

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE  
UNIVERSITÉ DU QUÉBEC

THESIS PRESENTED TO  
ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR  
A MASTER'S DEGREE IN ELECTRICAL ENGINEERING  
M. Eng.

BY  
Zeynab MIRZADEH

MODELING THE FAULTY BEHAVIOUR OF DIGITAL DESIGNS USING A FEED  
FORWARD NEURAL NETWORK BASED APPROACH

MONTREAL, 19<sup>TH</sup> SEPTEMBRE, 2014

© Copyright 2014 reserved by Zeynab Mirzadeh

© Copyright reserved

It is forbidden to reproduce, save or share the content of this document either in whole or in parts. The reader who wishes to print or save this document on any media must first get the permission of the author.

BOARD OF EXAMINERS  
THIS THESIS HAS BEEN EVALUATED  
BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Jean-Francois Boland, Thesis Supervisor  
Département de génie électrique à l'École de technologie supérieure

Mr. Yvon Savaria, Thesis Co-supervisor  
Département de génie électrique à l'École polytechnique de Montréal

Mr. Christian Belleau, President of the Board of Examiners  
Département de génie mécanique à l'École de technologie supérieure

Mr. Claude Thibeault, Member of the jury  
Département de génie électrique à l'École de technologie supérieure

THIS THESIS WAS PRESENTED AND DEFENDED  
IN THE PRESENCE OF A BOARD OF EXAMINERS AND PUBLIC  
29 AUGUST 2014  
AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE



## **FOREWORD**

The Consortium for Research and Innovation in Aerospace in Québec (CRIAQ) is a non-profit consortium founded in 2002. It was established with the financial support of industry and university toward carrying out collaborative industry research projects at university. CRIAQ's objectives are to improve knowledge of aerospace industry through education by training students in university. It improves competitiveness by enhancing knowledge on aerospace technology. ÉTS is involved in a number of CRIAQ projects and that project was carried as part of the CRIAQ AVIO403 project. The title of the AVIO403 project is Cosmic Radiation & Effect on Aircraft Systems. Bombardier Aerospace is the lead company involved in the project. Others partners in the project are MDA Corporation and the Canadian Space Agency. In addition, in this project, students and professors from the University of Montreal, École Polytechnique, Concordia, and the École de technologie supérieure are involved. The main objectives of the project are the development and validation of techniques and methodologies for verifying, testing and designing reliable systems subject to classes of radiations induced by cosmic rays.



## ACKNOWLEDGMENTS

There are no proper words to convey my deep gratitude and respect for my thesis and research advisor, Professor Jean-Francois Boland. He taught me the skills to successfully formulate and approach a research problem. I would also like to thank him for being an open person to ideas, and for encouraging and helping me to shape my interest and ideas.

I would like to express my deep gratitude and respect to Professor Yvon Savaria, my co-supervisor whose advices and insight was invaluable to me. He generously provides me constructive criticism, which helped me develop a broader perspective to my thesis.

I would like to thank Marc-Andre Leonard, my teammate in AVIO403 project, who was involved in this project from the start and was a great support for everything. Thanks to him for questioning me about my ideas, helping me think rationally and even for hearing my problems.

I am most grateful to all member of AVIO403 project, especially Remi Robache who was involved in this project as a great help in running the experiments. He was highly responsible and cooperative with solving the problems about running the studies.

I would like to thank my family, especially my mother and father for always believing in me, for their continuous love and their supports in my decisions. Without them, I could not have made it here.

Thanks to my supportive friends in LACIME laboratory, Parisa, Sara, Hossein, Aria, Hassan, Fanny, Mathieu, Normand, Smarjeet and others who made the lab a friendly environment for working.



# **MODÉLISATION DU COMPORTEMENT FAUTIF DE CIRCUITS NUMÉRIQUES BASÉE SUR L'APPROCHE PAR RÉSEAUX DE NEURONES DE TYPE CONNECTION-DIRECTE**

Zeynab MIRZADEH

## **RÉSUMÉ**

Les rayons cosmiques sont une source d'erreurs douces pour les circuits numériques. Dans le domaine aéronautique, aux altitudes des avions, le flux de neutrons provenant des radiations cosmiques est plus élevé qu'au niveau de la mer. Il est donc utile d'étudier le comportement fautif d'un circuit avant son implémentation afin d'analyser sa robustesse en présence de pannes. Le but de cette recherche consiste à développer une approche pour la modélisation du comportement fautif des circuits numériques. Cette approche peut être appliquée tôt dans le flot de conception avant l'étape de fabrication. Pour ce faire, nous devons tout d'abord extraire le comportement fautif du circuit à partir d'un modèle décrit à bas niveau en langage HDL. Ensuite, cette information est utilisée pour la phase d'apprentissage d'un réseau de neurones décrit en langage C/C++ ou MATLAB<sup>TM</sup>. Le modèle résultant servira à reproduire le comportement fautif du circuit en présence de pannes. Cette approche est propice à un développement d'une bibliothèque basée sur des modèles fautifs représentés par un réseau de neurones. Cette bibliothèque serait utilisée au niveau Matlab/Simulink et pourrait regrouper plusieurs classes de circuits. L'analyse de fiabilité devient donc possible à haut niveau et l'étude complète d'un design comprenant des sous-modules préalablement caractérisés est à la portée du concepteur.

La méthodologie développée dans le cadre de ce mémoire est basée sur des expérimentations effectuées sur deux circuits. Le premier est le ISCAS C17 avec lequel le comportement fautif a été généralisé par un réseau de neurones. Afin de valider notre méthodologie, les résultats ont été comparés avec une méthode préalablement développée basée sur génération de signatures. Par la suite, un multiplieur 4-bit est utilisé comme deuxième circuit plus élaboré. Les résultats de la modélisation du comportement fautif par réseaux de neurones montrent que la précision est augmentée comparativement à la méthode de génération de signatures. Pour le circuit C17, même en ne prenant que 30% des données générées par le simulateur de pannes LIFTING, le réseau de neurones est capable de reproduire le comportement fautif du circuit tout en préservant une erreur de modélisation sous les 6%.

**Mots clés:** SEU, réseau de neurones, comportement fautif, circuit numérique, C++, MATLAB<sup>TM</sup>.



# **MODELING THE FAULTY BEHAVIOUR OF DIGITAL DESIGNS USING A FEED FORWARD NEURAL NETWORK BASED APPROACH**

Zeynab MIRZADEH

## **ABSTRACT**

Cosmic rays lead to soft errors in electronic circuits. In avionic systems it is more critical, as the neutron flux that is caused by cosmic rays is stronger at high altitudes. It would be helpful to study the faulty behaviour of digital circuits before implementation to analyze their robustness in presence of faults. The goal of this research is to develop an approach for modeling the faulty behaviour of digital circuits. This proposed approach could be applied in a design flow before circuit fabrication to characterize the faulty behaviour of circuits for their early validation. This is achieved by extracting information about faulty behaviour of circuits from low-level models expressed in VHDL language. Afterwards the extracted information is used to train high-level artificial neural networks models expressed in C/C++ or MATLAB<sup>TM</sup> languages. The trained neural network becomes a model able to replicate the faulty behaviour of the circuit in presence of faults. Later, trained artificial neural network models could be used to develop a components characterization library available in Matlab/Simulink regrouping different classes of circuits. These pre-defined faulty component models could also be used in high-level models to conduct reliability analysis. Thus, the faulty behaviour of each sub-circuit and their effects on a system could be assessed.

The methodology adopted in this thesis is based on experiments done with two important benchmarks. First, the faulty behaviour of the C17 ISCAS circuit is modeled using a neural network approach. To validate our method, the results are compared with a previously reported faulty signature generation method. Then, our proposed technique is tested with a 4 bit multiplier design, which has a larger dataset. Results show that the neural network approach leads to models that are more accurate than the signature generation method. For the circuit C17, by taking only 30% of the dataset generated with the LIFTING fault simulator, the neural network is able to replicate the output of the circuit in presence of faults while keeping the mean absolute modeling error below 6%.

**Keywords:** SEU, Neural network, Faulty behaviour, Digital circuit, MATLAB<sup>TM</sup>, C++.



## TABLE OF CONTENTS

	Page
INTRODUCTION .....	1
CHAPITRE 1 LITERATURE REVIEW .....	5
1.1 Cosmic rays in the earth atmosphere .....	5
1.1.1 Effect of energetic neutrons on integrated circuits .....	7
1.1.2 Effectiveness of a shield .....	9
1.2 Faults caused by cosmic rays in digital systems.....	10
1.2.1 Classification of faults caused by cosmic rays .....	10
1.2.2 Faults, errors and failures.....	12
1.3 Digital circuit verification by fault injection .....	13
1.3.1 Fault models.....	14
1.3.1.1 Stuck-at (s-a) fault model.....	14
1.3.1.2 Bit flip fault model.....	15
1.3.2 Fault injection techniques .....	15
1.3.2.1 Hardware-based injection techniques .....	16
1.3.2.2 Emulation-based injection techniques .....	16
1.3.2.3 Simulation-based injection techniques .....	17
1.4 Developing fault behavioural model .....	18
1.5 Neural network and fault .....	19
1.6 Conclusion .....	21
CHAPITRE 2 NEURAL NETWORKS AND THEIR APPLICATIONS IN FAULT MODELS .....	23
2.1 A brief introduction to artificial neural network .....	23
2.2 The neuron.....	25
2.3 Neural network architectures.....	28
2.4 Multi-layer perceptron (MLP) networks .....	29
2.4.1 Preliminaries definitions for MLP learning algorithm explanation.....	31
2.4.1 MLP learning algorithm description.....	32
2.4.2 Neuronal activation functions .....	35
2.5 Multi-layer perceptron in practice .....	38
2.5.1 Data preparation.....	38
2.5.2 Size of training data .....	38
2.5.3 Number of hidden layers.....	39
2.5.4 Generalization and over-fitting.....	39
2.5.5 Training, testing, and validation .....	40
2.5.6 When to stop learning .....	41
2.5.7 Computing and evaluating the results.....	42
2.6 Conclusion.....	42

CHAPITRE 3	PROPOSED METHODOLOGY .....	43
3.1	Neural network approach and different scenarios .....	43
3.1.1	Concept of signature .....	44
3.1.2	Scenario 1: signature adjustment .....	46
3.1.3	Scenario 2: faulty output prediction.....	47
3.2	Proposal to use neural network for predicting faulty output of circuits .....	48
3.2.1	Dataset generation.....	51
3.2.2	Neural network training phase .....	52
3.2.3	Neural network validation.....	53
3.2.4	Using the trained neural network .....	54
3.3	Conclusion .....	55
CHAPITRE 4	IMPLEMENTATION OF PROPOSED MODEL .....	57
4.1	Dataset generation .....	57
4.2	MLP Neural network in MATLAB™ environment.....	58
4.3	MLP Neural network and OpenCV library written in C++ language .....	60
4.3.1	Parameters configuration .....	61
4.3.2	Layers definition .....	62
4.3.3	Network training .....	63
4.4	Case study for circuit C17 .....	63
4.5	Data sampling method for training neural network.....	65
4.5.1	Method of computation of errors .....	72
4.6	Neural network model for C17 in C++ environment .....	77
4.7	Results for a Multiplier circuit.....	78
4.8	Conclusion .....	82
CONCLUSION AND FUTURE WORK .....		83
APPENDIX I	C++ CODE FOR CIRCUIT C17 USING OPENCV .....	85
LIST OF BIBLIOGRAPHICAL.....		91

## LIST OF TABLES

	Page
Table 3.1	Computing error type for a circuit with following golden and faulty outputs .....45
Table 3.2	Computing signatures for the arbitrary circuit presented in table 3.1 .....45
Table 4.1	Fault truth table for circuit C17. First 17 Injected fault fields show stuck-at-0 for all 17 different fault sites and second 17 injected fault fields show stuck-at-1 for all .....64
Table 4.2	Errors of evaluation of the neural network model for C17 while using different percentages of sampling of the 1088 number of data .....67
Table 4.3	Errors of evaluation of the neural network based model for circuit C17 with sample sizes less than 10% of the 1088 number of data .....69
Table 4.4	Comparison between the errors of the neural network based model and model proposed by (ROBACHE 2013) for C17 circuit .....70
Table 4.5	Error types for circuit C17 .....73
Table 4.6	Matrix of <i>PR</i> for circuit C17 .....76
Table 4.7	Matrix of <i>PF</i> for circuit C17 while percentage of sampling is 70% .....76
Table 4.8	Comparison of speed between MATLABTM and C++ environment .....78
Table 4.9	Errors of evaluation of the neural network model for circuit 4 bit multiplier while using different percentages of sampling of the 82688 number of data .....79



## LIST OF FIGURES

		Page
Figure 1.1	Neutron flux at different altitude.....	6
Figure 1.2	Neutron flux in different latitude and altitude.....	7
Figure 1.3	The high-energy neutron strikes the silicon atoms of a CMOS NPN transistor and a trail of electron-hole pairs are created as a result of this interaction.....	8
Figure 1.4	Mitigation of the neutron flux as a function of the thickness of concrete in cm.....	9
Figure 1.5	Fault classification based on cause and effect.....	11
Figure 1.6	a) Fault which is masked in inner scope b) Fault which manifests in outer scope will be called error.....	12
Figure 1.7	The fault s-a-0 is activated on line G when logic value 0 is applied on line G Adapted from Patel (2005).....	14
Figure 1.8	The bit flip fault injected into memory.....	15
Figure 1.9	Classification of fault injection techniques.....	18
Figure 2.1	A small example of an MLP neural network with one hidden layer.....	25
Figure 2.2	A simple element which is called artificial neuron or perceptron. $X_1$ , $X_2$ , $X_3$ and $Y$ are neurons or perceptrons; The weights on the links to neuron $Y$ from neurons $X_1$ , $X_2$ and $X_3$ are $w_1$ , $w_2$ and $w_3$ .....	26
Figure 2.3	The sigmoid activation function.....	27
Figure 2.4	The different architectures for neural network: a) single layer feed forward, b) multi-layer perceptron feed forward, c) recurrent network.....	28
Figure 2.5	The multilayer perceptron neural network structure.....	30
Figure 2.6	Steps of Multi-Layer Perceptron training algorithm.....	33
Figure 2.7	Commonly used activation functions for a neuron, where O/P denotes the output of the neuron and I/P denotes the input.....	37
Figure 2.8	There are two data sets. White ones are training data and red ones are evaluation data. The lines are the estimation of training data set which	

	comes from the functions which are trained by the neural network. The left neural network has the ability of generalization but the right neural network exhibits over fitting. ....	40
Figure 2.9	Mean-square error during training .....	41
Figure 3.1	Proposed sketch for scenario 1 to adjust signatures .....	47
Figure 3.2	Diagram of the proposed approach to mimic the behaviour of a combinational circuit in the presence of faults.....	48
Figure 3.3	The flow diagram to develop a Neural Network model which is able to predict faulty output of circuit.....	50
Figure 3.4	The flow diagram to show the steps to generate the training dataset and .....	51
Figure 3.5	Flow diagram to show the training phase of the neural network .....	52
Figure 3.6	Flow diagram for validation phase of neural network .....	54
Figure 3.7	Flow diagram for using phase of neural network.....	55
Figure 4.1	Fault sites specified for circuit C17.....	58
Figure 4.2	Neural network based model for circuit C17 .....	65
Figure 4.3	Graphs of a) Error_category1, b) Error_category2, c) Error_category3 and d) mean_correlation coefficient.....	72
Figure 4.4	Graphs of a) Error_category1, b) Error_category2, c) Error_category3 and d) mean_correlation coefficient.....	82

## LIST OF ABBREVIATIONS

CMOS	Complementary Metal-Oxide-Semiconductor
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
LUT	Look Up Table
MTBF	Mean Time Between Failures
SEU	Single Event Upset
VHDL	VHSIC Hardware Description Language
SEE	Single Event Effect
SEL	Single Event Latch up
SEFI	Single Event Functional Interrupt
SET	Single Event Transient
RHA	Radiation Hardness Assurance
RBF	Radial Basis Function
S-a	Stuck-at
IC	Integrated Circuit
ANN	Artificial Neural Network
MLP	Multi-Layer Perceptron



## INTRODUCTION

FPGA-based circuits are very well-known in all fields of industry, including space, avionic and ground-level electrical instruments. For critical purposes, the reliability of such circuits is a primary concern. Therefore, the phenomena causing faults in circuits should be studied to ensure that measures are implemented to protect the system against these faults. One such class of phenomena is the single-event-upsets in flip-flops or memory cells. Neutrons are generated when cosmic particles strike nitrogen or oxygen atoms while travelling extremely fast in the upper layers of the atmosphere. When these neutrons hit silicon atom nucleus, they can generate secondary particles. Some of these particles are charged and they can generate trails of electron-hole pairs of a few microns in length. If one such trail is near a reversed-biased pn junction in a transistor, a voltage spike can be generated. Such a voltage spike can change the state of a memory cell or flip-flop. This state change is called a single-event-upset or soft error (Actel 2002). Before, this error was mainly considered in the space industry, but with the decreasing size of transistors, even ground-level instruments are susceptible to such problems. Though more robust FPGAs may be used (Brogliey 2009), their price compared to traditional FPGAs makes them unaffordable in many applications. An optimized solution to this problem is to use traditional FPGAs with a new strategy for circuit design to improve their robustness.

Cosmic rays and their effects are known to upset electrical instruments in avionic circuits as well as those at the ground level notably because of the decreasing size of transistor (Ostler, Caffrey et al. 2009). They change the state of flip-flops in integrated circuits leading to soft errors. Consequently, circuits behave differently in the presence of such faults. For verifying the reliability of circuits before production, it would be helpful to know about the faulty behaviour of circuits.

To augment the sensitivity of electrical devices in avionic circuits, improvement of their robustness becomes necessary. In addition, radiation tolerant integrated circuits are often unaffordable and therefore robustness of traditional integrated circuits has to be increased by

implementing necessary mitigation strategies. This would enable their use in lieu of more expensive alternatives (Laura Dominik 2008).

Studying current methodologies to design circuits should be done so that old strategies could be adapted to formulate new ones that answer the requirements mentioned above. Thus, reliability requirements to protect avionic circuits could be specified during the design steps in order to obtain reliable inexpensive integrated circuits.

The main objective of this project is to develop a faulty behaviour model for FPGA-based circuits described at a high-level of abstraction. By using neural networks, fault behaviour models are developed and their accuracy is validated. The developed models could be used to replace any component of the entire circuit with faulty versions of the components described at a high-level of abstraction. This ensures that the effect of faulty behaviour of each component on a system could be analyzed at a high-level of abstraction and the mitigation technique could be used to improve the robustness of more critical parts. The steps proposed in this project are part of the CRIAQ AVIO403 project. Sub-objectives of the present master project are explained below:

- The first sub-objective is to emulate faults in models described at a high-level of abstraction. Therefore, a suitable fault injection method is proposed;
- The second sub-objective is to explore if the type of faulty behaviour modeling that is the goal of the project could be done with neural networks. There are several possible tasks: for example, a neural network could be developed to produce the faulty output of a circuit, or to predict the probability of occurrence of faults for each bit of the circuit output;
- The third sub-objective is to develop the faulty behaviour model of different circuit components, such as multiplier and adder, using the neural network structure proposed in the previous sub-objective;
- The last sub-objective is to develop a library of faulty behaviour blocks of the circuit components composing a Simulink model. Therefore, each time designers need to

analyse possible faulty behaviour of a circuit at a high-level of abstraction, the library of components could be used.

### **Contributions**

This research proposes a fault behaviour model developed with a neural network concept in a novel way. The neural network structure is used to synthesize the faulty output of a circuit at a high-level of abstraction. All the strategies that are proposed in this research have novelty; and effort is exercised to find an appropriate structure for the neural network in this project.

Existing fault behaviour models for emulating faulty behaviour of circuits are not accurate enough and neural networks are proposed to enhance these models. This ensures a greater prediction accuracy while emulating faulty behaviour of avionic components.

### **Outline of the thesis**

In the first chapter, the literature review is presented, and also the essential knowledge for understanding this thesis is summarized. Details about the nature of cosmic rays and their effects on avionic circuits are provided. Different effects cause different errors and faults in the circuit, and this is described and classified. Different tools for fault injection at different levels of abstraction are also verified.

In the second chapter, the concept of artificial neural network is explained. Different structures of neural networks are classified and training algorithms are explained. The training algorithm selected for the work presented here is explained in greater detail, including the reason behind its selection.

In the third chapter, different paths followed to develop a possible solution for the stated problem are described. This chapter also includes a detailed description of the scenario implemented here. Note that there are several scenarios to solve this problem. Signature generation and signature improvement are considered and explained in further detail.

In chapter four, all the steps for signature generation scenario are analysed in detail. We introduce new concepts for modeling fault behaviour using neural network based signatures. These concepts are analysed through several case studies.

The fifth chapter compares the results of this study with the results obtained by colleagues, whose work strategies and related limitations are specified. This comparison highlights the advantages and disadvantages of the approaches explored by the research team.

The conclusion chapter presents the results of the methodology adopted in this thesis, and also offers suggestions for possible opportunities of future work in this area of research.

# CHAPITRE 1

## LITERATURE REVIEW

In this chapter, cosmic rays and their effect on electronic circuits are explained. Cosmic rays are first described and then their effects on integrated circuits are studied. The classification of failures caused by cosmic rays is explained, and the difference between a fault and an error is defined. At the end, neural network is introduced in terms of its utility as a tool for detection and classification of faults in digital and analog circuits.

### 1.1 Cosmic rays in the earth atmosphere

Galactic cosmic rays and solar rays trace their origins to deep space and the sun. As they approach the earth, they collide with gas atoms in the upper atmosphere, like those of oxygen and nitrogen. As a result of these collisions, oxygen and nitrogen atoms disintegrate into a variety of high energy particles. Most of these particles recombine quickly. Neutrons constitute the main proportion of the product of these atmospheric collisions. These neutrons tend not to recombine and are projected from the collisions at very high rates. They travel at a high speed until a second collision occurs; such a second collision could be with oxygen or nitrogen atoms in atmosphere, objects traveling in atmosphere or objects on the earth's surface (Velazco, Fouillat et al. 2007).

The amount of neutrons, which is called neutron flux, depends on the following factors (Actel 2002):

**Altitude:** At low altitudes, neutron flux decreases. Altitude is a significant factor affecting the neutron flux. In figure 1.1, neutron flux at different altitude is illustrated;

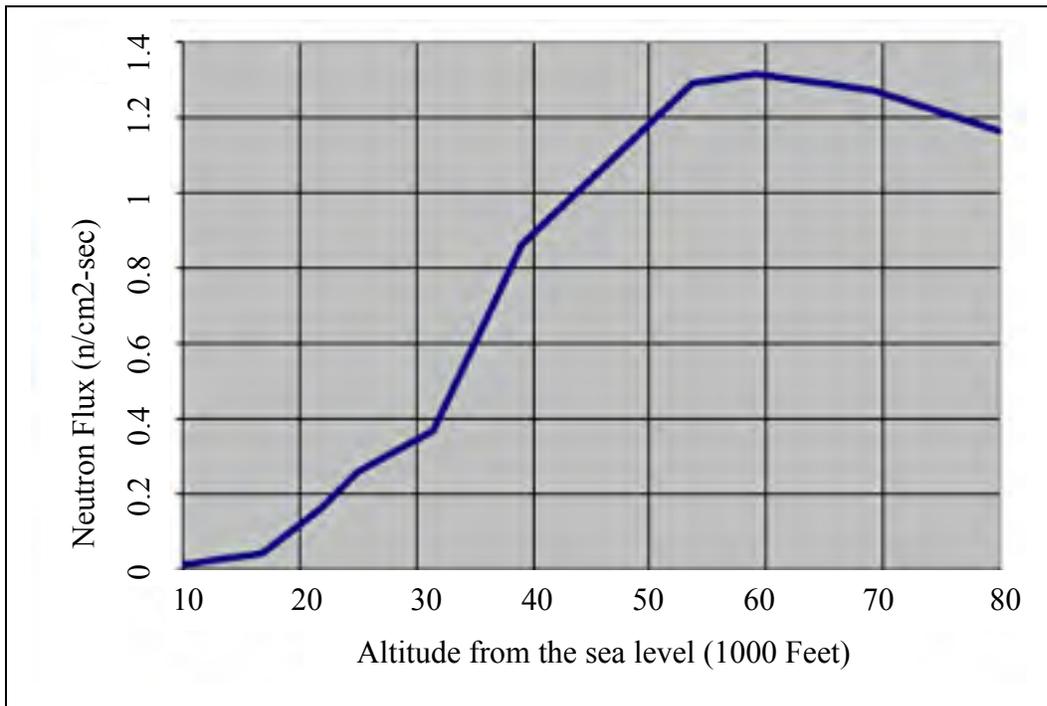


Figure 1.1 Neutron flux at different altitude  
Adapted from Actel (2002)

**Latitude:** The earth's magnetic field lines are closer together at the poles and hence facilitate a greater trapping of cosmic particles than at the equatorial latitude. In figure 1.2, neutron flux variation with latitude and altitude is illustrated;

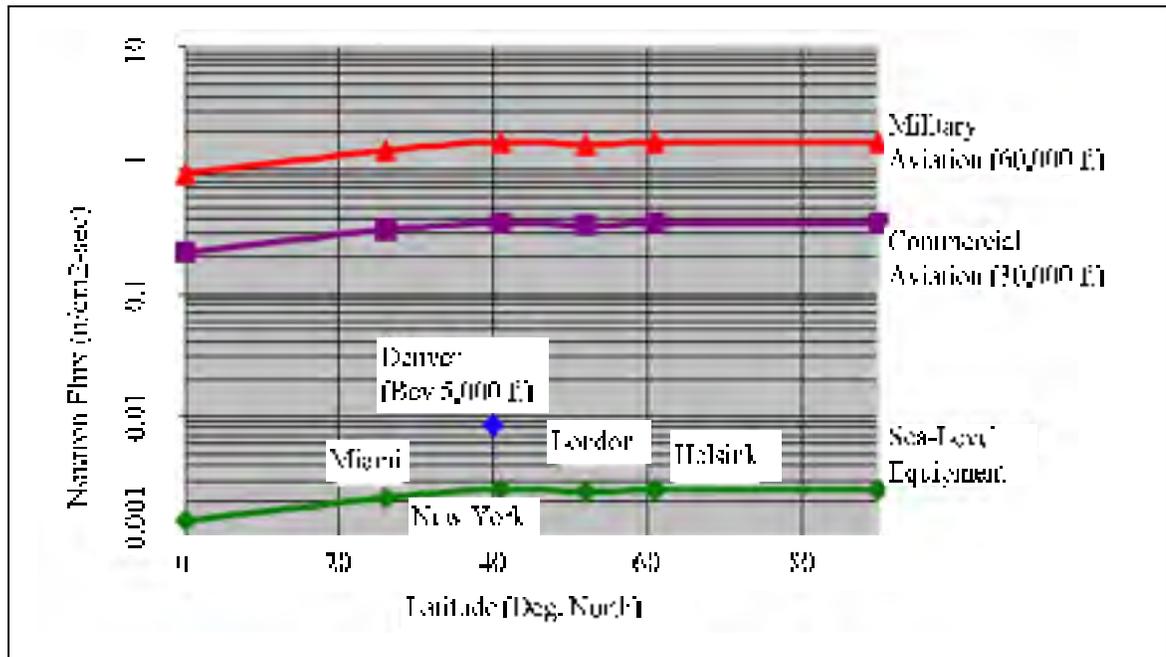


Figure 1.2 Neutron flux in different latitude and altitude  
Adapted from Actel (2002)

**Longitude:** It has some effects on the amount of cosmic rays but these effects are negligible compared to the previously mentioned factors.

As explained, altitude is a significant factor affecting the neutron flux. At aircraft altitude, the amount of cosmic rays is more than ground level. With respect to latitude, the amount of cosmic rays is significantly more at the polar latitude than at the equator. Longitude plays an insignificant role, and its effect is negligible compared to the other factors affecting neutron flux.

In the next section, the effect of cosmic rays on electronic circuits is explained. The importance of these studies for avionic circuits is more considering that the effect of cosmic rays at airplane flying altitudes is significantly harsher than at the sea level.

### 1.1.1 Effect of energetic neutrons on integrated circuits

Nowadays, the effect of cosmic rays on electronic systems is well known. Neutrons are the main cause for such effects (Normand 1996). Neutrons produced by collisions travel at

extremely high speed and can strike integrated circuits contained in avionic systems. Most of them will pass through integrated circuits without interacting with them. However, there are some neutrons that will pass close enough to a dopant or silicon atom, resulting in an impact. As a result of this disturbance, secondary particles are produced, creating a trail of electron-hole pairs. In figure 1.3, we see a transverse cut of a CMOS NPN transistor. When a high energy neutron strikes the silicon atoms, a trail of electron-hole pairs are created as a result of this interaction (Actel 2002).

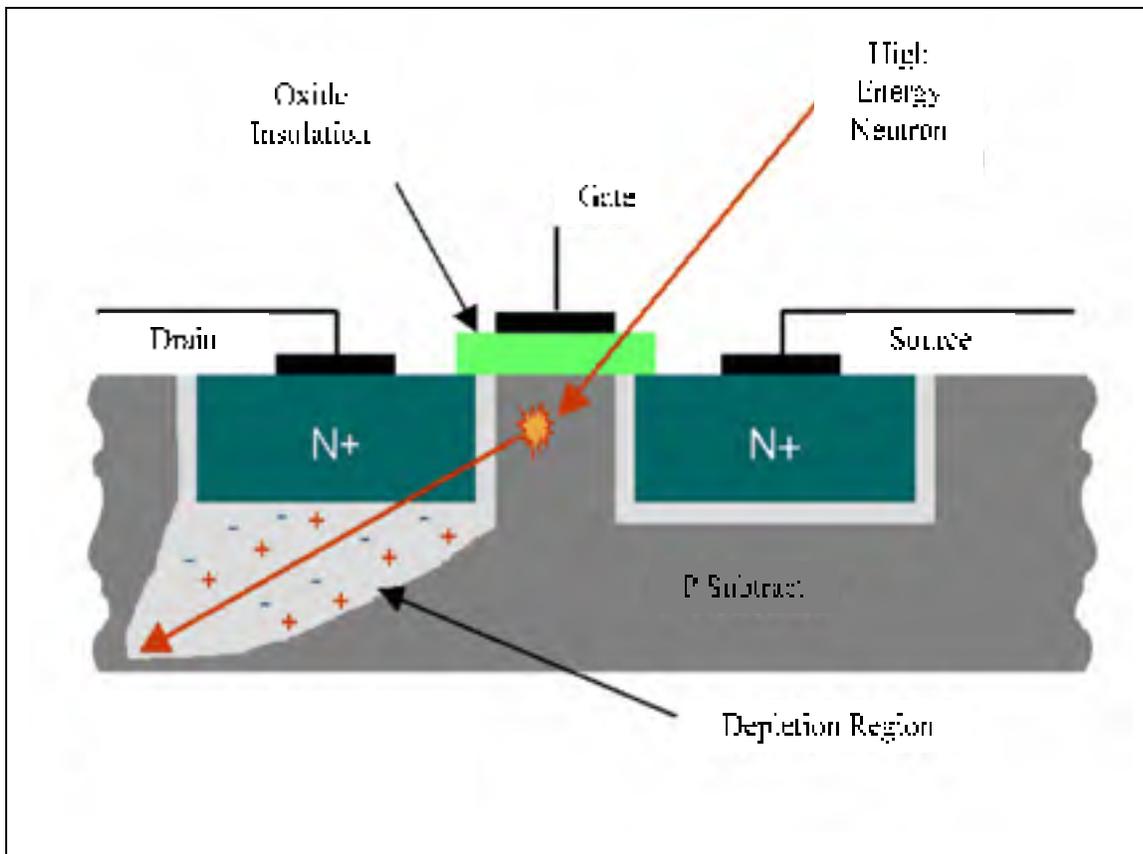


Figure 1.3 The high-energy neutron strikes the silicon atoms of a CMOS NPN transistor and a trail of electron-hole pairs are created as a result of this interaction  
Adapted from Actel (2002)

If the interaction happens near a reversed biased junction in a flip-flop or memory cell, a voltage spike could result. This effect could change the state of a memory element from zero to one or vice versa. This change is called single event upset (SEU). When only the state is

changed without any damage to the circuit, it is called a soft error. In the event of soft errors, the correct device operation may be restored by rewriting the memory element with the correct value.

The way in which circuits could be protected against high energy neutrons is by using shields, and is explained in the following section.

### 1.1.2 Effectiveness of a shield

It is very difficult to achieve an anti-neutron shield. In fact, even a thick concrete wall is not effective in reducing a neutron flux. The neutrons are slowed down and then absorbed by light nuclei. A material that contains hydrogen - such as water, polyethylene, paraffin, or concrete - is the best solution to mitigate such radiation. To do this effectively required a substantial thickness of material when neutron flux is high.

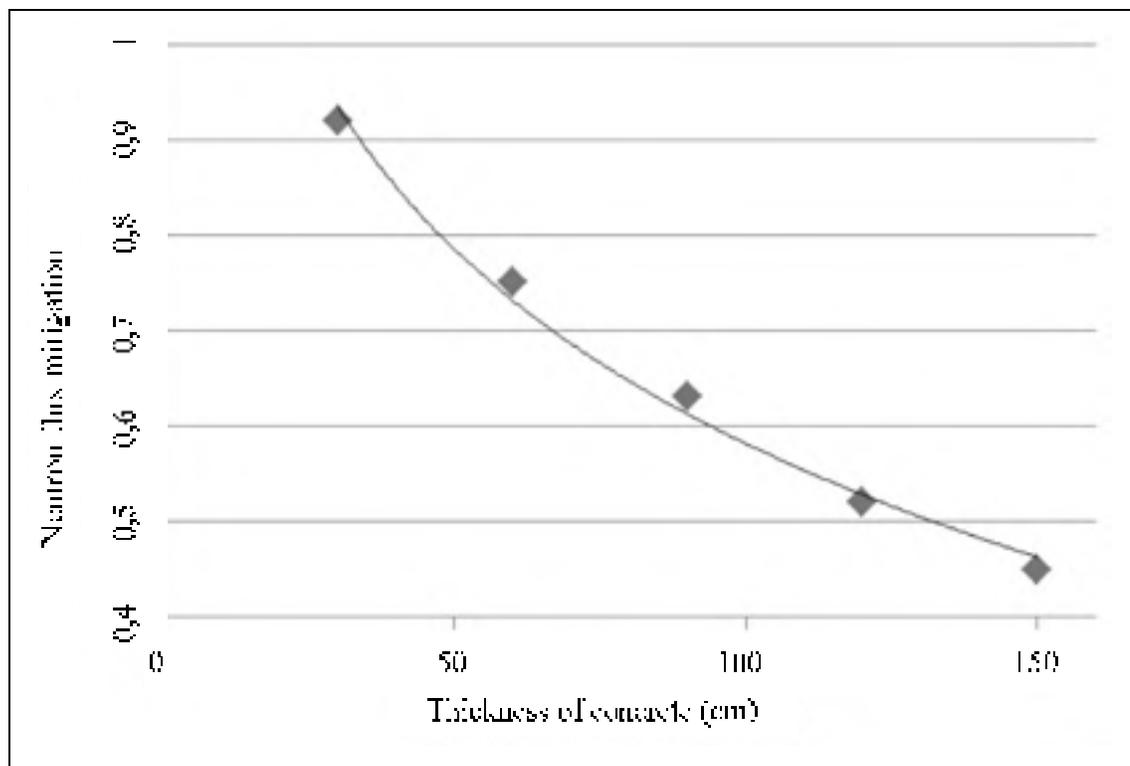


Figure 1.4 Mitigation of the neutron flux as a function of the thickness of concrete in cm  
Adapted from Dirk, Nelson et al. (2003)

Figure 1.4 represents the mitigation of the neutron flux as a function of the thickness of concrete as a shield. To reduce the flux to half, concrete with 1.3 m thickness should be used but providing such a thickness is very difficult in case of a lot of applications (Dirk, Nelson et al. 2003).

## **1.2 Faults caused by cosmic rays in digital systems**

One way to measure the effect of cosmic rays on a digital system is to measure the soft error rate (SER). SER is the rate at which a system expected to encounter soft errors (Vibishna, Beenamol et al. 2013). In the study by Taber and Normand in 1993, the soft error rate (SER) in a large amount of SRAM CMOS systems was  $1.2 * 10^{-7}$  soft fails per bit per day at 9 km altitude (Taber and Normand 1993). In the Rosetta project (Lesea, Drimer et al. 2005), implemented by Xilinx, the effect of cosmic ray on FPGAs in different positions from the earth is illustrated. For this project, 100 groups of different FPGA technologies were placed at different altitudes from the earth surface. They were tested to collect data to measure the effect of cosmic rays. It was found that at higher altitude, the effect of cosmic rays is more and therefore FPGA based avionic circuits will be more vulnerable to the presence of cosmic rays. In the following section, different faults caused by cosmic rays are classified.

### **1.2.1 Classification of faults caused by cosmic rays**

Faults are grouped into two main categories: non-recoverable and recoverable. Non-recoverable faults are the ones caused by cosmic rays with a destructive effect, whereas recoverable faults are those caused by cosmic rays without a destructive effect. For example, a faulty system can sometimes be fixed by reconfiguring the entire system, or reconfiguring only the faulty sub-system, to the same configuration used before the fault occurred (Carmichael, Caffrey et al. 2000). In presence of continuous radiation on the same part of a system, it is necessary to use different configurations so that the faulty functionality may be shifted when the fault is recoverable. Figure 1.5 presents the classification of faults based on cause and effect (Bolchini and Sandionigi 2010).

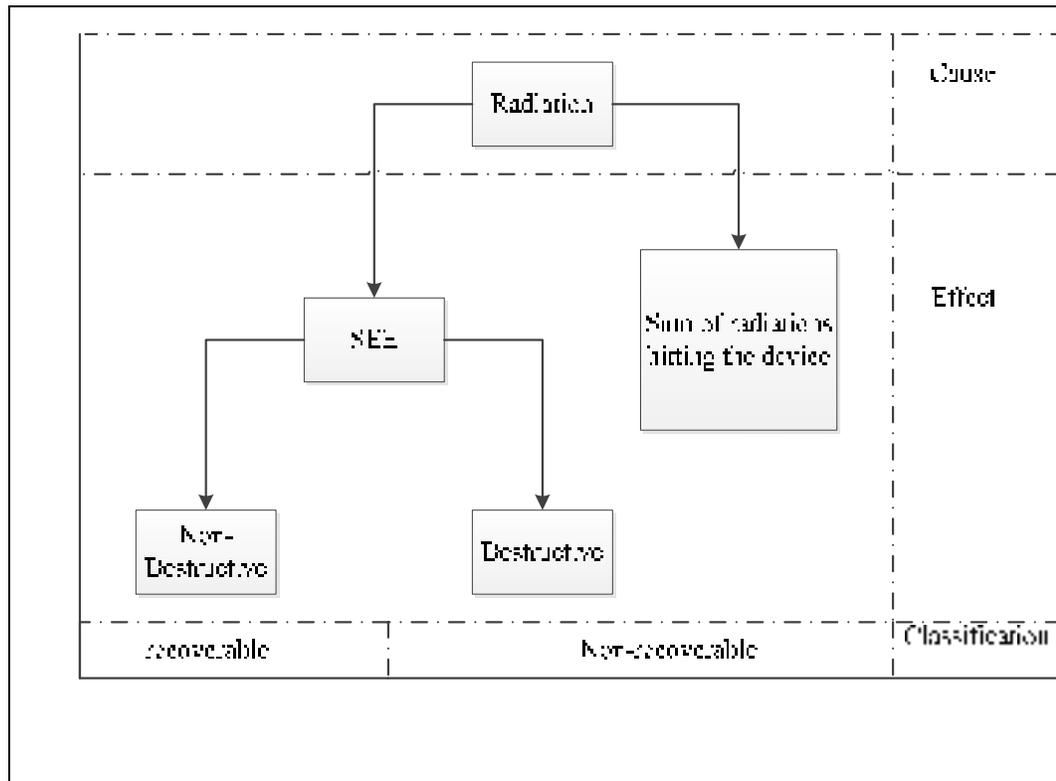


Figure 1.5 Fault classification based on cause and effect  
Adapted from Bolchini and Sandionigi (2010)

All effects on electronic circuits - non-recoverable or recoverable - which are induced by a single radiation event are called single event effect (SEE). Single event upset (SEU) is a subclass of SEEs (ALTERA 2013). SEUs occur when a high energetic particle attacks the depletion area of an N-P junction. This interaction may generate a voltage spike that can change the state (referred to as 'bit flip') of the storage element. This change in the state is called a single event upset. Because only the data in the storage is corrupted and it is temporary, it is termed a soft error (Vibishna, Beenamole et al. 2013). Note that soft errors or SEUs are non-destructive and recoverable. The following, introduces the reader to other subclasses of SEE (JESD89A 2006):

**Single Event Latch-up (SEL):** an irregular high current state in a storage cell which occurs when a high energetic particle attacks the sensitive area of the device, resulting in device malfunctioning;

**Single Event Functional Interrupt (SEFI):** a non-destructive error that causes the device to reset, lock-up, or other such circuit malfunctioning;

**Single Event Transient (SET):** a temporary increase in the voltage of a node caused by a single energetic particle attack.

Non-recoverable faults are not within the preview of this project and will therefore not be presented in detail.

### 1.2.2 Faults, errors and failures

This section aims to understand the difference between faults and errors. Note that errors are the manifestation of faults. Figure 1.6 illustrates the difference between faults and errors. When a fault is not masked by the component (inner scope) and manifests itself in the outer scope, error occurs (Mukherjee 2011, p. 7).

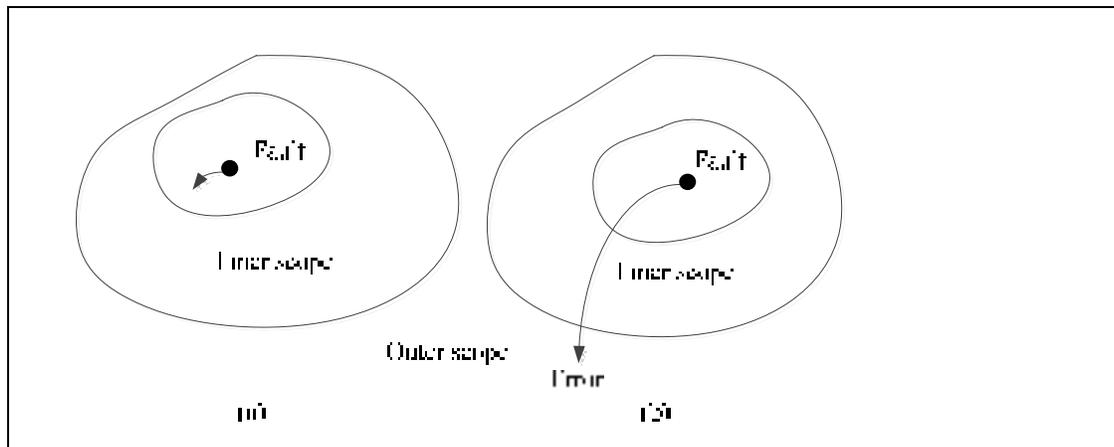


Figure 1.6 a) Fault which is masked in inner scope b) Fault which manifests in outer scope will be called error

Adapted from Mukherjee (2011), p. 8

When an error presents itself to the user, failure is said to occur. Failure may be attributed to system malfunction leading to a deviation from the correct behaviour of the system. Errors are classified as either permanent or transient: permanent faults cause permanent errors, while transient faults cause transient or soft errors (Mukherjee 2011, p. 8). In this thesis, soft

faults or soft errors are our concern. The following section explores the way in which circuits could be verified by fault injection.

### **1.3 Digital circuit verification by fault injection**

For modern digital circuit designs, one way to verify its behaviour is using a high-level language like VHDL. The design of the circuit should be evaluated before an actual implementation. This evaluation is based on a number of criteria such as area, power of consumption etc. By evaluating the design at a high-level of abstraction in presence of faults, the design could be modified before its actual implementation. This ability allows a desired design to be achieved after the actual implementation process. A fault injection tool could provide such ability. Note that there are two categories of faults: permanent and transient. During offline testing by the producer of the logic circuit, most of permanent faults of the circuits are recognized. Therefore permanent faults are the primary concern after a chip is purchased by the customer. To evaluate the performance of a non-line testable circuit, the capability to simulate the injection of a transient fault in the VHDL description of a circuit is necessary. This allows the performance of a circuit to be verified in faulty conditions before its actual implementation (Seward and Lala 2003).

Fault injection could be done at a low-level or a high-level of abstraction. Fault injection tools include those for fault simulation and fault emulation. Fault simulation tools are used for evaluating radiation effects and fault tolerance by applying analytical methods; however, fault emulation tools apply hardware methods. By using fault simulation and emulation tools, traditional radiation hardness assurance (RHA) techniques are improved. These steps ensure that circuit designs meet desired goals after implementation. By applying fault simulation and emulation tools, the test of the circuit design on the benchtop is allowed while ensuring that time and financial limitations for accelerated radiation testing are economized (Quinn, Black et al. 2013). In the following sections, different models used for fault injection are described and fault injection techniques are explained.

### 1.3.1 Fault models

Models emulate a physical reality using mathematical abstraction. For injecting faults in models of the digital circuit, fault models are defined. Two different fault models, Stuck-at and Bit-flip, are explored in this section (Borecky, Kohlik et al. 2011).

#### 1.3.1.1 Stuck-at (s-a) fault model

Stuck-at fault model is an abstract fault model. It models a functional fault. When stuck-at 1 model is applied a logic 0, a logical error is produced - this means the logic 0 becomes logic 1 after applying stuck-at 1 to a logic 0. The physical equivalence for this fault model are short and open connections (Patel 2005). In figure 1.7, the fault s-a-0 is activated on line G when logic value 0 is applied on line G. As illustrated, this fault changes the logic value 1 to 0 on line G.

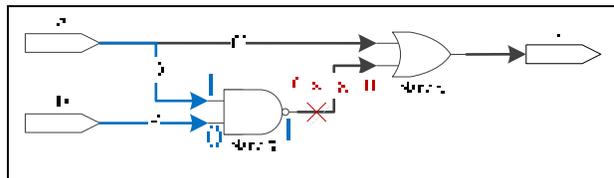


Figure 1.7 The fault s-a-0 is activated on line G when logic value 0 is applied on line G  
Adapted from Patel (2005)

In this thesis, single stuck-at fault model is used. In (Grosso, Guzman-Miranda et al. 2013), the authors establish a correlation between the effects of SEU and the Stuck-At (SA) faults model. According to their results, if injecting SA0 and SA1 at a specific location in a digital circuit doesn't produce a faulty output, SEUs won't produce error when they occur at the same location for a large majority of cases affecting the digital circuit it is a good model. However, simulation reveals that, by applying stuck-at fault model, sometimes fault injection simulator generates the same output obtained when no fault is injected. For example, when the value of a net in a circuit is 0, injecting a stuck-at 0 on that net does not have any observable effect on the circuit primary outputs.

### 1.3.1.2 Bit flip fault model

Bit flip fault model represents a modification in the state of memory. This change is caused by an SEU. As illustrated in figure 1.8, a bit flip fault changes the value of the bit memory from “0” to “1” (Borecky, Kohlik et al. 2011).

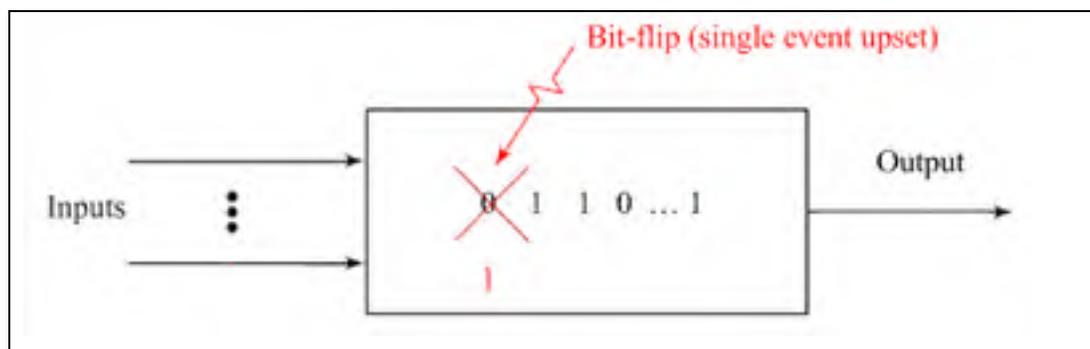


Figure 1.8 The bit flip fault injected into memory  
Adapted from Borecky, Kohlik et al. (2011)

It appears plausible that the bit flip fault injection model may be more efficient for this thesis. By applying bit flip fault injection, the state of the bit is always modified and there is greater correlation between bit flip fault model and SEU occurrences than between stuck-at fault model and SEU occurrences. Applying the bit flip fault model on the experimental results presented in this thesis offers the scope of future work.

### 1.3.2 Fault injection techniques

Fault injection is a validation technique of the dependability of fault tolerant systems, executed by observing the behaviour of the system in the presence of faults which are induced or injected into the system. There are several techniques to inject faults into a system model. They are classified into three main categories and are explained in the following sections.

### **1.3.2.1 Hardware-based injection techniques**

In the hardware-based fault injection technique, a system is tested by specially designed test hardware for the injection of faults into the target system and examining its effects. These kinds of faults are injected at the pin level of the Integrated Circuit (IC). For transistors, stuck-at, bridging, or transient faults are injected and circuit operation thereafter allows their effects to be examined.

Hardware fault injections are applied to the actual circuit after fabrication. By using some sort of interference which could produce and inject fault into the circuit, the behaviour of the circuit in the presence of fault is examined. This injection technique consumes time to prepare and thereafter test the circuit of interest. Testing the circuit by this method is faster than simulation, although the motivation behind testing a circuit before the delivery to the customer is pretty obvious (Alfredo Benso 2003).

In figure 1.9, some examples of this technique are presented (Ziade, Ayoubi et al. 2004). In the following sections, other techniques for fault injection are defined.

### **1.3.2.2 Emulation-based injection techniques**

FPGAs could be used to emulate the circuit behaviour at hardware speed and are therefore used to speed up the process of fault injection. The principle behind this technique is to execute tasks accomplished by simulation-based fault injection techniques using an FPGA. The problem with using such techniques is about retaining the flexibility of a simulation-based approach. With more tasks transferred from the computer to the FPGA, the controllability and observability of the process under test decreases. These approaches are faster than simulation-based techniques, but communication bottlenecks while data is transferring between the computer and the FPGA pose the main problem. In fact in these approaches, a major part of the task is accomplished using hardware, while the rest is

executed in the computer (Valderas, Garcia et al. 2007). There are two categories for fault injection by using FPGA emulation approaches - by using instrumented circuit technique, or partial reconfiguration technique. In partial reconfiguration methods, by reconfiguring the proper FPGA cell, a flip flop value is modified. It is usually done by resetting the flip flop (Antoni, Leveugle et al. 2002). In the instrumented circuit method, all circuit flip-flops are replaced by a more complex structure. In this way, fault injection may be controllable externally. The flip-flop substitution includes a fault mask and an injection enable signal. The fault mask allows the fault localisation to be selected, while the injection enable signal allows controlling the fault time instant (Lima, Rezgui et al. 2001).

SEU Controller is a tool created by Xilinx that can be included in any design using the Virtex-5 FPGA family or higher. This tool allows us to detect and correct SEU, but also used to emulate SEUs within the Virtex-5 device. By applying SEU controller, errors injection in a controlled and predictable way into the configuration memory can be done (Chapman 2010).

### **1.3.2.3 Simulation-based injection techniques**

Simulation-based fault injection is when faults are injected in models of the system using a fault simulator. Different modeling languages can be used like VHDL, SystemC, etc. In this way, the system dependability may be evaluated when only the model of the system is developed. By using different description languages, this approach could address different levels of abstraction (Ziade, Ayoubi et al. 2004). Figure 1.9 presents these different levels.

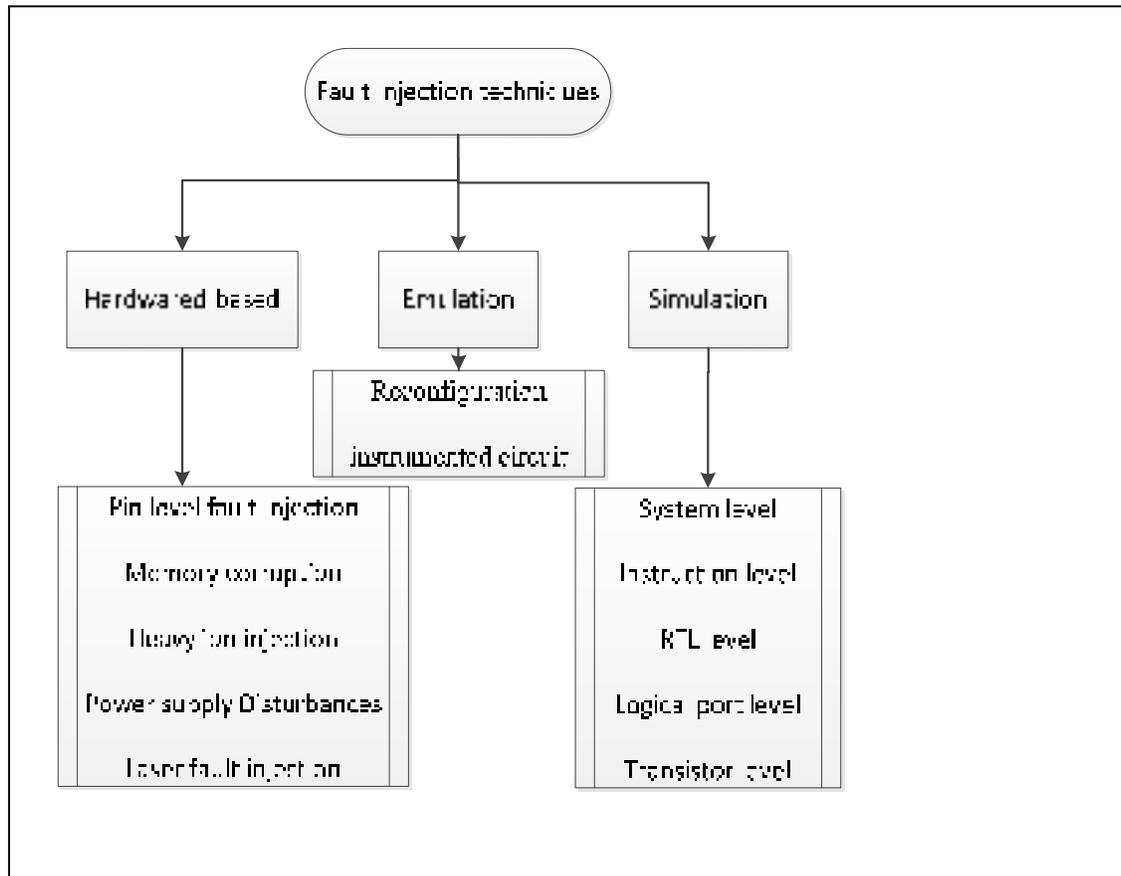


Figure 1.9 Classification of fault injection techniques  
Adapted from Ziade, Ayoubi et al. (2004)

Hardware simulation is used in this thesis and occurs at a low-level of abstraction of the circuit. Faults are injected into the gate level description of the circuits. Thereafter, the system is simulated and the response of the circuit to that particular fault is evaluated.

#### 1.4 Developing fault behavioural model

In this section, the two most popular fault injection techniques at a high-level of abstraction are discussed, and the way in which they are applied to the system is explained.

**Saboteur:** A saboteur is a VHDL component added to the original model, the goal of which is to change the value of one or more signals which is simulating the occurrence of a fault. To ensure normal operation of the system, this component may be made inactive. Ports of the

components in the model are affected by the saboteur. Therefore, for applying a saboteur, structural descriptions are needed. (Baraza, Gracia et al. 2008).

**Mutant:** A mutant is a component which replaces another component. While it is inactive, it behaves like the original component in existence of faults. Three ways may be used for developing mutation (Baraza, Gracia et al. 2008):

- in structural model descriptions, saboteur is added;
- structural descriptions are modified and sub-components are replaced. For example a NOR gate could be replaced by a NAND gate;
- in behavioral descriptions, syntactical structures are modified.

By using both techniques, fault behaviour models may be developed and used to find out how systems behave in presence of faults at high-level of abstraction. In this way, the reliability of the circuit could be studied early during the design flow itself. Our main concern in the work presented here is to capture the faulty behaviour of a component, so as to enable a realistic injection of fault. Neural networks are a viable solution for that. A neural network may be used to learn this faulty behaviour. In the following section, neural network and its advantages for capturing the faults in a system are studied.

### 1.5 Neural network and fault

The neural network has been used for fault diagnosing and fault clustering in digital and analog circuits before. The following describes some areas of using neural network approach for digital or analog system in presence of faults.

In (Al-Jumah and Arslan 1998), feed forward artificial neural networks are used for the diagnosis of multiple stuck-at faults in digital circuits. In their technique, a fault truth table is developed by injecting single random stuck-at fault in the circuit, and the result data is used to train the neural network. The results show that an efficient and flexible neural network is obtained, which is capable of diagnosing multiple stuck-at faults.

In (Whei-Min, Chin-Der et al. 2001), a new approach to detect fault types in a transmission line (which is an analog circuit component) is presented. A Neural network is used to detect various patterns of correspondent voltages and currents. In the paper, radial basis function (RBF) neural network is compared with back propagation neural network, and results show that RBF can provide the faster and more accurate model.

In (Lifen, He et al. 2010), a new fault detection method for analog circuits is presented. In their work, the output signal of the circuit under test is studied and the kurtoses and entropies of the signals are determined. These characteristics are used to measure the signal's higher order statistics. The kurtoses and entropies are used as inputs of neural network in training phase to develop fault classifier neural network model. The proposed method works with high accuracy for nonlinear analog circuits. This method can classify both soft and hard faults.

Previous studies show that the neural network approach is appropriate to be applied for fault clustering and fault diagnosis in digital and analog circuits. Neural networks are able to learn non-linear relations between data. After learning, Artificial Neural Network (ANN) structure could estimate a function for mapping input-output pairs. Furthermore, the trained ANN has the ability of generalization - which means that it could produce almost the desired outputs for inputs which are not participating in the training process.

For developing fault behaviour model of the digital circuit, it needs to have a component which is able to generate outputs of a circuit. The mentioned ability of ANN seems to be useful to develop that component. The faulty behavior of the circuit could be trained by ANN, following which the trained ANN may be used to generate faulty output of circuits.

In the course of this thesis, the possibility of applying the neural network to develop the faulty behaviour model of digital circuits in presence of faults or no is explored. The next chapter will explore neural networks, and present the information needed to develop one.

## **1.6 Conclusion**

In this section, cosmic rays are introduced, and explain how they cause faults in digital circuits. Afterwards, fault models are discussed and fault injection techniques are explained. Finally, neural network and its advantage with regard to fault diagnosis and fault clustering are discussed. As neural networks are used for fault recognition in digital circuit, it promises to be useful for developing fault behavioral model as well. In this thesis, a new way of developing fault behavioural model at a high-level of abstraction by using the neural network approach is explored. In the next chapter, neural networks are studied, and the benefits regarding the way in which neural network fault behavioral model maybe developed is explained.



## CHAPITRE 2

### NEURAL NETWORKS AND THEIR APPLICATIONS IN FAULT MODELS

In this chapter, neural networks are explained. First, the neuron is described and then neural network architectures are presented. Multilayer perceptron (MLP), which is one of the most common classes of neural network, is also explained in detail. To use neural networks efficiently, there are some tips that should be considered. Section 2.5 illustrates how to use the MLP neural network in practice. In short, this chapter presents how to develop a neural network for a specific problem.

#### 2.1 A brief introduction to artificial neural networks

A common problem in engineering fields is to estimate a function, which can map input-output pairs. This is referred to as supervised learning by the neural network (Cerny and Proximity 2001). The training set is composed of pairs of independent (input) and dependent (output) variables. The neural network represents the map function  $\varphi$  between input vectors ( $x$ ) and output vectors ( $y$ ) according to the following equation:

$$y = \varphi(x) \tag{2.1}$$

Where  $x$  is a vector for inputs and  $y$  is a vector for outputs of the system under investigation.

Neural network models have two different categories: supervised neural networks (such as the multilayer perceptron), or unsupervised neural networks (such as kohonen feature maps). For supervised neural networks, training and testing data are used to build the model, and the data includes input patterns with the corresponding output values. For unsupervised neural networks, the network decides what output values are best for the current input pattern, and there is no predetermined output value for any input pattern (Cerny and Proximity 2001).

In fact, a neural network is a huge parallel and distributed processor, which has the capability of storing information. It is similar to the human brain in three aspects:

- knowledge of neural network is acquired by learning process;
- synaptic weights, which are inter-neuron connection strengths, are responsible for storing knowledge;
- the network has the ability to generalize.

The process of learning is achieved through what is referred to as a learning algorithm. Changing synaptic weights to achieve a desired design goal is the objective of the algorithm. The trained network has the ability of generalization, which means that it can produce almost all the desired outputs for the inputs patterns that are not participating in the training process.

In the following, neural network characteristics which are relevant to this thesis are specified:

**Learning algorithm:** one of the more popular classes of learning algorithm is supervised learning. The network is fed with input samples, and weights of the network change so as to minimize the difference between the value observed at the output of the network and the desired output value. Learning is stopped when there are no more significant changes in the value of the weights;

**Ability of nonlinear mapping:** a neuron is a nonlinear element is a nonlinear activation function in a neuron which produces an output from the weighted input. Therefore, a neural network (which is composed of neurons) is a nonlinear mapping model. In the next section, a neuron and its activation function are explained in details;

**Ability of adaption:** a neural network is trained for a specific task in a specific environment (input-outputs pairs). However, it should be able to deal with a small change in the environment, and this is referred to as the ability of adaption of the neural network.

The most common neural network for supervised learning is the multilayer perceptron (MLP) which was established in 1986 (Jain, Mao et al. 1996). As illustrated in figure 2.1, the MLP is composed of feed-forward connections with adjustable weights. Training of the MLP refers

to the estimation of the best set of weights while the error between the network output and the desired output is minimized (Farhat 2003).

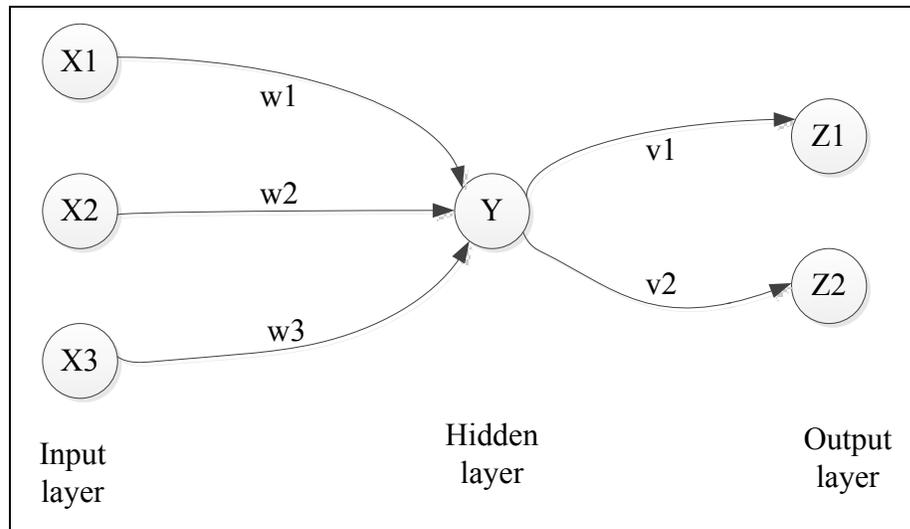


Figure 2.1 A small example of an MLP neural network with one hidden layer  
Adapted from Fausett (2006)

In figure 2.1, neuron  $Y$  sends its output  $y$  to neurons  $Z_1, Z_2$  with respective connection weights  $v_1$  and  $v_2$ . The received signals by  $Z_1$  and  $Z_2$  are different according to the different scales which come from the different weights  $v_1$  and  $v_2$  on their links. Note that this is a very simple neural network with a hidden layer and a nonlinear activation function (Fausett 2006).

## 2.2 The neuron

A neural network is an information processing system. It shares common characteristics with biological neural networks. In fact, it is a mathematical model of human cognition or neural biology. Such a model is based on the following assumptions:

- an artificial neuron (perceptron), which is shown in figure 2.2, is a very simple element where information processing occurs;

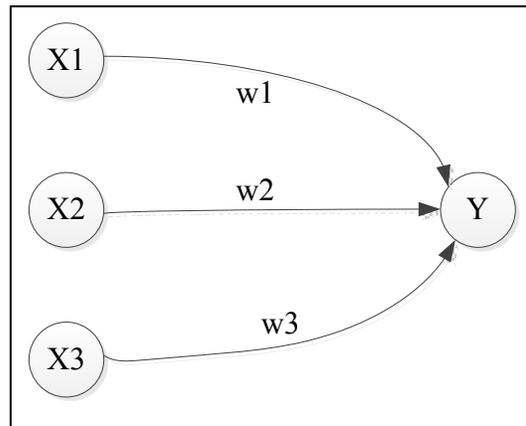


Figure 2.2 A simple element which is called artificial neuron or perceptron.  $X_1$ ,  $X_2$ ,  $X_3$  and  $Y$  are neurons or perceptrons; The weights on the links to neuron  $Y$  from neurons  $X_1$ ,  $X_2$  and  $X_3$  are  $w_1$ ,  $w_2$  and  $w_3$   
Redrawn from Fausett (2006)

- there is a connection link between any two neurons for passing signal;
- for each link, a weight is associated and this weight is multiplied to the signal transmitted;
- each neuron determines its output by applying an activation function to its input which will be discussed in details.

Each neuron behaves as a function; it transmits an input signal  $x$  to an output signal  $\varphi(x)$ . The function  $\varphi(x)$  could be assumed in different forms. It could model basic activation functions such as the sigmoid, the threshold or the radial basis function. In figure 2.3, the sigmoid activation function is illustrated. In the rest of this chapter, activation functions are further explained.

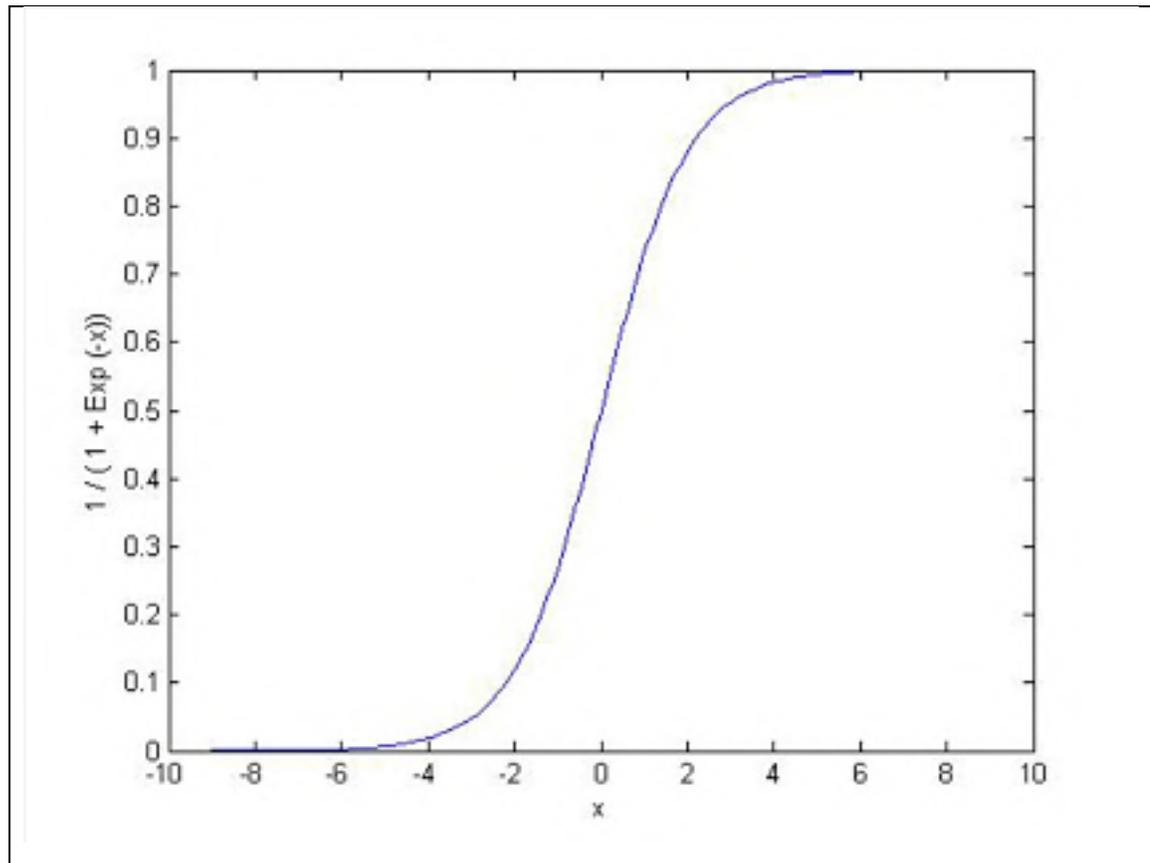


Figure 2.3 The sigmoid activation function

Each neuron has an internal state which is a response of the activation function to the input of the neuron. This internal state is called the activity level of the neuron, or the neuron's activation. The internal state of each neuron is broadcast as a signal to all the connected neurons via connection links.

In figure 2.2, consider the neuron  $Y$  that is connected to three neurons  $X_1$ ,  $X_2$  and  $X_3$ . The activation of these three neurons are  $x_1$ ,  $x_2$  and  $x_3$ . The weights on the links to neuron  $Y$  from neurons  $X_1$ ,  $X_2$  and  $X_3$  are  $w_1$ ,  $w_2$  and  $w_3$ . Input of neuron  $Y$ ,  $y_{in}$ , is the sum of the signals from neurons  $X_1$ ,  $X_2$  and  $X_3$ , that is:

$$y_{in} = w_1x_1 + w_2x_2 + w_3x_3 \quad (2.2)$$

Activation of neuron  $Y$ ,  $y = f(y_{in})$ , is calculated with a function of its input signals. As explained before, there are several activation functions and the logistic sigmoid (2.3) is one of them:

$$f(x) = \frac{1}{1 + \exp(-x)} \quad (2.3)$$

In the following section, neural network architectures are explained.

### 2.3 Neural network architectures

Neuron arrangement and the way in which they are connected together have a strong effect on the learning algorithm used to train the network. Neural network architectures are classified into three main categories as illustrated in figure 2.4 (Haykin 2001):

- single layer feed forward;
- multi-Layer Perceptron feed forward (MLP);
- recurrent network.

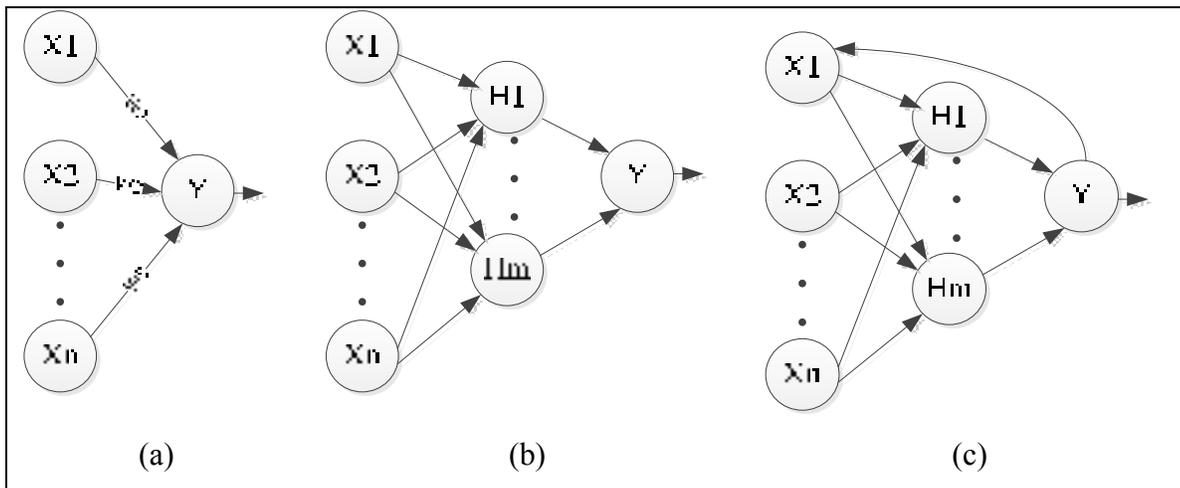


Figure 2.4 The different architectures for neural network: a) single layer feed forward, b) multi-layer perceptron feed forward, c) recurrent network  
Adapted from Haykin (2001)

In this thesis the MLP network architecture is used, which is also the most well-known and widely used topology. They are able to represent non-linear mapping between inputs and

outputs, and are universal approximations (Yu 2000, Cerny and Proximity 2001). Feed forward neural networks seem well suited for the purpose of the work here; as a black box, it facilitates the development of the fault behavior model of the circuit by considering only the inputs and the outputs of the circuit without any information of the circuit details. In addition, generalization ability of the neural network helps us to perform sampling by considering only part of the data, enabling the neural network model to map the remaining after training. In the following section, multi-layer neural networks trained by the back propagation algorithm are explained in detail.

#### 2.4 Multi-layer perceptron (MLP) networks

Consider  $n$  as the number of input neurons and  $m$  as the number of output neurons of a neural network. Let  $x$  be a  $n$  dimensional vector which contains all inputs of the neural network and  $y$  be an  $m$  dimensional vector which contains all outputs of the neural network. In addition, let  $w$  be a vector representing all weights used for internal connections between the neurons. The structure of a neural network is defined according to the choice of values for all weights in the vector  $w$ . This means that how the output of the neural network ( $y$ ) is computed from the input vector  $x$  is determined by choosing values for the elements of the vector  $w$ . The following equation (2.4) mathematically represents a neural network model (Zhang, Gupta et al. 2003).

$$y = y(x, w) \quad (2.4)$$

The learning by the neural network happens using the weights of the connection links. Therefore to solve more complicated problems, more links should be added. To increase the number of links, there are two options (Marsland 2011):

**Add backward connections:** using this technique, the output of the neural network is connected to the input of the neural network, and recurrent neural networks are developed as shown in figure 2.4 c);

**Add more neurons:** by adding more layers of neurons between the input and the output layers, multilayer neural network architecture is developed: figure 2.4 b).

A neural network can solve non-linear problems by including a hidden layer in its architecture. Neural networks with hidden layer have the ability to solve many more problems than can be done using a simple neural network without hidden layers. Such an ability may be attributed to the non-linear relation between its input and output neurons (Fausett 2006).

As illustrated in figure 2.5, neurons in MLP neural networks are grouped into layers. The first layer is the input layer and the last layer is the output layer. All the middle layers are called hidden layers.

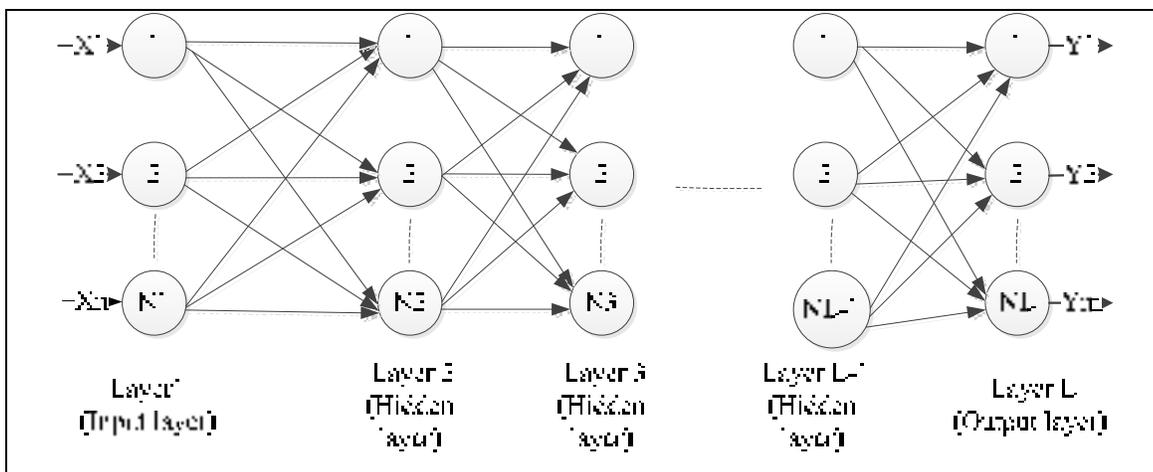


Figure 2.5 The multilayer perceptron neural network structure  
Adapted from Zhang, Gupta et al. (2003)

Suppose  $N_l$  is the number of neuron in the  $l$ th layer,  $l = 1, 2, \dots, L$ .  $w_{ij}^l$  is the weight of connection between  $j$ th neuron in the  $l - 1$ th layer and  $i$ th neuron of the  $l$ th layer.  $x_i$  is the  $i$ th input of the neural network and  $z_i^l$  is the output of the  $i$ th neuron of the  $l$ th layer.  $w_{i0}^l$  represents the bias for the  $i$ th neuron of the  $l$ th layer. In equation (2.5), vector  $w$  of the neural

network includes all  $w_{ij}^l$  while  $j = 0, 1, \dots, N_{l-1}$ ,  $i = 1, 2, \dots, N_l$ ,  $l = 2, 3, \dots, L$  (Zhang, Gupta et al. 2003).

$$w = [w_{10}^2, w_{11}^2, \dots, w_{N_L N_{L-1}}^L]^T \quad (2.5)$$

Each neuron has a threshold value  $\theta$  which determines when the neuron state will be changed. When the input of a neuron is more than the threshold  $\theta$ , the neuron fires : this means that the state of the neuron is changed. When the neuron input is less than the threshold  $\theta$ , nothing happens and the neuron remains in its current state. Threshold  $\theta$  should be adjustable in order to have a changeable value for firing each neuron. For enabling this capability, we add an extra input weight to each neuron with fixed inputs. This weight is updated during the learning phase, similar to other weights, in order to adjust if the neuron fires or not. This input node is called bias. The bias node has weight  $w_{0j}$  connecting to  $j$ th neuron (Marsland 2011).

#### 2.4.1 Preliminaries definitions for MLP learning algorithm explanation

Some preliminary definitions are now explained which will be used to explain the MLP learning algorithm:

**Inputs  $x$ :** it is the input vector with elements  $x_i$ , where  $i$  is from 1 to input dimension ( $m$ );

**Weights  $w_{ij}$ :** it is the weight value of the connection signal between node  $i$  and  $j$ ;

**Outputs  $y$ :** it is the output vector with elements  $y_j$ , where  $j$  is from 1 to output dimension ( $n$ );

**Target  $t$ :** it is the target vector with elements  $t_j$ , where  $j$  is from 1 to output dimension ( $n$ ).

We will use this vector during supervised training. It contains the correct outputs that a neural network uses during training phase;

**Activation function  $g$ :** it is a mathematical function which works like the threshold function described earlier. It expresses the firing of a neuron as a reply to weighted inputs;

**The learning rate  $\eta$ :** it describes how much the weights are changing in order to decrease errors between current outputs and target outputs. It lies in the range specified by  $0.1 < \eta < 0.4$  (Marsland 2011) and depends on the amount of error we expect from the input.

### **2.4.1 MLP learning algorithm description**

The following describes the steps of Multi-Layer Perceptron training algorithm. This is also called back propagation learning (Marsland 2011). In figure 2.6, the steps of this algorithm are illustrated.

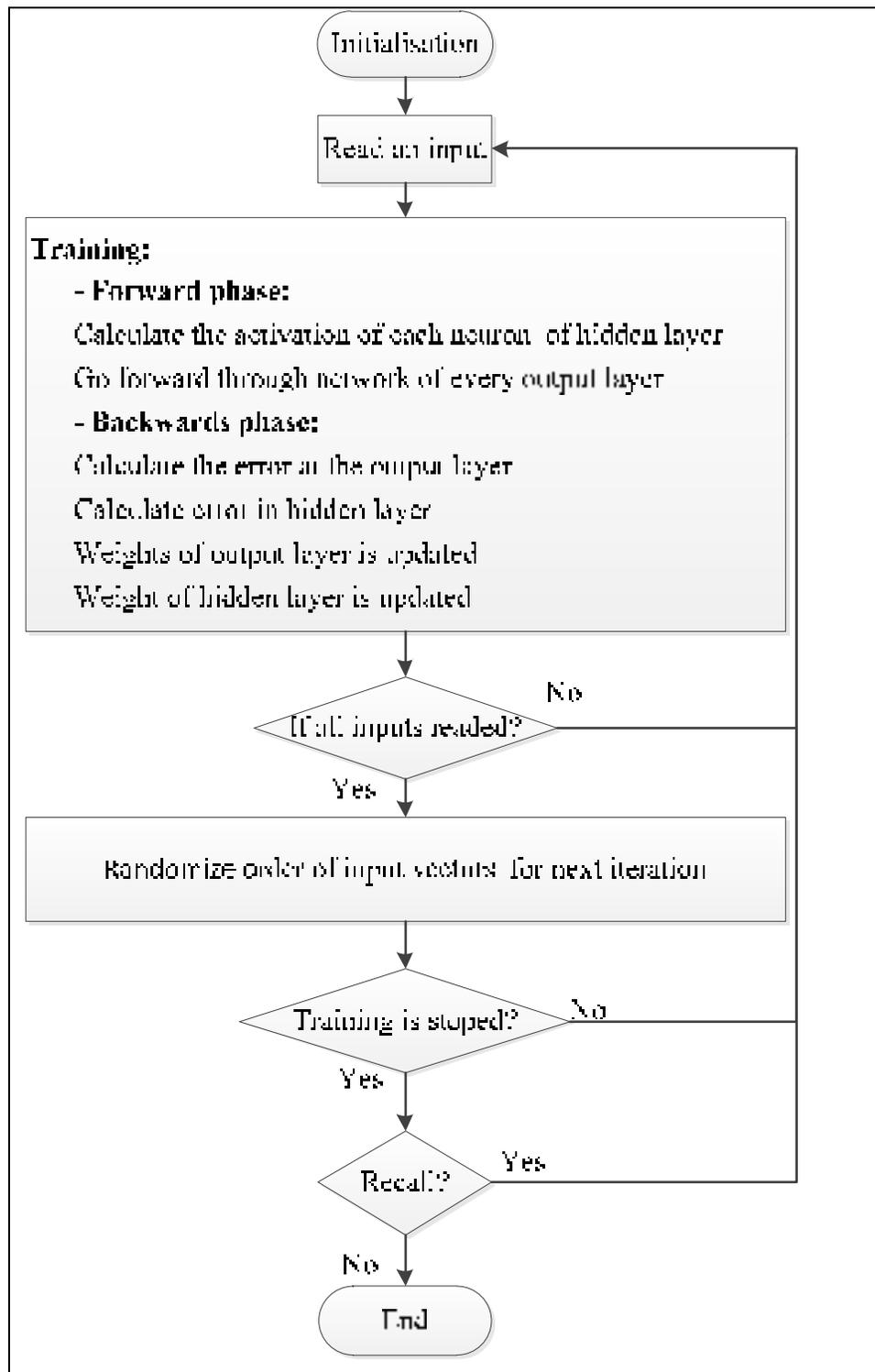


Figure 2.6 Steps of Multi-Layer Perceptron training algorithm

The steps of the MLP algorithm are now explained in greater detail.

In the Initialisation step, all weights are initialized to small random values (positive and negative).

In the Training step there are two phases:

**Forward phase:** calculate the activation  $a_j$  of each neuron  $j$  of the hidden layer by the following formula ( $\beta$  is a positive parameter). First,  $h_j$  (which is the addition of all weighted inputs of neuron  $j$  of the hidden layer) is calculated by formula (2.6), and then by using it in formula (2.7) which gives activation function ( $a_j$ ), the output of neuron  $j$  ( $g(h_j)$ ) is calculated;

$$h_j = \sum_i x_i v_{ij} \quad (2.6)$$

$$a_j = g(h_j) = \frac{1}{1 + \exp(-\beta h_j)} \quad (2.7)$$

Go forward through network such that output layer has the following activation function:

$$h_k = \sum_i a_i w_{jk} \quad (2.8)$$

$$y_k = g(h_k) = \frac{1}{1 + \exp(-\beta h_k)} \quad (2.9)$$

**Backwards phase:** calculate the error at the output layer with the formula (2.10):

$$\delta_{ok} = (t_k - y_k)y_k(1 - y_k) \quad (2.10)$$

Calculate error in hidden layer with following formula:

$$\delta_{nj} = a_j(1 - a_j) \sum_k^n w_{jk} \delta_{ok} \quad (2.11)$$

Weights of output layer is updated using the following: ( $w$  is the weight of the first hidden layer)

$$w_{jk} \leftarrow w_{jk} + \eta \delta_{ok} a_j^{\text{hidden}} \quad (2.12)$$

Weight of hidden layer is updated as following: ( $v$  is the weight of the first hidden layer)

$$v_{ij} \leftarrow v_{ij} + \eta \delta_{hj} x_i \quad (2.13)$$

In the next section, activation function in neuron element is explained in details.

## 2.4.2 Neuronal activation functions

The functions  $\varphi$  in the hidden layer are not identical as the ones in the output layer. There are a lot of different activation functions to produce an output from the weighed input. The activation function is selected according to the task of the neuron. In figure 2.7, the most common activation functions are illustrated.

**Hard limit activation function:** if the neuron input reaches some threshold, this function sets the neuron output to 1, and otherwise to 0.

**The symmetrical hard limit activation function:** if the neuron input reaches pre-specified threshold, this function sets the neuron output to 1, and otherwise to -1.

**The linear activation function:** this function multiplies neuron input signal by a constant scale and after adding bias value to it, the signal is transmitted to the neuron output.

**The saturated linear activation function:** this function works the same way as the linear activation function while the neuron input is between  $[-1, 1]$ ; the neuron output is set to 1 when the neuron input is more than 1, and the neuron output is set to -1 when the neuron input is less than -1.

**The log sigmoid:** by using this function, while input varies between  $-\infty$  to  $\infty$ , the neuron output is between  $[0, 1]$ . This is a differentiable function and is useful for back propagation training algorithm.

**The tan-sigmoid activation function:** by using this function, while input varies between  $-\infty$  to  $\infty$ , the neuron output is between  $[-1, 1]$ .

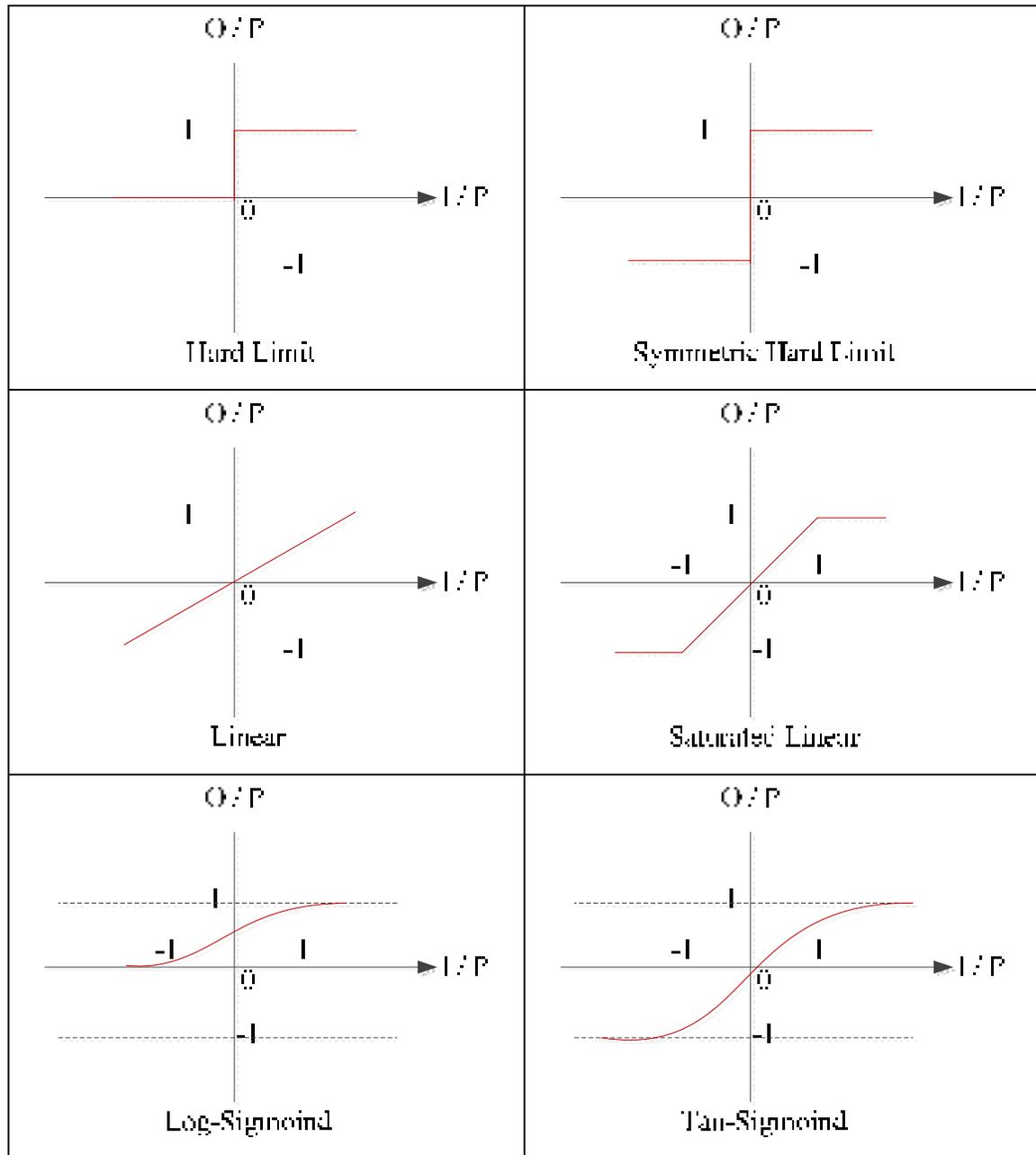


Figure 2.7 Commonly used activation functions for a neuron, where O/P denotes the output of the neuron and I/P denotes the input  
Adapted from Farhat (2003)

Generally, the linear functions are suitable for the neuron in the output layer while the sigmoid functions are suitable for the neuron in the hidden layer.

## 2.5 Multi-layer perceptron in practice

In this section, the process to determine the parameters to develop the network is explained. According to difference in number for these parameters, the neural network will be adjusted to find solutions for different kinds of problems.

### 2.5.1 Data preparation

With pre-processing on input and target datasets before training, most of the neural network and machine-learning algorithms work with greater precision. For example, if the activation function for output is the sigmoid function, the target values should be between 0 and 1. Regardless of which activation function used, it is normal to scale target values between 0 and 1 to ensure weights do not get too large. Also, the input dataset may be scaled to avoid increasing the number of required weights. The most common method for MLP is to scale each data dimension to have a zero mean and a unit variance. Scaling data while maximum is 1 and minimum is -1 will result in a zero mean value and unit variance value. This scaling is called normalization or standardization. Though normalization for MLP is not necessary and only ensures better results, for some neural network it is necessary since without normalization, they cannot learn the relation between input and output sets (Marsland 2011).

### 2.5.2 Size of training data

For a neural network with one hidden layer, the number of weights is  $(m + 1) \times n + (n + 1) \times p$ , where  $m$  is the number of nodes in the input layer,  $n$  is the number of nodes in the hidden layer and  $p$  is the number of nodes in the output layer. The extra +1s are for bias nodes which have adjustable weights. The weights of connection links between neurons are set during the training phase. Back propagation algorithm is responsible for setting the values of weights and these values are determined from the error calculated from data during the training phase. It is obvious that with more training data, learning is better although the time of training phase is consequently more. Unfortunately, there is no way to determine the minimum amount of training data, and it depends on the problem. Based on rule of thumb, the amount of training data should be at least ten times the number of connection links. This

is a large number and training the neural network is a computationally expensive operation (Marsland 2011).

### **2.5.3 Number of hidden layers**

There are two more considerations for neural networks: the number of hidden layers, and the number of nodes in them. Later, it is explained why two hidden layers is the maximum amount that an MLP neural network needs. Note that there is no theoretical guide for selecting the number of nodes in hidden layers and they could be specified practically. In practice, by testing the neural network with different number of nodes in the hidden layers, the best number is found. For the back propagation algorithm, it is possible to have any number of hidden layers, but as mentioned before, normally, two hidden layers is the maximum which is needed. The reason for this is that approximate the mapping of any smooth functional by a linear combination of sigmoidal activation functions is possible. Therefore, any decision boundary (besides linear ones) may be approximated by a neural network. Here, the lines which classify different data are called decision boundary (Marsland 2011).

### **2.5.4 Generalization and over-fitting**

Generalization is the whole purpose of using neural networks to compute the output for all possible input patterns based on training samples. We have to be sure that the neural network has been trained enough to be able generalize well. There is also the danger of over fitting. If training is performed for too long, data will be over fitted, which means that besides the actual function, noise is also learnt. When over fitting happens the trained neural network is too complicated and cannot generalize. Figure 2.8 shows two data sets. White ones are training data and red ones are evaluation data. The lines are the estimation of training data set which comes from the functions trained by the neural network. The right figure shows over fitting results. The estimated line passes all training data (white ones). This line is too complicated and cannot predict evaluation data (red ones). In contrast, the left figure shows the ability of generalization. The line doesn't pass all the training data (white ones), but is an

estimation of training data. This neural network didn't train noise and has the ability to predict evaluation data (red ones).

Solution for over fitting problem has two parts. First, the training has to be stopped before over fitting occurs. Second, the neural network has to be tested (Marsland 2011). In the next section, this is explained in details.

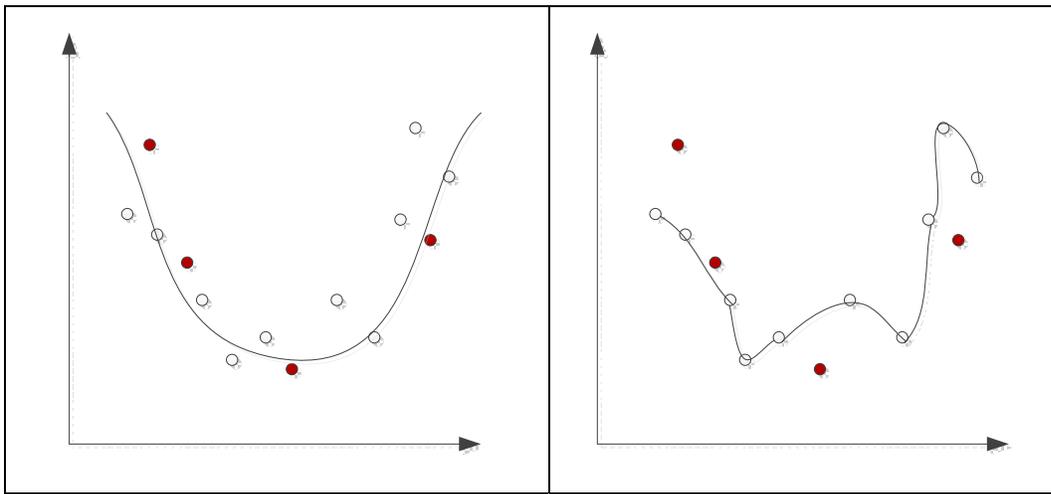


Figure 2.8 There are two data sets. White ones are training data and red ones are evaluation data. The lines are the estimation of training data set which comes from the functions which are trained by the neural network. The left neural network has the ability of generalization but the right neural network exhibits over fitting.

Adapted from Johnson ( 2013)

### 2.5.5 Training, testing, and validation

The error of a trained neural network is calculated by the sum-of-squares error between the computed output and the targeted output. The test should be done with the data, which is not used for training in order to see how well the trained neural network generalizes and if overfitting has occurred. Therefore, we have to keep a reserve dataset for testing that has not been used for training.

During training, it is necessary how well the neural network works, and to decide if it is a good time to stop or not. For this purpose, if we use the training data set, over-fitting cannot

be detected. Test data set is not good either, since it has to be used for the final test. Therefore, another data set should be prepared for this purpose, and it is called the validation data set. It is used for validating the already conducted training.

The proportion of data to train, data for validation and data to test parts is arbitrary but typically, it is in the ratio 50:25:25 (when data is aplenty), and otherwise it is in the ratio 60:20:20 (Marsland 2011).

### 2.5.6 When to stop learning

For learning, the MLP algorithm runs on all data several times while changing the weight of connection links to reduce the error for the next iteration. However, the question to decide when to stop learning is significant, since predefined number of iterations may lead to over fitting or to stop training while learnt is still insufficient. Predefined value of minimum error may lead to non-terminating training or over fitting. However, learning may be terminated when error stops decreasing. Validation data set may be used for monitoring generalization ability in each epoch or each iteration.

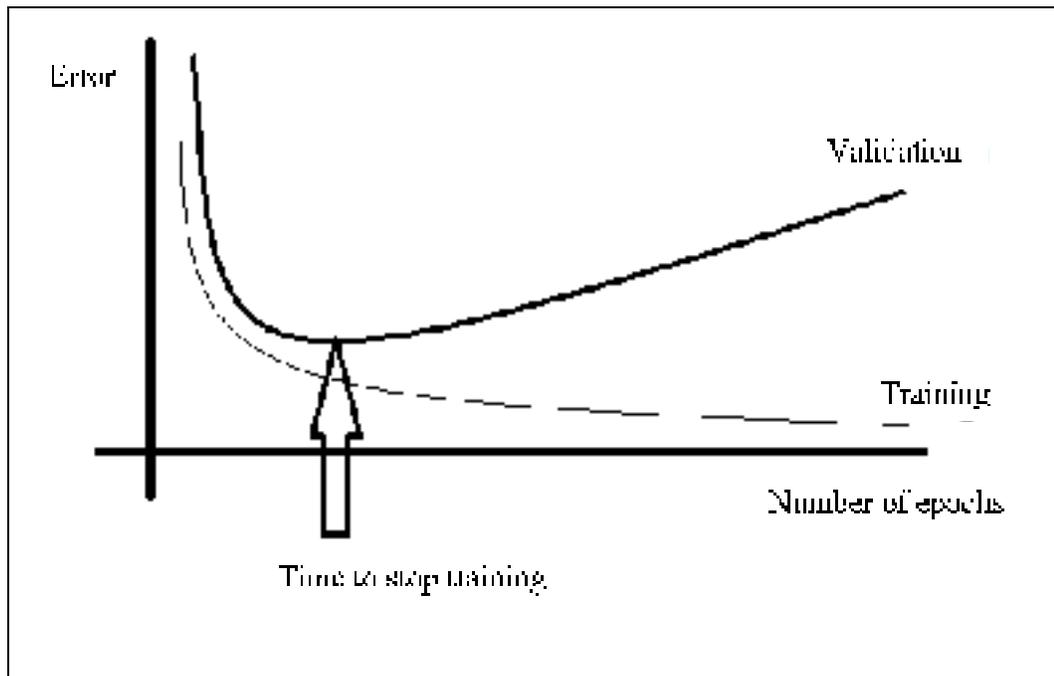


Figure 2.9 Mean-square error during training  
Adapted from Marsland (2011)

According to figure 2.9, mean-square error decreases suddenly in several first iterations after which the rate of decreasing slows down. For deciding about a good time to stop learning, validation set is useful. With validation data, the ability of generalization of neural network is evaluated. When errors of validation line increase, it means the neural network has learnt the actual function and is beginning to learn noise. It is therefore a good time to terminate learning (Marsland 2011).

### **2.5.7 Computing and evaluating the results**

Till this step, learning is completed and it is time to analyse the trained neural network. The test data set is useful for this purpose. Trained neural network runs on the test data set. The error is evaluated by comparing the predicted output with the target output. For a regression problem, all mean square error could be added while for a classification problem, the number of correct predictions (when predicted output is the same as the target output) for each class could be computed (Marsland 2011).

## **2.6 Conclusion**

Different approaches are explored in literature for modeling of circuits at a high-level of abstraction. In this thesis, neural network approach is applied. As explained in the literature, neural networks have the ability to approximate complex non-linear mappings with high accuracy. They are also very flexible with data that is incomplete or noisy, or if some of the data is missing. In addition, performance of neural networks may be automated, thereby minimizing human participation (Cerny and Proximity 2001). Considering the merits of the aforementioned advantages, the neural network is explored as a viable approach in this thesis. Here, all the efforts are directed towards using neural networks to find a way to model the faulty behaviour of the circuit at a high-level of abstraction. For developing the faulty behaviour model of the circuit, the neural network works as a black box, i.e. only the input and the output of the circuit are considered without any information about the circuit details. In the following chapter, the proposed methodology to applied neural network for developing the faulty behavioral model at a high-level of abstraction is be explained.

## CHAPITRE 3

### PROPOSED METHODOLOGY

The goal of developing a behavioural fault model of a circuit is to create a library of faulty components reusable at high-level of abstraction. Components from this library can then be used to replace different parts of a high-level model of the system to replicate the faulty behaviour observed at lower level of abstraction. This facilitates determining the sensitive parts of the model. For example, if on replacing one part of the overall circuit with the correspondent faulty component results in too many errors in the output of the overall circuit, the replaced part is a sensitive part. To improve reliability of the overall circuit, mitigation technique could be applied on these sensitive parts.

In this chapter, different paths are investigated to determine a useful scenario for developing a neural network based model, which is explained in the following sections. The chosen method is described in details.

#### **3.1 Neural network approach and different scenarios**

In the work described in this thesis, all the efforts are directed at using neural network to find a way to model the faulty behaviour of a circuit at high level of abstraction. Firstly, the concept of signatures, proposed by (ROBACHE 2013), is introduced. Signatures are the compressed information about faulty behaviour of the circuit. There are two options going ahead: the first option is to improve the signatures produced, and the second option is to propose a different method to have faulty behaviour information of the circuit.

Neural networks as computational models have the capability to generalize, to learn or to organize data, based on parallel processing. Multi-layer feed-forward neural networks are the most well-known used topology. They are able to represent non-linear mapping between inputs and outputs, and they are universal approximations (Yu 2000, Cerny and Proximity 2001). Feed forward neural network seems well suited for the purpose of the work here:

while it works as a black box, it facilitates the development of the fault behavior model of the circuit by considering only the inputs and the outputs of the circuit without any information of the circuit details. In addition, generalization ability of the neural network helps us to perform sampling; by considering only part of the data, our neural network model could map the rest after training.

In the following sections, two scenarios are proposed: the first one is about using neural network to improve accuracy of signatures, and the second one is about proposing a method to predict faulty output of the circuit by applying neural network concept. Both scenarios are analyzed and discussed in the context of the present work.

### 3.1.1 Concept of signature

A Signature is a concept that has been developed in (ROBACHE 2013). They are generated from the information collected in the reports generated by a fault injection tool. In fact, signatures are the compress form of this information and will be used at a higher level of abstraction. In the work described in this thesis, Lifting is the tool used for fault injection and it is a logic level simulator. The following paragraphs help explain what signatures are.

Error type is defined as the following:

$$\text{Error type} = \text{GoldenOutput} - \text{FaultyOutput} \quad (3.1)$$

As shown above, error type is the difference between the output of fault free circuit (GoldenOutput) and the output of faulty circuit (FaultyOutput). In table 3.1 GoldenOutput, FaultyOutput and error type for an arbitrary circuit is presented.

Table 3.1 Computing error type for a circuit with following golden and faulty outputs

GoldenOutput	FaultyOutput	Error type
00	11	-3
00	10	-2
00	01	-1
00	00	0
01	00	1
10	00	2
11	00	3

For this arbitrary circuit, the percentage of probability of occurrence of different error types is given in table 3.2. Table 3.2 is the table of signatures and these signatures are computed using the method of (ROBACHE 2013). To decrease the complexity and size of signatures, the number of parameters used in signature is minimized. Each signature contains the observed error in the circuit outputs and their corresponding probability. In this way, signatures are an array of errors and the corresponding number of their occurrences generalized for all inputs.

Table 3.2 Computing signatures for the arbitrary circuit presented in table 3.1

Signature	Error type	Percentage of probability
1	-3	1,69%
2	-2	8,01%
3	-1	8,89%
4	0	67,25%
5	1	6,86%
6	2	6,25%
7	3	1,01%

In the presented work, input of circuits will be considered; all efforts are for improving or generating these signatures by neural network approach. Towards the end of this thesis, the results from this neural network approach will be compared with the results obtained by (ROBACHE 2013).

### **3.1.2 Scenario 1: signature adjustment**

In this section, improvement of the accuracy of probabilities of different error types for a specific input is explored. The proposed sketch for this scenario is illustrated in figure 3.1. In this sketch shown in figure 3.1, a saboteur module contains the array of signatures; it is used to inject fault at the outputs of the healthy circuit by considering signatures. Here, the neural network block is used to improve the signatures such that the faulty behaviour of the healthy circuit is made as similar to output of the fault injected one. As illustrated, the output of the saboteur is compared to the output of the fault injected circuit, and the result of comparison is sent to the neural network as feedback. By considering differences between the mentioned outputs, the neural network improves the signature as it is connected to the input of circuit. As was mentioned in the previous section, signatures obtained by (ROBACHE 2013) don't see the input of the circuit; however, in the proposed sketch shown in figure 3.1 improvement in signatures is made possible by allowing the input of the circuit to be modified. In fact, by adding more characteristics for generating signatures, further improvement could be possible.

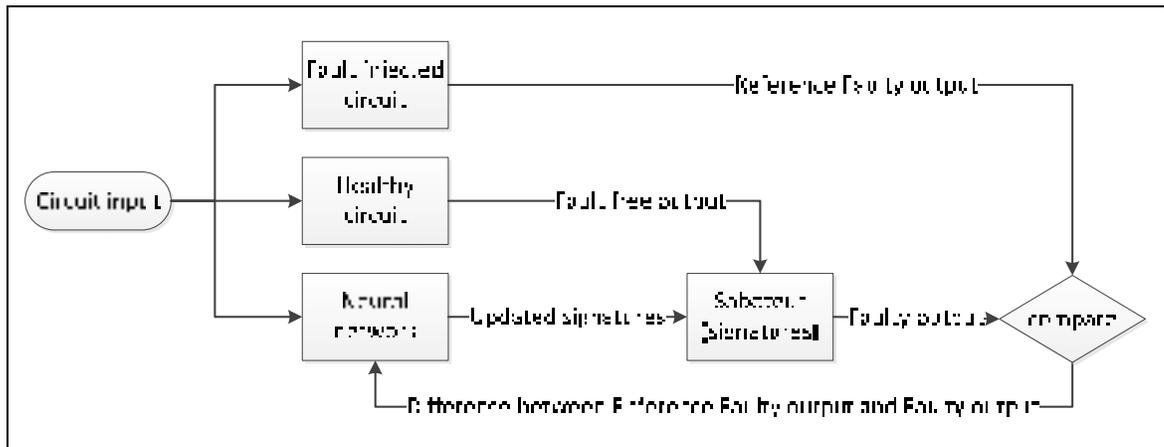


Figure 3.1 Proposed sketch for scenario 1 to adjust signatures

In figure 3.1, when the input is active, it is sent to all three connected blocks: fault injected circuit, healthy circuit and neural network circuit. The signatures already contained in the saboteur are obtained by (ROBACHE 2013) and they are generalized for all inputs. Saboteur is used to generate faulty output from fault free output which is produced by healthy circuit. By sending the difference between outputs of the fault injected circuit (reference faulty output) and the saboteur (faulty output) while the input is being specified, the neural network block will try to improve the signatures and produce updated signatures.

However, in the proposed sketch, the injected fault in the fault injected circuit is not specified; the output of the fault injected circuit and the saboteur is random since a random fault could be injected to them. The output of the fault injected circuit and the saboteur is random, and hence could not be compared since it is not possible to compare two random occurrences. In short, the proposed sketch is rejected and in the next section, an alternative scenario is explored.

### 3.1.3 Scenario 2: faulty output prediction

In this scenario, a neural network is used to predict the faulty output of the circuit. In figure 3.2, the diagram of the proposed approach for predicting the faulty output of the circuit is presented. As illustrated, random input vectors are generated and transmitted to the inputs of a properly trained neural network and to the primary inputs of the combinational circuit.

Also, a random stuck-at 0/1 fault is generated and injected in the combinational circuit. This fault may propagate or not to the primary outputs of the combinational circuit, thus affecting the result. The goal here is to have a neural network able to predict this result observed at the primary outputs of the circuit in the presence of faults. This way, the neural network model mimics the behaviour of the fault injected circuit. For developing the neural network model to mimic such behaviour, appropriate training is necessary. In the next section, different phases to develop a neural network model with the ability of predicting the behaviour of the fault injected circuit is explained in details.

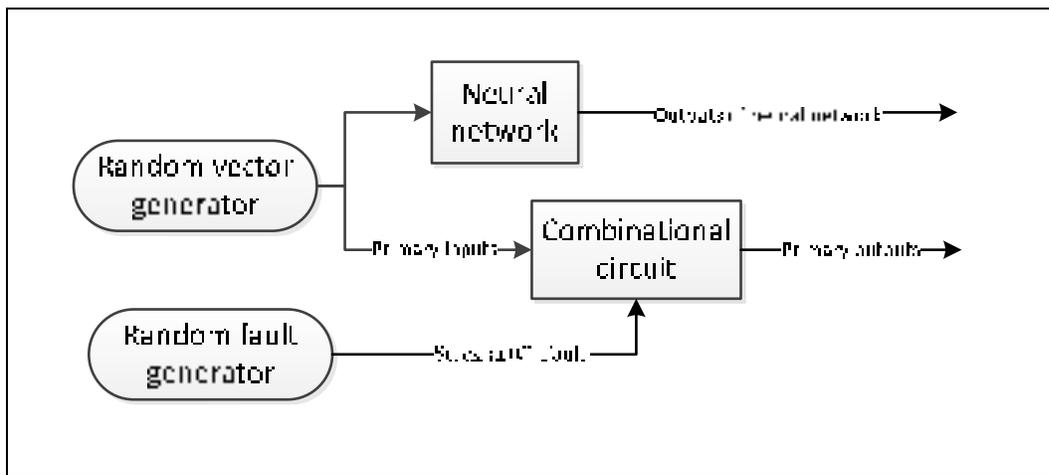


Figure 3.2 Diagram of the proposed approach to mimic the behaviour of a combinational circuit in the presence of faults

### 3.2 Proposal to use neural network for predicting faulty output of circuits

In figure 3.3, the flow diagram for the proposed approach for the second scenario is illustrated. In the first step, the circuit is simulated with the fault simulator for all stuck-at faults on all circuit nodes and for all input vectors. Also, a golden simulation run is done to generate the fault free outputs of the circuit. The simulator generates fault injection reports, and in the next step all those reports are read. In the second step, an array which contains the set of all faulty outputs is defined. Each row in the array contains the input of circuit, the type of injected fault and the corresponding faulty output. Subsequently, rows of the array are randomized. The reason for randomization is to enable random access to fields of the array

while the array is read sequentially. In the fourth step, the array is split into two datasets: the training dataset and the testing dataset. Later, these datasets are used to train and test the neural network. Finally, in the 7th step, the trained neural network is evaluated to see if it works properly or not.

The following sections present in details the dataset generation process, the learning phase and the validation phase.

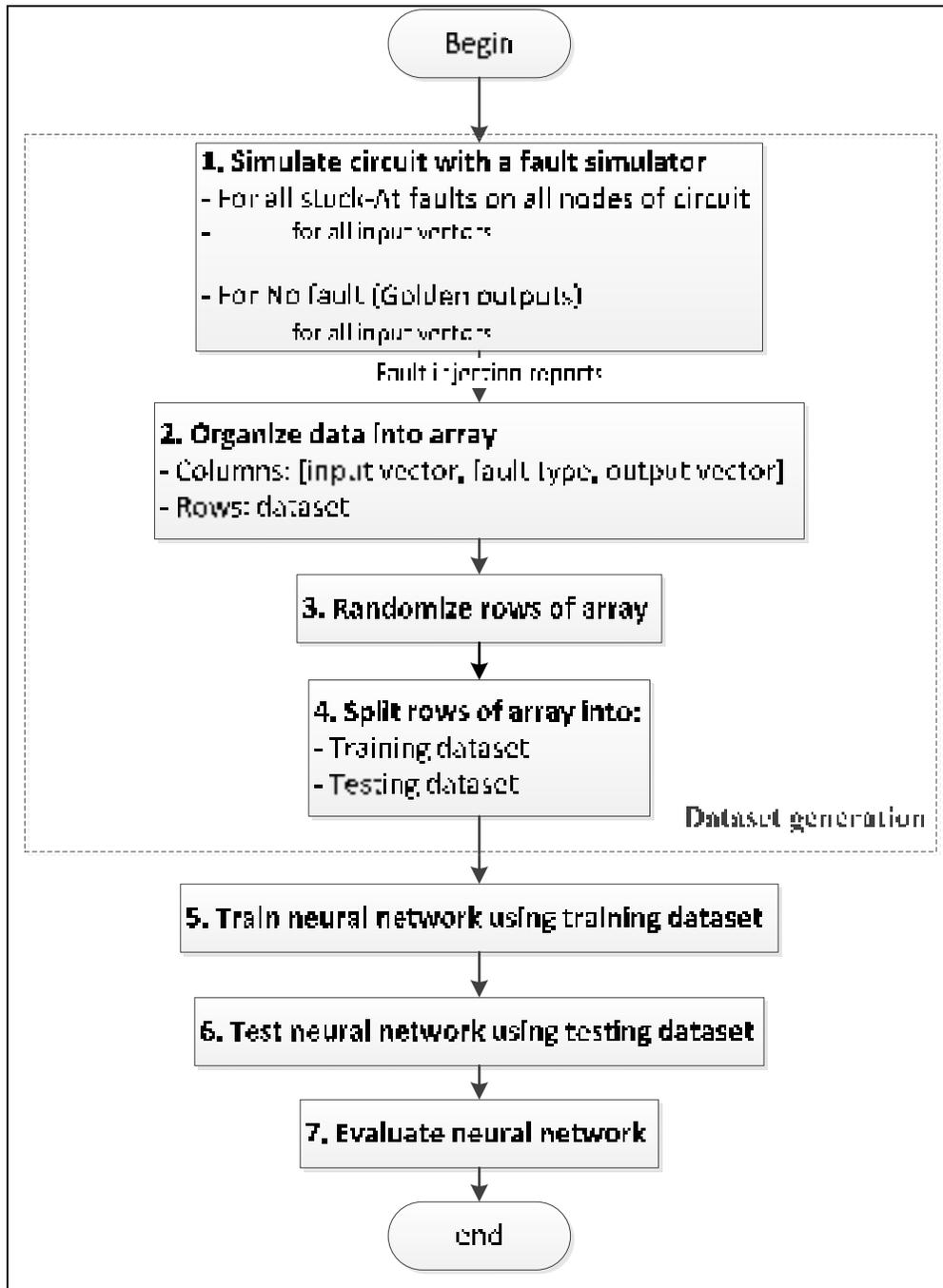


Figure 3.3 The flow diagram to develop a Neural Network model which is able to predict faulty output of circuit

### 3.2.1 Dataset generation

Figure 3.4 shows the steps to generate the training and the testing datasets. All input vectors and all stuck-at faults are sent to the fault simulator. Circuit netlist in Verilog language is also sent to the fault simulator. The golden outputs report and the faulty outputs report are generated by the fault simulator. Faulty outputs report is used by dataset generator block to produce the training and testing dataset. In fact 70% of the report fields are used to generate the training dataset and 30% of the report fields are used to generate the test dataset. The format of the training and testing dataset matrix is presented in the following:

Columns: [input vector, fault type, output vector]

Rows: dataset

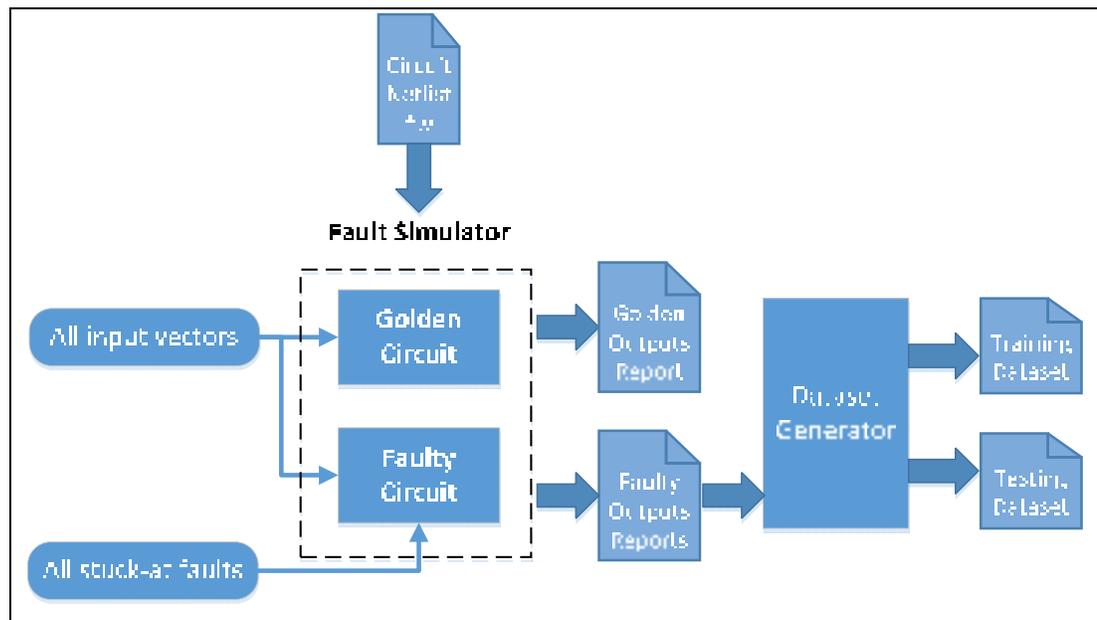


Figure 3.4 The flow diagram to show the steps to generate the training dataset and the testing dataset

The golden outputs report is used in validation phase and it will be explained in the neural network validation section.

### 3.2.2 Neural network training phase

The training phase of the neural network is shown in figure 3.5. In training phase, the input vectors and the fault type parts of the training dataset should be sent to the neural network. The output vector part of the training dataset is used as a reference output to applied supervised training. In supervised training expected output is provided. In this way, after each update on the weight of the links through the neural network, estimated output by neural network is compared with the expected output which is exist in training dataset output vector part. The comparison results are sent back to the neural network as a feedback error. By considering this error neural network could be updated its links' weight to reduce feedback error in next the iteration.

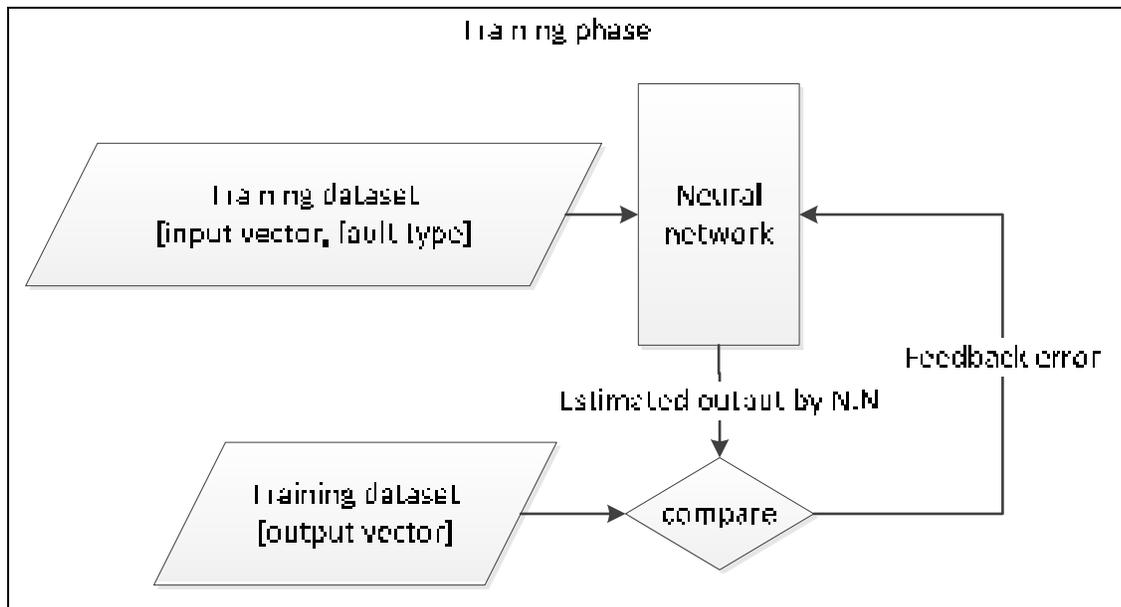


Figure 3.5 Flow diagram to show the training phase of the neural network

In the next section, the way in which the trained neural network could be validated is discussed.

### 3.2.3 Neural network validation

First of all, signature generator block in figure 3.6 will be explained. Signature generator is used to compute error type probability matrix  $P$  (3.2):

$$P = \begin{bmatrix} p_{0,-3} & \cdots & p_{0,3} \\ \vdots & \ddots & \vdots \\ p_{31,-3} & \cdots & p_{31,3} \end{bmatrix} p_{i,j} \text{ while } i = \text{input of circuit, } j = \text{error type} \quad (3.2)$$

Here,  $p_{i,j}$  is probability of having error type  $j$  while the input of the circuit is  $i$ .

Formula (3.3) is used to compute the error type:

$$\text{Error type} = \text{GoldenOutput} - \text{FaultyOutput} \quad (3.3)$$

In figure 3.6, the proposed diagram for validating the neural network is illustrated. Signature generator 1 is used to generate reference error type probability matrix  $P^R$  from simulator reports (testing dataset [outputs vector]). For computing  $P^R$  faulty output in formula (3.3) is testing dataset [outputs vector]). Signature generator 2 is used to generate faulty error type probability matrix  $P^F$  neural network output (estimated output by N.N). For computing  $P^F$  faulty output in formula (3.3) is estimated output by N.N. The trained neural network could produce estimated outputs from the testing dataset. By comparing  $P^R$  and  $P^F$ , accuracy of the neural network for producing faulty outputs of circuit is examined. To measure the accuracy with which the neural network works, the probability of different error types from the neural network and reports from the simulator should be almost similar. By comparing these two probability matrixes, the reliability of the neural network is estimated.

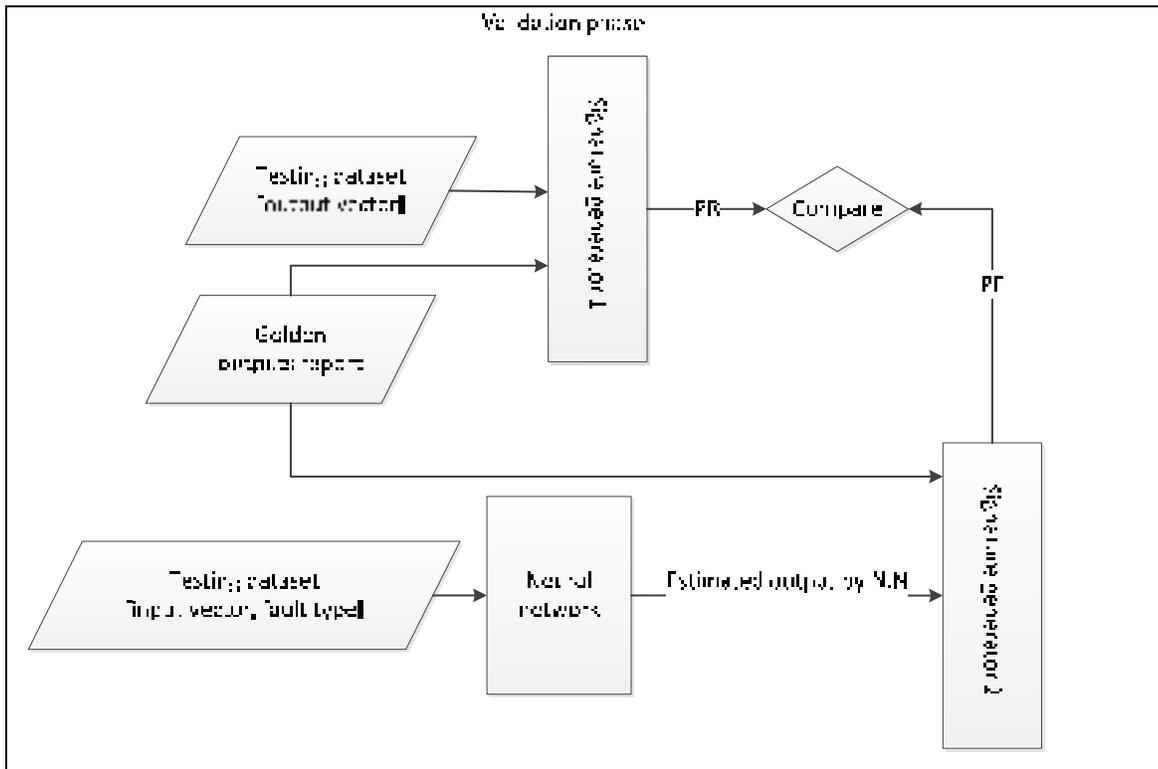


Figure 3.6 Flow diagram for validation phase of neural network

In the discussion till now, the trained neural network is validated and is ready to be used. The following section describes how this trained neural network could be used.

### 3.2.4 Using the trained neural network

In figure 3.7, the diagram shows how the trained neural network is used in other applications. As shown, by sending the input of the circuit to the neural network block, faulty outputs are produced. In the next chapter, the steps to develop the neural network model are explained clearly with the example of a case study.

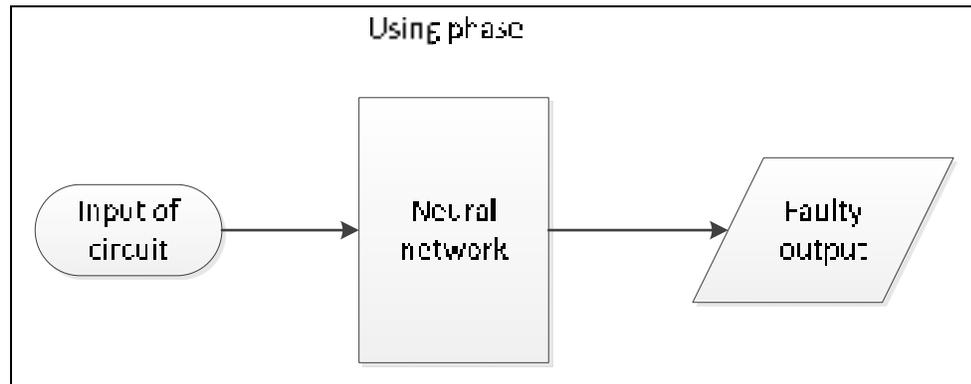


Figure 3.7 Flow diagram for using phase of neural network

### 3.3 Conclusion

During the course of the work presented in this thesis, different paths are explored to find a useful scenario for developing a neural network based model. In the next sections, all such efforts are explained, with the one finally accepted described in detail. Signatures refer to compressed information about fault behavior of the circuit, and probabilities of all error types are expressed by the signature. These signatures are generated by my teammate and they are explained in his thesis with detail (ROBACHE 2013). The first scenario explored is about signature adjustment and which describes the way to improve the accuracy of signatures by considering the input of circuits. In fact, my teammate generalized signatures for all possible inputs in the work presented in his thesis. However, the first scenario is rejected since it presents the possibility of two random faulty outputs being compared, whereas it is not possible to compare two random occurrences. In the second scenario, the goal is to generate faulty output prediction. Neural network approach is used to predict the faulty output of the circuit. Training, validation and the way of using the developed neural network is explained in details. The following chapter enables a clearer understanding of all the steps for developing the neural network model by describing the development of the neural network for circuit C17.



## CHAPITRE 4

### IMPLEMENTATION OF PROPOSED MODEL

In this chapter, implementation of the proposed flow diagram of figure 3.3 (the flow diagram to develop a Neural Network model which is able to predict faulty output of circuit) shown in the previous chapter is presented. Initially, programming is done in MATLAB<sup>TM</sup>; however, C++ is later used to overcome the limitations of the MATLAB<sup>TM</sup> environment. Simulation of circuits by a simulator is necessary to generate the fault reports required for modeling the behavior of the circuits. To explain all the steps shown in figure 3.3 the combinational circuit C17 is used as a case study and the neural network model of circuit C17 is developed step by step. Note that C17 is a very simple circuit with a few fault reports. To prove the methodology presented in this thesis, circuits with bigger dataset should be analyzed such as a multiplier component, which is chosen.

The first section presents the steps to generate the dataset used for the neural network training. Then, the procedure to develop the neural network in MATLAB<sup>TM</sup> and C++ is explained. In the following sections, combinational circuit C17 is used as a simple example to validate the proposed methodology of this thesis. All the steps to develop and validate the neural network model corresponding to the faulty behaviour of circuit C17 are explained and analysed in detail. Finally, the analysis of a 4-bit multiplier neural network model is presented as an example of a bigger data set case study.

#### 4.1 Dataset generation

A neural network needs a relevant dataset for proper training. For this purpose, a simulator that is able to perform fault simulations on a circuit netlist description for single and multiple stuck-at faults (Bosio and Di Natale 2008).

The netlist file of the circuit in Verilog language is sent to the LIFTING simulator, and it generates golden output and reference faulty output. As mentioned previously, the golden

output is the output of the fault free circuit. The reference faulty output contains the set of output files, generated by the simulator, in the presence of faults on all nodes for all inputs. This implies that the number of files in the reference faulty output is equal to the number of fault sites in the circuit. Fault sites are explained next. In figure 4.1 all fault sites for circuit C17 are illustrated by numbers. As shown, there is a fault site on each branch of the circuit and on each input of gates (Bushnell and Agrawal 2000), for a total of 17 fault sites for the circuit C17 shown in figure 4.1. At each fault site, stuck-at zero and stuck-at one can be injected; thereby with 17 fault sites, the number of different faults is  $17 \times 2 = 34$ .

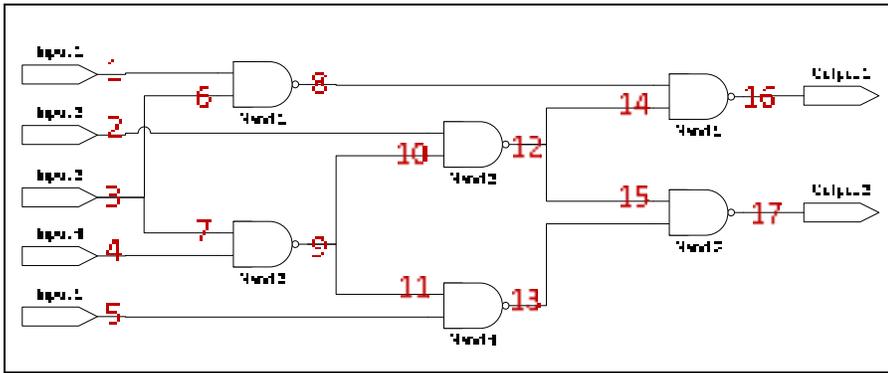


Figure 4.1 Fault sites specified for circuit C17

In the reference faulty output file for circuit C17, there are 17 files and each file contains the output of the circuit in the presence of specific faults for all inputs. To sum up, LIFTING generates two outputs: golden output and reference faulty output, which are used for training the neural network using the steps enumerated in the following sections.

## 4.2 MLP Neural network in MATLAB<sup>TM</sup> environment

For generating a multi-layer perceptron neural network in the MATLAB<sup>TM</sup> environment, the command *newff* is used:

$$net = newff(PR, [S1 S2 \dots SNL], \{TF1 TF2 TFNL\}, BTF) \quad (4.1)$$

The command *newff* as the following parameters (4.1):

- *PR* is the  $R \times 2$  matrix of min and max amounts for  $R$  input elements;
- *Si* is the size of  $i^{\text{th}}$  layer, given that there are  $Nl$  layers;
- *TFi* is the type of transfer function of  $i^{\text{th}}$  layer. The default transfer function used is '*tansig*';
- *BTF* is the type of training function. The default training function is '*traingdx*';
- *net* refers to the neural network that is initialized by the MATLAB<sup>TM</sup> command *newff*.

For training the neural network, the MATLAB<sup>TM</sup> command *train* is used:

$$net1 = train(net, x, y) \quad (4.2)$$

The command *train* as the following parameters (4.2):

- *net* is the initialized MLP neural network generated by command *newff*;
- *x* is the measured input vector;
- *y* is the measured output vector.

The trained MLP neural network is tested by the *sim* command. The output of the trained MLP neural network *net1* is simulated by the *sim* command:

$$ytest = sim(net1, x) \quad (4.3)$$

The command *sim* as the following parameters (4.3):

- *net1* is the trained neural network;
- *x* is the input vector.

For the validation of the trained neural network, the measured output *y* is compared to the result from the *ytest* command; the *ytest* command gives the output of the trained neural network. In (4.4) below, *e* is the error and is a measure of how good the trained neural network is (Koivo 2006).

$$e = y - y_{test} \quad (4.4)$$

However, there are some limitations for implementing neural networks in MATLAB™. Speed and memory shortage are the significant limitations for the MATLAB™ environment. For example when C17 dataset (3.18 KB) and 4 bit multiplier dataset (724 KB) are loaded there is no memory shortage but when 5 bit multiplier dataset (43.8 MB) is loaded, memory shortage error is happened.

In the following section, implementation of neural network for circuit C17 in C++ is explained. Note that running the program in the C++ environment offers more speed performances and memory capacity compared to the MATLAB™ environment (Mirzadeh, Mehri et al. 2010) because the model is compiled in native machine code. In (Mirzadeh, Mehri et al. 2010), the simulation result shows implementation of a loop in MATLAB takes two times more than the same loop in C++. In spite of C functions, the OpenCV2 functions take less time, so OpenCV that is an open source computer vision library written in C and C++ (and it is an active development on interfaces for MATLAB) would be useful for getting more speed. After implementation the "for" loops of the algorithm using OpenCV library, the simulations show the run time decreases a lot.

### **4.3 MLP Neural network and OpenCV library written in C++ language**

OpenCV (Open Source Computer Vision) is an open source computer vision library that provides a common platform for communication between different computers based applications, thereby accelerating the use of machine vision for developing commercial products. OpenCV is a BSD-licensed product and therefore allows academic and commercial projects to use and modify it for their own applications.

OpenCV was started by Intel in 1999 and contains more than 2500 optimized machine vision and computer vision algorithms. OpenCV provides a programming interface to C, C++ and Java, besides supporting Windows, Linux, Android and Mac OS operating systems. Written

in C++, OpenCV focuses on real-time image processing and the latest computer vision algorithms (Wagner 2012).

MLP (Multi-Layer Perceptron) neural network is implemented in OpenCV as an instance of CvANN\_MLP (Wagner 2012).

### 4.3.1 Parameters configuration

The following parameters are used to determine the performance of the MLP neural network. For each parameter, an arbitrary value is assigned here only as an example.

```
CvTermCriteria criteria ;
criteria .max_iter = 100;
criteria . epsilon = 0.00001 f;
criteria . type = CV_TERMCRIT_ITER | CV_TERMCRIT_EPS;
CvANN_MLP_TrainParamsparams ;
params .train_method = CvANN_MLP_TrainParams :: BACKPROP;
params .bp_dw_scale = 0.05 f;
params .bp_moment_scale = 0.05 f;
params .term_crit = criteria ;
```

*term\_crit* (termination criteria) identifies the number of iterations, and the change of associated weights between any two iterations for the training algorithm.

*train\_method* defines the training algorithm. It can be CvANN\_MLP\_TrainParams::BACKPROP (sequential back propagation algorithm) or CvANN\_MLP\_TrainParams::RPROP (RPROP algorithm). In this thesis sequential back propagation algorithm is used because this is used in MATLAB<sup>TM</sup> version. To compare the results of programming in C++ and programming in MATLAB<sup>TM</sup> same algorithm should be used.

*bp\_dw\_scale* is only for Back propagation training algorithm. The computed weight gradient is multiplied by this coefficient. The value of this parameter is determined after several trials. It could be started from 0.05 f and increased little by little till the trained neural network in the evaluation step gives us least error.

*bp\_moment\_scale* is also only for Back propagation training algorithm. The difference between weights for the two previous iterations is multiplied by this coefficient. The amount of this parameter is determined after several trials. It could be started from 0.05 f and increase little by little till the trained neural network in the evaluation step gives us least error.

### 4.3.2 Layers definition

The purpose of neural networks is generalization which refers to the ability of the neural network to predict the outputs values for inputs vectors which are not used during training (Sarle 2002). With networks having few neurons, approximating the function is a problem while networks with too many neurons result in an over-fit, thereby producing random errors due to the inability to find the correct relationship (Heaton 2008). The number of neurons depends on the application and it could be determined by trials and errors. A starting point could be chosen between number of neurons in input and output layers (Heaton 2008). After setting different number of neurons in hidden layer, the neural network is trained and the performance of that is evaluated. When the trained neural network gives us least error in the evaluation step, its number of neuron is the appropriate amount.

A row-ordered *cv :: Mat* stores the number of neurons per layer. In the following example, *layers* is a matrix which size is  $4 \times 1$ : row (0) in *layers* which is shown first layer of neural network has 2 neurons, second layer (row (1) ) has 10 neurons, third layer (row (2) ) has 15 neurons and the last layer (row (3) ) has 1 neuron. Last command is used to create MLP neural network from *layers* which was defined before.

```
cv :: Mat layers = cv :: Mat (4 , 1 , CV_32SC1 );
```

```

layers . row (0) = cv :: Scalar (2) ;
layers . row (1) = cv :: Scalar (10) ;
layers . row (2) = cv :: Scalar (15) ;
layers . row (3) = cv :: Scalar (1) ;
mlp . create ( layers ) ;

```

### 4.3.3 Network training

The Application Programming Interface (API) for training the MLP needs training data, training classes, which shows amount of different classes for data separation, as well as a structure for the parameters. For more details appendix I should be checked.

```

mlp . train ( trainingData , trainingClasses , cv :: Mat () , cv :: Mat () , params ) ;

```

#### **Prediction:**

Activation of the output layer is stored in `cv :: Mat` response. This enables multiple neurons in the output layer.

```

mlp . predict ( sample , response ) ;
float result = response . at <float >(0 ,0) ;

```

In the following section, circuit C17 is used as an example for exploring all the steps mentioned previously. The results are analyzed and the proposed methodology is hence certified.

## 4.4 Case study for circuit C17

In this section, the proposed methodology is explained in greater detail keeping in context the circuit C17.

A fault truth table is developed to include all possible outputs of the circuit in presence of the different faults (Al-Jumah and Arslan 1998). In table 4.1, part of the fault truth table for the circuit C17 is shown. As can be seen, there are  $(34 + 1) \times 32 = 1120$  fields in the table; 34 is number of different faults that could be injected, 32 is the number of different inputs for

the circuit C17, and the extra 1 which is added to 34 is for fault free outputs (which is for showing the output of circuit when there are no injected fault).

Table 4.1 Fault truth table for circuit C17. First 17 Injected fault fields show stuck-at-0 for all 17 different fault sites and second 17 injected fault fields show stuck-at-1 for all 17 different fault sites.

Field	Input of circuit						Injected fault	Output of circuit	
1	0	0	0	0	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
32	1	1	1	1	1	1	0	0	0
33	0	0	0	0	0	0	1	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
64	1	1	1	1	1	1	1	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1086	0	0	0	0	0	0	34	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1120	1	1	1	1	1	1	34	1	1

The neural network based model for circuit C17 developed according to the proposed methodology is now explained with the help of figure 4.3. This neural network has 3 layers: the input, the hidden and the output layers. There are 11 nodes in the input layer. The first five nodes are for representing inputs of the circuit and the next six nodes are for representing faults number. For circuit C17 there are 34 faults; value 34 in decimal format needs 6 bits in binary format because  $5 < \log_2 34 < 6$ . For determining number of neuron in the hidden layer the trained neural network is evaluated. When the trained neural network in the evaluation step gives us least error, its number of neurons in its hidden layer is the appropriate amount. There are 200 nodes in the hidden layer which are determined during programing after several trials; by choosing different number of neurons from 30 to 550, I have found by choosing the amount of neurons between 150 to 220 the best validation

performance will be achieved. There are 2 nodes in the output layer which are the number of outputs for circuit C17.

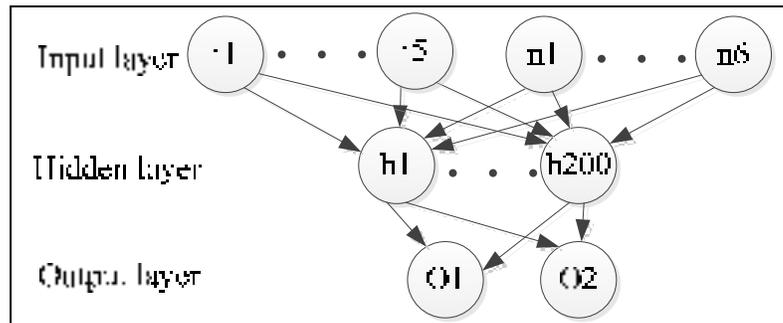


Figure 4.2 Neural network based model for circuit C17

For training the proposed neural network, data from the fault truth table is used. The number of fields in the fault truth table of C17 is 1120 and though it is possible to use the entire available data to train the neural network, the method of sampling is proposed in the following section for training circuits with huge fault truth tables. This training method allows the proposed neural network model to be more easily trained for complex circuits with huge amount of data in their fault truth tables.

#### 4.5 Data sampling method for training neural network

Sampling is a way to decrease the size of the data set. By applying data sampling on a fault truth table, the amount of data could be decreased thereby enabling less data to be selected as a representative of the huge data set. Sampling could be either random or weighted. The validation of random sampling method for training of the fault truth table of circuit C17 is now provided.

**Step 1:** The fault truth table for circuit C17 is made. Note that for training the neural network, the part of the table which represents fault free output is not used. Therefore there are 1088 fields in the fault truth table used for training the neural network.

**Step 2:** The fault truth table is organised randomly which ensures that the data is not in any particular pre-determined order. By going through the table sequentially, random fields are read.

**Step 3:** 10% to 80% of fields from the fault truth table are read. Note that reading always starts from the beginning of the truth table.

**Step 4:** Step 2 and Step 3 are repeated for the 2nd, 3rd, ..., 5th trials. This ensures the availability of several trials for the experiment.

Table 4.2 lists the errors of evaluation encountered when applying the sampling method for evaluating the neural network based model for circuit C17. Three different categories of errors are defined and they will be explained in details in the next section:

***Error<sub>category1</sub>***: this category of errors is the mean of the cumulative frequencies of the error types that appear in the high-level of abstraction but do not appear in the low-level of abstraction for specific inputs;

***Error<sub>category2</sub>***: this category of errors is the mean of the cumulative frequencies of the error types that appear in the low-level of abstraction but do not appear in the high-level of abstraction for specific inputs;

***Error<sub>category3</sub>***: this category of errors is the mean of the cumulative frequencies of the difference between the error types that appear in both the low-level and the high-level of abstraction for specific inputs. In fact, it is the mean absolute error.

Table 4.2 Errors of evaluation of the neural network model for C17 while using different percentages of sampling of the 1088 number of data

	Percentage of data	10%	20%	40%	50%	60%	70%	80%
1 <sup>st</sup> try	Error <sub>category1</sub>	<b>10.29</b>	<b>6.37</b>	4.41	7.35	2.94	2.94	<b>0</b>
	Error <sub>category2</sub>	8.28	5.63	3.92	2.94	4.90	2.94	<b>0</b>
	Error <sub>category3</sub>	<b>12.25</b>	<b>8.14</b>	5.01	4.30	4.81	2.74	<b>2.08</b>
	Mean <sub>correlation coefficient</sub>	<b>0.89</b>	<b>0.94</b>	0.97	0.98	0.98	0.99	<b>0.99</b>
2 <sup>nd</sup> try	Error <sub>category1</sub>	13.36	13.12	8.82	8.82	3.52	2.94	0
	Error <sub>category2</sub>	18.62	4.11	3.78	5.88	2.94	2.94	0
	Error <sub>category3</sub>	19.05	9.75	5.91	4.58	4.06	3.26	2.12
	Mean <sub>correlation coefficient</sub>	0.79	0.94	0.97	0.98	0.99	0.99	0.99
3 <sup>rd</sup> try	Error <sub>category1</sub>	13.08	7.05	4.90	4.11	2.94	0	0
	Error <sub>category2</sub>	8.08	5.58	3.78	4.70	<b>2.94</b>	2.94	2.94
	Error <sub>category3</sub>	20.98	10.56	5.71	4.53	3.58	2.59	2.64
	Mean <sub>correlation coefficient</sub>	0.77	0.91	0.97	0.98	0.99	0.99	0.99
4 <sup>th</sup> try	Error <sub>category1</sub>	10.41	10.39	4.90	5.88	3.52	<b>0</b>	0
	Error <sub>category2</sub>	<b>7.64</b>	<b>3.52</b>	2.94	3.92	2.94	<b>2.94</b>	0
	Error <sub>category3</sub>	16.05	10.60	4.99	5.49	3.78	<b>2.53</b>	2.35
	Mean <sub>correlation coefficient</sub>	0.82	0.93	0.98	0.98	0.99	<b>0.99</b>	0.99
5 <sup>th</sup> try	Error <sub>category1</sub>	13.49	6.76	<b>4.41</b>	<b>3.52</b>	<b>0</b>	4.41	2.94
	Error <sub>category2</sub>	16.17	5.88	<b>2.94</b>	<b>2.94</b>	3.52	2.94	0
	Error <sub>category3</sub>	18.21	12.19	<b>4.62</b>	<b>3.31</b>	<b>3.17</b>	3.60	2.55
	Mean <sub>correlation coefficient</sub>	0.84	0.89	<b>0.98</b>	<b>0.99</b>	<b>0.99</b>	0.99	0.99
Best try	Error <sub>category1</sub>	10.29	6.37	4.41	3.52	0	0	0
	Error <sub>category2</sub>	7.64	3.52	2.94	2.94	2.94	2.94	0
	Error <sub>category3</sub>	12.25	8.14	4.62	3.31	3.17	2.53	2.08
	Mean <sub>correlation coefficient</sub>	0.89	0.94	0.98	0.99	0.99	0.99	0.99

In table 4.2, results from the execution of all the listed steps are presented, and the same sequence of steps has been executed 5 times. In the 6<sup>th</sup> row, the best results are presented. Now, the sample size is decreased further to less than 10%. Table 4.3 lists the results of different categories of error when using sample sizes less than 10%.

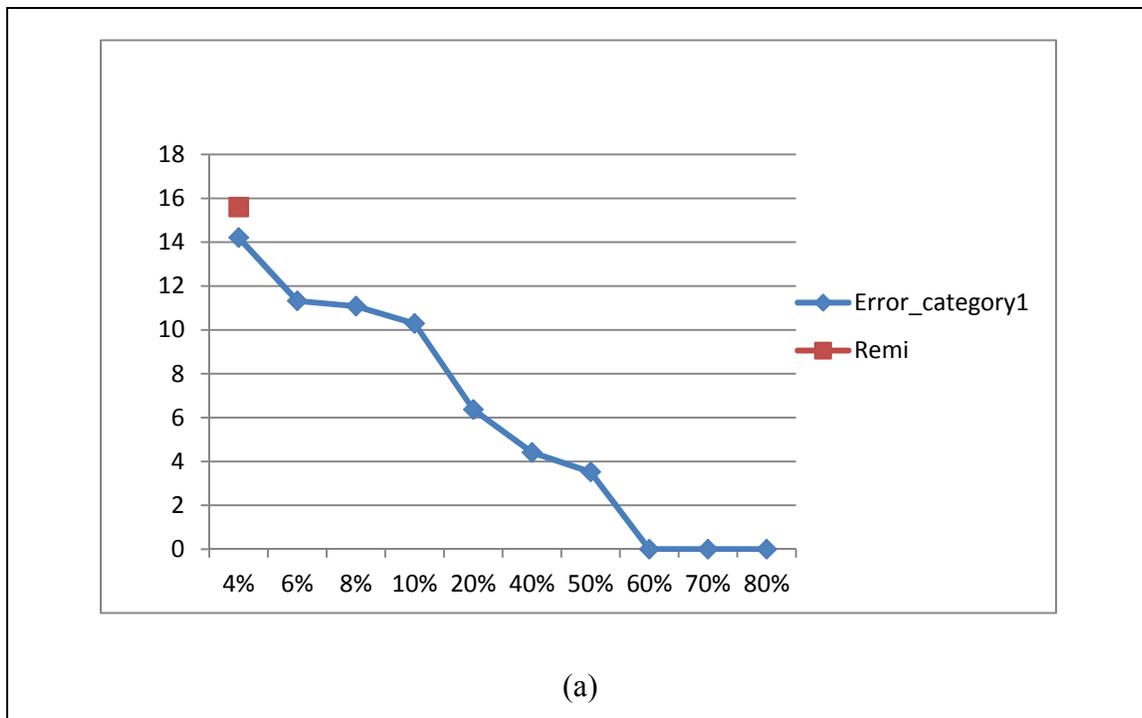
Table 4.3 Errors of evaluation of the neural network based model for circuit C17 with sample sizes less than 10% of the 1088 number of data

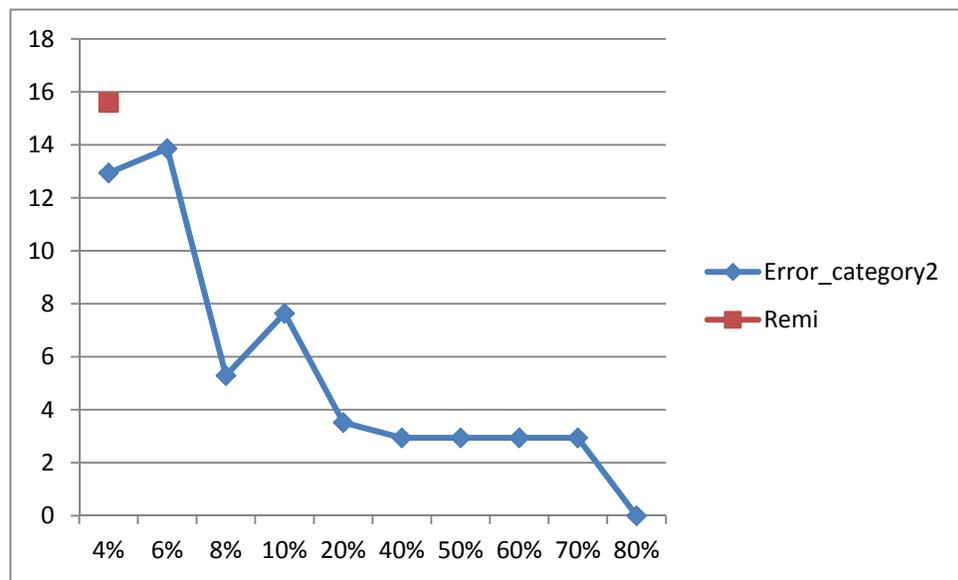
	Percentage of data	4%	6%	8%	10%
1 <sup>st</sup> try	Error <sub>category1</sub>	<b>14.21</b>	18.55	14.21	<b>10.29</b>
	Error <sub>category2</sub>	<b>12.94</b>	18.71	13.86	8.28
	Error <sub>category3</sub>	19.69	<b>12.65</b>	17.26	<b>12.25</b>
	Mean <sub>correlation coefficient</sub>	0.67	<b>0.79</b>	0.77	<b>0.89</b>
2 <sup>nd</sup> try	Error <sub>category1</sub>	20.49	13.36	11.76	13.36
	Error <sub>category2</sub>	19.70	16.17	8.08	18.62
	Error <sub>category3</sub>	<b>17.63</b>	21.81	<b>14.72</b>	19.05
	Mean <sub>correlation coefficient</sub>	<b>0.70</b>	0.65	<b>0.89</b>	0.79
3 <sup>rd</sup> try	Error <sub>category1</sub>	16.54	<b>11.32</b>	<b>11.08</b>	13.08
	Error <sub>category2</sub>	34.92	<b>13.86</b>	18.01	8.08
	Error <sub>category3</sub>	21.44	19.80	16.51	20.98
	Mean <sub>correlation coefficient</sub>	0.62	0.69	0.78	0.77
4 <sup>th</sup> try	Error <sub>category1</sub>	18.96	19.51	12.20	10.41
	Error <sub>category2</sub>	16.17	20.95	<b>5.29</b>	<b>7.64</b>
	Error <sub>category3</sub>	30.03	28.72	18.42	16.05
	Mean <sub>correlation coefficient</sub>	0.61	0.59	0.76	0.82
5 <sup>th</sup> try	Error <sub>category1</sub>	15.02	20.43	15.78	13.49
	Error <sub>category2</sub>	13.23	22.22	14.70	16.17
	Error <sub>category3</sub>	26.63	23.52	16.95	18.21
	Mean <sub>correlation coefficient</sub>	0.66	0.66	0.75	0.84
Best try	Error <sub>category1</sub>	14.21	11.32	11.08	10.29
	Error <sub>category2</sub>	12.94	13.86	5.29	7.64
	Error <sub>category3</sub>	17.63	12.65	14.72	12.25
	Mean <sub>correlation coefficient</sub>	0.70	0.79	0.89	0.89

Table 4.4 Comparison between the errors of the neural network based model and model proposed by (ROBACHE 2013) for circuit C17

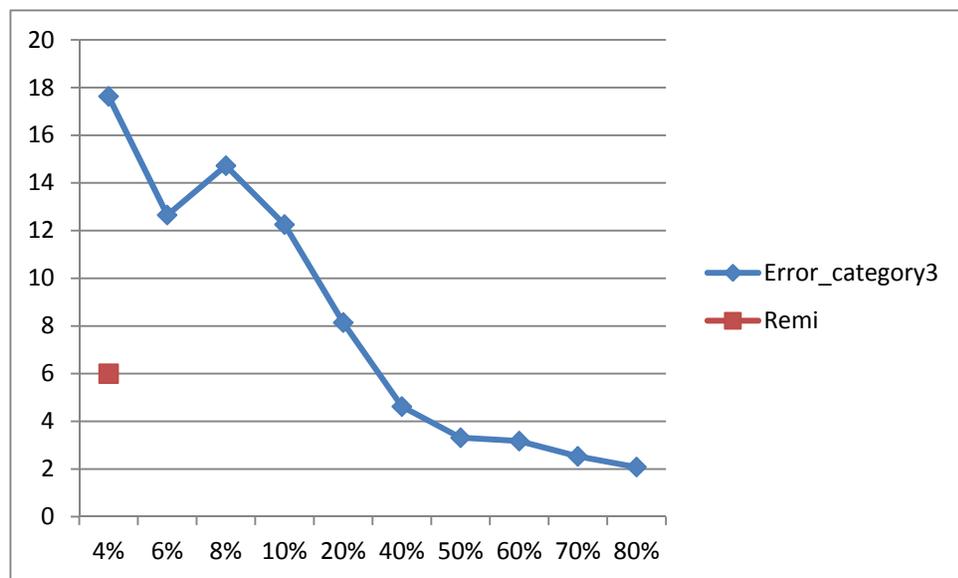
	Percentage of data	10%	20%	40%	50%	60%	70%	80%	Remi
Best try	Error <sub>category1</sub>	10.29	6.37	4.41	3.52	0	0	0	15.6
	Error <sub>category2</sub>	7.64	3.52	2.94	2.94	2.94	2.94	0	8.32
	Error <sub>category3</sub>	12.25	8.14	4.62	3.31	3.17	2.53	2.08	6.00
	Mean <sub>correlation coefficient</sub>	0.89	0.94	0.98	0.99	0.99	0.99	0.99	0.96

Figure 4.4 illustrates the graphs of  $Error_{category1}$ ,  $Error_{category2}$  and  $Error_{category3}$  for the best try. As may be seen, reducing the percentage of sampling increases the number of errors in most of the cases. In all the graphs, the value of errors obtained by (ROBACHE 2013) for the same circuit is shown with a red square.





(b)



(c)

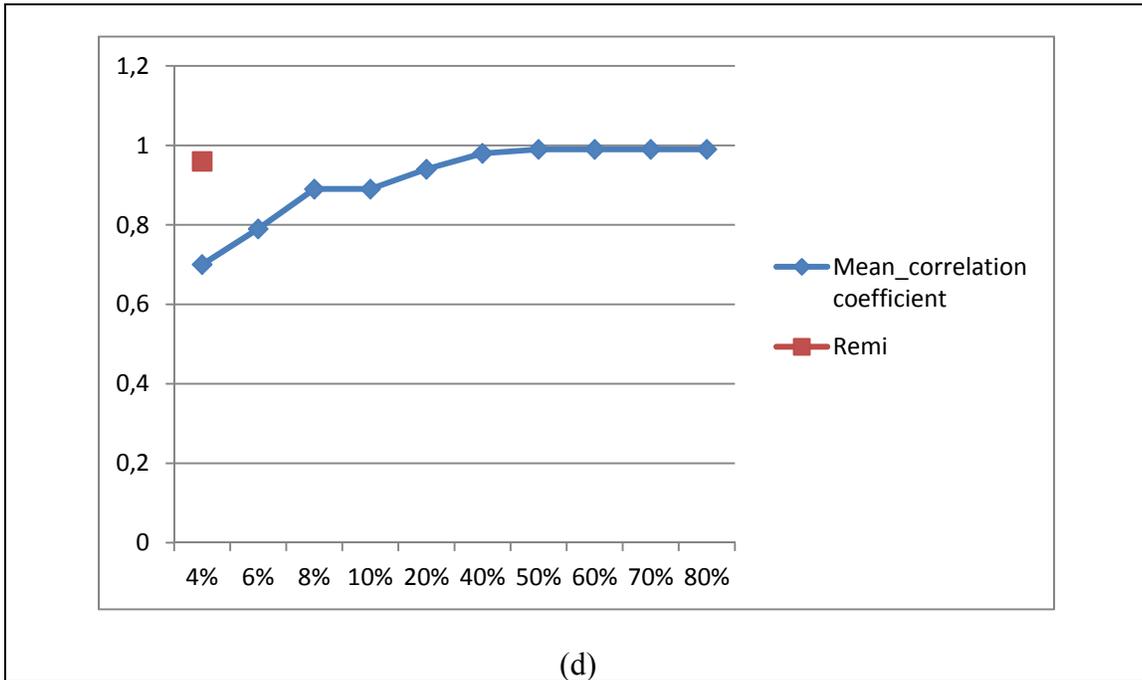


Figure 4.3 Graphs of a) Error\_category1, b) Error\_category2, c) Error\_category3 and d) mean\_correlation coefficient

By taking only 30% of the dataset generated by the software LIFTING, the trained neural network is able to replicate the output of the circuit in presence of faults while keeping the mean absolute error ( $Error_{category1}$ ) below 6% (which is the mean absolute error obtained by (ROBACHE 2013)), and the mean\_correlation coefficient more than 0.96 (which is the mean\_correlation coefficient of Robache's method). Also, by considering only 4% of the dataset for training the neural network,  $Error_{category1}$  and  $Error_{category2}$  from this method are less than that obtained by (ROBACHE 2013). The results presented verify the assumption that this method works better than the method proposed by (ROBACHE 2013). However, more circuits have to be used for a more rigorous validation.

#### 4.5.1 Method of computation of errors

Errors are a manifestation of the presence of faults; however, faults which are masked in the inner scope do not result in errors. To compute error, the output of the fault injected circuit is compared to the output of the healthy circuit (called the golden output as has been mentioned

previously). Table 4.5 shows that the error types for circuit C17 vary from  $-3$  to  $+3$ . For example, when the output of the healthy circuit (golden output) is 00 and the output of the fault injected circuit is also 00, there is no error and the error type is 0. Similarly, when the output of the healthy circuit (golden output) is 00 and the output of the fault injected circuit is 11 and, the error type is  $-3$  (obtained as  $-3 = 00 - 11$ ). Table 4.5 lists all the error types for circuit C17.

Table 4.5 Error types for circuit C17

Golden Output	Faulty Output	Error type
00	11	-3
00	10	-2
00	01	-1
00	00	0
01	00	1
10	00	2
11	00	3

Formula (4.5) is used to compute the error type:

$$\text{Error type} = \text{GoldenOutput} - \text{FaultyOutput} \quad (4.5)$$

Here, Golden Output is the output of the healthy circuit and Faulty Output is the output of the fault injected circuit. To calculate the probability of different error types for specific input, error type probability matrix  $P$  is defined (4.6):

$$P = \begin{bmatrix} p_{0,-3} & \cdots & p_{0,3} \\ \vdots & \ddots & \vdots \\ p_{31,-3} & \cdots & p_{31,3} \end{bmatrix} p_{i,j} \text{ while } i = \text{input of circuit, } j = \text{error type} \quad (4.6)$$

Here,  $p_{i,j}$  is probability of having error type  $j$  while the input of the circuit is  $i$ .

Two error type probability matrices are defined:

- Reference error type probability matrix  $P^R$  (4.7), which is the reference probability matrix and is computed by data from the low-level of abstraction. It is produced by using files from LIFTING which contain outputs of the fault injected circuit, while  $p^R_{i,j}$  is the probability of having error type  $j$  when the input of the circuit is  $i$ ;

$$P^R = \begin{bmatrix} p^R_{0,-3} & \cdots & p^R_{0,3} \\ \vdots & \ddots & \vdots \\ p^R_{31,-3} & \cdots & p^R_{31,3} \end{bmatrix} \quad (4.7)$$

- Faulty error type probability matrix  $P^F$  (4.8), which is called signature. It is computed by using outputs from the neural network based model of circuit C17.  $p^F_{i,j}$  is the probability of having error type  $j$  when the input of the model is  $i$ .

$$P^F = \begin{bmatrix} p^F_{0,-3} & \cdots & p^F_{0,3} \\ \vdots & \ddots & \vdots \\ p^F_{31,-3} & \cdots & p^F_{31,3} \end{bmatrix} \quad (4.8)$$

$Error_{category1}$  (4.9) is computed by using both probability matrices  $P^R$  and  $P^F$ . When the mean of the cumulative frequencies of the error types that appear in the high-level of abstraction ( $p^F_{i,j} > 0$ ) do not appear in the low-level of abstraction ( $P^R_{i,j} = 0$ ) for specific

inputs. In the formula (4.9),  $n_i$  is the number of instances of  $p_{i,j}^F$  which are considered to be added in each row and  $n$  is the amount of  $\frac{1}{n_i} \sum_{j=-3}^{+3} p_{i,j}^F$ .

$$Error_{category1} = \frac{1}{n} \sum_{i=0}^{31} \left\{ \frac{1}{n_i} \sum_{j=-3}^{+3} p_{i,j}^F \text{ when } p_{i,j}^F > 0 \text{ and } p_{i,j}^R = 0 \right\} \quad (4.9)$$

$Error_{category2}$  (4.10) is computed by using both probability matrices  $P^R$  and  $P^F$  again when the mean of the cumulative frequencies of the error types that appear in the low-level of abstraction ( $p_{i,j}^R > 0$ ) do not appear in the high-level of abstraction ( $p_{i,j}^F = 0$ ) for specific inputs. In the formula (4.10),  $n_i$  is the number of instances of  $p_{i,j}^R$  which are considered to be added in each row and  $n$  is the amount of  $\frac{1}{n_i} \sum_{j=-3}^{+3} p_{i,j}^R$ .

$$Error_{category2} = \frac{1}{n} \sum_{i=0}^{31} \left\{ \frac{1}{n_i} \sum_{j=-3}^{+3} p_{i,j}^R \text{ when } p_{i,j}^R > 0 \text{ and } p_{i,j}^F = 0 \right\} \quad (4.10)$$

$Error_{category3}$ , in fact is the mean absolute error (MAE) and is computed by using both probability matrices  $P^R$  and  $P^F$  (4.11). It refers to the value of differences between errors which appear in both matrixes ( $p_{i,j}^R > 0$ ,  $p_{i,j}^F > 0$ ).  $n_i$  is the number of instances of  $|p_{i,j}^F - p_{i,j}^R|$  which are considered to be added in each row and  $n$  is the amount of  $\frac{1}{n_i} \sum_{j=-3}^{+3} |p_{i,j}^F - p_{i,j}^R|$  which are considered to be added.

$$Error_{category3} = \frac{1}{n} \sum_{i=0}^{31} \left\{ \frac{1}{n_i} \sum_{j=-3}^{+3} |p_{i,j}^F - p_{i,j}^R| \text{ when } p_{i,j}^R > 0 \text{ and } p_{i,j}^F > 0 \right\} \quad (4.11)$$

Mean correlation coefficient between the two matrices shows how much the two are related to each other. To find the amount of this relation between  $P^R$  and  $P^F$ , mean correlation coefficient of them are computed using the formula (4.12):

$$\text{corr}(P^F, P^R) = \frac{\text{cov}(P^F, P^R)}{\sigma_{P^F}\sigma_{P^R}} \quad (4.12)$$

Here,  $\text{cov}(P^F, P^R)$  is the covariance between  $P^R$  and  $P^F$ ,  $\sigma_{P^F}$  is the variance of  $P^F$  and  $\sigma_{P^R}$  is the variance of  $P^R$ .

The concept of error computation is clarified further with the help of the following example.  $P^R$  is shown in table 4.6 for circuit C17:

Table 4.6 Matrix of  $P^R$  for circuit C17

<u>Error types</u> <u>input of circuit</u>	-3	-2	-1	0	1	2	3
0	0	11,76	14,70	73,52	0	0	0
1	0	11,76	0	70,58	17,64	0	0
2	0	11,76	14,70	73,52	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
29	0	0	0	73,52	17,64	8,82	0
30	0	0	20,58	61,76	0	17,64	0
31	0	0	20,58	61,76	0	17,64	0

The matrix of  $P^F$  for circuit C17 is computed while percentage of sampling is 70% and it is shown in table 4.7.

Table 4.7 Matrix of  $P^F$  for circuit C17 while percentage of sampling is 70%

<u>Error types</u> <u>input of circuit</u>	-3	-2	-1	0	1	2	3
0	2,94	5,88	14,70	76,47	0	0	0
1	0	11,76	0	70,58	17,64	0	0
2	0	8,82	23,52	67,64	0	0	0

⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
30	0	0	20,58	64,70	0	14,70	0
31	0	0	20,58	64,70	0	14,70	0

Example 4.1 is the MATLAB<sup>TM</sup> script for computing  $Error_{category1}$  is presented. By using this script and the values of  $P^F$  and  $P^R$  from table 4.6 and table 4.7,  $Error_{category1}$  is calculated to be 2.94, and it is specified by yellow color.

#### Example 4.1 Matlab script to compute $Error_{category1}$

```
%compute  $Error_{category1}$ 
% PF is probability matrix for fault injected circuit
% PR is reference error type probability matrix
sizeNum = 0;
temp=zeros(size(PF,2),1);
fori= 1:1:size(PF,2)
for j = 1:1:size(PF,1)
if (PF(j,i) > 0 &&PR(j,i)==0)
temp(i)=PF(j,i)+temp(i);
sizeNum = sizeNum + 1;
end
end
if ( sizeNum> 0 )
temp(i) = temp(i)/sizeNum;%compute average for each row
sizeNum = 0;
end
end
sumTemp = sum(temp);%add all the elements in the matrix
nonZeroAmount = find(sumTemp);
avgTemp = sumTemp / size(nonZeroAmount,1);%compute average of the matrix
```

## 4.6 Neural network model for C17 in C++ environment

As explained before, programming in MATLAB<sup>TM</sup> environment has limitations: speed and memory shortage are the two main concerns for programming in MATLAB<sup>TM</sup>, especially when the code requires a lot of computation. Running the program in C++ environment is faster and also required less memory when compared to MATLAB<sup>TM</sup> environment. In table 4.8, the comparison of speed between running the program written using C++ and MATLAB<sup>TM</sup> for modeling the fault behaviour of circuit C17 by neural network approach is provided.

Table 4.8 Comparison between MATLABTM and C++ environment

		C++ environment	MATLAB environment
C17	Elapsed time	1.18 s	2.32 s
	Memory usage	3 MB	48 MB
4-bit Multi	Elapsed time	-	332.88 s
	Memory usage	-	144 MB

Table 4.8, clearly shows that programming in C++ language is more efficient and also requires less elapsed time for running the program. For circuit C17, which is programmed in both C++ and MATLAB, elapsed time and memory usage are significantly reduced. For the 4-bit multiplier, the elapsed time and memory usage are much greater than the C17. Consequently, it is expected that for more complex circuit MATLAB programming would lead to excessive processing time and memory usage and it is better to model all circuits in C++. The internal database for an 8-bit multiplier was also created and it consumes larger than 1 Giga byte. When that database was read in MATLAB, an out of memory error was received and it was impossible to generate results for that benchmark. Note that the system used for this experiment is equipped with an 8.0 GB RAM memory and a 3.40 GHz CPU.

#### 4.7 Results for a Multiplier circuit

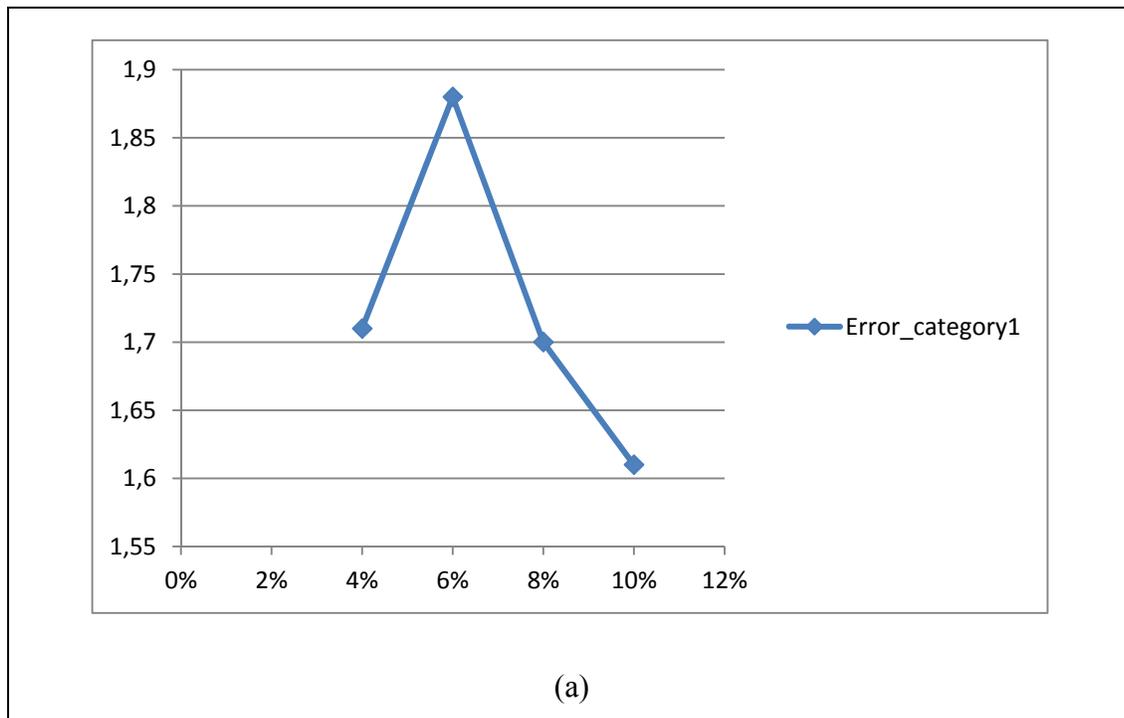
For validating the proposed method with more circuit, 4-bit multiplier will be explored in this section. Table 4.9 lists the errors of evaluation encountered when applying the sampling method for evaluating the neural network based model for circuit 4-bit multiplier.

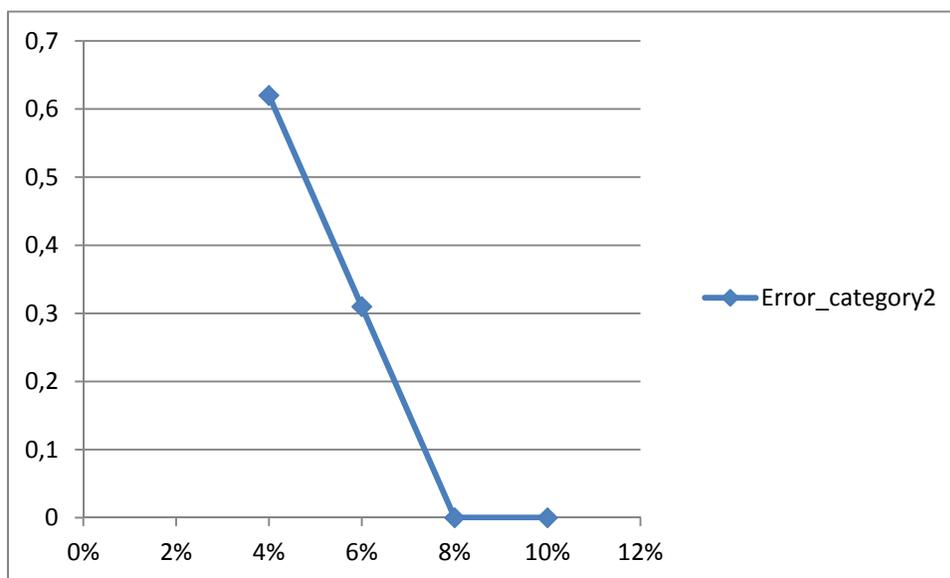
Table 4.9 Errors of evaluation of the neural network model for circuit 4-bit multiplier while using different percentages of sampling of the 82688 vectors

	Percentage of data	4%	6%	8%	10%
1 <sup>st</sup> try	Error <sub>category1</sub>	6.90	1.49	1.70	2.28
	Error <sub>category2</sub>	0.31	0.46	0.0	0.0
	Error <sub>category3</sub>	11.25	8.73	5.87	5.98
	Mean <sub>correlation coefficient</sub>	0.89	0.95	0.98	0.99
2 <sup>nd</sup> try	Error <sub>category1</sub>	6.75	1.57	6.87	6.78
	Error <sub>category2</sub>	0.31	0.31	3.41	0.62
	Error <sub>category3</sub>	12.28	9.84	10.00	10.59
	Mean <sub>correlation coefficient</sub>	0.89	0.94	0.90	0.90
3 <sup>rd</sup> try	Error <sub>category1</sub>	8.84	1.88	6.58	2.13
	Error <sub>category2</sub>	0.0	0.31	0.0	0.0
	Error <sub>category3</sub>	15.19	6.14	11.07	5.68
	Mean <sub>correlation coefficient</sub>	0.87	0.98	0.90	0.99
4 <sup>th</sup> try	Error <sub>category1</sub>	1.70	6.80	6.85	1.61
	Error <sub>category2</sub>	0.0	0.0	0.62	0.0
	Error <sub>category3</sub>	7.47	11.13	10.64	5.13
	Mean <sub>correlation coefficient</sub>	0.97	0.90	0.90	0.99
5 <sup>th</sup> try	Error <sub>category1</sub>	1.71	1.78	4.44	2.12
	Error <sub>category2</sub>	0.62	0.62	0.0	0.0
	Error <sub>category3</sub>	6.97	6.20	10.24	5.71
	Mean <sub>correlation coefficient</sub>	0.98	0.98	0.93	0.99
B	Error <sub>category1</sub>	1.71	1.88	1.70	1.61

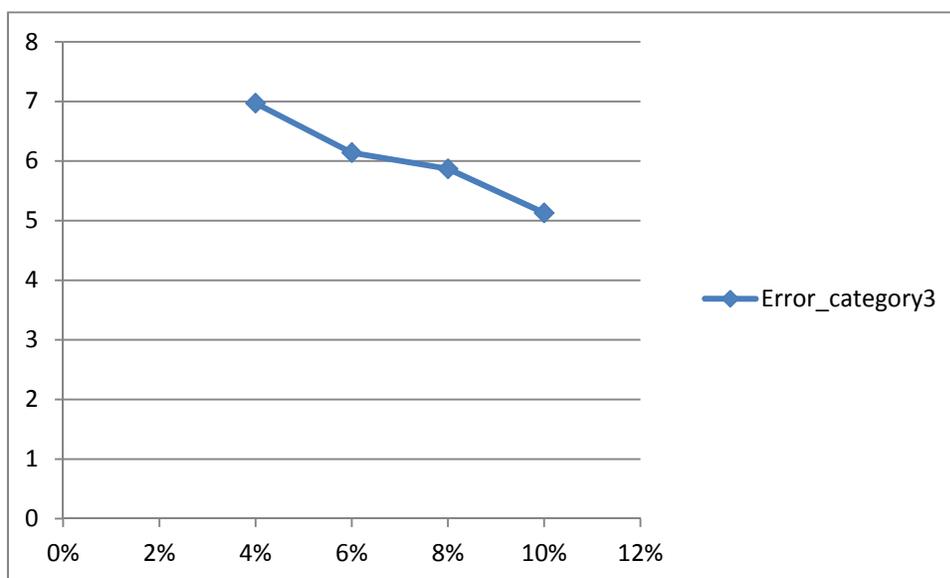
	$Error_{category2}$	0.62	0.31	0.0	0.0
	$Error_{category3}$	6.97	6.14	5.87	5.13
	$Mean_{correlation\ coefficient}$	0.98	0.98	0.98	0.99

In table 4.9, results from the execution of all the listed steps are presented, and the same sequence of steps has been executed 5 times. Figure 4.5 illustrates the graphs of  $Error_{category1}$ ,  $Error_{category2}$  and  $Error_{category3}$  for the best try. As may be seen, reducing the percentage of sampling increases the number of errors in most of the cases. In figure 4.5 the last graph shows  $Mean_{correlation\ coefficient}$  and reducing the percentage of sampling decreases it in most of the cases.





(b)



(c)

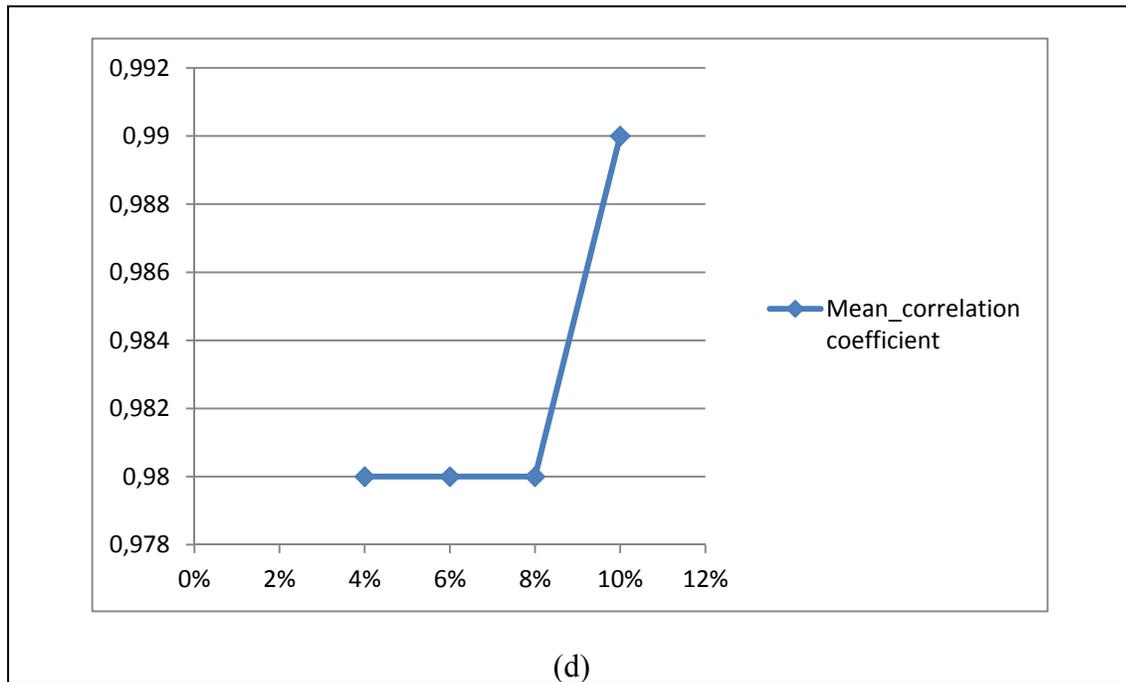


Figure 4.4 Graphs of a) Error\_category1, b) Error\_category2, c) Error\_category3 and d) mean\_correlation coefficient

## 4.8 Conclusion

In this chapter, the proposed methodology of the previous chapter is implemented. Programming is done in MATLAB<sup>TM</sup> as well as C++ programming environment using OpenCV. The latter is suggested for the clear improvements in speed and overcoming the shortage of memory that is faced when utilising the MATLAB<sup>TM</sup> environment. As explained in this chapter, programming in C++ is more efficient and also requires less elapsed time for running the program. It was shown, that for more complex circuit programming in MATLAB leads to problems and it is better to model all circuits in C++. The circuit C17 is used as a case study for the implementation of the proposed methodology, and the accuracy of its neural network model is evaluated. Three different type of errors are used for benchmarking the accuracy: Error<sub>category1</sub>, Error<sub>category2</sub> and Error<sub>category3</sub>. Mean\_correlation coefficient is also evaluated. All the results are compared with the results obtained by (ROBACHE 2013) and the analysis reveals that this methodology results in greater accuracy and allows better estimation of the faulty behaviour of circuits.

## CONCLUSION AND FUTURE WORK

Cosmic rays lead to faults in electrical systems. This is more significant when such systems are located at a higher altitude, such as avionic systems. Indeed, at higher altitudes, the cosmic rays intensity is higher and therefore the probability that they affect avionic circuits is higher. The faults caused by cosmic rays decrease the reliability of electronic systems. The goal of this research was to develop an approach for modeling the faulty behaviour of a digital circuit in the presence of cosmic rays. The proposed approach can be applied to a design flow before circuit fabrication to ensure early validation. The netlist file of the circuit is used by a fault simulator to generate the golden output and the reference faulty output of the circuit. These outputs are used for training the neural network model in MATLAB<sup>TM</sup> or C++ environment. Later, the trained neural network models can be used to develop pre-established libraries in SIMULINK. These pre-defined components can be used by a designer to substitute different part of the whole circuit to see the effect of faulty behaviour of each sub-circuit on a system.

The method proposed in this thesis details how to predict the faulty output of a circuit based on a neural network approach. The inputs of the circuit go to the neural network and the fault injected circuit. The objective of the method is to ensure that the neural network model mimics the behaviour of the fault injected circuit. To achieve this goal, appropriate training of the neural network is necessary.

The proposed methodology was validated with two case studies. The first uses a simple example which consists of modeling the faulty behaviour of the C17 ISCAS benchmark circuit. To validate our method, the results of the neural network approach are compared with those obtained from a faulty behavior Signature generation method that works at a lower level of abstraction presented in (ROBACHE 2013). The second case study uses a more complex example: the proposed technique is applied to another design, a 4-bit multiplier. Results show that the neural network approach leads to a more precise model. For the C17 circuit, by taking only 30% of the data set generated by the Lifting software, the neural

network is able to replicate the output of the circuit in presence of faults while keeping the mean absolute error below 6%. For the 4-bit multiplier circuit, by taking only 8% of the data set generated by the Lifting software, the neural network is able to replicate the output of the circuit in presence of faults while keeping the mean absolute error below 6%. The neural network approach involves more complexity when compared to the faulty behavior Signature method presented in (ROBACHE 2013). Thus simulation time is larger but results are more precise.

For future work, more circuits may be analyzed. The adder, subtractor and divider for 4-bit binary digit are just some examples of circuits for which the model may be developed. The 8-bit multiplier could also be explored as a more complicated example.

In order to find the best structure of neural network for this purpose, different architectures of neural network needs to be studied to develop the faulty behaviour model of the circuit. In this thesis, only the MLP neural network is explored. As part of future work, the investigation into developing different architectures for the neural network - such as Hopfield and recurrent neural network - is worth exploring. After analysing the results obtained from such architectures, it may enable us to choose the best architecture for the purpose under study.

MATLAB<sup>TM</sup> is the computing language that is used for modeling neural network in this thesis. However, for future work, porting the model to C/C++ is highly recommended, since as explained already in the course of the thesis, running a C++ is significantly faster and it also requires less memory compared to the performance observed using the MATLAB<sup>TM</sup> environment.

## APPENDIX I

### C++ CODE FOR CIRCUIT C17 USING OPENCV

```
#include < iostream >
#include < fstream >
#include < math.h >
#include < string >
#include <stdlib.h>
#include <tuple>

#include <opencv\cv.h>
#include <opencv\highgui.h>
#include <opencv\ml.h>

using namespace cv ;
using namespace std ;

int numTrainingPoints = 900;
int numTestPoints = 188;
int size = 200;
int numAttribute = 11;
int numOutputs = 2;
int eq =0;// Function to learn  $y > \sin(x * 10) ? -1 : 1$ ;
// accuracy
float evaluate(cv::Mat & predicted, cv::Mat & actual ) {
    assert( predicted.rows == actual.rows );
    int t = 0;
    int f = 0;
    for(int i = 0; i < actual.rows ; i ++ ) {
        for(int j = 0; j < numOutputs ; j ++ ) {
            float p = predicted.at <float >(i ,j) ;
            cout << "predict: " << p << " " ;
            float a = actual.at <float >( i ,j) ;
            cout << "target: " << a << " " << endl;
            if(( p >= 0.0 && a >= 0.0) || ( p <= 0.0 && a <= 0.0) ) {
                t ++;
            } else {
                f ++;
            }
        }
    }
    return (t * 1.0) / (t + f );
}

// plot data and class
void plot_binary( cv::Mat & data , cv::Mat & classes , string name ) {
    cv::Mat plot ( size , size , CV_8UC3 );
    plot.setTo ( cv::Scalar (255.0 ,255.0 ,255.0) );
    for(int i = 0; i < data.rows ; i ++ ) {
        float x = data.at <float >(i ,0) * size ;
        float y = data.at <float >(i ,1) * size ;
        if( classes.at <float >(i , 0) > 0) {
```

```

        cv::circle ( plot , Point (x ,y) , 2, CV_RGB (255 ,0 ,0) ,1) ;
    } else {
        cv::circle ( plot , Point (x ,y) , 2, CV_RGB (0 ,255 ,0) ,1) ;
    }
}
cv::imshow ( name , plot );
}

```

```

void mlp ( cv::Mat & trainingData , cv::Mat & trainingOutputs , cv::Mat & testData ,
cv::Mat &
testOutputs ) {

    int layers_d[] = { numAttribute, 20, numOutputs};
    Mat layers = Mat(1,3,CV_32SC1);
    layers.at<int>(0,0) = layers_d[0];
    layers.at<int>(0,1) = layers_d[1];
    layers.at<int>(0,2) = layers_d[2];

    // create the network using a sigmoid function with alpha and beta
    // parameters 0.6 and 1 specified respectively (refer to manual)

    CvANN_MLP* nnetwork = new CvANN_MLP;
    nnetwork->create(layers, CvANN_MLP::SIGMOID_SYM, 0.6, 1);

    // set the training parameters

    CvANN_MLP_TrainParams params = CvANN_MLP_TrainParams(

        // terminate the training after either 1000
        // iterations or a very small change in the
        // network wieghts below the specified value

        cvTermCriteria(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 1000, 0.000001),

        // use backpropogation for training

        CvANN_MLP_TrainParams::BACKPROP,

        // co-efficients for backpropogation training
        // (refer to manual)

        0.1,
        0.1);

    // train the neural network (using training data)

    //printf( "\nUsing training database: %s\n", "C17TrainDataset.csv");

    nnetwork->train(trainingData, trainingOutputs, Mat(), Mat(), params);

    cv::Mat response (1, numOutputs , CV_32FC1 );
    cv::Mat predicted ( testOutputs.rows
        , numOutputs, CV_32F );
    for(int i = 0; i < testData . rows ; i ++ ) {

```

```

        cv::Mat response (1, numOutputs , CV_32FC1 );
        cv::Mat sample = testData.row ( i);

        nnetwork->predict( sample , response );
        for (int jj = 0 ; jj < numOutputs ; jj ++ )
            predicted.at <float >(i ,jj) = response.at <float >(0 ,jj) ;
    }
    cout << "Accuracy = " << evaluate ( predicted , testOutputs ) << endl ;
    plot_binary ( testData , predicted , "Predictions Backpropagation");
}

std::tuple<double*, double*, double*> processLine( string &line)
{
    //create new pattern and target
    double* pattern = new double[numAttribute];
    double* target = new double[numOutputs];
    double* dData = new double[(numOutputs+numAttribute)];

    //store inputs
    char* cstr = new char[line.size()+1];
    char* t;
    strcpy(cstr, line.c_str());

    //tokenise
    int i = 0;
    t= strtok (cstr, ",");

    while ( t!=NULL && i < (numAttribute + numOutputs) )
    {
        if ( i < numAttribute ) patten\n[i] = atof(t);
        else target[i - numAttribute] = atof(t);

        //move token onwards
        dData[i] = atof(t);
        t = strtok(NULL, ",");
        i++;
    }

    //add to records

    return make_tuple(dData, pattern, target);
}

std::tuple<Mat, Mat, Mat> read_data_from_csv(const char* filename, int n_samples )
{
    Mat data ( n_samples , (numAttribute + numOutputs) , CV_32FC1 );
    Mat dataIn ( n_samples , numAttribute , CV_32FC1 );
    Mat dataOut( n_samples , numOutputs , CV_32FC1 );
    double tmp;
    string line = "";
    int lineNum = 0;
    int sizeData = numOutputs + numAttribute;
    double* dataLine = new double[(numAttribute + numOutputs)];

```

```

    double* dataInLine = new double[numAttribute];
    double* dataOutLine = new double[numOutputs];

ifstream f;
f.open(filename, ios::in);
    // if we can't read the input file then return 0
    //FILE* f = fopen( filename, "r" );
    if( !f.is_open() )
    {
        printf("ERROR: cannot read file %s\n", filename); // all not OK
    }

    // for each sample in the file
    while(std::getline(f, line))
    {

        // for each line in the file

            //getline(f, line);
            std::tie(dataLine, dataInLine, dataOutLine) = processLine(line);

            for (int i = 0 ; i < sizeData; i++)
                data.at<float>(lineNum, i) =(float) dataLine[i];

            for (int i = 0 ; i < numAttribute; i++)
                dataIn.at<float>(lineNum, i) =(float) dataInLine[i];

            for (int i = 0 ; i < numOutputs; i++)
                dataOut.at<float>(lineNum, i) =(float) dataOutLine[i];

            lineNum++;

    }
f.close();

    return make_tuple(data, dataIn, dataOut); // all OK
}

int main () {

    cv::Mat trainingData ( numTrainingPoints , numAttribute , CV_32FC1 );
    cv::Mat testData ( numTestPoints , numAttribute, CV_32FC1 );

    cv::Mat trainingInputs ( numTrainingPoints , numAttribute , CV_32FC1 );
    cv::Mat testInputs ( numTestPoints , numAttribute, CV_32FC1 );

    cv::Mat trainingOutputs ( numTrainingPoints , numOutputs , CV_32FC1 );
    cv::Mat testOutputs ( numTestPoints , numOutputs, CV_32FC1 );

    freopen("report.txt", "w", stdout);

    std::tie(trainingData, trainingInputs, trainingOutputs) =
read_data_from_csv("C17TrainDataset.csv", numTrainingPoints);

```

```
    for (int i = 0 ; i < numTrainingPoints; i++)
        cout << trainingData.row(i) << endl;

        std::tie(testData,          testInputs,          testOutputs)          =
read_data_from_csv("C17TestDataset.csv", numTestPoints);

        mlp ( trainingInputs , trainingOutputs , testInputs , testOutputs );
        cv::waitKey ( ) ;
        return 0;
}
```



## LIST OF BIBLIOGRAPHICAL

- Actel (2002). Effects of Neutrons on Programmable Logic.–a white paper. Technical report, Actel corporation.
- Al-Jumah, A. and T. Arslan (1998). Artificial neural network based multiple fault diagnosis in digital circuits. Circuits and Systems, 1998. ISCAS'98. Proceedings of the 1998 IEEE International Symposium on, IEEE.
- Alfredo Benso, P. P. (2003). Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Springer.
- ALTERA (2013). Introduction to Single-Event Upsets.–a white paper. Technical report, ALTERA corporation.
- Antoni, L., et al. (2002). Using run-time reconfiguration for fault injection in hardware prototypes. Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings. 17th IEEE International Symposium on, IEEE.
- Baraza, J.-C., et al. (2008). "Enhancement of fault injection techniques based on the modification of VHDL code." Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 16(6): 693-706.
- Bolchini, C. and C. Sandionigi (2010). "Fault classification for SRAM-Based FPGAs in the space environment for fault mitigation." Embedded Systems Letters, IEEE 2(4): 107-110.
- Borecky, J., et al. (2011). Fault models usability study for on-line tested fpga. Digital System Design (DSD), 2011 14th Euromicro Conference on, IEEE.
- Bosio, A. and G. Di Natale (2008). LIFTING: A flexible open-source fault simulator. Asian Test Symposium, 2008. ATS'08. 17th, IEEE.
- Brogley, M. (2009). "FPGA reliability and the sunspot cycle." White Paper, Actel.
- Bushnell, M. and V. D. Agrawal (2000). Essentials of electronic testing for digital, memory, and mixed-signal VLSI circuits, Springer.

Carmichael, C., et al. (2000). "Correcting single-event upsets through Virtex partial configuration." Xilinx Corporation.

Cerny, P. A. and M. A. Proximity (2001). "Data mining and neural networks from a commercial perspective." Auckland, New Zealand Student of the Department of Mathematical Sciences, University of Technology, Sydney, Australia.

Chapman, K. (2010). "SEU strategies for Virtex-5 Devices."

Dirk, J., et al. (2003). "Terrestrial thermal neutrons." Nuclear Science, IEEE Transactions on 50(6): 2060-2064.

Farhat, I. (2003). Fault detection, classification and location in transmission line systems using neural networks, Concordia University.

Fausett, L. (2006). Fundamentals of Neural Networks: Architectures, Algorithms, and Applications, Pearson Education India.

Grosso, M., et al. (2013). "Exploiting Fault Model Correlations to Accelerate SEU Sensitivity Assessment."

Haykin, S. (2001). Neural Networks: A Comprehensive Foundation. , Pearson Education Singapore Pte. Ltd.

Heaton, J. (2008). Introduction to neural networks with Java, Heaton Research, Inc.

Jain, A. K., et al. (1996). "Artificial neural networks: A tutorial." IEEE computer 29(3): 31-44.

JESD89A, J. S. (2006). "Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices." Solid State Technology Association.

Johnson, J. ( 2013). General regression and over fitting.–a blog post. The Shape of Data 2014.

- Koivo, H. N. (2006). Basics using MATLAB Neural Network Toolbox, London: Verlag Springer.
- Laura Dominik, H. I., Minneapolis, MN (2008). "SYSTEM MITIGATION TECHNIQUES FOR SINGLE EVENT EFFECTS." IEEE.
- Lesea, A., et al. (2005). "The Rosetta experiment: atmospheric soft error rate testing in differing technology FPGAs." Device and Materials Reliability, IEEE Transactions on 5(3): 317-328.
- Lifen, Y., et al. (2010). "A New Neural-Network-Based Fault Diagnosis Approach for Analog Circuits by Using Kurtosis and Entropy as a Preprocessor." Instrumentation and Measurement, IEEE Transactions on 59(3): 586-595.
- Lima, F., et al. (2001). On the use of VHDL simulation and emulation to derive error rates. Radiation and Its Effects on Components and Systems, 2001. 6th European Conference on, IEEE.
- Marsland, S. (2011). Machine learning: an algorithmic perspective, CRC Press.
- Mirzadeh, Z., et al. (2010). A fast method for video deblurring based on a combination of gradient methods and denoising algorithms in Matlab and C environments. Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series.
- Mukherjee, S. (2011). Architecture design for soft errors, Morgan Kaufmann.
- Normand, E. (1996). "Single-event effects in avionics." Nuclear Science, IEEE Transactions on 43(2): 461-474.
- Ostler, P. S., et al. (2009). "SRAM FPGA reliability analysis for harsh radiation environments." Nuclear Science, IEEE Transactions on 56(6): 3519-3526.
- Patel, J. H. (2005). "Stuck-At Fault: A Fault Model for the next Millennium?". Retrieved March 1, 2014, from [http://courses.engr.illinois.edu/ece543/docs/stuck\\_at\\_fault\\_6per\\_page.pdf](http://courses.engr.illinois.edu/ece543/docs/stuck_at_fault_6per_page.pdf).

Quinn, H. M., et al. (2013). "Fault Simulation and Emulation Tools to Augment Radiation-Hardness Assurance Testing." Nuclear Science, IEEE Transactions on 60(3): 2119-2142.

ROBACHE, R. (2013). "MISE EN OEUVRE ET CARACTÉRISATION D'UNE MÉTHODE D'INJECTION DE PANNES À HAUT NIVEAU D'ABSTRACTION."

ROBACHE, R. (2013). MISE EN OEUVRE ET CARACTÉRISATION D'UNE MÉTHODE D'INJECTION DE PANNES À HAUT NIVEAU D'ABSTRACTION.—a thesis.

Sarle, W. S. (2002). "comp.ai.neural-nets FAQ, Part 3 of 7: Generalization." Retrieved 4 May, 2013, from <http://www.faqs.org/faqs/ai-faq/neural-nets/part3/>.

Seward, S. and P. K. Lala (2003). Fault injection for verifying testability at the VHDL level. 2013 IEEE International Test Conference (ITC), IEEE Computer Society.

Taber, A. and E. Normand (1993). "Single event upset in avionics." Nuclear Science, IEEE Transactions on 40(2): 120-126.

Valderas, M. G., et al. (2007). Advanced simulation and emulation techniques for fault injection. Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on, IEEE.

Velazco, R., et al. (2007). Radiation effects on embedded systems, Springer.

Vibishna, B., et al. (2013). "Understanding single-event effects in FPGA for Avionic system design." IETE Technical Review 30(6): 497.

Wagner, P. (2012). "Machine Learning with OpenCV2."

Whei-Min, L., et al. (2001). "A fault classification method by RBF neural network with OLS learning procedure." Power Delivery, IEEE Transactions on 16(4): 473-477.

Yu, Z. (2000). Feed-forward neural networks and their applications in forecasting, University of Houston.

Zhang, Q.-J., et al. (2003). "Artificial neural networks for RF and microwave design-from theory to practice." *Microwave Theory and Techniques, IEEE Transactions on* 51(4): 1339-1350.

Ziade, H., et al. (2004). "A survey on fault injection techniques." *Int. Arab J. Inf. Technol.* 1(2): 171-186.