal document, International Computer Science Institute and UC Berkeley, 1993.

[5] K.Asanović. *Torrent Architecture Manual.* Internal document, International Computer Science Institute and UC Berkeley, 1993.

[6] K.Asanović, J.Beck, T.Callahan, J.Feldman, B.Irissou, B.Kingsbury, P.Kohn, J.Lazzaro, N.Morgan, D.Stoutamire and J.Wawrzynek. *CNS-1 Architecture Specification.* ICSI Technical Report TR-93-021, April 1993.

[7] 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 Microeletronics for Neural Networks, Münich, Germany, October 1991.

[8] H.Boulard and Y.Kamp. *Auto-association by Multilayer Perceptrons and Singular Value Decomposition.* Byological Cybernetics, No. 59, 1988.

[9] D.D.Caviglia, M.Valle, G.M.Bisio. *Effect of Weight Discretization on the Back Propagation Learning Method: Algorithm Design and Hardware Realization.* IJCNN 90, San Diego, USA, June 1990.

[10] A.Corana, C.Rolando, S.Ridella. *A Highly Efficient Implementation of Back-propagation Algorithm on SIMD Computers.* In High Performance Computing, J.-L.Delhaye and E.Gelenbe (Eds.), Elsevier, 1989.

[11] J.Dongarra. *Linear Algebra Library for High-Performance Computers.* Frontiers of Supercomputing II. K.R.Ames and A.Brenner (Eds.), University of California Press, 1994.

[12] E.Fiesler, A.Choudry, and H.J.Caulfield. *A Universal Weight Discretization Method for Multi-Layer Neural Networks.* To appear in IEEE Trans. on SMC.

[13] D.Hammerstrom. *A VLSI architecture for High-Performance, Low-Cost, On-Chip Learning.* IJCNN 90, S.Diego, USA, June 1990.

[14] M.Hoehfeld and S.E.Fahlman *Learning with Numerical Precision Using the Cascade-Correlation Algorithm.* IEEE Trans. on NN, Vol.3, No.4, July 1992.

[15] P.W.Hollis, J.S.Harper, and J.J.Paulos. *The effect of precision constraints in a back-propagation learning network.* Neural Computation, Vol.2, No.3, 1990.

[16] N.Kambhatla and T.K.Leen. *Fast Non-Linear Dimension Reduction.* NIPS 6, J.D.Cowan, G.Tesauro and J.Alspector (Eds.), Morgan Kaufmann, 1994.

[17] G.Kane, J.Heinrich. *MIPS RISC architecture.* Prentice Hall, 1992.

[18] P.Ienne and M.A.Viredaz. *GENES IV: A Bit-Serial Processing Element for a Multi-Model Neural-Network Accelerator.* available in Neuroprose.

[19] M.Moller. *Supervised Learning on Large Redundant Training Sets.* Int. J. of Neural Systems, Vol.4, No.1, 1993.

[20] N.Morgan and H.Boulard. *Factoring Networks by a Statistical Method.* Neural Computation, Vol.4, No.6, Nov. 1992.

[21] U.Ramacher et al. (Eds). *VLSI Design of Neural Networks.* Kluwer Academic, 1991.

[22] 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.

[23] S.Renals and N.Morgan. *Connectionist Probability Estimation in HMM Speech Recognition.* ICSI Technical Report TR-92-081, Dec. 1992.

[24] S.Sakaue, T.Kohda, H.Yamamoto, S.Maruno, and Y.Shimeki. *Reduction of Required Precision Bits for Back-Propagation Applied to Pattern Recognition.* IEEE Trans. on NN, Vol.4, No.2, March 1993.

[25] T.J.Sejnowsky and C.R.Rosenberg. *Parallel Networks that Learn to Pronounce English Text.* Complex Systems 1, 1987.

[26] G.Tesauro and B.Janssens. *A Neural Network That Learns to Play Backgammon.* NIPS, D.Z.Anderson (Ed.), 1988.

[27] T.Tollenaere. *SuperSAB: fast adaptive back propagation with good scaling properties.* Neural Networks, Vol.3, No.5, 1990.

[28] M.A.Viredaz. *MANTRA I: An SIMD Processor Array for Neural Computation.* Euro-ARCH 93, München, October 1993.

[29] T.P.Vogl, J.K.Mangis, A.K.Rigler, W.T.Zink, and D.L.Alkon. *Accelerating the Convergence of the Back-Propagation Method.* Biological Cybernetics 59, 1989.

[30] J.Wawrzynek, K.Asanović, and N.Morgan. *The Design of a Neuro-Microprocessor.* IEEE Trans. on NN, Vol.4, No.3, May 1993.