

Recomputing by Operand Exchanging: a Time-redundancy Approach for Fault-tolerant Neural Networks

Yuang-Ming Hsu & Earl E. Swartzlander, Jr.
Dept. Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX, USA

Vincenzo Piuri
Dept. Electronics and Information
Politecnico di Milano
Milano, Italy

Abstract

The use of neural networks in mission-critical applications requires concurrent error detection and correction at architectural level to provide high consistency and reliability of system's outputs. Time redundancy allows for fault tolerance in digital realizations with low circuit complexity increase. In this paper, we propose the use of REcomputation with eXchanged Operands - an approach based on operands' rotation - to introduce concurrent error detection and correction, when timing constraints are not particularly strict. Different architectural approaches for neural design are considered to match the implementation constraints and to show the versatility of the proposed solutions.

1. Introduction

Massive computing in neural networks is an attractive solution for several problems, e.g., in the areas of signal and image processing, of robotics, and of real-time control [1]. Physical realizability of dedicated components for on-board applications is nowadays feasible due to the advances of integration technologies.

Validation of the network's outputs becomes a mandatory issue in designing neural architectures for mission-critical applications, e.g., in aerospace applications, in advanced robotized manufacturing plants, and in critical control systems. In such cases, consistency and correctness need to be continuously checked and certified in order to guarantee error confinement and to avoid complete application failure due to errors generated by the neural component. Suited configuration of some neural paradigms (e.g., multi-layered feed-forward networks) allows for a limited amount of intrinsic correction capability [2]; this is accomplished by enforcing the information distribution and computation in the whole network, by means of a modified version of the learning procedure. However, this capability does not provide any information both on the correctness of the current network's output and on the location of the faulty component. Intrinsic error masking may fail for some input sets and from a specific time on, but no signal is generated to prevent the use of the erroneous data generated by the network in the critical application. On the other hand, no assumption can be a priori defined on the error magnitude with respect to the normal magnitude of the network results, so that any error can be extremely critical for the application.

This implies that we need an architectural support to introduce error detection and correction in these computing structures. Besides, when the neural outputs are immediately used by the other components of the application system, we need to certify their correctness as soon as they are generated in order to avoid the use of erroneous outputs for subsequent critical operations. Concurrent techniques must therefore be adopted to provide fault tolerance.

Several approaches have been considered in the literature for the case of digital implementations of neural paradigms. Efficient algorithm-based techniques have been

proposed in [3, 4], but they are either too application-specific or quite expensive in terms of circuit complexity. Data coding techniques has discussed in [5-7] for high-performance digital architectures, even if the high circuit complexity increases the realization difficulties.

In this paper, time redundancy is adopted to introduce fault tolerance in digital neural networks at a low cost in terms of circuit complexity, when the timing performances required to the system are not very strict. Many approaches to time redundancy have been discussed in the literature for fault tolerance in general digital circuits, but no analysis is presently available for neural architectures. The main goal is repetition of the same computation several times by using the system components in a different way for each repetition; results are compared to detect the presence of errors due to hardware faults and, possibly, to correct them. Alternating logic during recomputation was used in [8] to detect errors in adders and multipliers, but it is not able to provide correction since only two alternative computational ways are available. REcomputation with Shifted Operands (RESO), REcomputing with Swapped Operands (RESWO), and REcomputing with Circularly-shifted Operands (RECO) were proposed in [9], [10], and [11], respectively; however, they are expensive even if some extensions allow for dealing with error correction. REcomputation with Duplication With Comparison (REDWC) and REcomputation with Triplication With Voting (RETWV) were proposed in [10, 12] to detect and correct errors at a limited but not very low increase of circuit complexity.

In this paper, we focus our attention on a technique based on circular rotation of operands during the redundant computational phases; we call this approach REcomputation with eXchanged Operands (REXO). While RESO, RESWO, and RECO apply rotation at the level of operand's bits, REXO considers rotation at the level of the whole set of operands required to generate the individual result by a complex computation (e.g., as in convolution or in weighted summation within neurons). A related approach based on simple shifting without recirculation was proposed in [13] for array processors, but requiring the use of additional processing elements and additional circuits to feed the operands. Section 2 introduces the general architectural schemes implementing the neural paradigm, with particular reference to multi-layered feed-forward networks (even if the techniques can be adopted for any kind of neural paradigm). Section 3 presents REXO and the related neural architectures for concurrent error detection; Section 4 proposes the REXO technique for concurrent error correction.

2. Neural paradigms and digital architectures

Digital implementation of neural paradigms has been extensively studied to achieve massive computational capabilities while preserving the physical realizability by using the current VLSI integration technologies. Many architectures were presented in the literature, ranging from the dedicated structures to the general-purpose neurocomputers. Since we are concerned with application-specific systems, we consider only digital realizations of dedicated neural architectures. These structures are capable of performing only one class of neural computation. The topology and the neuron's operations are fixed; at most, the interconnection weights can be changed by downloading them before the beginning of the network operation. As an example, we consider the multi-layered feed-forward networks [1], even if the approaches can be applied to any kind of neural paradigm. Operation of neuron i is defined by $y_i = f_i(\sigma_i - \theta_i) = f_i(\sum_k w_{ik} x_k - \theta_i)$, being x_k the neuron's input from neuron k , w_{ik} the interconnection weight from neuron k to neuron i , y_i the neuron's output, θ_i the threshold, σ_i the weighted summation of neuron's inputs, and f_i the non-linear activation function. No restriction is imposed both on the nature of the non linear function and on the values of the interconnection weights. Operands are assumed integer or full-fractional real numbers in the

two's complement representation; both neuron's inputs and weights are represented with n bits (minor changes are required for operands with different size) and are managed in bit-parallel way (extension to bit-serial and to serial/parallel cases are possible).

The architectural approaches proposed in the literature may be partitioned in three main classes (see Fig. 1): *direct-mapping* structures, *pipelined* architectures, and *time-multiplexed* solutions. Direct implementation of the neural paradigm can be achieved by one-to-one mapping of neural operators onto digital components [14]. In each neuron, every arithmetic module M_k generates one product term of the input weighted summation from one neuron's input and the related interconnection weight; this term is then added to the partial summation produced by the previous module. The new partial summation is propagated to the subsequent module, until the last module fed the complete weighted summation into the circuit evaluating the activation function. Finally, the neuron's output is delivered to the receiving neurons.

To increase the system throughput, pipelining can be adopted; the basic operation within the individual neuron are cascaded and separated in time by suitably latching in registers the data transferred between adjacent arithmetic modules, so that the period of input presentation is reduced to the latency of the single pipeline stage. Registers are also introduced at neurons' outputs to provide the suited data presentation to the receiving neurons.

To decrease the overall circuit complexity, time-multiplexing of some components can be exploited, despite an increased latency and a reduced throughput: for example, only one multiplier and one adder-accumulator can be used for each neuron, providing a suited feeding of interconnection weights and neuron's inputs. Weights are circularly rotated in the registers' file within each neuron. Inputs presentation is achieved by multiplexing the interconnection structure between neurons in adjacent layers. Only one data path including the neuron's output registers is used between adjacent layers: neurons' outputs are stored in parallel in the output registers and, then, they are circularly shifted during each clock period, so as to present all of them at the proper time to the receiving neurons.

By abstracting from the specific implementation technology, we consider the problem of detecting and correcting any transient or permanent error in the result of any arithmetic unit and interconnection path, whenever faults affect the computation of only one neuron. Any error multiplicity (i.e., any number of bits may be erroneous in the result) should be covered.

3. The use of REXO for concurrent error detection

Recomputation with eXchanged Operands works at a high abstraction level on the arithmetic operations performed within the neural computation. We consider each operand as an atomic entity and we apply the ideas of partitioning and rotated allocation to subsets of operands on different arithmetic units. In other words, we focus our attention not on the simple arithmetic operations (addition and multiplication) to identify the scope of partitioning and rotation. We afford such a problem at the level of the complex arithmetic operation required to generate the excitation input for the non-linear activation function starting from neuron's inputs and weights: namely, the weighted summation, or - in mathematical terms - the inner product operation between inputs and weights.

Consider first protection of the linear part of the neural computation. In the individual neuron i , we define the input set X as $X = \{x_1, x_2, x_3, \dots, x_N\}$ and the weight set W as $W = \{w_{i1}, w_{i2}, w_{i3}, \dots, w_{iN}\}$, being x_k the neuron's inputs and w_{ik} the interconnection weights. Application of REXO for concurrent error detection consists of partitioning both the set X and the set W in two subsets. We obtain the least-significant part and the most-significant part of the input set (denoted by LSI and MSI, respectively), and the least-significant part and the

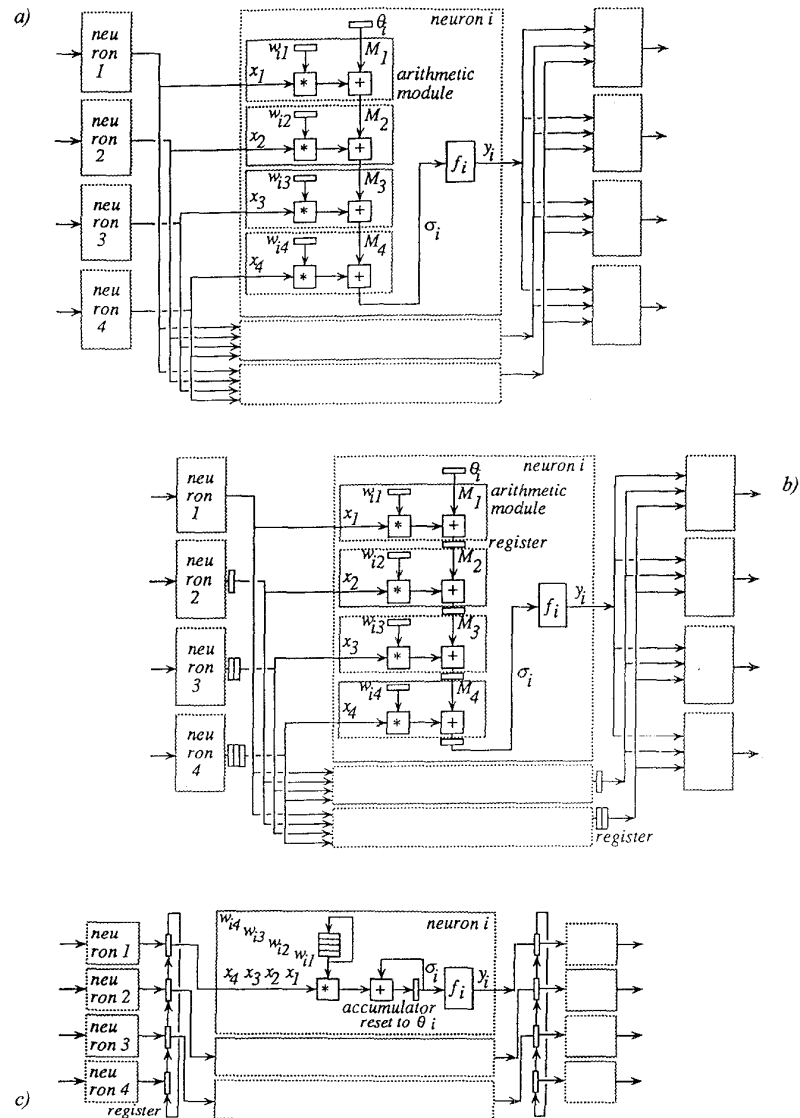


Fig. 1 - The neural architectures: (a) direct-mapping approach, (b) pipelined architecture, (c) time-multiplexed structure

most-significant part of the weight set (denoted by LSW and MSW, respectively). They are defined as: $X^L = \{x_1, x_2, x_3, \dots, x_K\}$, $X^M = \{x_{K+1}, x_{K+2}, x_{K+3}, \dots, x_N\}$, $W^L = \{w_{i1}, w_{i2}, w_{i3}, \dots, w_{iK}\}$, $W^M = \{w_{i,K+1}, w_{i,K+2}, w_{i,K+3}, \dots, w_{i,N}\}$, being $K = \lfloor N/2 \rfloor$.

The complete complex arithmetic unit computing the inner product on the nominal operands' sets is divided in two parts of equal size: the least-significant part and the most-significant part are created by including K nominal arithmetic modules in each of them. Therefore, each of them is capable of performing half of the complete inner product on half of the operands' sets. Two different approaches for direct-mapping architectures are shown in Figg. 2 and 3. Their general structures are identical: the main difference is in merging the partial results generated by the two parts of the inner product units. The corresponding solutions for the pipelined network architecture can be easily derived from the case of direct-mapping network by introducing suited pipeline registers.

During the nominal phase, the LSI and the LSW are presented to the least-significant part of the inner product unit, while the MSI and the MSW are fed into the most-significant part of the inner product unit. Since each arithmetic module deals with two alternative neuron's inputs according to the operation phase, an input multiplexer is used to select the input corresponding to the current phase. In the j -th module of the least-significant part of the inner product unit, the input multiplexer selects the j -th neuron's input during the nominal phase and the $(j+K)$ -th neuron's input during the redundant one; in the $(j+K)$ -th module, the input multiplexer performs the opposite choice. Interconnection weights must be treated in a similar way.

The least-significant part of the inner product unit apply the inner product to the first part of the operands' sets to generate the partial weighted summation $\sigma_i^L = \sum_{k=1, K} w_{ik} x_k$, while the most-significant part of the inner product unit performs the same operation on the second part of the operands' sets producing the partial weighted summation $\sigma_i^M = \sum_{k=K+1, N} w_{ik} x_k$.

To compute the final result $\sigma_i - \theta_i$, these two partial summation and the possible threshold θ_i must be added together. This merging operation can be executed by a dedicated three-operands adder - namely, the two partial summations and the threshold - (e.g., by the adder on the left side of Fig. 2). In the case all arithmetic modules are identical, i.e., all of them (including the first module of the least-significant part and the first module of the most-significant part of the inner product unit) contain the adder, the threshold can be incorporated in the weighted summation by presenting it at the summation input of the first module of least-significant part of the inner product unit; the dedicated adder becomes a two-operands adder.

An alternative approach is based on suited propagation of intermediate results between the complex arithmetic sub-systems (see Fig. 3). The threshold can be fed in the first module of least-significant part of the inner product unit as in the nominal architecture. The partial summation σ_i^L is propagated from the last module of least-significant part of the inner product unit into the first module of most-significant part of the inner product unit, as it was a threshold. The final result $\sigma_i - \theta_i$ is automatically generated at the output of the last module of most-significant part of the inner product unit.

During the redundant phase, the LSI and the LSW are used by the most-significant part of the inner product unit, while the MSI and the MSW are given to the least-significant part of the inner product unit. The most-significant part of the inner product unit produces the second copy of σ_i^L , while the least-significant part of the inner product unit evaluates σ_i^M .

If N is odd, the least-significant part and the most-significant part of the inner product unit have anyway the same number K of modules to accommodate the operands LSI and LSW during both phases. An extra weight $w_{i,N+1}$ is set to 0 so that, even if x_{N+1} is undefined, the term $w_{i,N+1}x_{N+1}$ - evaluated by the $(N+1)$ -th module - gives always a null contribution to σ_i .

Fig. 2 - The neural error-detecting architecture for the case of the direct-mapping structure.

The structure shows the approach when $f_i(\cdot)$ is not protected by REXO and dedicated adders are used for merging.

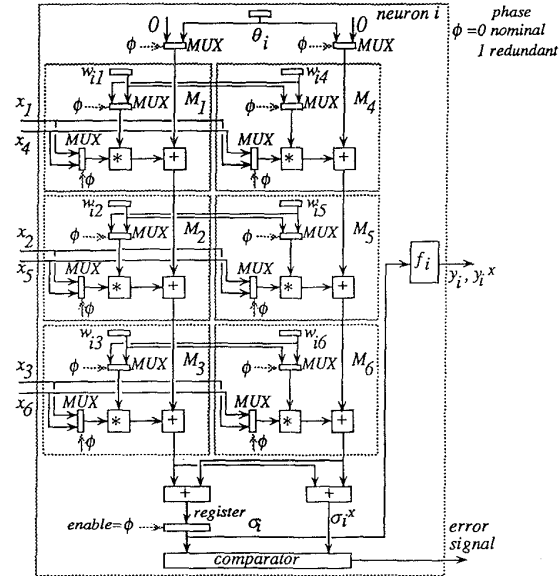
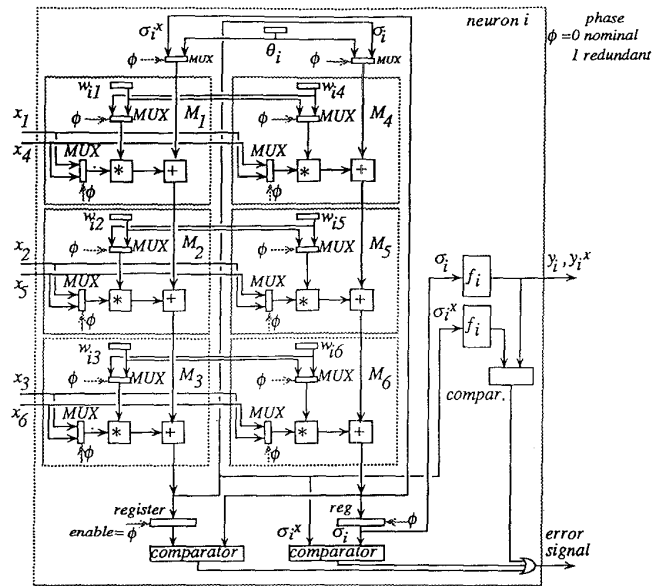


Fig. 3 - The neural error-detecting architecture for the case of the direct-mapping structure. The structure shows the approach when $f_i(\cdot)$ is protected by duplication within REXO and recirculation is used for merging.



Reliability is not particularly affected by such a register since only a very limited amount of additional complexity is required with respect to the complexity of the whole inner product unit, while regularity and modularity are preserved (this is particularly useful to implement configurable devices in which weights are downloaded at the beginning of the computation). Alternately, complexity can be reduced and reliability lightly enhanced by customizing the multiplexers connected to the redundant zero weight and by removing the extra register.

The two partial summation and the threshold must be merged to create the final result. Merging can be executed by the same dedicated adder used in the first phase.

Alternately, suited propagation of intermediate results can be adopted, as in Fig. 3: the threshold is presented to the first module of most-significant part of the inner product unit, while the partial summation σ_r^L is propagated from the last module of most-significant part into the first module of least-significant part. The final result is now produced at the output of the last module of least-significant part of the inner product unit.

Output checking is performed by comparing the final results generated during the two phases. The REXO checker is composed by a register and by a comparator; the register input is connected to the component generating the result of the nominal phase (namely, the dedicated adder or the last module of most-significant part of the inner product unit, according to the architectural approach adopted). The comparator's inputs are connected to the register holding the nominal result and to the circuit generating the redundant copy of the weighted summation. The output of the comparator is valid only when both nominal and redundant results have been presented at the comparator inputs.

A self-checking checker should be adopted to protect also this circuit; to such purpose, we can duplicate the whole checker, or duplicate the register and use a self-checking comparator with three inputs (i.e., the registers and the function generator).

If the same circuits are used to generate both nominal and redundant values of the weighted summation (as described above), error detection is not fully guaranteed.

If the final result is generated by the single dedicated adder on the left side of Fig. 2, faults in such an adder may not be detected for some operands' values since the same circuit is used in both phases. Duplication of this adder (one for each phase) guarantees the use of two independent ways for computing the final result. The leftmost dedicated adder in Fig. 2 is used during the first phase and its output is stored in the checker's register; its output is ignored during the redundant phase. The second dedicated adder on the right side is considered only during the second phase. The checker's comparator is connected to such register and to the output of the second dedicated adder, so that checking is always effective to detect errors in the whole linear part of the neuron.

In the case of data propagation, the chain of adders in the arithmetic modules is used in both phases, even if with operands in different locations. This allows for possible error masking, according with the specific operands' values. To avoid this drawback, we compare both the final results of the weighted summation and the intermediate values propagated from one subsystem of arithmetic modules to the other (see Fig. 3). Two checkers are adopted: one for the final weighted summations (as discussed above), the other one for intermediate results. The second checker contains a register to hold the intermediate result generated at the output of the last module of the least-significant part of the inner product unit during the nominal phase (i.e., the result propagated to the first module of the most-significant part); its comparator is connected to the output of such a register and to the output of the last module of the most-significant part of the inner product unit, so that it can compare the value stored in the register to the intermediate result that is propagated from most-significant part of the inner product unit to least-significant part during the redundant phase.

The register holding the interconnection weights are critical for error detection. Since they are unique in each neuron, they cannot be protected by REXO. If no other protection is adopted, they are an hard-core. A possible solution (even if it is expensive) is duplication of the registers themselves: each pair of registers stores the same weight value. In direct-mapping or pipelined architectures, the two instances are connected to the multiplier by a multiplexer: one register is used during the nominal phase, the other during the redundant one.

Protection of the activation function $f_i(\cdot)$ is performed by considering the specific kind of non-linearity. In the case of a linear function $f_i(\cdot)$, the weighted summation is multiplied by a gain coefficient and a possible displacement coefficient is added. Implementation can be obtained by modifying the interconnection weights and the threshold; the distributive property of the product with respect to the addition allows for applying the gain coefficient directly to the weights and by incorporating the displacement into the threshold. No checking is needed on the weighted summation since possible errors are propagated to the neuron's output due to the linearity of all operations; for the same reason, no checking is required on the neuron's outputs (except on the final network's outputs). If no checker is used in the neurons, the fault location is loosen, since no information is available on the functioning state of every neuron; besides, paths' reconvergence may lead to intrinsic error masking. If only neuron's output are checked, identification of the faulty neuron is feasible by observing the checkers' error signal.

In the case the circuit evaluating $f_i(\cdot)$ is very large (see the example shown in Fig. 2), we can choose not to protect the non-linear function by assuming it as hard-core. To avoid a multiplexer at the input of the function generator, the value stored in the checker's register is used as unique input by the function generator during both phases without any decrease in the detection capabilities. Checking is applied only on the weighted summations. If the fault is in the circuits evaluating the weighted summation, identification of the faulty neuron is feasible; if the fault is in the activation function generator, we can identify only the layer containing the fault since the error is propagated to all neurons in the subsequent layer and all receiving neurons have active error signal, while the faulty neuron has no active error signal.

For some implementation of $f_i(\cdot)$, specific techniques can be adopted to detect errors in this circuits; the linear and the non-linear parts are checked separately. Fault location is feasible.

If the circuit evaluating $f_i(\cdot)$ has a limited complexity (e.g., the single-step function), we can protect the non-linear function by using duplication (see Fig. 3). The outputs of the two instances are exactly identical in the absence of errors during both phases; the output of only one of them needs to be propagated as neuron's output. By connecting the generator input to the output of the register holding the weighted summation produced during the first phase, the output of the function generator used during the first phase is still valid also during the second one for the receiving neurons. No multiplexer is required to select the neuron's output for the corresponding phase. An error in the weighted summation is detected; the result delivered by the faulty neuron and the results generated by the whole network are not relevant. Checking must be applied to the output of the two function generators to protect them; the register in the checker can be avoided since the function value generated by the first phase is valid during the second one. This second checker provides - with the first one - the identification of the faulty neuron. When checking on neuron's output is not performed, each function generator is connected to the corresponding weighted summation and a multiplexer is used to select the generators' output that must be propagated as neuron's output. Fault location is reduced.

REXO is not efficient for time-multiplexed neurons: contemporaneous execution of the two inner products (even if having reduced size) requires duplication of all circuits involved in weighted summation. The circuit complexity arises so that REXO is equivalent to duplication with comparison.

4. The use of REXO for concurrent error correction

Extension of REXO to error correction is straightforward from the simple detection. In the individual neuron i , the input set X is partitioned in three subsets: the least-significant part LSI, the intermediate-significant part ISI, and most-significant part MSI. Similarly, the weight set W is partitioned into the least-significant part LSW, the intermediate-significant part ISW, and the most-significant part MSW. They are $X^L = \{x_1, \dots, x_K\}$, $X^I = \{x_{K+1}, \dots, x_{2K}\}$, $X^M = \{x_{2K+1}, \dots, x_N\}$, $W^L = \{w_{i1}, \dots, w_{iK}\}$, $W^I = \{w_{iK+1}, \dots, w_{i2K}\}$, $W^M = \{w_{i2K+1}, \dots, w_{iN}\}$, respectively, being $K = \lceil N/3 \rceil$. The arithmetic unit computing the inner product on the nominal operands' sets is divided in three parts of equal size: the least-significant part, the intermediate-significant part, and the most-significant part. Each of them includes K arithmetic modules (null extra-weights are introduced when N is not multiple of 3). The same two architectural approaches available for detection in the direct-mapping structure can be used also for correction, as shown in Figg. 4 and 5. The internal structure of the individual arithmetic units is identical to the nominal one. Use of REXO for the time-multiplexed neuron's structure is not efficient, as in detection.

During the nominal phase, LSI and LSW are presented to the least-significant part of the inner product unit, ISI and ISW are used by the intermediate-significant part, and MSI and MSW are fed into the most-significant part. In each arithmetic module, a three-input multiplexer selects the suited neuron's input according to the operation phase, as well as another three-input multiplexer selects the corresponding weight. In the j -th module of the least-significant part of the inner product unit, each of them chooses the j -th value in the related set during the nominal phase, the $(j+K)$ -th value during the first redundant phase, and the $(j+2K)$ -th value during the second redundant phase. In the $(j+K)$ -th module, the selections are the $(j+K)$ -th, the $(j+2K)$ -th, and the j -th values, respectively. In the $(j+2K)$ -th module, they are the $(j+2K)$ -th, the j -th, and the $(j+K)$ -th values, respectively.

During the nominal phase, the least-significant part of the inner product unit, the intermediate-significant part, and the most-significant part apply the inner product operation to the correspondent pair of operands' subsets and generate the three partial weighted summations $\sigma_i^L = \sum_{k=1,K} w_{ik} x_k$, $\sigma_i^I = \sum_{k=K+1,2K} w_{ik} x_k$, and $\sigma_i^M = \sum_{k=2K+1,N} w_{ik} x_k$, respectively.

The final result $\sigma_i - \theta_i$ is obtained by merging the three partial summations and the possible threshold. Two approaches may be envisioned as we did in Section 3: a dedicated multi-operands adder or suited propagation of the intermediate results. If a dedicated adder is used, three instances are introduced to provide enough redundancy for error correction (Fig. 4).

Propagation is shown in Fig. 5. The threshold is fed in the first module of the least-significant part of the inner product unit, σ_i^L is propagated from the last module of the least-significant part into the first module of the intermediate-significant part, σ_i^I is propagated from the last module of the intermediate-significant part into the first module of the most-significant part, and the final result $\sigma_i - \theta_i$ is generated at by the last module of the most-significant part.

During the first redundant phase, LSI and LSW are used by the most-significant part, ISI and ISW by the least-significant part, and MSI and MSW by the intermediate-significant part. The most-significant part, the least-significant part, and the intermediate-significant part produce the second copy of σ_i^L , σ_i^I , and σ_i^M , respectively. Merging is performed by the second dedicated adder of the propagation chain; the threshold is presented to the first module of the most-significant part, σ_i^L is propagated from the last module of the most-significant part into the first module of the least-significant part, σ_i^I is propagated from the last module of the least-significant part into the first module of the intermediate-significant part, and σ_i^M is generated by the last module of the intermediate-significant part.

During the second redundant phase, LSI and LSW are used by the intermediate-significant part of the inner product unit, ISI and ISW by the most-significant part, and MSI and MSW by the least-significant part. The intermediate-significant part of the inner product unit, the most-significant part, and the least-significant part compute σ_i^L , σ_i^I , and σ_i^M , respectively. Merging is performed by the third dedicated adder or by the propagation chain (the threshold is given to the first module of the intermediate-significant part of the inner product unit, σ_i^L is propagated from the last module of the intermediate-significant part to the first module of the most-significant part, σ_i^I is propagated from the last module of the most-significant part to the first module of the least-significant part, and σ_i^M is generated at the output of the last module of the least-significant part of the inner product unit).

Checking is performed by comparing the final results generated during the three phases. The checker for the case of REXO error correction contains two registers and a three-inputs comparator; these registers store the results generated by the nominal phase and by the first redundant one. The comparator's inputs are connected to the registers and to the circuit generating the second redundant copy of the weighted summation. The approach based on dedicated adders is shown in Fig. 4. For data propagation, we need three checkers to avoid possible masking: one for the final summations and one for each of the triplets of σ_i^L and σ_i^I .

Correction of the weighted summation can be applied instead of simple checking when the characteristics of the activation function may lead to decrease the correction capabilities. The correcting circuit is similar to the checking one: two registers hold the values generated during the nominal phase and the first redundant one, while the majority voter evaluates the correct result from the values stored in the registers and the current value at the input of the correcting circuit during the second redundant phase (i.e., the third copy of the weighted summation). To avoid possible error masking, detection can be adopted for the partial summations.

Protection of the activation function $f_i(\cdot)$ is related to the function's characteristics, as in the case of detection. Identification of the faulty neuron suffers from the same problems discussed in the previous section. If the generator has limited complexity, triplication is used (Fig. 4). Whenever the function characteristics do not reduce the correction capabilities, simple error checking is applied to the weighted summation. The input of the nominal instance of the function generator and the input of the first redundant one are connected to the output of the corresponding register within the checking circuit discussed above. The input of the third instance is connected with the checker's input. The output of these instances is valid only at the end of the corresponding phase. A simple majority voter is connected at the outputs of the function generators to produce the corrected neuron's output.

If simple error checking of the weighted summation is not suited, correction must be adopted: a correcting circuit replaces the checker on the summation bits and a register is introduced to hold the corrected weighted summation during the three phases of the receiving neurons. All the three instances are connected to the output of the summation voter and a simple majority voter is connected at their outputs. In both cases, a register at the voter's output holds the neuron's output value for the receiving neurons during the three phases. Such a register is hard-core since it is unique: triplication (one instance for each function generator) could be considered despite of a massive increase of circuit complexity.

If the function generator is hard-core (Fig. 5), voting is applied only to the weighted summation to preserve the correction capabilities from the negative influence both of nonlinearities and of the reconvergence of the computation through the subsequent network layers. Location of the faulty neuron is possible if the fault occurs in the circuits of the weighted summation. For specific implementation of $f_i(\cdot)$, dedicated techniques can be adopted for separate error correction or, at least, detection; in this case, location of the faulty neuron is

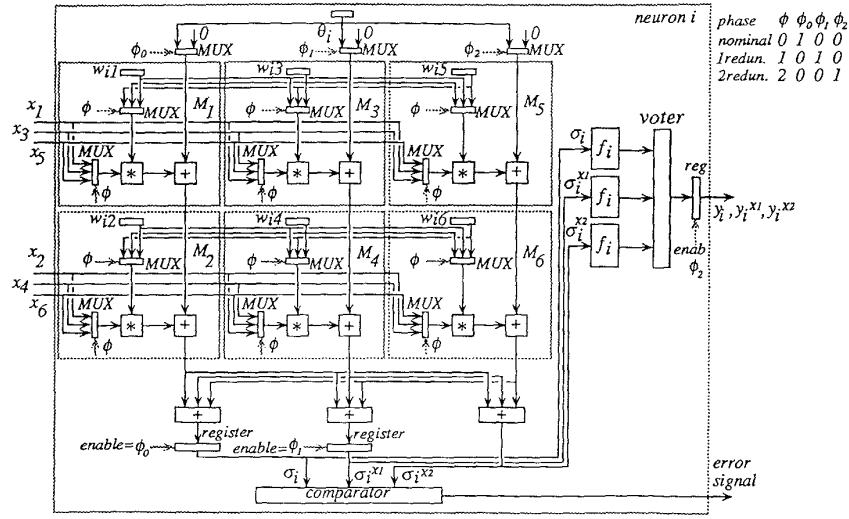


Fig. 4 - The neural error-correcting architecture for the case of the direct-mapping structure. The structure shows the approach when $f_i(\cdot)$ is protected by triplication within REXO and dedicated adders are used for merging.

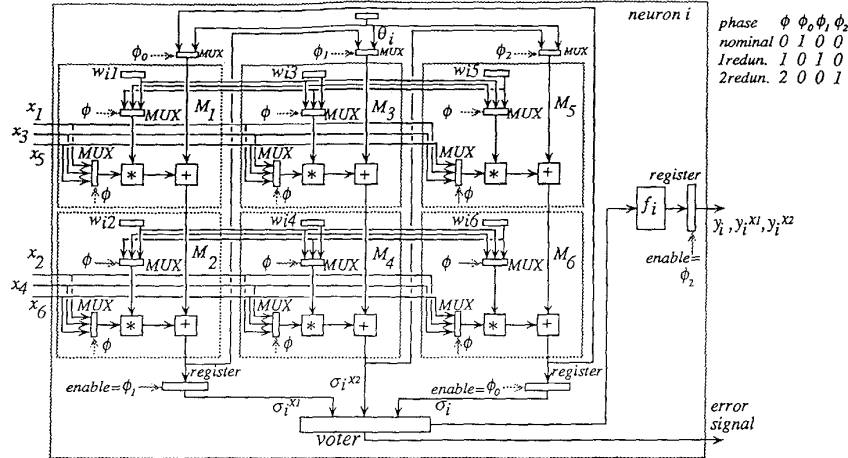


Fig. 5 - The neural error-correcting architecture for the case of the direct-mapping structure. The structure shows the approach when $f_i(\cdot)$ is not protected by REXO and recirculation is used for merging.

always feasible. The function generator receives the voter's output and stores the neuron's output in an output register.

If the activation function $f_i(\cdot)$ is linear, checking is not required within the individual neuron since the error is always propagated to the output of the activation function. Only the correcting circuit at the neuron's output must be used to preserve the correction capabilities.

5. Conclusions

Design of fault-tolerant dedicated architectures for neural networks has been discussed when minimization of the circuit complexity is mandatory to allow physical realizability both for the silicon area occupied and for power consumption, while time performance of the network is not so strict. Continuous validation and, whenever possible, correction of the network's outputs has been provided by recomputation with exchanged operands, a time-redundancy technique.

REXO needs a small redundant circuit complexity except for the time-multiplexed neural architecture; in fact, the internal structure of the arithmetic units is not modified. However, the interconnection structure is quite relevant; therefore, identification of the most suited approach must take into account not only the pure circuit complexity (e.g., evaluated by considering gates' count), but also the interconnections. Both detection and correction are fully guaranteed.

References

1. J. Hertz, A. Krogh, *Introduction to the Theory of Neural Computation*, Wesley Publishing Co, 1991
2. M. Stevenson, R. Winter, B. Widrow, "Sensitivity of feedforward neural networks to weight errors", *IEEE Trans. on Neural Networks*, Vol. 1, N. 1, March 1990
3. V. Piuri, "An algorithmic approach to concurrent error detection in artificial neural networks", *IEEE Proc. Int'l Workshop on VLSI Signal Processing*, Napa, CA, Oct. 1992
4. L. Breveglieri, V. Piuri, "Error detection in digital neural networks: an algorithm-based approach for inner product protection", *Proc. 1994 SPIE Conference - Advance Signal Processing*, S. Diego, CA, July 1994
5. V. Piuri, M. Sami, R. Stefanelli, "Arithmetic codes for concurrent error detection in artificial neural networks: the case of AN+B codes", *IEEE Proc. Int'l Workshop on Defect and Fault Tolerance in VLSI Systems*, Dallas, TX, Nov. 1992
6. V. Piuri, M. Villa, "Residue codes for concurrent error detection in artificial neural networks", *Proc. IJCNN'93*, Portland, OR, July 1993
7. S. Bettola, V. Piuri, "The use of redundant binary representation for fault tolerant neural networks", *Proc. Int'l Conference on Neural Networks*, Orlando, FL, July 1994
8. D. A. Reynolds and G. Metzger, "Fault detection capabilities of alternating logic", *IEEE Trans. Comput.*, Vol. 27, pp. 1093-1098, 1978
9. J. H. Patel and L. Y. Fung, "Concurrent error detection in ALU's by recomputing with shifted operands", *IEEE Trans. Comput.*, Vol. 31, pp. 589-595, 1982
10. B. W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, 1989
11. L.G. Chen and T.H. Chen, "Fault-tolerant serial-parallel multiplier", *IEEE Proc.-E*, Vol. 138, pp. 276-280, 1991
12. Y.-M. Hsu and E.E. Swartzlander, Jr., "Time redundant error correcting adders and multipliers", *IEEE Proc. Int'l Workshop on Defect and Fault Tolerance in VLSI Systems*, Dallas, USA, 1992
13. A. Majumdar, C.S. Raghavendra, and M.A. Breuer, "Fault tolerance in linear systolic arrays using time redundancy", *IEEE Trans. on Comput.*, Vol. 39, pp. 269-272, 1990
14. C. Lehmann, F. Blayo, "A VLSI implementation of a generic systolic synaptic building block for neural networks", *Proc. Int'l IFIP Workshop on VLSI for Artificial Intelligence and Neural Networks*, St. Paul de Vence, France, 1990