High Performance Fault-Tolerant Digital Neural Networks

Simone Bettola, *Member*, *IEEE*, and
Vincenzo Piuri, *Senior Member*, *IEEE*

Abstract—Efficient implementation of neural networks requires high-performance architectures, while VLSI realization for mission-critical applications must include fault tolerance. Contemporaneous solution of such problems has not yet been completely afforded in the literature. This paper focuses both on data representation to support high-performance neural computation and on error detection to provide the basic information for fault tolerance by using the redundant binary representation with a three-rail logic implementation. Costs and performances are evaluated referring to multilayered feed-forward networks.

Index Terms—Neural architecture, redundant binary representation, three-rail logic, concurrent error detection, unidirectional errors, high-performance architecture.

1 Introduction

ARTIFICIAL neural networks [1] are an attractive solution in several applications requiring massively-parallel computation (e.g., signal and image processing, sensor fusion, robotics, real-time control) when no algorithmic approach is known or when it cannot be easily formalized while the desired system operation can be specified through examples.

A neural paradigm is composed by a number (usually quite high) of functional operators (*neurons*) which are interconnected by weighted links (*synapses*) [1]. In any static network, the neuron n_i is modeled by:

$$x_i = f_i(\sigma_i - \vartheta_i) = f_i\left(\sum_{j=1}^{N} w_{i,j} \cdot x_j - \vartheta_i\right),$$

where x_i is the neuron's output, $w_{i,j}$ is the interconnection weight from neuron n_j to neuron n_i, $\sigma_i = \sum_{j=1}^{N} w_{i,j} \cdot x_j$ is the inputs' weighted summation, ϑ_i is the neuron's threshold, and $f_i(\cdot)$ is the nonlinear activation function that generates the new neuron's output from the incoming excitations. As an example, we consider only multilayered feed-forward networks [1] realized with fixed-point bit-parallel arithmetic units, but the approach can be extended to the any network topology.

In the literature, a number of digital architectures and implementations have been proposed. In this paper, we consider only the very frequent case of dedicated structures for applications in which learning can be perfected off-line *once* and the configured network can be used without changing its parameters. The solution proposed in [2] maps each neural operator onto an individual circuit so as to achieve a distributed architecture with high parallelism degree, high performances, but high circuit complexity. At the opposite extreme, the architecture presented in [3] time-multiplexes many components within each processing element to minimize the overall circuit complexity, at reduced performances.

The use of neural architectures in mission-critical real-time applications (e.g., in aerospace environments and in critical control systems) implies the capability of guaranteeing the consistency and the reliability of each *individual* output result generated by the underlying architecture before it is used for the subsequent system operations. The testing techniques are not suited to achieve these goals, since application of the input vectors capable of exciting all (or, according to the desired coverage, at least a large class of) errors in the fault model requires too much time during which the normal operations must be suspended (this is not acceptable for real-time applications). On the other hand, it is not necessary to certify the absence of any fault during the computation of an individual neural network result: It is enough to verify that no fault affected such a result.

The intrinsic correction capability of some neural paradigms is often claimed as an obvious solution to the fault tolerance issue, but we cannot rely on it for the applications mentioned above. Some authors tried, in fact, to achieve error masking by distributing the neural computation among a larger number of neurons so that the individual contribution to the final result is small: In the case of faults, the survived neurons are supposed to be able to generate the correct (or near correct) final result. This is true only if the error is always small; unfortunately, this assumption is usually not realistic [4]. Besides, too many neurons make it difficult to configure the network, due to the too high number of degrees of freedom, and decrease the generalization ability.

Learning can introduce some intrinsic error masking ability by forcing the final working points of the neurons towards the saturation regions of the nonlinear activation functions, so that even a large variation of the weighted summation affects the neuron output marginally or not at all. However, this ability does not cover many relevant error classes or is not present in several neural paradigms. An experimental analysis of the effectiveness of intrinsic error masking is given in [4]. Besides, the statistical analysis [5], [6] gives only the probability that the output is correct even in the presence of a fault, but it is neither able to certify the actual correctness of the individual network result nor to forecast the exhaustion of the intrinsic error masking ability, allowing the use of possibly-erroneous network results in the mission-critical applications, eventually, with dangerous consequences.

To *guarantee* error detection for the *individual* neural result, concurrent error detection techniques are required. Algorithm-based approaches [7] are either too application-specific or expensive in terms of circuit complexity. Time redundancy [8], [9] limits the circuit complexity overhead, but throughput is reduced and latency is increased. Data coding [10], [11] has been shown effective for moderate throughput at moderately-large circuit complexity only for single errors.

In this paper, we discuss the use of the Redundant Binary Representation [12], realized by means of a three-rail logic implementation [13], as an effective and efficient approach to concurrent error detection of all unidirectional stuck-at faults on multiple input and/or output lines [12], [14], when high throughput is mandatory. The adopted fault model is statistically sound if the diagnostic checks are frequent enough; no restriction is imposed on the multiplicity of the faults and of the generated errors. The massive throughput increase due to the carry-free RBR operations and the excellent detection capabilities make acceptable the complexity increase in several mission-critical applications.

Section 2 reviews RBR and presents the arithmetic units for the neural networks. Section 3 introduces the architectures with concurrent error detection capabilities. In Section 4, throughput and circuit complexity are evaluated at the functional level to provide figures of merit independent from the implementation technology.

• *The authors are with the Department of Electronics and Information, Politecnico di Milano, Piazza L. da Vinci 32, I-20133 Milano, Italy.*
E-mail: piuri@elet.polimi.it.

Manuscript received 19 July 1995.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 106213.

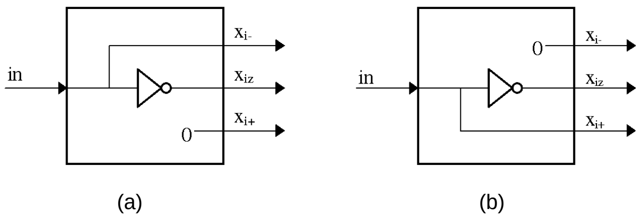


Fig. 1. The encoder: the cells of the sign bit (a) and for the other bits (b).

2 The Redundant Binary Representation and Units for the Neural Networks

The *Redundant Binary Representation* [12], [14] is a signed-digit number representation in the fixed radix 2 [12], with the digit set $\{\overline{1}, 0, 1\}$ ($\overline{1}$ being a compact notation for -1). An n-digit redundant binary integer is represented by $[x_{n-1}\ x_{n-2}\ \dots\ x_1\ x_0]$, where $x_i \in \{\overline{1}, 0, 1\}$; its value in the decimal notation is $\sum_{i=0}^{n-1} x_i \cdot 2^i$. An integer thus has a multiple representation allowing it to avoid the carry propagation. The fixed-point real numbers in the full-fractional notation can be represented in the same notation.

Two binary digits are sufficient to represent each RBR digit, but the arithmetic distance [15] between any of the two digit representations is not sufficient for error detection. The *three-rail logic* [13] (i.e., a *1-out-of-3* code) for each RBR digit is suited to solve this problem by supporting error detection of each individual digit. We represent the RBR values $\overline{1}$, 0, and 1 by using the codewords 100, 010, and 001, respectively, since such a codeword space has the maximum arithmetic distance and is unordered [13]. The result of any unit is correct if every output digit, checked independently from the others, is correct; individual checking of the RBR digits maximizes the computational parallelism.

It was proved that a combinatorial circuit is *strongly fault secure* [14] if the circuit is essentially inverter free (i.e., if it does not contain inverters but in the encoders) and the output codeword space is unordered for given fault model and coding technique. This implies that all sequences ϕ (having a given maximum length) of faults inducing errors can be detected, since the circuit's output is either correct in the absence of faults or not a codeword in the presence of any of such sequences. From the above remarks and definitions, it was shown [13], [14] that the unidirectional stuck-at fault model and the RBR with three-rail logic are well suited to design strongly fault secure combinatorial circuits.

The neural networks execute sequences of arithmetic operations (addition and multiplications) and nonlinear operations (the activation functions). The structure of coding and arithmetic units needs therefore to avoid all unnecessary transformations from the binary to the RBR space and vice versa. To save both circuit complexity and latency, being relevant only the final neural outputs, we limit coding only at the primary input and decoding whenever it is necessary to guarantee error detection, while the arithmetic units operate with RBR operands in the three-rail logic notation.

Coding of unsigned binary data into RBR is very simple; in [14], it was shown that the case of two-rail logic representation for adders and multipliers. Extension to the cases of integer and fixed-point numbers in the full-fractional two's-complement representation with the three-rail logic is straightforward: The *encoder* for the individual bit is shown in Fig. 1, being the n-bit encoder composed of an array of such a component. The circuit complexity and the latency can be practically neglected.

The *decoder* from unsigned RBR with three-rail logic into the binary representation is very complex and was presented in [14]; the binary result and its one's complement are generated so that each pair of bits composed by one output bit and its ones complement is a 1-out-of-2 coding of the output binary digit. Since such a code is not ordered and no inverter is present in the decoder, this circuit is strongly fault secure. Extension to the integer and fixed-point numbers needs to preserve such a characteristic. Therefore, we designed a decoder capable of transforming RBR values into binary numbers in the sign-modulus representation; a decoder generating numbers in the two's complement notation needs inverters and, as a consequence, it is not strongly fault secure. The RBR integer or fixed-point number is first converted in the RBR sign-modulus representation and, then, the modulus is decoded by using the circuit of [14]; the positive (negative) sign is denoted

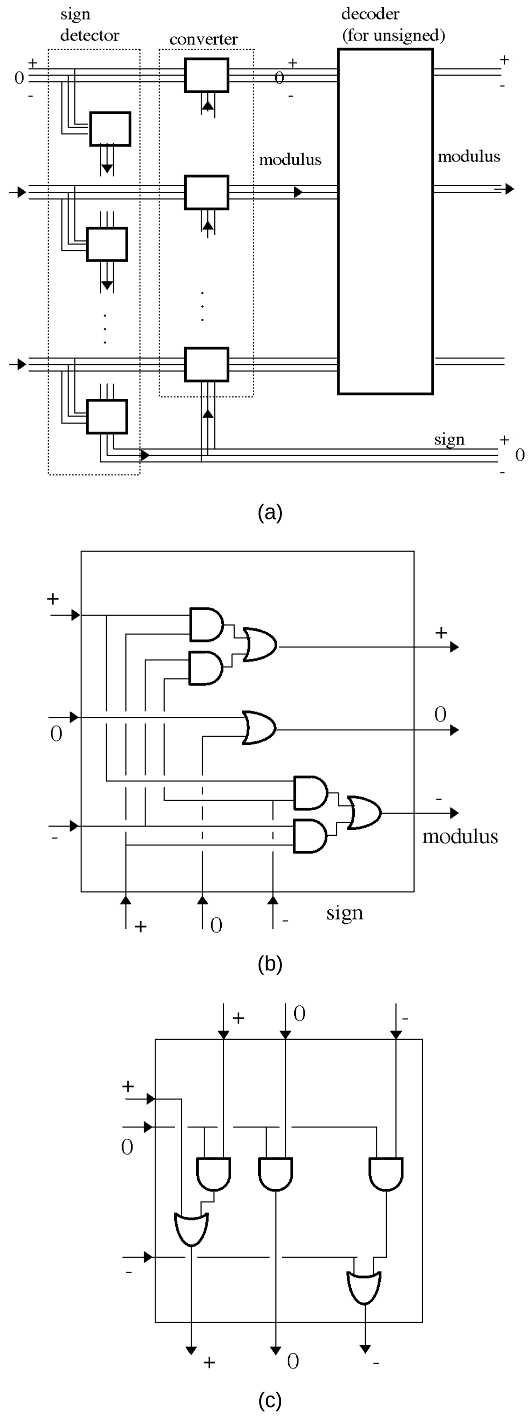


Fig. 2. The decoder: the overall structure (a), the converter cell (b), and the sign detector cell (c).

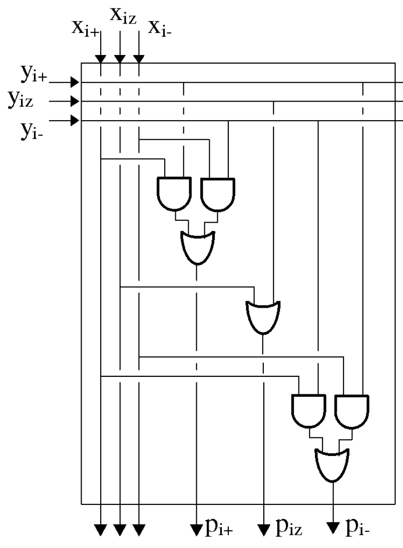


Fig. 3. The partial product generator.

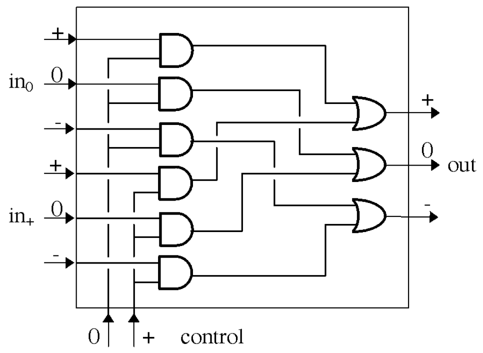


Fig. 4. The multiplexer.

by 001 (100). The overall circuit is shown in Fig. 2a: Figs. 2b and 2c present the basic cell of the converter and the one of the sign detector, respectively.

The *adder* for integer numbers was presented in [14], but it can be directly used also for fixed-point full-fractional numbers. The carry-save operation due to RBR allows for a very low latency, which is constant with the number of operands bits.

The *multiplier* was introduced in [14] for the case of unsigned operands represented in the two-rail logic. At first, the partial products for each digit are computed in parallel in a constant time; then, the partial products are added by using a binary tree of two-input RBR adders, generating a result in the three-rail logic in a time which is proportional to $\log_2 n$. For the integer and fixed-point numbers, the partial products generation must be modified so that input operands are in the three-rail logic. The new single-digit cell for the partial product generator is shown in Fig. 3: It is strongly fault secure and has circuit complexity and latency similar to [14].

To store intermediate results or coefficients according to the three-rail logic scheme, it is necessary to modify the registers holding the coefficients, the master-slave registers used as accumulators, and the shift registers supporting data transfer among neurons. For each RBR digit, three flip-flops are required. Any register or accumulator for the three-rail (or two-rail) logic is self-checking for all unidirectional faults on its input/output lines when it is used only with totally self-checking circuits, except for faults on the clock signal. Data delivered by one of these registers do not need to be checked at the register's output, since they will be implicitly checked after the operations performed by the subsequent arithmetic units. A shift register with the self-checking property is obtained by cascading as many master-slave memory elements in the three-rail logic as the number of stages in the shift register.

A strongly fault secure *multiplexer* can be obtained by using the two-rail logic for the input selection when the inputs are represented in the three-rail logic, as shown in Fig. 4.

The circuit implementing the *activation function* is strictly related to the envisioned nonlinearity; only the widely-used cases are considered here. The linear activation function is shown in Fig. 5a: The function generator is composed by a multiplier and an adder in the three-rail logic, while the function parameters are stored in registers. Fig. 5b presents the general case of the step function in which the step is not in the origin and has parametric height; the sign of the weighted summation is computed by the strongly-fault secure sign detector shown in Fig. 2a. The symmetric step centered in the origin and the step centered in the origin but starting from zero can be easily derived from this circuit.

To verify the correctness of RBR results, an error *checker* was proposed in [14] as a two-rail logic circuit comparing the two outputs of the decoder. Checkers for 1-out-of-3 codes are more complex and were widely studied in the literature, e.g., by using an asynchronous sequential machine [16], other coding techniques [17], [18], and the pass-transistor logic [19], [20]. The clock signal can be verified, e.g., by means of the circuit proposed in [21] for periodic signals.

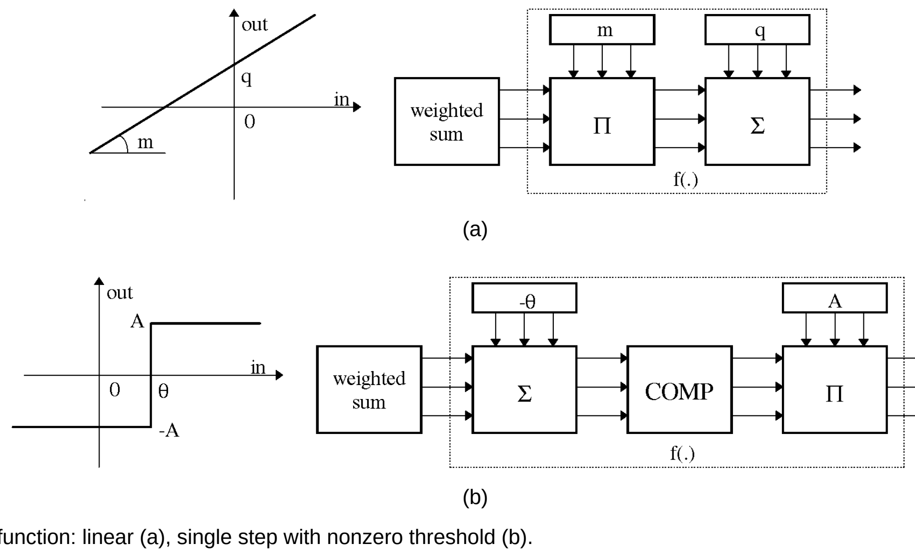


Fig. 5. The activation function: linear (a), single step with nonzero threshold (b).

3 The Neural Network Architectures with Concurrent Error Detection Capability

By using the components analyzed in Section 2, we can create a wide spectrum of neural architectures implementing the multilayered feed-forward paradigm, as well as other neural networks. In [13], it was shown that it is possible to connect strongly fault secure adders and multipliers in any cascaded structure by preserving such a property for the whole system when no inverter is introduced but in the encoders. Therefore, the subsystem computing the weighted summation can always be realized as a strongly fault secure circuit for any unidirectional stuck-at fault on multiple input/output lines.

If the activation function generator holds this property (e.g., see Fig. 5), the whole neural architecture becomes strongly fault secure; otherwise, the weighted summation must be checked in each neuron to verify its correctness, while the nonlinear part needs to be protected by using a different technique (e.g., the modular redundancy). In this section, we present the implementation details of the two extreme schemes introduced in Section 1, namely, the distributed and the time-multiplexed neural architectures.

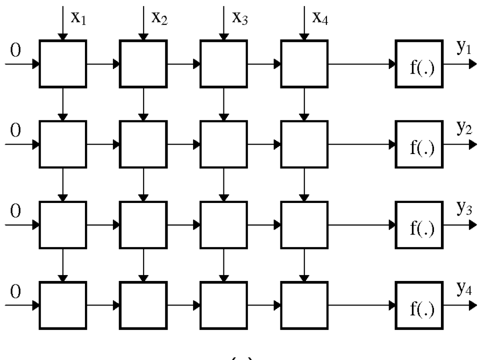
The overall structure of one layer of the distributed neural architecture is shown in Fig. 6a. Each processing element in the last column produces the neurons' outputs by applying the activation function, while the other processing elements (Fig. 6b) contribute to generate the weighted summation. Data are transferred among the processing elements in the three-rail logic. No encoder is necessary but at the primary inputs. Within the processing elements related to the weighted summation, no decoder is needed, as well as (if localization of the faulty processing element is not required) no checker. Decoding and checking are mandatory before the activation function only if the activation function generator is not protected by RBR, otherwise, they can be introduced only at the network's outputs by saving a large amount of circuit complexity and latency. Checking the neurons' outputs allows for localizing the faulty neuron. The interconnections are obviously increased; to limit their complexity and the silicon area occupied, two-rail logic or uncoded data could be adopted between the processing elements, despite a massive circuit complexity increase due to the possible encoders, the decoders, and the checkers now required within each processing element.

The overall structure for the time-multiplexed architectures is shown in Fig. 7a. Each processing element computes the whole neuron's output and is composed by the following subsystems:

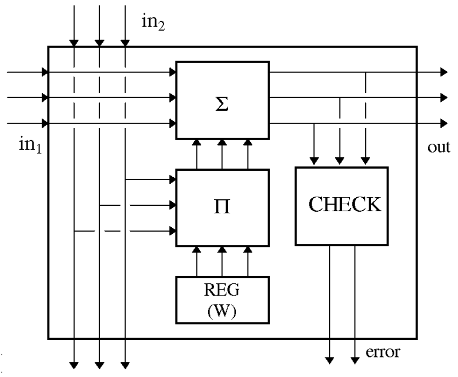
- The *input synapses* multiply the input values by the corresponding weight. The circular shifting of the weight register file needs to be synchronized with the circular shifting of the registers holding the layer's inputs. Besides, this second operation is interleaved with periodic loading of the new layer's inputs coming from the neurons of the previous layer.
- The *evaluator of the neuron's excitation* adds the weighted inputs. The adder is connected to an accumulator holding the partial summations. The possible neuron's threshold is stored in a dedicated register and loaded in the accumulator during the initialization of the excitation evaluation.
- The *neuron's output evaluator* generates the new output from the excitation and delivers it to the neurons in the subsequent layer by iterating the addition

Design of such a solution requires an accurate operation timing, a high synchronization, and a detailed analysis of the adopted coding scheme to guarantee detection even if the neuron is a sequential circuit. Two approaches are available: the sequential and the time-compressed architectures.

The neuron for the *sequential architecture* is shown in Fig. 7b. The operation of the above subsystems is performed sequentially. Each layer waits until all outputs of the previous layer are available before



(a)



(b)

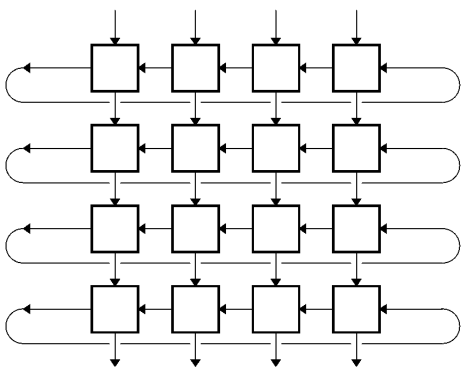
Fig. 6. The distributed architecture for one layer: the overall structure (a), the processing element for the weighted summation (b).

starting its own operations. The layer computation needs as many clock cycles as the neuron's inputs. The shift register must be as long as the number of these clock cycles to present the suited input with the corresponding weight. The clock cycle τ_{seq} is given by $\tau_{seq} = \tau_{mux} + \tau_{acc} + \tau_{\Pi} + \tau_{\Sigma} + max(\tau_{check}, \tau_f)$, where τ_{mux}, τ_{acc}, τ_{Π}, τ_{Σ}, τ_{check}, and τ_f are the latencies of the multiplexer, the accumulator, the partial product generator, the adder, the checker, and the activation function generator, respectively. The individual clock cycle is not fully exploited by all units: some of them are in fact not working in several time intervals. Synchronization between layers is quite simple since all neurons receive the same clock signal.

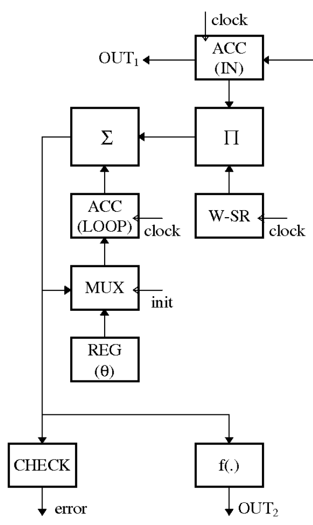
The neuron for the *time-compressed architecture* is shown in Fig. 7c. The operations are identical to the previous case, but the timing is different to better exploiting the clock cycle. The main architectural differences are the position of the checker (moved in the adder feedback loop) and the register before the activation function generator. The circuit complexity is practically equivalent to the sequential architecture, but the operation parallelism is higher. The layer operation requires one clock cycle more than the previous case, since the activation function is evaluated separately from the other operations in an its own cycle. The synchronization between layer's inputs and weights can be obtained with one of the following approaches:

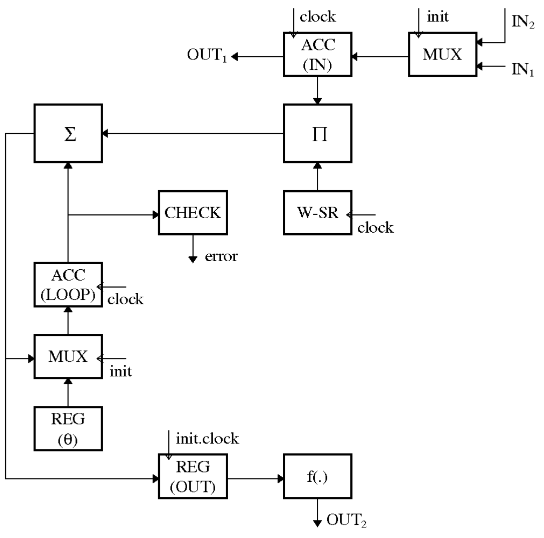
- By avoiding the rotation of the weight register file during the additional clock cycle, since the required operations concern the output stage only; this needs an additional control signal.
- By introducing an additional value in the weight register file so that its length is equal to the number of clock cycles; this preserves the periodicity of the synchronization signal.

All neurons can receive the same clock signal, since all data outgoing from a layer are always available before the receiving layer



(a)





(c)

Fig. 7. The time-multiplexed architecture: the overall structure (a), the sequential neuron (b), the time-compressed neuron (c).

starts its own computation. The additional clock cycle does not influence the overall performances since the increased parallelism even allows to reduce the clock cycle τ_{compr} to $\tau_{\text{compr}} = max(\tau_{\text{acc}} + \tau_{\Pi} + \tau_{\Sigma}; \tau_{\text{acc}} + \tau_{\text{check}}; \tau_{\text{reg}} + \tau_{f})$. The throughput is higher than both the case of the binary data representation and the sequential structure.

Massive reuse of the same components may lead to aliasing if a suited checking design is not considered to prevent exhaustion of the detection capacity for the adopted coding. In particular, the intermediate results generated by an iteration of the time-multiplexed architectures need to be checked so that the computation of combinatorial circuit implementing such that an iteration, as well as the accumulator register, are verified. This strategy practically substitutes the verification of the final result generated by the sequential circuit with a sequence of verifications of the intermediate results, each generated by the combinatorial part during the individual iteration. The detection capacity and the strongly-fault secure property of the overall structure viewed are thus preserved. This is obtained by inserting the checker in the accumulation loop (either at the output of the adder or at the output of the accumulator).

4. Evaluation and Conclusions

In this paper, the redundant binary representation in the three-rail logic has been shown to be attractive for designing neural architectures with concurrent error detection capabilities, with specific reference to the case of multilayered feed-forward neural networks. To verify the effectiveness and the efficiency of our approaches, we considered two extreme structures (namely, the distributed and the time-multiplexed architectures), all the other solutions being in between. The main results achieved are:

- the complete protection against all unidirectional stuck-at faults on multiple input/output lines, not only for the arithmetic operations but also for the whole interconnections;
- the relevant reduction of latency and, as a consequence, the massive increase of throughput achieved by exploiting the carry-save operations of RBR and the computation parallelism.

The cost we have to pay is the increase of the circuit complexity (due to RBR and three-rail logic) and of the interconnections among neurons (due to three-rail logic).

To evaluate our approach, the architectures proposed here have been designed at a functional level and compared to similar structures based on the traditional binary representation and carry-look-ahead units. The RBR solutions have been completely designed in the three-rail logic, with a step function having a nonzero threshold as an activation function; checkers have been introduced within each neuron to localize the faulty neuron. As an example, each neuron is assumed to have 10 inputs; this does not limit the generality of our results, since it can easily be seen that the latency is proportional to the number of neuron's inputs, as well as the circuit complexity in the distributed architecture.

In the case of the distributed architecture, the percentage increase of the circuit complexity is shown in Fig. 8a: It is about 200 percent.

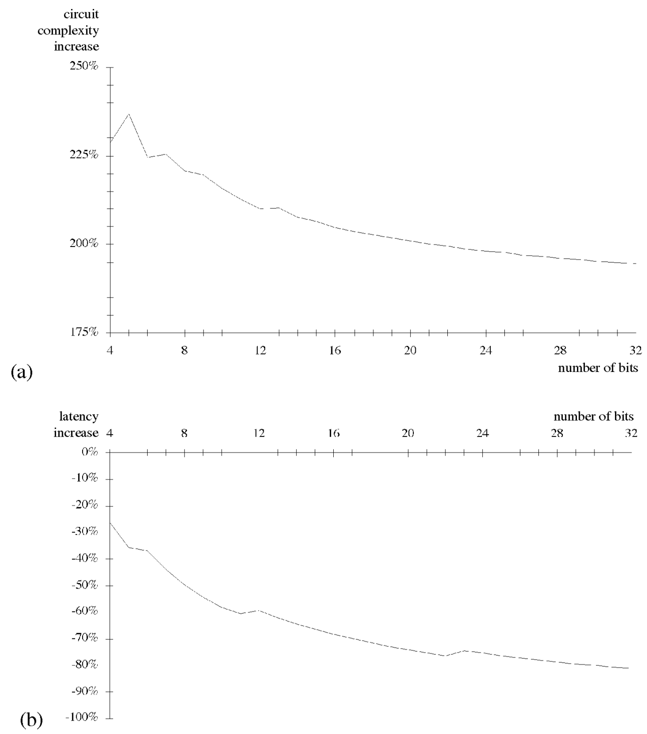


Fig. 8. Evaluation of the distributed architecture: circuit complexity (a), latency (b).

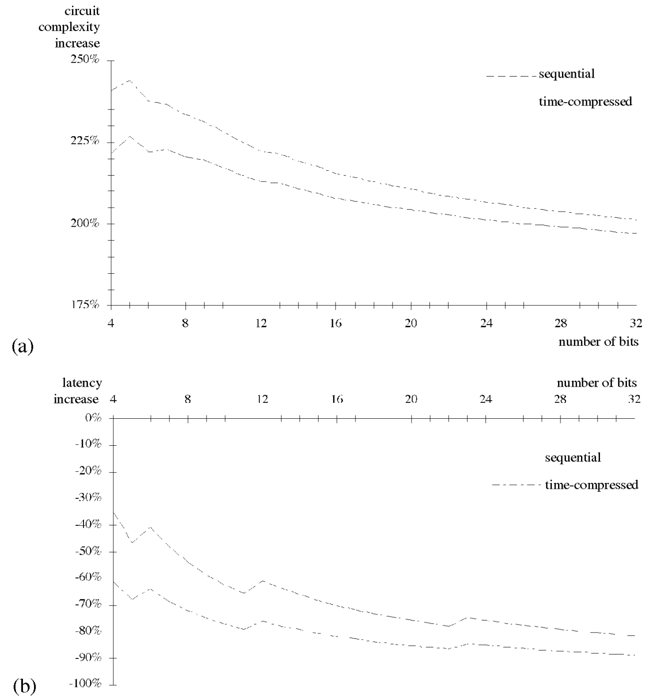


Fig. 9. Evaluation of the time-multiplexed architecture: circuit complexity (a), latency (b).

The percentage increase (actually, a decrease) of the latency is given in Fig. 8b: The distributed architecture is about 60 percent faster and tends to 80 percent, as the number of the operands' bits increases.

For the time-multiplexed architecture, the percentage increase of the circuit complexity is slightly higher than in the distributed solution (see Fig. 9a), while the difference between the sequential and the time-compressed approaches becomes marginal as the number of operands' bits increases. In Fig. 9b, we show the impressive percentage decrease of the latency due to the massive use of carry-save operations.

Therefore, with both architectural approaches, the concurrent detection capability for all unidirectional stuck-at faults on multiple input and/or output lines is achieved by using the presented approaches based on the RBR and the three-rail logic. This technique introduces a circuit complexity increase, but it allows for high performance since the latency greatly decreases, as well as the throughput being enhanced. The interconnection complexity among neurons does not usually become impractical, due to the use of the three-rail logic, since most of the interconnections are local in the proposed architectures; as a consequence, the additional silicon area required by such interconnections is quite limited.

References

[1] J. Hertz and A. Krogh, *Introduction to the Theory of Neural Computation*. Wesley Publishing Co., 1991.

[2] C. Lehmann and F. Blayo, “A VLSI Implementation of a Generic Systolic Synaptic Building Block for Neural Networks,” *Proc. Int'l Workshop VLSI for Artificial Intelligence and Neural Networks*, pp. G7.1-10, Oxford, U.K., Sept. 1990.

[3] S.Y. Kung and J.N. Hwang, “A Unified Systolic Architecture for Artificial Neural Networks,” *Proc. Int'l Conf. Systolic Arrays*, pp. 163-170, San Diego, Calif., 1988.

[4] V. Piuri, M. Sami, and R. Stefanelli, “Fault Tolerance in Neural Networks: Theoretical Analysis and Simulation Results,” *IEEE Proc. Compeuro 1991*, pp. 429-436, Bologna, Italy, May 1991.

[5] M. Stevenson, R. Winter, and B. Widrow, “Sensitivity of Feed Forward Neural Networks to Weight Errors,” *IEEE Trans. Neural Networks*, vol. 1, no. 1, pp. 81-92, Mar. 1990.

[6] C. Alippi, V. Piuri, and M. Sami, “Sensitivity to Errors in Artificial Neural Networks: A Behavioral Approach,” *IEEE Trans. Circuits and System: 1 Fundamental Theory and Applications*, vol. 42, no. 6, pp. 358-361, June 1995.

[7] L. Breveglieri and V. Piuri, “Error Detection in Digital Neural Networks: An Algorithm-Based Approach for Inner Product Protection,” *Proc. 1994 SPIE Conf.—Advance Signal Processing*, pp. 809-820, San Diego, Calif., July 1994.

[8] Y.-M. Hsu, V. Piuri, and E.E. Swartzlander Jr., “Time-Redundant Multiple Computation for Fault-Tolerant Digital Neural Networks,” *1995 IEEE Proc. Int'l Symp. Circuits and Systems*, pp.II.977-II.980, Seattle, Wash., Apr. 1995.

[9] Y.-M. Hsu, V. Piuri, and E.E. Swartzlander Jr., “Recomputing by Operand Exchanging: A Time Redundancy Approach for Fault-Tolerant Neural Networks,” *1995 IEEE Proc. Int'l Conf. Application-Specific Array Processors*, pp. 54-65, Strasbourg, France, July 1995.

[10] V. Piuri, M. Sami, and R. Stefanelli, “Arithmetic Codes for Concurrent Error Detection in Artificial Neural Networks: The Case of AN+B Codes,” *IEEE Proc. Int'l Workshop Defect and Fault Tolerance in VLSI Systems*, pp. 277-286, Dallas, Tex., Nov. 1992.

[11] V. Piuri and M. Villa, “Residue Codes for Concurrent Error Detection in Artificial Neural Networks,” *Proc. IJCNN '93*, pp. IV.825-IV.830, Portland, Ore., July 1993.

[12] A. Avizenis, “Signed-Digit Number Representations for Fast Parallel Arithmetic,” *IRE Trans. Electronic Computers*, vol. 10, pp. 389-400, Sept. 1961.

[13] J.E. Smith and G. Metze, “Strongly Fault Secure Logic Networks,” *IEEE Trans. Computers*, vol. 27, pp. 491-499, June 1978.

[14] N. Takagi and S. Yajima, “On-Line Error-Detectable High-Speed Multiplier Using Redundant Binary Representation and Three-Rail Logic,” *IEEE Trans. Computers*, vol. 36, no. 11, pp. 1,310-1,317, Nov. 1987.

[15] T.R.N. Rao, *Error Coding for Arithmetic Processors*. New York: Academic Press, 1974.

[16] R. David, “Totally Self-Checking 1-out-of-3 Code Checker,” *IEEE Trans. Computers*, vol. 27, pp. 570-572, June 1978.

[17] P. Golan, “Design of Totally Self-Checking Checker for 1-out-of-3 Code,” *IEEE Trans. Computers*, vol. 33, no. 3, p. 285, Mar. 1984.

[18] A.M. Paschalis, C. Efstathiou, and C. Halatsis, “An Efficient TSC 1-out-of-3 Code Checker,” *IEEE Trans. Computers*, vol. 39, no. 3, pp. 407-411, Mar. 1990.

[19] H. Hao and E.J. McCluskey, “Resistive Shorts within CMOS gates,” *Proc. IEEE Int'l Test Conf.*, pp. 292-301, 1991.

[20] C. Metra, M. Favalli, P. Olivo, and B. Riccò, “A Highly Testable 1-out-of 3 CMOS Checker,” *Proc. IEEE Int'l Workshop Defect and Fault Tolerance in VLSI Systems*, pp. 279-286, Venice, Italy, Oct. 1993.

[21] A.M. Usas, “A Totally Self-Checker Design for the Detection of Errors in Periodic Signals,” *IEEE Trans. Computers*, vol. 24, pp. 483-488, May 1975.