A NEW LEARNING APPROACH TO DESIGN FAULT TOLERANT ANN: FINALLY A ZERO HW-SW OVERHEAD

Djones Lettnin[1], Fabian Vargas[2], Diogo Brum[3], Dárcio Prestes[4]

Catholic University – PUCRS
Electrical Engineering Dept.
Av. Ipiranga, 6681. 90619-900 Porto Alegre, Brazil
[1]djones@ee.pucrs.br, [2]vargas@computer.org, [3]diogo@ee.pucrs.br , [4]darcio@ee.pucrs.br

Abstract

We present a new approach[*] to design fault tolerant artificial neural networks (ANNs). Additionally, this approach allows estimating the final network reliability. This approach is based on the Mutation Analysis technique and it is used during the training process of the ANN. The basic idea is to train the ANN in the presence of faults (single-fault model is assumed). To do so, a set of faults is injected into the code describing the ANN. This procedure yields mutation versions of the original ANN code, which in turn are used to train the network in an iterative process with the designer until the moment when the ANN is no more sensible to the single faults injected. In other words, the network became tolerant to the considered set of faults. A practical example illustrates the proposed methodology, where an ANN is used to classify electrocardiogram (ECG).

Keywords: *Artificial Neural Networks; Mutation Analysis; Fault-Tolerant Computing Systems; Electrocardiogram Classification.*

1. Introduction

The use of ANNs in critical applications implies adoption of specific techniques to ensure system's reliability. In this paper, we present a new learning approach used to train the ANN in the presence of faults. The ultimate goal of this approach is to render the ANN tolerant to the set of faults for which it was trained.

When compared to other techniques found in the literature used to implement fault tolerant ANNs [1], the proposed approach does not implies in system's performance degradation for the case the ANN is implemented in the form of software, or even area overhead, if the ANN is mapped into hardware (i.e., an ASIC or FPGA). The primary reason for such an attractive advantage is the fact the proposed approach affects only the learning process of the ANN, thus, only the values of the weights associated with the neuron inputs are modified. The whole ANN architecture is maintained as in the original form, in terms of interconnections and internal blocks. According to our knowledge, it is also worth to mention that, so far, commercially available HW-implemented ANNs [2,3] do not have been designed by integrating fault tolerance techniques into their architecture [4,5]. This situation is economically very interesting because commercial chips can become fault tolerant by using the proposed approach without any HW design modification. This means that from the user's point of view, system is unchanged, and only the training procedure is modified to attend the requested fault-tolerant criteria.

2. Correlating the Mutation Analysis Technique and the Proposed Approach

In the following, we propose an adaptation of the mutation analysis approach, originally proposed for software testing in 1978 by DeMillo *et al*. [6,7]. At that time, Mutation Analysis was proposed as a method for evaluating the adequacy of a set of test vectors for a program. Informally, test vectors are considered *mutation-adequate* for a program if they can distinguish the program from programs that differ from it by small syntactic changes [6]. Although mutation analysis is the basis for a test vectors selection criterion, in our work we are concerned with its use as a criterion for *fault-tolerance insertion/reliability estimation by means of training/simulating the ANN in the presence of faults*.

If the obtained reliability level attends the specification, then ANN can be compiled for the target processor (or synthesized, if it is to be implemented in hardware). At this moment, it is said that the *ANN is tolerant to the assumed set of faults*

For this purpose, a set of mutation operators dedicated to C++ has been developed and considered as a functional fault model (see Table 1).

Type	Description	Type	Description
AOR	Arithmetic Operator Replacement	ROR	Relational Operator Replacement
ABS	Absolute Value Insertion	VCR	Variable for Constant Replacement
CR	Constant Replacement	VR	Variable Replacement
CVR	Constant for Variable Replacement	UOI	Unary Operator Insertion
LOR	Logical Operator Replacement	BOR	Bit Operator Replacement

Table 1. Mutation operators set for C++ functional descriptions.

To inject a fault, a mutation operator should be applied on the original C++ description. Fig. 1 gives an example of eight mutants for routine *neuronbad*. This routine describes the behavior of the neuron used in the ANN case study. Note that each of the eight mutant statements is executed one at a time during the fault simulation process. Additionally, it is worth to mention that ***Mutants 1*** to ***5***: ***AOR***, ***VCR***, ***VR***, ***ABS*** and ***ROR*** were used to train the case study during the learning process in the presence of faults (i.e., during the fault simulation process).

```
float neuronbad::SumFunction(float *input) { //Neuron Core:: Summation Function
        float output_SF = 0.0;
        for (int i=0; i< w_size; i++){
            output_SF += weight[i]*input[i];
            Δoutput_SF -= weight[i]*input[i];   // AOR: Mutant 1
            Δoutput_SF += 1*input[i];           // VCR: Mutant 2
            Δoutput_SF += weight[i]*input[w_size];  // VR: Mutant 3
        }
        return output_SF;
}
void neuronbad::ActFunction(float output_SF) {    // Activation Function
        float u;
        u = -TWO*B*output_SF;
        Δu = -TWO*B*fabs(output_SF);          // ABS: Mutant 4

        if (u>UMAX) u = UMAX;
        Δif (u>UMAX) u = output_SF;           // CVR
        Δif (u<UMAX) u = UMAX;                // ROR: Mutant 5
        Δif (u<UMAX) u = B;                   // CR

        output = (TWO*A/(1+exp(u))-A);
        Δoutput = (TWO*A/(1+exp(++u))-A);     // UOI

}
```

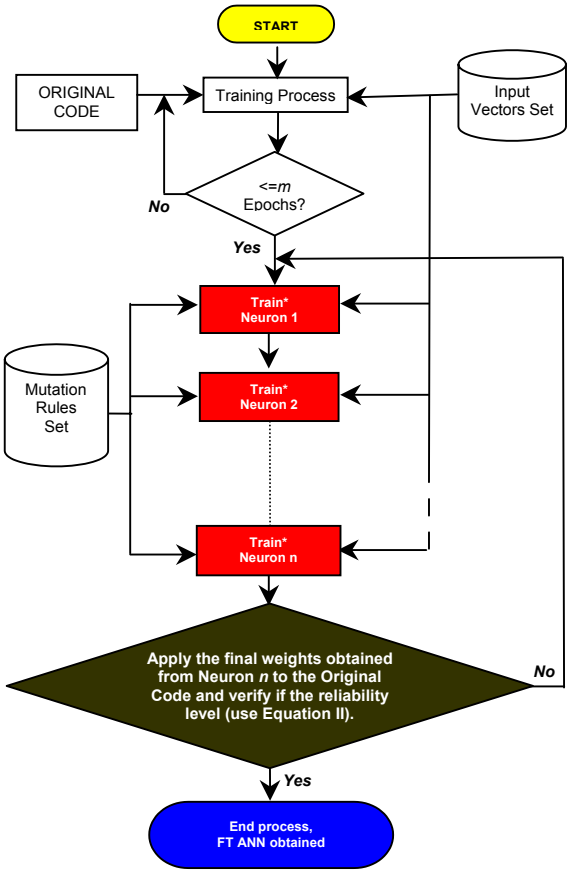
Fig. 1. Example of fault injection in a neuron described in the C++ language. (The symbol Δ identifies mutated statements.)

3. The Proposed Methodology

The development of an artificial neural network is basically divided into three steps: a) *Architecture Specification; b)* Definition of the *Learning Paradigm c) Selection of the test patterns and validation.*

[*] *This work is partially supported by CNPq and FAPERGS.*

The proposed approach is applied to the second and third steps of the ANN design flow. More precisely, it is applied to the *supervised learning process*, as seen in Fig. 2. With this goal in mind, we have proposed the following methodology:



*: Before moving to train the subsequent mutant, ensure that the current mutant has been trained for all the network neurons.

Fig. 2. Proposed Methodology: *Fault-Tolerant ANN Design Flow*.

a) Initially, we execute the original program in order to train the network with respect to a given *Input Vectors Set*. In this step, identically to any other conventional procedure, a given number of epochs is used to train the ANN. At the end of this process, save the final weights of all neurons of the network to be used in the next step of the methodology.

b) Next, we inject faults into the original code by generating mutated versions of it. These mutated versions are based on the *Mutation Rules Set*, and are one by one, trained with respect to the same set of *Input Vectors Set*. In this procedure, all neurons of the network are trained, one by one as follows (consider initially the neuron input weights determined in step **(a)**):

   (i) Inject mutant 1 into neuron 1 and start the training procedure for the whole network. When the network is trained, take off mutant 1 from neuron 1, move it to neuron 2 and inject the same mutant 1 into it. Execute the network training procedure with this fault injected into neuron 2. Follow this procedure until all the network neurons have been trained for mutant 1.

   (ii) Choose another mutant (say mutant 2) from the *Mutation Rules Set* and repeat step **(i)**. Stay in this training loop for the network until all the mutants from the *Mutation Rules Set* have been applied to all the neurons of the network.

c) Just after leaving step **(b)**, there is a checkpoint used to verify the obtained reliability level for the network, with respect to the *Mutation Rules Set* considered. The reliability verification process is performed as follows: take the final set of neuron input weights determined for the network after training all the neurons for all the mutants and apply these weights into the original code. Then, take the same subset of mutants used to train the network in step **(b)**, and add new mutants to this subset from the *Mutation Rules Set*. Based on this larger set of mutants, generate mutated versions of the original code, simulate one by one, check the outputs, and based on an expected behavior, decide if the obtained reliability level was attained or not for the network. If yes, then the process is finished and the network is compiled for the target processor (or synthesized, if it is to be implemented in hardware). Otherwise, it is necessary to move backwards and train again the network with respect to all the mutants, as described in step **(b)** items **(i)** and **(ii)**. Note that in this case, the initial neuron input weights are not the initial values obtained from step **(a)**, but those obtained from the end of step **(b)**, item **(ii)**. This new training process is based on two additional actions:

   - Add complementary mutation rules to the "*Mutation Rules Set*" in order to increase the types of faults that will be injected into the ANN;
   - Add complementary vectors to the "*Input Vectors Set*" in order to increase the size of epochs used to train the ANN.

This design loop between steps **(c)** and **(b)** is repeated until a given runtime or an iterations number is not exceeded.

4. Experimental Results

This section presents a computation example that we have developed to illustrate the proposed approach. With this purpose, we implemented and trained an ANN to recognize electrocardiogram (ECG). By doing so, the primary goal of this section is to verify the ANN effectiveness to recognize two of the most important cardiac dysfunctions that can be detected by means of an EGC record: *Ventricular Flutter* (**VF**), and *Premature Ventricular Contraction* (**PVC**), even if the ANN is suffering from single faults.

ECG is a bio-signal, which is due to the electrical activity of the human heart that is transmitted to the body surface. It can be recorded using various systems. The simplest one is the orthogonal 3-lead system that records three subcomponent signals which are called lead X, Y, and Z, respectively [8]. Each ECG lead is composed of a number of cardiac cycles. The electrocardiography patterns that constitute a cardiac cycle and must be recognized are the **QRS** complex, the inter-wave segments **P** and **T**, and the cardiac intervals [8].

One of the ANN architectures we implemented in this work is a *Multi-Layer Perceptron* (MLP), with one hidden layer (fifteen neurons) and one output layer (one neuron). The algorithm used to train the ANN was the *Back-Propagation* with the *Hyperbolic Tangent Activation Function*. After specifying and training the ANN, but before applying the proposed technique, the ANN provided us with a recognition success of 92% at the output (left-hand column, fig. 3). For the fault tolerant (FT) version of this network, i.e., after training it according the proposed approach, it was obtained the score of 91% (right-hand column, fig. 3). Both recognition scores shown in fig. 3 were obtained for a fault-free operation, i.e., the network was not operating in the presence of faults when these scores were obtained. Therefore, the most important conclusion we can take from the data depicted in fig. 3 is that the proposed approach does result in a negligible degradation of the network recognition capability when compared to the conventional technique used for training the network.

In order to obtain the score of 92%, we have trained the ANN for 800 epochs, where one epoch is equal to 1218 input vectors, as seen in fig. 2 .

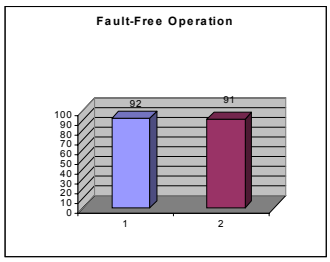


Fig. 3. Recognition success for the 16-neuron network *before* (92%) and *after* (91%) using the proposed approach. Results for a fault-free operation.

It was used 5 mutants from the *Partial Mutation Rules Set*: **AOR**, **VCR**, **VR**, **ABS**, **ROR**, (fig. 1, Mutants *1* to *5*, respectively) to train the ANN in the presence of single faults. During this procedure, the mutants were injected into each of the 16 neurons of the ANN, one mutant at a time. For each mutant injected, we executed one epoch, i.e., 1218 input vectors were applied to the network during the training process. Then, the computational complexity associated with the whole training process in terms of the total number of *input vectors* applied to the network can be computed as follows: *5 mutants per mutant* **x** *16 neurons* **x** *1218 vectors per mutant/neuron* = 97.440.

The Figure 4 illustrates results for two mutants injected into the ANN code, and the obtained reliability (in terms of recognition scores) for two different situations: *before* and *after* it was trained according to the proposed approach. In figs. 4(A1) and 4(B1), there are two results (columns) for each of the sixteen neurons: the *left-hand column* indicates the network recognition success for a fault injected in the respective neuron *before* training the ANN with the proposed approach. The *right-hand column* indicates the same situation, but *after* training the ANN with the proposed approach. Note that in all the 16 cases of these figures (except for neuron 4 in fig. 4(B1)) the ANN presents better recognition scores when it was trained with the proposed approach.

The two mutants used in fig. 4 are: **ABS** and "**Output Null**" were used as ***closed test*** and ***open test*** respectively to verify the reliability. The "**Output Null**" mutant represents the *disconnection of the neuron from the network*.

Figs. 4(A2) and 4(B2) summarize the *network average recognition success* for each one of the four faults injected into the network. For instance, for the ABS mutant, the score of 66% represents the averaged value of the 16 scores obtained when the ABS mutant was injected into the 16 network neurons of the original version (**Not FT**). This result was obtained *before* training the network with the proposed approach. For the **FT** version of this network, i.e., *after* training it with the proposed approach, similar reasoning can be taken for the score of 83% as indicated by the right-hand column in fig. 4(A2).

Additionally, the reader should also note that the left-hand column in fig. 4(B1) indicates the respective neuron contribution importance for the ANN operation. In other words, when it is disconnected from the network as the consequence of the fault "**Output Null**", it cannot contribute for the recognition process. Following this thought, neurons 3, 5, 8, 10 and 11 are the most important for the ANN operation.

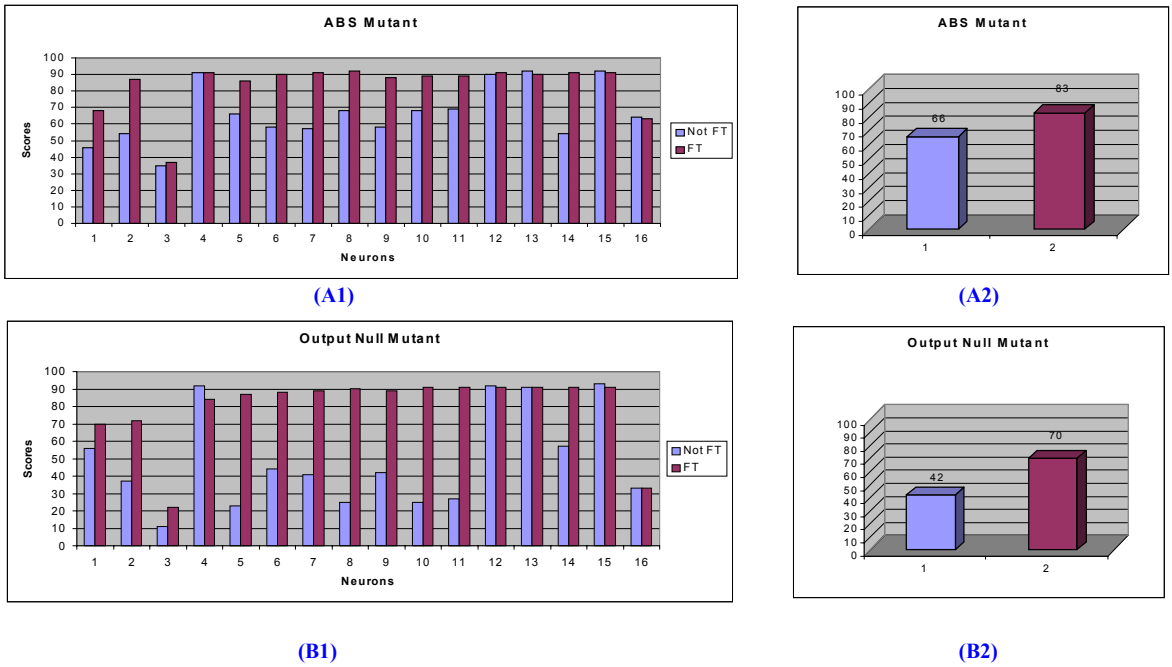


Fig. 4. Recognition success for the network *before* and *after* using the proposed approach. Results for single fault injection in the 16-neuron network, one at a time. **(A)**: *closed test*; **(B)**: *open test*.

5. Final Considerations & Future Work

We presented a new approach to design fault tolerant artificial neural networks (ANNs). Additionally, this approach allows estimating the final network reliability. This approach is based on the Mutation Analysis technique, and it is used during the training process of the ANN.

The basic idea is to train the ANN in the presence of faults. In this case, the single fault model is assumed. To do so, a set of faults (one at a time) is injected into the code describing the ANN. This procedure yields mutation versions of the original ANN code, which in turn are used to train the network in an iterative process with the designer until the moment when the ANN is no more sensible to the single faults injected. In other words, the network became tolerant to the considered set of faults.

A practical example where an ANN is used to recognize electrocardiogram (ECG) was presented to illustrate the proposed

methodology. The architecture used to implement the case study was a *Multi-Layer Perceptron* (MLP), in two different topologies:

a) one hidden layer (fifteen neurons) and one output layer (one neuron);

b) one hidden layer (twenty-five neurons) and one output layer (one neuron).

In both cases, the algorithm used to train the ANN was the *Back-Propagation* with the *Hyperbolic Tangent Activation Function*. The results obtained indicate that the proposed approach is a powerful tool to the ANN with respect to a given set of faults. If the ANN is implemented in software, these faults may be the result of design specification mistakes, or compilation errors for instance. Alternatively, if the ANN is to be implemented in hardware, these faults can be mapped as single stuck-at faults at the gate level.

For the continuation of this work, other network architectures will be considered. For instance, instead of using *feed-forward* ANNs, we will consider *recurrent/feedback* ones. Additionally, we will implement two equivalent ANN versions: one in software to run in a DSP processor, and one in hardware, by mapping a programmable logic component (FPGA). The primary goal of this task is to perform electromagnetic compatibility tests with these software/hardware components in order to more accurately evaluate the benefits provided by the proposed approach for the network when it is operating in real noisy environments such as those affected by strong electromagnetic interference.

6. References

[1] Demidenko, S.; Piuri, V. *On-line Testing In Digital Neural Networks*. IEEE Proceedings of the Asian Test Symposium (ATS), 1996.

[2] Heemkerk, J. N. H. *Overview of Neural Hardware*. Ph.D. Thesis, Unit of Experimental and Theoretical Psychology - Leiden University, Chapter 3, 1995.

[3] *Philips L-Nero Chips 1.0 and 2.3*. Laboratories d'Electronic Philips (LEP). 22, Av. Descates. BP 15, 94453 Himeil-Brevannes Cedex, France.

[4] Pradhan, D. K. *Fault-Tolerant Computer System Design*. Prentice-Hall, 1996. 544p.

[5] Crouch, A. L. *Design for Test for Digital IC's and Embedded Core Systems*. Prentice Hall PTR, Upper Saddle River, NJ 07458, 1999. 349p.

[6] Weiss, S. N.; Fleyshgakker, V. N. *Improved Serial Algorithms for Mutation Analysis*. International Symposium on Software Testing and Analysis - ACM-ISSTA, Cambridge - MA, June 1996, pp. 149-158.

[7] Offutt, A. J. *Investigations of the Software Testing Coupling Effect*. ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 1, January 1992, pp.5-20.

[8] Trahanias, P.; Skordalakis, E. *Syntactic Pattern Recognition of the ECG*. IEEE Transactions on Pattern Analysis and Machine, Vol. 12 No. 7, Jul. 1990, pp.648-657.