ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Ανίχνευση Δυναμικών Σεναρίων Συστήματος Σε Ασύρματες Εφαρμογές με χρήση Νευρωνικών Δικτύων

Διπλωματική Εργασία
του
Ευάγγελου Ν. Ζαφειράτου

Επιβλέπων : **Δημήτριος Σούντρης**
Αν. Καθηγητής Ε.Μ.Π

Αθήνα, Μάρτιος 2015

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Ανίχνευση Δυναμικών Σεναρίων Συστήματος Σε Ασύρματες Εφαρμογές με χρήση Νευρωνικών Δικτύων

Διπλωματική Εργασία
του
Ευάγγελου Ν. Ζαφειράτου

Επιβλέπων : **Δημήτριος Σούντρης**
Αν. Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 30η Μαρτίου 2015.

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
………………………	………………………	………………………
Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Γεώργιος Οικονομάκος
Αν. Καθηγητής Ε.Μ.Π	Καθηγητής Ε.Μ.Π	Επ. Καθηγητής Ε.Μ.Π

Αθήνα, Μάρτιος 2015

....

Ζαφειράτος Ευάγγελος

Διπλωματούχος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

Copyright © ΖΑΦΕΙΡΑΤΟΣ ΕΥΑΓΓΕΛΟΣ, 2015
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου

Περίληψη

Τα τεχνητά νευρωνικά δίκτυα κερδίζουν σε δημοφιλία τα τελευταία χρόνια, καθώς οι μοντέρνοι επεξεργαστές εξελίσσονται με βάση την προσέγγιση στην παραλληλοποίηση. Οι παραδοσιακοί, σειριακοί ψηφιακοί υπολογισμοί επικρατούν σε πολλούς τομείς, αλλά είναι λιγότερο επιτυχείς για άλλου τύπου προβλήματα. Η εξέλιξη των νευρωνικών δικτύων ξεκίνησε πριν από 60 χρόνια περίπου, με κίνητρό να προσπαθήσουμε να καταλάβουμε αλλά και να μιμηθούμε τον εγκέφαλο, και συνεχώς κερδίζουν έδαφος, αφού η εξέλιξη των σύγχρονων Hardware πλατφόρμων προσφέρει νέες δυνατότητες.

Τα σενάρια συστήματος είναι επίσης ένας εξελισσόμενος τομέας στην επιστήμη του Hardware, που έχει σαν σκοπό να μετατρέψει την αυξανόμενα δυναμική φύση των ενσωματωμένων συστημάτων σε ευκαιρία βελτιστοποίησης αντί για πιθανό πρόβλημα. Η χρήση των σεναρίων συστήματος στις μοντέρνες συσκευές μας επιτρέπει να διανέμουμε τους πόρους του συστήματος με έναν αποδοτικό τρόπο, αφού κάθε εφαρμογή που εκτελείται έχει και διαφορετικές απαιτήσεις. Γνωρίζοντας το σενάριο εκτέλεσης, είναι δυνατό με δυναμική ανάθεση πόρων, να πετύχουμε καλύτερη απόδοση.

Ο στόχος της παρούσας Διπλωματικής Εργασίας είναι να παρέχει μία υλοποίηση, η οποία χρησιμοποιεί τεχνητό νευρωνικό δίκτυο ως το βασικό δομικό στοιχείο, και πραγματοποιεί εντοπισμό σεναρίων σε πραγματικές εφαρμογές. Η επιλογή των νευρωνικών δικτύων έγινε εξαιτίας της παράλληλης δομής τους, και της ικανότητάς τους να αναπτύσσουν δυναμική συμπεριφορά. Η υλοποίηση παρουσιάζεται συγκριτικά με μία στατική υλοποίηση με σκοπό να αναδείξουμε και να επισημάνουμε τις διαφορές και τα πλεονεκτήματα της καθεμιάς.

Λέξεις Κλειδιά
Σενάρια συστήματος; δυναμική ανάθεση; Νευρωνικά δίκτυα, εντοπισμός; fpga; vhdl.

Abstract

Artificial Neural Networks gain popularity in recent years, as modern processors evolve towards a parallel approach. Traditional, sequential, logic-based digital computing excels in many areas, but has been less successful for other types of problems. The development of artificial neural networks began approximately 60 years ago, motivated by a desire to try both to understand the brain and to emulate some of its strengths and is constantly gaining attention as modern Hardware platforms evolve and offer new promising capabilities for Neural Networks development.

System Scenarios is also a developing field in science of Hardware which aims to convert the increasingly dynamic nature of embedded systems into an optimization opportunity instead of a potential problem. The use of system scenarios scheduling in modern devices allows us to exploit resources of the system in a sophisticated manner, since every different form of execution differs in terms of hardware requirements. Acknowledging the scenario to be executed, it is possible to modificate resources allocation and achieve greater performance.

The goal of this diploma thesis is to provide a sufficient hardware/software co-design implementation which enables neural networks as the basic unit of a structure that detects Scenarios in real applications. The choice of neural networks was made because of their inherited parallelism and their ability to develop dynamic behavior. The implementation with Neural Networks is presented side by side with a straight – forward implementation in order to feature the advantages of each and highlight the differences.

The thesis is organized as follows:

In Chapter 1, there is an introduction in Wireless Systems and System Scenarios, along with a proposed methodology (Zompakis et al, 2012) for using System Scenarios in real applications. A description of Scenario detection in real - time follows accompanied by related work on this problem. Finally, an outline of the suggested solution by current thesis is presented.

Chapter 2 is a brief description of Artificial Neural Networks. Historical background, topologies, and types of ANNs are examined. Special emphasis is given to training methods and more specifically, to Levenberg – Marquadt algorithm, which is the selected training function.

Analytical methodology for our solution is presented in Chapter 3. The workflow shows the steps sequentially towards the final implementation. The said chapter also contains extended justification for the neural network selected specifications. The last part is a detailed analysis of the VHDL modules of the implementation, which apart from technical information also include timeline diagrams. The intention for using

timeline diagrams for each module separately is to analytically present in a schematic way the exact tasks performed in the inferred hardware.

Chapter 4 is dedicated to the presentation and analysis of the results of our case study. Important implementation parameters, such as operating frequency, chip area and dynamic ability are measured and compared for the two separate solutions.

Finally, Chapter 5 summarizes the results and conclusions of the current study and suggests future work for the improvement of the existent implementation.

Keywords
System Scenarios; Dynamic Scheduling; Neural Networks; detection; fpga; vhdl.

Table of Contents

Chapter 1 Introduction
1.1 Embedded Systems 10
1.1.1 Overview 10
1.1.2 SDR Operation Specs 11
1.2 System Scenarios 13
1.2.1 Overview 13
1.2.2 Description and Methodology 15
1.3 Motivation – Problem Statement 18
1.4 Proposed Solution 19
Chapter 2 Neural Networks
2.1 Overview 22
2.2 Neural Networks Fundamentals 24
2.2.1 Definition 24
2.2.2 Characteristics 25
2.2.3 Network Architecture 26
2.3 Neural Networks Types 27
2.3.1 Overview 27
2.3.2 Perceptron 27
2.3.3 ADELINE, MADELINE 28
2.3.4 Backpropagation 30
2.3.5 Hopfield 31
2.3.6 ART 31
2.3.7 Cascade Correlation 32
2.4 Fundamentals of Learning and Training functions 33
2.4.1 Learning methods 33
2.4.2 Training functions 34
2.4.2.1 Levenberg – Marquadt Algorithm 34
2.5 Hardware adaptation of Neural Networks 35
2.5.1 Hardware Platforms Overview 35
2.5.2 ASIC 36
2.5.3 FPGA 36
2.5.4 Neural Networks in Hardware 37
2.5.5 FPGA and Neural Networks 38
Chapter 3 Implementation
3.1 Implementation Aspects 39
3.1.1 Neural Network Architecture 39
3.1.2 Data Discretization 41
3.1.3 Input Normalization 42
3.2 Methodology 42

3.2.1 Overview 42
3.2.2 Static Implementation 43
3.2.3 Dynamic Implementation 44
3.2.4 Neural Networks Builder 46
3.3 Anatomy of the Design 46
3.3.1 Project Hierarchy 46
3.3.2 Neural Library Module 47
3.3.3 Log Sigmoid Module 49
3.3.4 Hidden LUTs Module 54
3.3.5 Output LUTs Module 56
3.3.6 Hidden Node Module 57
3.3.7 Output Node Module 61
3.3.8 Ann Module 63
3.3.9 Hybrid Module 67
Chapter 4 Case Study
4.1 System Modeling 70
4.2 Case Study (I) 73
4.3 Case Study (II) 77
Chapter 5 Conclusions & Future Work 80
References 81
Appendix A 84
Appendix B 92
Appendix C 96

Chapter 1 Introduction

1.1 Embedded Systems

1.1.1 Overview

In recent years, the wireless technology has opened new horizons in the means and ways that users communicate [1]. We are living in a very competitive environment, where the radio devices become outdated soon after their engineering. Radios exist in a multitude of items such as cell phones, vehicles, tablet pcs and digital TVs. The different types of applications demand different type of communication standards. Although all these systems have almost similar components, the ways these components behave differ greatly. To cope with these challenges, communication systems adopt open architectures with flexible interfaces. The new specifications are introduced to the existing infrastructure without requiring new expenditures. Thus, while migrating from one generation to the next, the new devices are compatible with the conventional and the state of the art networks. The modern 4G networks provide high quality of services (QoS) exploiting new innovative products, which combine smart transceivers and high performance signal processing elements [2]. This trend highlights challenges that the classic hardware-based radios cannot cope with.

More precisely, the traditional radio chips are designed for specific operations each of them is realized through a single communication standard. A typical handset has several chips to establish a variety of wireless links, one to talk to a cell phone, another to communicate with a Wi-Fi base station, a third to process GPS signals. All these chips support particular spectrum areas and modulation schemes. Thus, after the device engineering, they are exploitable only for the purpose that they are designed. This confines the scalability of a potential radio device and restricts the update capabilities at the improvement of the user interface without providing real operation extensions. However, this approach was not able to answer the ever-changing requirements of the modern transceivers.

In addition, the standardization at the development of the new handsets is a key issue, which occupies the radio industry. This is highly desirable because it allows new products come quickly into the market limiting the design and the development cost. It is fact that a family of products with common hardware architecture will require much less implementation effort. In this direction, the particular functionality can be performed by modifiable software. The software definition of the functionality opens significant opportunities at the follow-on-support services. New features and capabilities can be added to the existing devices without requiring any extra hardware equipment. Software upgrades can remotely activate new revenue

generating features. Bug-fix and reprogramming services are able to reduce the costs while a device is in service. Thus, the cost reduction in the end-users allows them to communicate with whomever they need, whenever they need to and in whatever manner is appropriate.

Another open issue is the efficient utilization of the available spectrum area. Radio bandwidth is a scarce resource, which have to be distributed with a dynamic way. The conventional radios, which are modifiable only by physical interventions, don't provide the necessary flexibility. Thus, the interest to explore ways using the spectrum with a more efficient way is quite high. The right exploitation of the frequency bandwidth depends on a number of factors, which combine the geographical characteristics of the area and the transmission activity in it. The main reason for insufficient bandwidth utilization is the spectrum fragmentation. Even in an environment with high density of wireless transmissions, the spectrum exploitation can be poor. The reason is the substantial amounts of unused spectrum segments "white spaces" which are congested by gaps between the transmission channels, which ensure the avoidance of the interference. Wireless devices being able to access unused or restricted spectrum segments that may be available for usage in other geographical areas or under other regulatory regimes, can improve the spectrum utilization. In this regard, reconfigurability is the key point for the radio industry.

Taking into consideration all the previous challenges, wireless industry requires a multiband reconfigurable implementation with an open architecture capable to cope with the rapid development of the communication standards. The reconfigurability refers to a radio that supports multiple frequencies bands and multiple modulation schemes which adapt its configuration at the running state. An extra motivation for such an implementation is the fact that the standard wireless processes like filtering, decoding, signal modulation, can also benefit from the reconfigurability offered by a general-purpose architecture [36]. A well-known example of a platform with these capabilities is Software Defined Radio (SDR) [37], which combines numerous communication standards in a single device. Many of its functionalities are implemented in software, running on one or multiple generic processors, leaving only the high performance functions implemented in hardware. These kinds of software radios will be future proof as the whole system will be based on reprogramming, leading the same hardware behaving differently at different instances.

1.1.2 SDR Operation Specs

Software Define Radio (SDR) is an efficient merging of technologies, which combines software and hardware in such a way that the physical layer functions are modifiable. The Wireless Innovation Forum, in collaboration with the Institute of Electrical and Electronic Engineers (IEEE) P1900.1 group, establishes a definition of SDR that provides a clear view of the technologies involved and their benefits. Software Defined Radio is defined as: "Radio in which some or all of the physical

layer functions are software defined" [2]. SDR defines a collection of hardware and software technologies where some or all of the radio's operating functions (also the physical layer processing) are implemented through modifiable software or firmware operating on programmable processing technologies. The use of SDR technologies enables greater degree of freedom in adaptation, higher performance levels and better quality of service. Adaptation has the notion of sensing the operations changes, calibrating the system parameters for succeeding a better performance. This characteristic makes software-defined radios remarkable flexible. In a theoretical basis, the right software in a SDR chip can implement every individual function, which takes place in a wireless device. The idea is to transfer the critical wireless functions in software, allowing adding new operations without hardware changes. Thus, SDR architectures tend to become a general purpose platform which can realize every wireless implementation.

After a long period from the first introducing of the Software Defined Radio concept [37] SDR seems to be a promising solution for integrating the existing and the emerging communication standards into one platform. The first SDR approach limited only at the level of the replacement parts of the radio hardware by ones that are reconfigurable and reprogrammable. After this concept was extended including reconfiguration of applications and services, as well as network-based reconfiguration support, provided by a dedicated network infrastructure. The cause of this development is that applications and services are likely to be affected by changing transmission quality and changing Quality of Service (QoS) resulting from vertical handover from one radio mode to another and, therefore, service aspects have to be taken into account in handover decision-making.

The advanced SDR technology has to handle not only the primary performance challenges but also the restrictions of the mobility. In the last decades, SDR devices have become much more complex due to the introduction of a lot of new functionality in one application, and due to supporting various services simultaneously including a wide range of communication protocols and services. Thus, the SDR platforms communicate with other platforms using multiple complex communication schemes. The connection flexibility is restricted mainly by the tight platform constrains. These handsets have stringent requirements on size, performance and energy consumption. Optimizing energy efficiency is key for maximizing battery lifetime between recharges. In addition, the modern SDR system architectures enlarge the gap between average and worst-case execution time of applications to increase total performance. An efficient utilization of the available resources based on the running situations and with the minimum configuration cost is needed. System adaptation can be implemented either at application level, selecting an effective task mapping technique, or at platform level, e.g. with dynamic frequency scaling technique (DFS).

Thus, the development of proper methods in resource scheduling is without doubt, an imperative need. Traditional design approaches based on the worst-case leave a lot of room of optimization if the increasing resource usage dynamism can be properly predicted at runtime.

1.2 System Scenarios

1.2.1 Overview

In the past years, the functions demanded for embedded systems have become so numerously and complex that the development time is increasingly difficult to predict and control [3]. This complexity, together with the constantly evolving specifications, has forced designers to consider implementations that they can change rapidly. For this reason, and also because the hardware manufacturing cycles are more expensive and time-consuming than before, software implementations have become more popular. As often the application source code is already written, the trend is to reuse the applications, as this is the best approach to improve the quality and the time to market for the products a company creates and, thereby, to maximize profits [4]. Most of these applications are written in high level languages to avoid the dependency on any type of hardware architecture and to increase developers' productivity.

In the context of this software intensive approach, the job of the embedded designers is to evaluate multiple hardware architectures and to select the one that fits best given the application constraints and the final product requirements (i.e., price, energy, size, performance). The explored architectures lay between fixed single processor off-the-shelf architectures and fully design time configurable multi-processor hardware platforms [5]. The off-the-shelf components are cheaper to use, as no extra development is needed, but they are not very flexible (e.g., video accelerators) or cannot be tuned for a specific application (e.g., general-purpose processors, if performance is considered). Hence, they usually are good candidates for simple systems that are produced in small volumes. On the other extreme, configurable multi-processor platforms offer more flexibility in tuning, but they imply an additional design cost. Hence they are used when the production volume is large enough for economically viable manufacturing, or when no existing off-the-shelf component is good enough.

Given an embedded system application, to find the most suitable architecture, or to fully exploit the features of a given one under the real-time constraints, estimations of the amount of resources required by each part of the application are needed. To give guaranties for the system quality, the estimations should be pessimistic, and not optimistic, as over-estimations are acceptable, but underestimations are generally not. Currently used design approaches use worst case estimations, which are obtained by statically analyzing the application source or object code [6]. However, these techniques are not always efficient when analyzing complex applications (e.g., they do not look at correlations between different application components), and they lead to system over-dimensioning.

Hence, the problem System Scenarios aiming to resolve is :
"The need for a systematic methodology that, given a dynamic streaming application with many operation modes, finds and efficiently exploits the most suitable hardware architecture under the final system constraints (i.e., performance, price, size and energy consumption), without ending in an explosion problem".

This problem is quite broad, as it ranges from single to multi-processor architectures, and it covers multiple types of resources (e.g., computation, communication, storage) and constraints.

1.2.2 Description and Methodology

Scenario based design has been used for a long time in different design areas [38] and especially at the development of the embedded system domain [7]. Scenarios describe, in an early design phase of a development process, the future system functionality including the interaction with the user. The scenarios are narrative descriptions of envisioned usage episodes. In case of object oriented software engineering a unified modelling language (UML) and use-case diagram enumerate, from functional and timing point of view, all possible user actions and the system reactions that are required to meet a proposed system function. These scenarios are called use-case scenarios [7]. In our study, we concentrate on a different kind of scenarios, so-called system scenarios, which characterize the system from the resource usage perspective.

The system scenario methodology has been described in a fully systematic way in [4]. The aim is to capture the data dependent dynamic behavior inside a thread in order to better schedule a multi-thread application on a heterogeneous multi-processor architecture. Usually, most of these applications are streaming and have to deliver a given throughput, which imposes specific time constraints. [8] presents a design methodology that provides a systematic way of detecting and exploiting system scenarios for streaming applications. A scenario is defined as the application behavior for a specific type of input data, i.e. a group of execution paths for that particular group of input data. The system scenario concept was also outlined in [9], where the tasks are written using a combination of a hierarchical finite state machine (FSM) with a synchronous dataflow model (SDF). The disadvantage of this method is that the applications must be written using a limited model, which is a time consuming and error-prone operation.

The system scenario methodology is a design approach for handling the complexity analysis of applications with multidimensional costs and strict constraints. The main challenges are: 1) the optimal application mapping on the platform and 2) the efficient management of the platform resources. The methodology key points are: 1) the splitting of the design problems in separate steps at design time and 2) the implementation of only the optimal solutions at run time. In particular, by classifying and clustering the possible system executions into system scenarios, a run-time resource manager can heavily reduce the average cost resulting from this execution

compared to the conventional worst-case bounding approach, while still meeting all constraints.

As a first step in explaining the methodology, we have to introduce the concept of a Run-Time Situation (RTS). As RTS we define a piece of system execution that is treated as a unit because it has uniform behavior internally. The system scenario methodology comprises 5 individual steps, 1) RTS identification, 2) RTS characterization, 3) RTS clustering into system scenarios, 4) scenario detection and, 5) scenario switching.

1) **RTS identification** This methodology starts with the characterization of all possible RTSs, which occur in the system. We identify all the variables (RTS parameters) that affect the state of the system from a functionality or implementation point of view. System variables can be classified in two categories; control and data variables. Control variables define the execution paths of an application and determine which conditional branches are taken or how many times a loop will iterate. They have a higher impact on execution time, as they decide how often each part of the program is executed. Hence we focus on them. The data variables represent the data processed by the application.

2) **RTS characterization** In most cases, the cost characterization of the RTSs is not a simple determination of one cost value but it leads to a Pareto surface of potential exploitation points in the multidimensional exploration space. Each RTS can be characterized by a number of cost factors obtained from profiling the application on a platform or by using high-level cost estimators. Cost axes may include quality level, user benefit, code size, execution time, total energy consumption, including the impact of the system operating conditions. It quantifies all the costs for each different platform configuration per RTS. The two typical costs for a system are: 1) the energy consumption, 2) the performance as it is expressed by the total delay (latency) for an operation execution. Hence the exploration space is usually two dimensional.

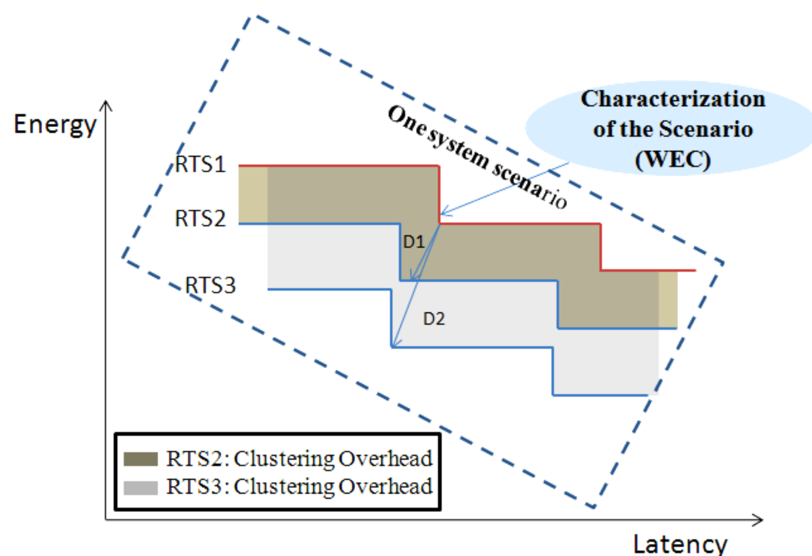


Figure 1.1 Clustering Overhead Representation [1, p.45]

3) Clustering of RTSs in System Scenarios An individually handling of every RTS, would lead to excessive overheads at run-time, since the source code and all configuration settings would need to be stored for each RTS and applied at run-time. So they have to be clustered into scenarios. But clustering introduces overestimation, which is characterized as clustering overhead, and is caused by the deviation between the real cost of the RTS and the estimated cost which is the representative cost for the scenario of the RTS. This overestimation will be incurred in every appearance of this RTS. Thus, the total overestimation will be proportional not only to the distance between RTS cost and scenario cost but also to the frequency of this RTS.

The similarity between costs of different RTSs or in general sets of RTSs (scenarios) has to be quantified e.g., by defining the normalized, potentially weighted, distance between two N-dimensional Pareto surfaces as the size of an N-dimensional volume that is present between these two sets. Based on this distance, the quality of potential scenario options can be quantified, e.g., to decide whether or not to cluster RTSs in different scenarios [5]. Clustering is implemented using a cost function related to the target objective optimization and takes into account: 1) how often each RTS occurs at run-time and 2) the distance of their Pareto curves. The scenario characterization (Pareto curve) results from taking the worst-case cost point among the RTSs.

4) Detection of System Scenarios After the generation of system scenarios the next step is the realization of a detection algorithm, which can recognize at run-time the scenario to be executed. The detection mechanism will be embedded in the middleware (e.g. RTOS) of the targeted platform adding some overhead on both execution time and memory footprint. It is critical to keep this overhead small while maintaining the benefits by exploiting the knowledge from the scenario recognition. The detection is implemented by monitoring the changes of the RTS parameters at run-time. Their value range has great impact on the final overhead. The challenge is to discover heuristic techniques which can detect the scenarios with minimum cost.

Figure 1.2 illustrates the implementation of a detection algorithm for a given application with 3 RTS parameters (bandwidth, number of antennas, coding). The detection algorithm starts from inner node ξ1, if the current bandwidth is equal to 20 MHz. If the condition is true the detection goes to line 3. At the new instruction line, we are at the inner node 2 and we have a new RTS parameter (number of antennas) to check and a new instruction to run. The procedure continues until the decision diagram reaches a detected system scenario.

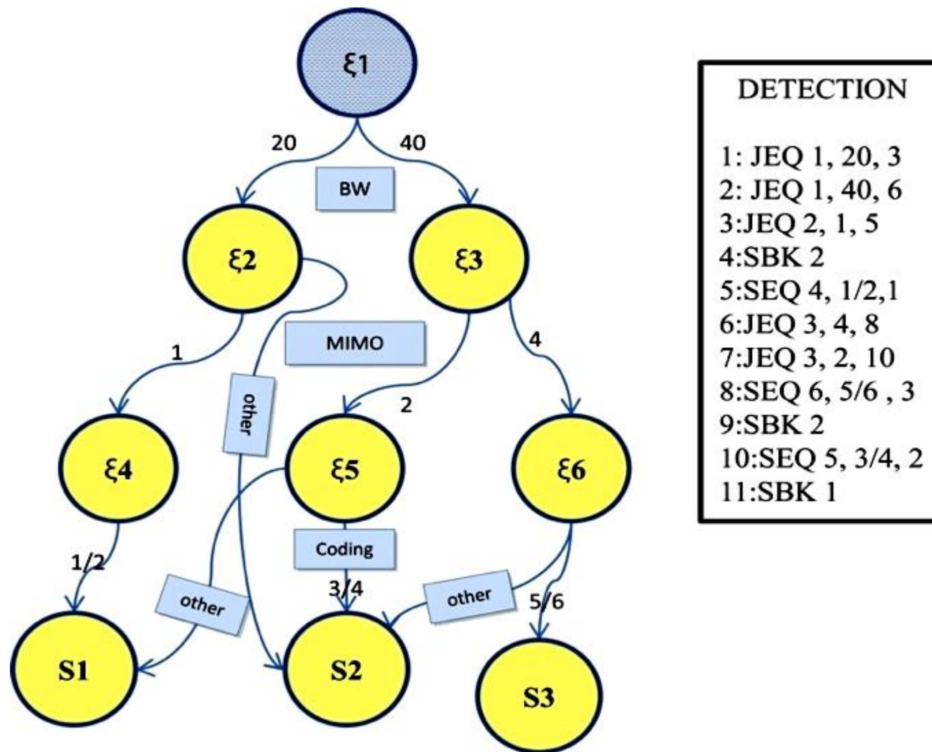


Figure 1.2. Decision diagram of a wireless Application [1, p.47]

5) Switching Having identified the system scenarios and the suitable detection approach, the next step is the implementation of a run-time algorithm, which will decide on the switching of the system configuration in real time. From the identification part, we have characterized every scenario so we can estimate, at design time, the tuning configuration for every scenario which respects the application constrains with the minimum energy cost. The tuning configurations can be related with the voltage scaling and the frequency scaling or other power saving techniques like processor resizing [10] and cache resizing [11]. So every system scenario corresponds to an optimal set of system configurations (e.g. an E-T Pareto curve of potential working points) and this information is stored in the system scenario list.

What we need now is the implementation of a mechanism which will react to the detection of a new scenario being triggered, and then decide whether to switch from the current scenario or not, while exploiting this information and taking into consideration the switching cost. If the new scenario is not expected to last very long and the gain G is limited then we cannot afford a high switching cost because that will probably be lower than G. As switching cost, we define the cost for the switching from one scenario to another. This cost will normally depend heavily on the initial and final state.

1.3 Motivation – Problem Statement

System Scenarios methodology steps are the following : 1) RTS identification, 2) RTS characterization, 3) RTS clustering into system scenarios, 4) scenario detection and, 5) scenario switching. The subject of the current thesis is to feature the demands and characteristics of the step referring to scenario detection and develop efficient solutions that could be used in real – time applications.

The step of detection is directly dependent on the previous step of clustering. There could be many different approaches regarding RTS clustering, e.g a fully analytical approach that includes many RTSs in its exploration would make the procedure of detection more demanding than an approach that includes only a few RTSs. Taken this into account, we can come to the first conclusion that a universal detector is not suitable for every case, as we have specific requirements that result from each problem.

Another important aspect is this of integration. The development of a mechanism that will run in parallel to the main implementation and recognize at run-time the Scenario that the specific combination of RTSs define is the key point for a successful implementation of run – time scheduling in wireless devices. This mechanism is not directly part of the device hardware; it is complementary and its function is to interact with elements from the main architecture and this interaction is critical to have response time which will be significantly lower than the average time of Scenario execution. Since response time is a prerequisite, external circuits to perform this task are not considered as possible solutions. This mechanism should be embedded to the system so as to share resources and transfer data more efficiently.

Moreover, there is high demand for accuracy. The process of detecting the current scenario is deterministic and should be treated as such. Recognition of a false scenario could trigger a change to an unsuitable state where resource allocation is not sufficient for the current task. Using a hypothetical probabilistic approach, there would be mispredictions of two types: (i) over-prediction, when a scenario with a higher cost is selected, and (ii) under-prediction, when a scenario with lower cost is selected [4] . The first type does not produce critical effects, just leading to a less cost effective system; the second type often reduces the system quality, e.g., by increasing the number of deadline misses when the cost is a cycle budget for an MP3 decoder application.

A proposed solution (Gheorghita et al 2007) is to construct a graph as a decision diagram, and make use of a restricted programming language to prevent added overhead, as shown in Figure 1.3.

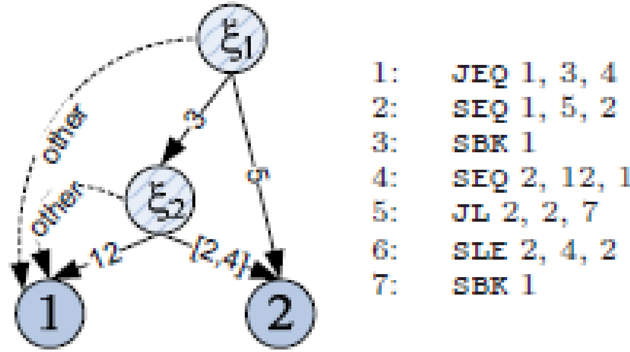


Figure 1.3 Example of detector implementation [4]

It examines, for the current frame to process, the values of a set of variables, and based on them it predicts in which scenario the application runs. In this approach, the decision diagram is implemented as a program in a restricted programming language, and it is executed by a simple execution engine. The program is in the application source represented by a data array. This split allows an easy calibration of the decision diagram, which consists of changing the values of several array elements.

This approach is a straight – forward implementation of the detection scheme and while it looks suitable at occasions where RTS identification and clustering involves a limited amount of parameters, in case of a broader RTS identification, the additional overhead and cost of the decision diagram is a restraining factor of the specific implementation. Thus, we will suggest alternative methods that adjust the final solution depending on the scaling of the problem.

1.4 Proposed Solution

Our goal is to propose a scenario detection methodology and proceed towards developing the tools needed for its implementation. The solution is focused towards minimizing the detection overhead. The latter is the most critical parameter that we should take into consideration, because it affects in direct way the performance of our system. Achieving timing closure in our implemented mechanism enables the supported system to recognize scenarios and switch states at run – time in a pace that maximizes the gains of this process.

A hardware implementation was preferred instead of software implementation. This decision was due to two main reasons: a) the already reported need to reduce the timing overhead and b) recent evolution of reconfigurable Hardware (FPGAs) provides with the necessary flexibility for the design and parameterization of the specific task. Moreover, the detection scheme is designated to be used in real applications of wireless devices, so a direct hardware implementation seems more usable.

Two separate solutions were developed in order to exploit the features that appear when using System Scenarios. The first solution is a straight – forward approach, a deterministic LUT which accepts as input the pre-defined combination of RTSs and returns in its output the specific scenario. The second solution is a Neural Network with the minimum number of layers in order to prevent additional overhead. The input and output stages of the second solution are the same with the ones of first solution, but the internal stages are by far different than the simplified LUT implementation. The most interesting part was to study the trade-offs that these implementations introduce among response time, implementation cost and dynamic behavior. These trade-offs were explicitly researched within the case study presented in Chapter 4.

The LUT implementation is perfectly suitable when the stage of clustering produces a dataset of RTSs and Scenarios that are manageable in terms of size. The final product is a circuit that performs input – output mapping in order to identify the coded Scenario at every moment. We use compression techniques to reduce its size and complexity, while exploiting the advantages of modern synthesizers which have the capability to handle and simplify large logic functions.

An alternative solution which enables Neural Networks as detectors is introduced and thoroughly examined through its various aspects. The specific implementation takes advantage of the well – known ability of neural networks to generalize via training and thus provide correct output results for unknown data. Migration of Neural Networks from conventional processors to hardware platforms boosts their performance, but it is always a demanding and complicated task, so much effort was put on to optimize the parameters of the Neural Network so as to adapt in a more efficient way into Hardware environment. In order to achieve a highly flexible solution, there was developed a special software along with a graphical user interface, which acts as a Neural Network generator. Experimenting with various parameters of the Hardware implementation enables us to come to useful conclusions as far as the trade-offs are concerned.

Finally, a full methodology is introduced which targets to evaluate by using specific measurements such as response time and chip area, the tradeoffs among the different variations of implementing the scheme of detection. This methodology is analyzed and explained step by step in its theoretical level in Chapter 3, while Chapter 4 contains analytical results of the Case Studies in which the methodology was tested.

The flowchart of the described methodology is given in Figure 1.4, where each step is presented in a separate box. The main idea behind this methodology is to generate an optimal Scenario Detection solution, according to the user's desired style of implementation. Unlike the static implementation, which is as simple as it is shown, with only few sequential steps required, the finding of the optimal dynamic implementation demands a repetitive process, which summarizes in the following steps :

i) Normalize the values of RTS Parameters
ii) Define specific combination of RTS values that do not trigger a change in Scenarios (optional)
iii) Choose the size of the hidden layer and train the Network using the largest fraction of the Dataset.
iv) Simulate the Neural Network using the whole Dataset.
v) Evaluate the prediction percentage and compare with the previous measurement. If a better prediction is achieved, repeat the process adding nodes. If not, recall the previous instantiation and proceed to the next step.
vi) The optimal solution of the implementation is achieved, and is followed by the sequential steps of Synthesis, Implementation and Bitstream Generation.

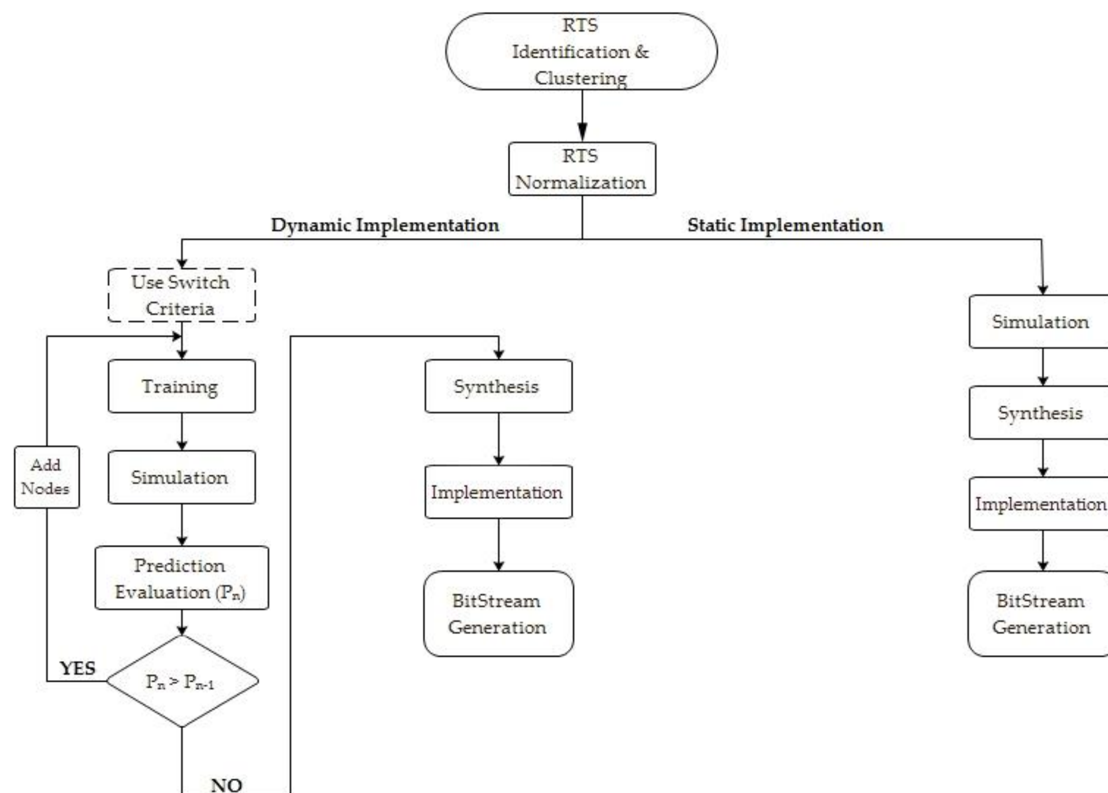


Figure 1.4 Flowchart of the proposed Methodology

Chapter 2 Neural Networks

2.1 Overview

Today's computers can perform complicated calculations, handle complex control tasks and store huge amounts of data [24]. However, there are classes of problems which a human can solve easily, but a computer can only process with high effort. Examples are character recognition, image interpretation or text reading. These kinds of problems have in common, that it is difficult to derive a suitable algorithm.

Unlike computers, the human brain can adapt to new situations and enhance its knowledge by learning. It is capable to deal with incorrect or incomplete information and still reach the desired result. This is possible through adaption. There is no predefined algorithm, instead new abilities are learned. No theoretical background about the problem is needed, only representative examples.

The neural approach is beneficial for the above addressed classes of problems. The technical realization is called neural network or artificial neural network. They are simplified models of the central nervous system and consist of intense interconnected neural processing elements. The output is modified by learning. It is not the goal of neural networks to recreate the brain, because this is not possible with today's technology. Instead, single components and function principles are isolated and reproduced in neural networks.

The development of artificial neural networks began approximately 60 years ago but early successes were overshadowed by rapid progress in digital computing. Also, claims made for capabilities of early models of neural networks proved to be exaggerated, casting doubts on the entire field.

Recent renewed interest in neural networks can be attributed to several factors. Training techniques have been developed for the more sophisticated network architectures that are able to overcome the shortcomings of the early, simple neural networks. High-speed digital computers make the simulation of neural processes more feasible. Technology is now available to produce specialized hardware for neural networks. However, at the same time that progress in traditional computing has made the study of neural networks easier, limitations encountered in the inherently sequential nature of traditional computing have motivated some new directions for neural network research.

Neural networks are of interest to researchers in many areas for different reasons [12]. Electrical engineers find numerous applications in signal processing and control theory. Computer engineers are intrigued by the potential for hardware to implement neural networks efficiently and by applications of neural networks to robotics. Computer scientists find that neural networks show promise for difficult

problems in areas such as artificial intelligence and pattern recognition. For applied mathematicians, neural networks are a powerful tool for modeling problems for which the explicit form of the relationships among certain variables is not known.

Biological Inspiration

The model for the neural processing elements is nerve cells. A human brain consists of about 10^{11} of them. All biological functions—including memory—are carried out in the neurons and the connections between them. The basic structure of a neuron cell is given in Figure 2.1.

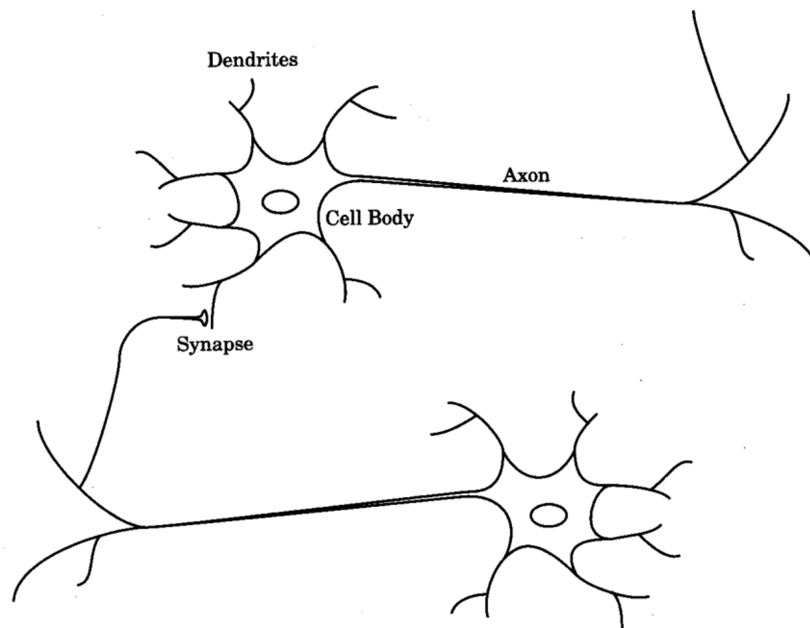


Figure 2.1. Schematic drawing of biological neurons

Dendrites	Carry electric signals from other cells into the cell body
Cell Body	Sum and threshold the incoming signals
Axon	Signal transfer to other cells
Synapse	Contact point between axon and dendrites

Every neuron receives electrochemical impulses from multiple sources, like other neurons and sensor cells. The response is an electrical impulse in the axon which is transferred to other neurons or acting organs, such as muscles. Every neuron features about 100–10.000 connections.

There are two types of synapses: excitatory and inhibitory. The neural activity depends on the neuron's intrinsic electric potential. Without stimulation, the potential rests at about −70mV. It is increased (excitatory synapse) or decreased (inhibitory synapse) by the collected inputs. When the sum of all incoming potentials exceeds the threshold of the neuron, it will generate an impulse and transmit it over the axon to other cells.

The interaction and functionality of biological neurons is not yet fully understood and still a topic of active research. One theory about learning in the brain suggests metabolic growth in the neurons, based on increased activity. This is expected to influence the synaptic potential.

2.2 Neural Network Fundamentals

2.2.1 Definition

Neural Network is an interconnected group of artificial neurons that uses a mathematical or computational model for information processing based on a connectionist approach to computation [24]. To achieve good performance, neural networks employ a massive interconnection of simple computing cells referred to as "neurons" or "processing units." We may thus offer the following definition of a neural network viewed as an adaptive machine:

“A neural network is a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:
1. Knowledge is acquired by the network from its environment through a learning process.
2. Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.”

The procedure used to perform the learning process is called a *learning algorithm,* the function of which is to modify the synaptic weights of the network in an orderly fashion to attain a desired design objective.

Each neuron is connected to other neurons by means of directed communication links, each with an associated weight. The weights represent information being used by the net to solve a problem. Each neuron has an internal state, called its activation or activity level, which is a function of the inputs it has received. Typically, a neuron sends its activation as a signal to several other neurons. It is important to note that a neuron can send only one signal at a time, although that signal is broadcast to several other neurons.

For example, consider a neuron Y, illustrated in Figure 2.2, that receives inputs from neurons X_1, X_2 and X_3. The activations (output signals) of these neurons are X_1, X_2, and X_3 respectively. The weights on the connections from X_1, X_2 and X_3 to neuron Y are W_1, W_2, and W_3, respectively. The net input, y_in, to neuron Y is the sum of the weighted signals from neurons X_1, X_2 and X_3, i.e., $y_in = w_1x_1 + w_2x_2 + w_3x_3$ [Eq 2.1]. The activation y of neuron Y is given by some function of its net input, $y = f(y_in)$

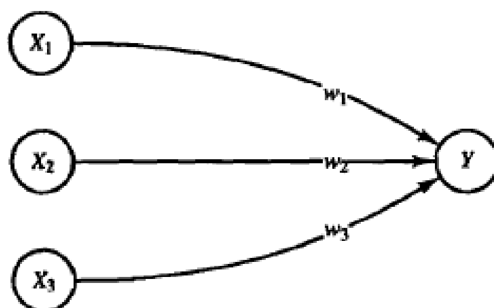


Figure 2.2. A simple (artificial) neuron

Common transfer functions fall into the following categories:

Linear The simplest case. Examples are identity and linear function with saturation.
Threshold A threshold function generates binary outputs. Unipolar or bipolar coding is possible. Another name is hard limit function.
Sigmoid Functions in the sigmoid class are continuous, differentiable, monotone and have a limited co-domain, usually in the range of [0;1] or [−1;1]. Examples are logistic function and the sigmoid function itself.

2.2.2 Characteristics

Artificial neural networks, apart from their complex structure, are encountered in literature in a huge variation of architecture and implementation aspects. However, we could highlight their main common attributes and briefly explain them [13].
Learning Neural Networks must be trained to learn an internal representation of the problem.
Generalization This attribute refers to the neural network producing reasonable outputs for inputs not encountered during training (learning). This information-processing capability makes it possible for neural networks to solve complex (large-scale) problems.

Associative Storage Information is stored according to its content.

Distributed Storage The redundant information storage is distributed over all neurons.

Robustness Sturdy behavior in the case of disturbances or incomplete inputs.

Performance Massive parallel structure which is highly efficient.

VLSI Implementability The massively parallel nature of a neural network makes it potentially fast for the computation of certain tasks. This same feature makes a neural network well suited for implementation using very-large-scale-integrated

(VLSI) technology. One particular beneficial virtue of VLSI is that it provides a means of capturing truly complex behavior in a highly hierarchical fashion [1000].

2.2.3 Network Architecture

The performance of neural networks originates from the connection of individual neurons to a network structure which can solve more complex problems than the single element. Literature [25] suggests that it is possible to distinguish between two network topologies:

1. Feed – forward networks
   - First Order
   - Second Order
2. Recurrent networks

They are illustrated in Fig 2.4.

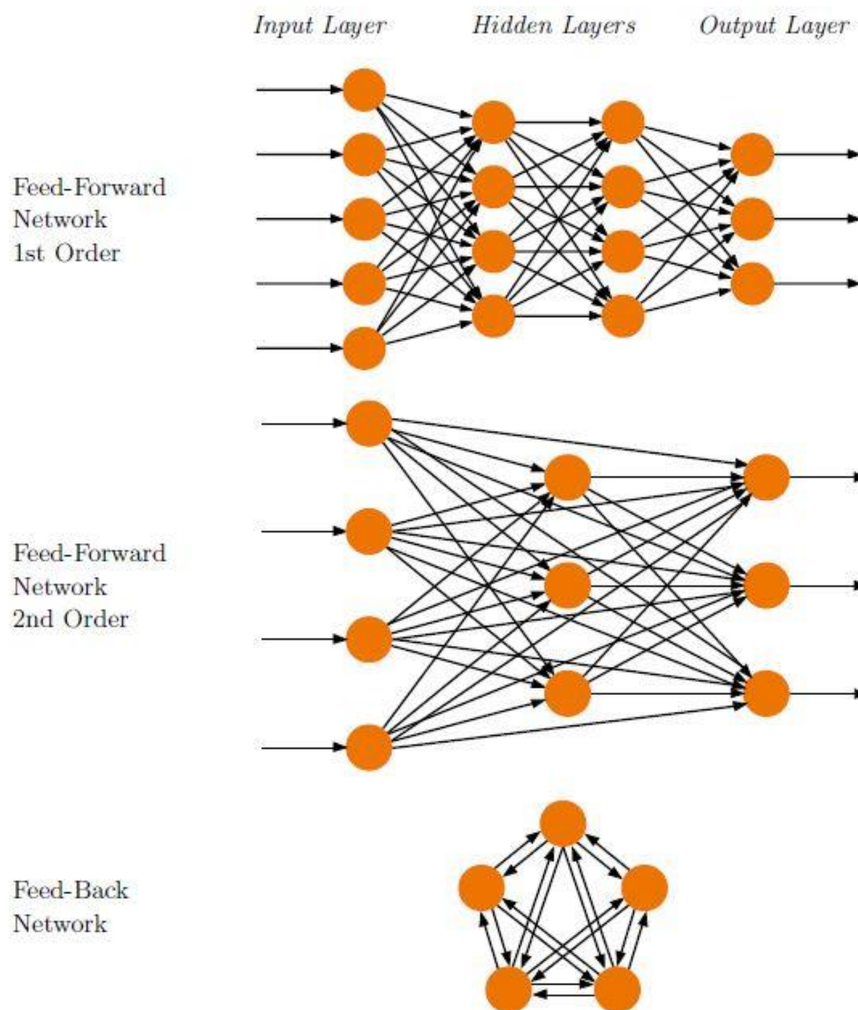


Figure 2.4 Neural Networks Architectures

1. Feed-Forward Networks

Feed-forward networks organize the neurons in layers. Connections are only allowed between neurons in different layers and must be directed toward the network output. Connections between neurons in the same layer are prohibited. Feed-forward networks of first order only contain connections between neighboring layers. In contrast, second order networks permit connections between all layers.

The network inputs form the input layer. This layer does not include real neurons and therefore has no processing ability. It only forwards the network inputs to other neurons. The output layer is the last layer in the network and provides the network outputs. Layers in between are called hidden layers, because they are not directly reachable from the outside.

2. Recurrent Networks

Opposite to feed-forward, recurrent networks also allow connections from higher to lower layers and inside the same layer. In many cases, the organization into layers is completely dropped. For example, a recurrent network may consist of a single layer of neurons with each neuron feeding its output signal back to the inputs of all the other neurons. The presence of feedback loops has a profound impact on the learning capability of the network and on its performance. Moreover, the feedback loops involve the use of particular branches composed of unit-delay elements which result in a nonlinear dynamical behavior, assuming that the neural network contains nonlinear units.

2.3 Neural Network Types

2.3.1 Overview

There are many different neural network types which vary in structure, application area or learning method. Among them the networks in the following page should be presented here. They were selected according to their significance and to show the neural network variety.

2.3.2 Perceptron

The Perceptron neuron was introduced 1958 by Frank Rosenblatt [26]. It is the oldest neuronal model which was also used in commercial applications. Perceptrons could not be connected to multi-layered networks because their training was not possible yet. The neuron itself implements a threshold function with binary inputs and outputs. It is depicted in Figure 2.5.

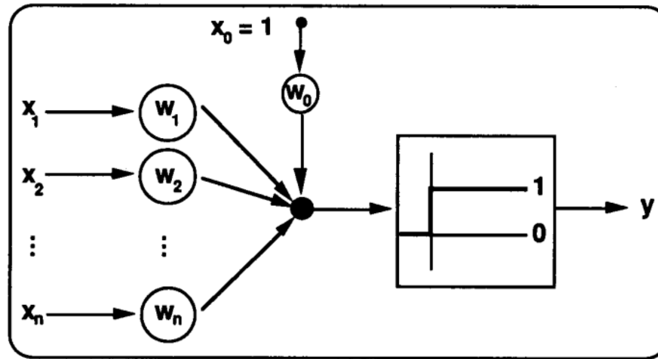


Figure 2.5 Perceptron Neuron

Neuron training is possible with different supervised learning methods e.g. perceptron learning rule, Hebb rule or delta rule. The Perceptron can only handle linear separable problems. Graphically speaking, the problems are separated by a line for 2 inputs or by a plane for 3 inputs, as visualized in Figure 2.6.

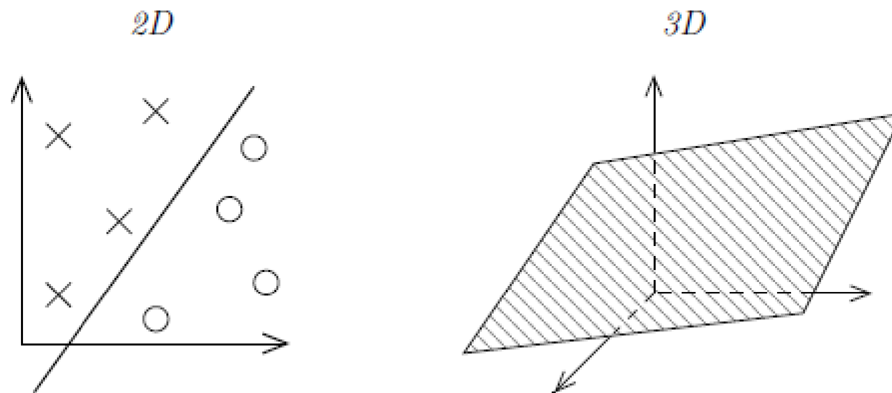


Figure 2.6 Linear separable problems

2.3.3 ADELINE, MADELINE

The ADALINE is also a single neuron which was introduced in 1960 by Bernhard Widrow. "ADALINE" stands for "Adaptive Linear Neuron" and "Adaptive Linear Element", respectively.

The ADALINE neuron implements a threshold function with bipolar output. Later it was enhanced to allow continuous outputs. Inputs are usually bipolar, but binary or continuous inputs are also possible. In functionality it is comparable to the Perceptron. The major field of application is adaptive filtering, as shown in Figure 2.7. The neuron is trained with the delta rule.

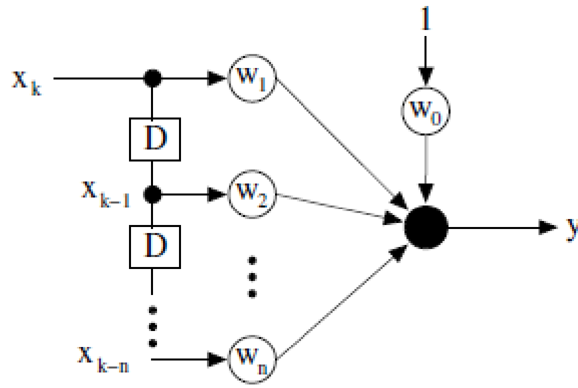


Figure 2.7 ADALINE neuron as adaptive filter

MADALINE

“MADALINE” spells “Many ADALINEs” – many ADALINEs whose outputs are combined by a mathematical function. This approach is visualized in Figure 2.8. MADALINE is no multi-layered network, because the connections do not carry weight values. Still, through the combination of several linear classification borders more complex problems can be handled. The resulting area shape is presented in Figure 2.9.

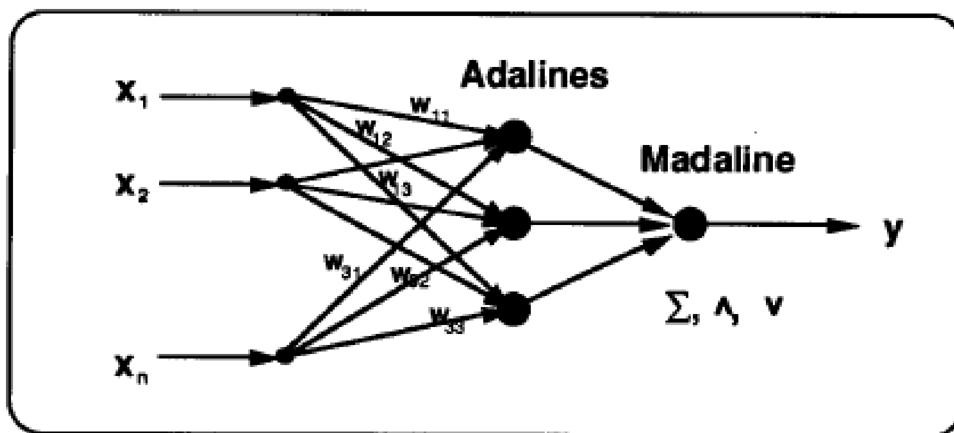


Figure 2.8 MADALINE

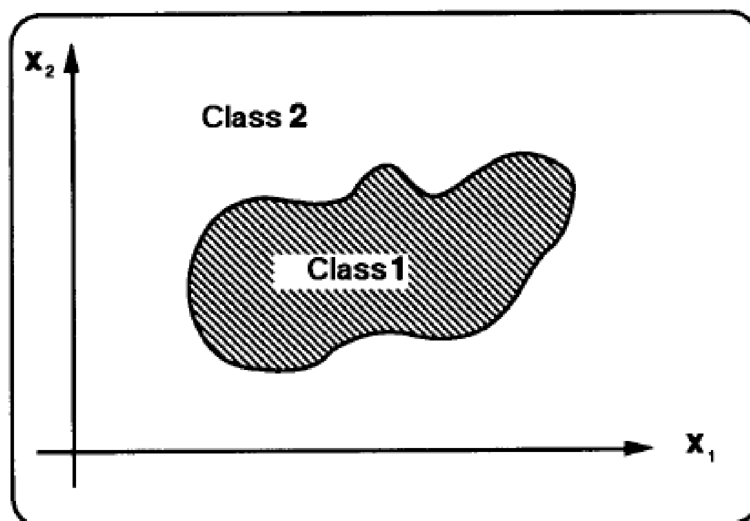


Figure 2.9 Complex contiguous classification areas

2.3.4 Backpropagation

The most popular neural network type is the Backpropagation network. It is widely used in many different fields of application and has a high commercial significance. Backpropagation was first introduced by Paul Werbos in 1974 [27]. Until then it was impossible to deal with disjointed complex classification areas, like the ones in Figure 2.10. For this purpose hidden layers are needed, but no training method was available. The Backpropagation algorithm now enables training of hidden layers.

The term "Backpropagation" names the network topology and the corresponding learning method. In literature, the network itself is often called "Multi-Layer Perceptron Network". The Backpropagation network is a feed-forward network of either 1st or 2nd order. The neuron type is not fixed, only a sigmoid transfer function is required.

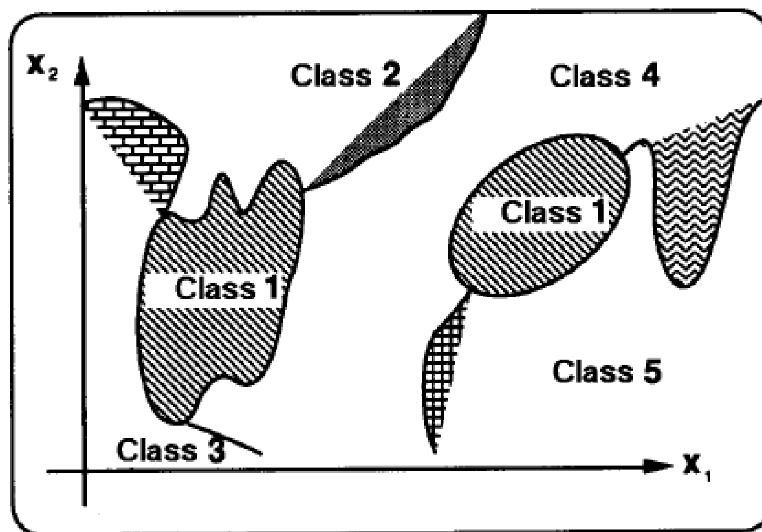


Figure 2.10 Disjointed complex classification areas

Standard Backpropagation learns very slow and possibly reaches only a local minimum. Therefore variants exist which try to improve certain aspects of the algorithm [28, Chapter 12].

2.3.5 Hopfield

The Hopfield network was presented in 1982 by John Hopfield [29]. It is the most popular neural network for associative storage. It memorizes a number of samples which can also be recalled by disturbed versions of themselves. This is exemplarily depicted in Figure 2.11.

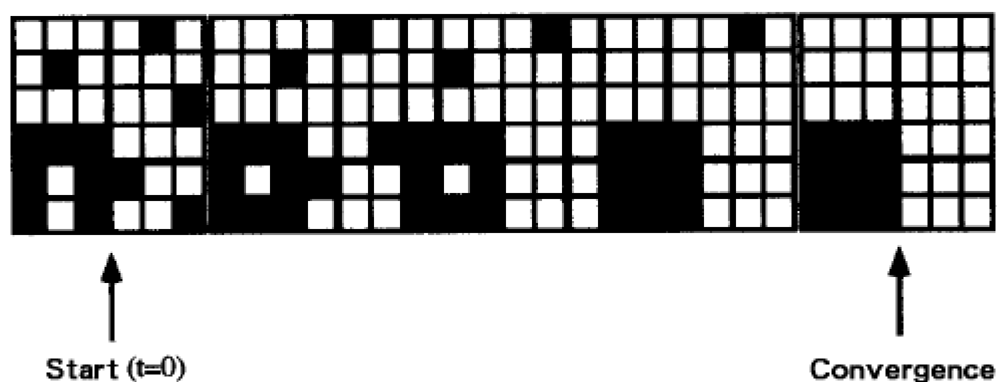


Figure 2.11 Associative pattern completion

The structure is sketched in Figure 2.12. It is a feed-back network, where every neuron is connected to all other neurons. The connection weights between two neurons are equal in both directions. The neuron implements a binary or bipolar threshold function. The input and output co-domains match the threshold function type.

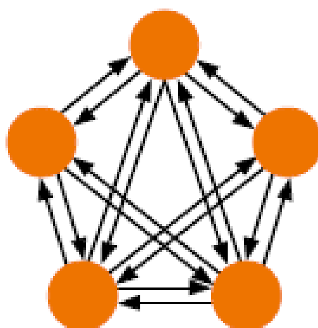


Figure 2.12 Hopfield Network

Learning is possible by calculating the weight values according to the Hopfield learning rule.

2.3.6 ART

Adaptive Resonance Theory (ART) is a group of networks which have been developed by Stephen Grossberg and Gail Carpenter since 1976. ART networks learn unsupervised by subdividing the input samples into categories. Most unsupervised learning methods suffer the drawback that they tend to forget old samples, when new ones are learned. In contrast, ART networks identify new samples which do not

fit into an already established category. Then a new category is opened with the sample as starting point. Already stored information is not lost.

The disadvantage of ART networks is their high complexity which arises from the elaborate sample processing. The structure is presented in Figure 2.13. Various versions of ART networks exist which differ in structure, operation and input value co-domain.

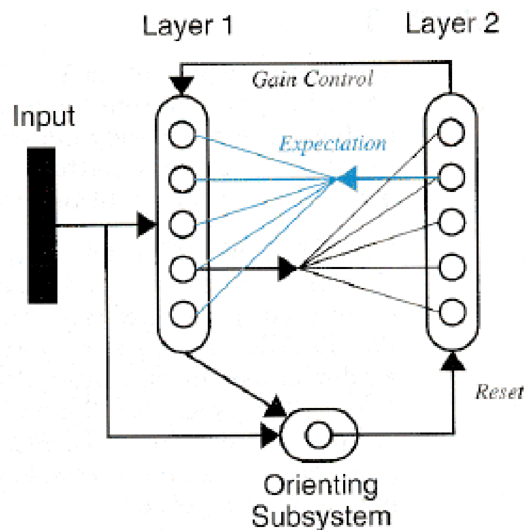


Figure 2.13 ART Network [28, p.16-3]

2.3.7 Cascade Correlation

The Cascade Correlation network was developed in 1990 by Scott E. Fahlman and Christian Lebiere [30]. It is an example of a growing network structure. Usually it is difficult to find a suitable network structure for a given problem. In the majority of cases try-and-error is used, possibly supported by heuristic methods. In Cascade Correlation networks the structure is part of the training process. Starting from the minimal network, successive new neurons are added in hidden layers. The new neurons are trained while previously learned weights are kept. The overall network structure is feed-forward 2nd order as depicted in Figure 2.14.

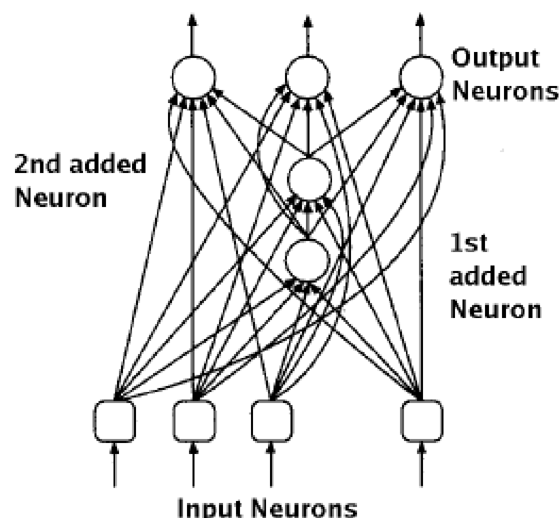


Figure 2.14 Cascade Correlation Network

2.4 Fundamentals of Learning and Training functions

2.4.1 Learning Methods

The most interesting characteristic of neural networks is their capability to familiarize with problems by means of training and, after sufficient training, to be able to solve unknown problems of the same class. This approach is referred to as generalization. We introduce some essential paradigms of learning by presenting the differences between their regarding training sets. A training set is a set of training patterns, which we use to train our neural network.

Unsupervised Learning It is the biologically most plausible method, but is not suitable for all problems. Only the input patterns are given; the network tries to identify similar patterns and to classify them into similar categories. The training set only consists of input patterns, the network tries by itself to detect similarities and to generate pattern classes. The most popular example is Kohonen's self-organizing maps [31], [32].

Reinforcement Learning In this specific type of learning the network receives a logical or a real value after network receives reward or punishment completion of a sequence, which defines whether the result is right or wrong. Intuitively it is clear that this procedure should be more effective than unsupervised learning since the network receives specific criteria for problem-solving. The training set consists of input patterns, after completion of a sequence a value is returned to the network indicating whether the result was right or wrong and, possibly, how right or wrong it was.

Supervised Learning In supervised learning the training set consists of input patterns as well as their correct results in the form of the precise activation of all output neurons. Thus, for each training set that is fed into the network the output, for instance, can directly be compared with the correct solution and the network weights can be changed according to their difference. The objective is to change the weights to the effect that the network cannot only associate input and output patterns independently after the training, but can provide plausible results to unknown, similar input patterns, i.e. it generalizes.

2.4.2 Training Functions

Supervised learning suggests that there must be a defined pattern (training function) based on which, a neural network is trained and adjusts the value for its weights. The scheme for this procedure is as follows :

- Entering the input pattern (activation of input neurons)
- Forward propagation of the input by the network, generation of the output
- Comparing the output with the desired output (teaching input), provides error vector (difference vector)
- Corrections of the network are calculated based on the error vector
- Corrections are applied.

2.4.2.1 Levenberg Marquadt Algorithm

The Levenberg – Marquadt algorithm is a numerical optimization method, more specifically it is a variation of Newton's method that was designed for minimizing functions that are sums of squares of other nonlinear functions. This is very well suited to neural network training where the performance index is the mean squared error. A flowchart of the algorithm is presented in following figure, while analytical mathematical background is provided in Appendix A.

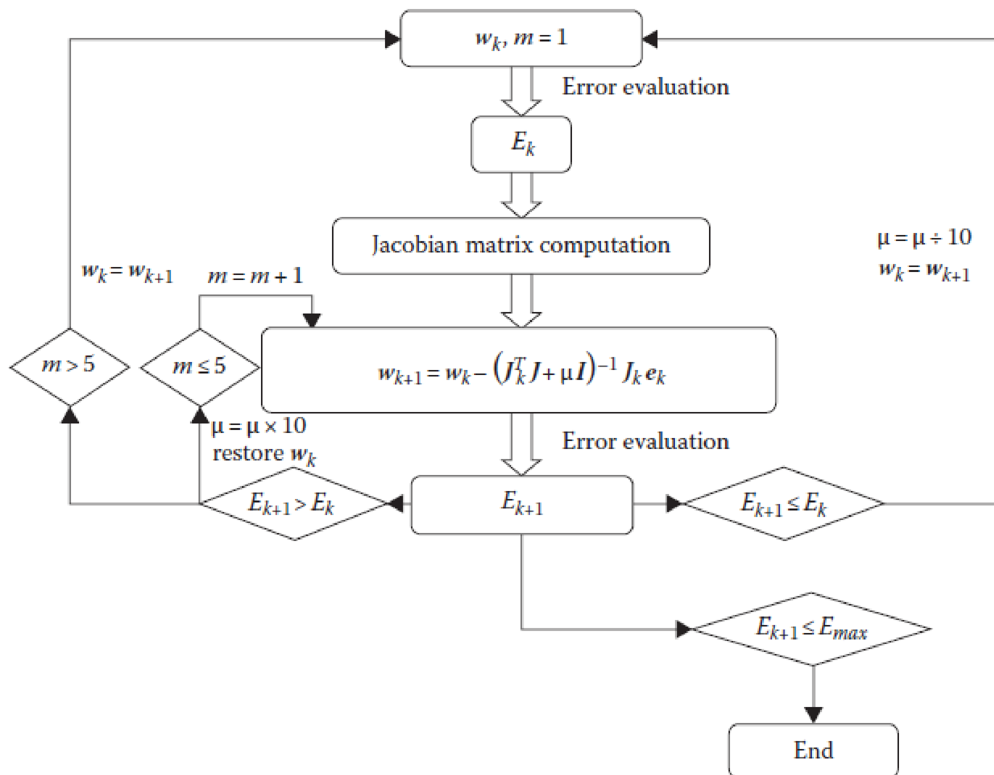


Figure 2.15 Block diagram for training using Levenberg–Marquardt algorithm [23]

Therefore, the training process using Levenberg–Marquardt algorithm could be designed as follows:

i. With the initial weights (randomly generated), evaluate the total error (SSE).
ii. Do an update as shown in the Equation to adjust weights.
iii. With the new weights, evaluate the total error.
iv. If the current total error is increased as a result of the update, then retract the step (such as reset the weight vector to the precious value) and increase combination coefficient μ by a factor of 10 or by some other factors. Then go to step ii and try an update again.
v. If the current total error is decreased as a result of the update, then accept the step (such as keep the new weight vector as the current one) and decrease the combination coefficient μ by a factor of 10 or by the same factor as step iv.
vi. Go to step ii with the new weights until the current total error is smaller than the required value.

2.5 Hardware adaptation of Neural Networks

2.5.1 Hardware Platforms Overview

With the passing of time, integrated circuit (IC) technology has provided a variety of implementation formats for system designers [14]. The implementation format defines the technology to be used, how the switching elements are organized and how the system functionality will be materialized. The implementation format also affects the way systems are designed and sets the limits of the system complexity. Today the majority of IC systems are based on complementary metal-oxide semiconductor (CMOS) technology. In modern digital systems, CMOS switching elements are prominent in implementing basic Boolean functions such as AND, OR, and NOT. With respect to the organization of switching elements, regularity and granularity of elements are essential parameters. The regularity has a strong impact on the design effort, because the reusability of a fairly regular design can be very simple. The problem raised by the regularity is that the structure may limit the usability and the performances of the resource. The granularity expresses the level of functionality encapsulated into one design object. Examples of fine-grain, medium-grain, and coarse-grain are logic gates, arithmetic and logic units (ALUs), and intellectual property components (processor, network interfaces, etc.), respectively. The granularity affects the number of required design objects and, thereby, the required design or integration effort.

Depending on how often the structure of the system can be changed, the three main approaches for implementing its functionality are dedicated systems, reconfigurable systems, and programmable systems. In a dedicated system, the structure is fixed at the design time, as in application-specific integrated circuits (**ASIC**s). In programmable systems, the data path of the processor core, for example, is

configured by every instruction fetched from memory during the decode-phase. The traditional microprocessor-based computer is the classical example. In reconfigurable systems, the structure of the system can be altered by changing the configuration data, as in field programmable gate arrays (**FPGA**s).

2.5.2 ASIC

Application-specific integrated circuits (ASICs) refer to those integrated circuits specifically built for preset tasks [6]. Why use an ASIC solution instead of another off-the-shelf technology—programmable logic device (PLD, FPGA), or a microprocessor/microcontroller system? There are, indeed, many advantages in ASICs with respect to other solutions: increased speed, lower power consumption, lower cost (for mass production), better design security (difficult reverse engineering), better control of I/O characteristics, and more compact board design (less complex PCB, less inventory costs). However, there are important disadvantages: long turnaround time from silicon vendors (several weeks), expensive for low-volume production, very high NRE cost (high investment in CAD tools, workstations, and engineering manpower), and, finally, once committed to silicon the design cannot be changed. Application-specific components can be classified into full-custom ASICs, semi-custom ASICs, and field programmable ICs.

2.5.3 FPGA

The field-programmable gate array (FPGA) is a semiconductor device that can be programmed after manufacturing. Instead of being restricted to any predetermined hardware function, an FPGA allows you to program product features and functions, adapt to new standards, and reconfigure hardware for specific applications even after the product has been installed in the field—hence the name "field-programmable". You can use an FPGA to implement any logical function that an application-specific integrated circuit (ASIC) could perform, but the ability to update the functionality after shipping offers advantages for many applications.

Unlike previous generation FPGAs using I/Os with programmable logic and interconnects, today's FPGAs consist of various mixes of configurable embedded SRAM, high-speed transceivers, high-speed I/Os, logic blocks, and routing. Specifically, an FPGA contains programmable logic components called logic elements (LEs) and a hierarchy of reconfigurable interconnects that allow the LEs to be physically connected. You can configure LEs to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flipflops or more complete blocks of memory.
As FPGAs continue to evolve, the devices have become more integrated. Hard intellectual property (IP) blocks built into the FPGA fabric provide rich functions while lowering power and cost and freeing up logic resources for product differentiation. Newer FPGA families are being developed with hard embedded processors, transforming the devices into systems on a chip (SoC).

Compared to ASICs or ASSPs, FPGAs offer many design advantages, including:

- Rapid prototyping
- Shorter time to market
- The ability to re-program in the field for debugging
- Lower NRE costs
- Long product life cycle to mitigate obsolescence risk

2.5.4 Neural Networks in Hardware

Pure software solutions on general-purpose processors tend to be slow because they do not take advantage of the inherent parallelism, whereas hardware realizations usually rely on optimizations that reduce the range of applicable network topologies, or attempt to increase processing efficiency by means of low-precision data representation. For the development of neural networks software simulators are sufficient. On the other hand, in production use computer based simulation is not always acceptable.

Compared to software simulation, hardware implementation benefits from the following points:

- Higher operation speed by exploring intrinsic parallelities
- Reduced system costs in high volume applications
- In stand-alone installments no PC needed for operation
- Optimization toward special operation conditions possible, e. g. small size, low power, hostile environment

The highly interconnected nature of neural networks prohibits direct structure mapping to hardware for all but very small networks. Direct mapping also requires many processing elements. In particular, one multiplier for each neuron input. Alternative approaches are required to reduce connections and hardware costs.

Classification

It is possible to split up the hardware approaches into two groups:

- Fixed network structure in hardware, targeting one particular task
- Flexible neurocomputer, suitable for many different network types and structures

Another division follows the appearance of the implementation :

Neurocomputers as complete computing systems based on neural network techniques
PC Accelerator Boards to speed up calculations in PC, either accelerating the operation of a software simulator or as stand-alone neural network PC card
Chips for system integration

Cell Libraries/IP for System-On-Chip (SoC) with the need for a neural network component
Embedded Microcomputers implementing software neural networks

2.5.5 FPGA and Neural Networks

The traditional hardware approach leads to a fixed network structure. The implementations are usually small and fast, but some applications need more flexibility. Especially in the course of development it is advantageous to evaluate a number of different implementations. This can be achieved by using Field Programmable Gate Arrays (FPGAs) which are in-system reconfigurable.

This reconfiguration feature can be exploited in a number of ways [16]:

- Rapid prototyping of different networks and parameters
- Build a multitude of neural networks and load the most appropriate one on startup
- Recent FPGAs can be reconfigured at runtime, this allows density enhancements by dynamic reconfiguration. Usually time-multiplex of different processing stages (like learning and propagation) is performed.
- Topology adaption at runtime or start-up is imaginable

FPGA implementations of neural networks have a great develop in recent years, because of its reconcilability and short design time, such as FPGA neurocomputers (Omondi et al., 2006), Arithmetic precision for implementing BP networks on FPGA (Moussa et al., 2004), FPGA Implementation of Very Large Associative Memories (Hammerstrom et al., 2006), and so on. But there remains a performance problem. If the problem could be solved, the FPGA approach will make hardware ANN a bright future.

Chapter 3 Implementation

Traditional programming languages such as C/C++ (augmented with special constructions or class libraries) are sometimes used for describing electronic circuits. They do not include any capability for expressing time explicitly and, consequently, are not proper hardware description languages. Nevertheless, several products based on C/C++ have appeared: Handel-C, System-c, and other Java-like based such as JHDL or Forge. Using a proper subset of nearly any hardware description or software programming language, software programs called synthesizers can infer hardware logic operations from the language statements and produce an equivalent netlist of generic hardware primitives to implement the specified behavior.

However, a specialized hardware description language, such as VHDL, is more suitable for an exact depiction on Hardware because it provides the designer with a higher level of control on the final netlist. Thus we choose VHDL as the language to develop our project.

In order to validate and complete the implementation we also need a Software based simulation for Neural Networks. There are many suitable software for this purpose, which allow custom Neural Network building while offering a high degree of parameterization. After experimenting with some of this Software, we arrived at the decision that MatLab is the most suitable of all. MatLab environment contains a powerful tool for Neural Networks [17], which is called "*nntool*". It can simulate various kinds of ANNs, as well as different learning methods and activation functions, already implemented in MatLab language and provided as built–in functions. This diversity was exploited by our need for a highly accurate implementation.

3.1 Implementation Aspects

3.1.1 Neural Network Architecture

As far as neural networks are concerned, their diversity is so vast, as we have already seen in Chapter 2, that we should specify the basic architecture that we are going to use for our design. Those decisions are justified in the next paragraphs.

1) Ann Structure

The problem described is purely deterministic; actually we need to build a 'black box' which will be able to resolve a complicated non-linear function. Judging from relative implementations in literature regarding Classification problems, a multilayer feedforward ANN seems the most reasonable choice to perform such a task.

2) Number of Inputs

While the number of ANN inputs is defined by the number of RTS of the dataset, what needs to be determined is the length of bits for each input. The latter is critical to the precision of our final implementation, and while the minimum amount of bits is dependent on the maximum value we encounter in the entire dataset, it is helpful to introduce a user-defined level of precision (number of bits), which will enhance the system with greater stability.

3) Number of Layers

ANNs can possibly have as many layers wanted, actually the deeper the network, the better its learning capability is. There are however, two separate factors that are determinant for the decision of the number of layers.

- It is generally proven, that a single hidden layer with the appropriate number of neurons is sufficient for an ANN that is constructed to resolve non-linear functions [18].
- The existence of two or more hidden layers puts on delay in the implementation, since there are more stages of processing from the input layer to the output neurons.

The above converge to the decision of using a *single hidden layer*.

4) Number of Output Nodes (Neurons)

A hardware implementation of input-output mapping should include an output layer which shows the stage selected by the combination of inputs. One possible implementation is to use as many neurons as the number of unique stages included in the output stage, with each neuron acting as a switch, YES('1') or NO('0'). In that case, only one neuron should be activated each time, while the others should be turned off('0').

However, there is a different approach that requires even fewer resources. This approach also involves output nodes acting as switches, but it uses the minimum number of them. The amount of output nodes is determined by the number of unique Scenarios, using the following type :
$N_OUTPUTS = ceil(\log 2(N_SCENARIOS))$.

For instance, if we were to implement an ANN for a dataset with 4 Scenarios, we would simulate our ANN with 2 output nodes.

5) Number of Hidden Nodes (Neurons)

The number of hidden nodes is a decision that we cannot be certain of. It depends on three parameters, the most important of them non measurable. Number of Inputs, Number of Outputs and last but not least, the complexity of the data.

A trial and error procedure will specify the number of hidden nodes to be used in the final implementation. Firstly, we make a rough estimation about the number. Depending on the results of the training, we modify this number. If training produces very little or no errors, we remove nodes until we reach the minimum number adequate for the ANN to be efficient. Otherwise, if training produces many errors, we add nodes until errors are minimized.

6) Activation Function

The function that seems more suitable for a hardware implementation is the logistic sigmoid function (logsig). It is a function that drives input in the range [0, 1], an attribute that is convenient because the two edges represent the two binary states. After experimentation, we also found that the specific activation function provided more accurate results when training networks in software (MATLAB), compared to the results of a) hyperbolic tangent function (tansig)
and b) combinations of tansig and logsig in hidden and output layers.

7) Training Function

Since we use a Neural Network to perform a deterministic task and not just as a predictor as its primary usage usually is, there is demand for the maximum accuracy achievable. If we chose to train our network in hardware (on-chip learning), besides the obvious difficulty, we would reduce dramatically the efficiency of the network, due to the restrictions introduced by the specification of the chips (lack of adequate memory resources, which are necessary for the sophisticated training algorithms that are used).

There is a lot of software suitable for neural network training; surely one of the most extensive is MatLab, via Neural Network Toolbox. After experimentation with some of the training functions provided, we came to Levenberg – Marquadt algorithm, which is a backpropagation variation. Its advantage is that it converges faster compared to other algorithms and its drawback is that it uses large matrixes for computations, so it requires more memory resources compared to others. However, there are no restrictions on the size of network that we can train using this algorithm.

3.1.2 Data Discretization

Most software simulators use floating point values for neural network calculation. This is not suitable for hardware implementations, because floating point computations are hardware-expensive. Fixed point data is preferred for fast and resource efficient hardware implementations. However Xilinx tools do not directly support fixed point library, as the latter became part of IEEE library only recently, in VHDL – 2008 edition, while Xilinx compilers are oriented to previous VHDL

versions. So, we have to manually add the specific libraries and add some modifications, in order to enhance better performance:

1. When specifying the rounding routine to use in fixed point operations, there are two options: *round* and *truncate*. Rounding provides more accurate results, but with the cost of added logic. So, we make the choice of truncating, while keeping in mind that we should have adequate bits so as not to lose critical information due to truncation.
2. Overflowing routine also offers two options: *Saturate* and *wrap*. Saturation is more accurate routine, but in terms of hardware consumes important resources, so we go with *wrap* option.

3.1.3 Input Normalization

Convergence in Neural Networks is usually faster if the average of each input variable over the training set is close to zero. To see this, consider the extreme case where all the inputs are positive. Weights to a particular node in the first weight layer are updated by an amount proportional to δx where δ is the (scalar) error at that node and x is the input vector. When all of the components of an input vector are positive, all of the updates of weights that feed into a node will be the same sign (i.e. sign(δ)). As a result, these weights can only all decrease or all increase *together* for a given input pattern. Thus, if a weight vector must change direction it can only do so by zigzagging which is inefficient and thus very slow.
This normalization will be performed in various ways, depending on the implementation. After instantiating many networks, we consider as most effective the normalization of input values in the range [-1,25 1,25].

3.2 Methodology

3.2.1 Overview

The following flowchart describes a methodology to create a detection scheme based on the needs of the problem and evaluate its hardware footprint. There are two separate implementations proposed, the one that is static and uses a straight – forward approach, and the one that simulates the function of a neural network, with dynamic behavior. The static implementation is ideal in cases where we are aware of all the cases of combined RTSs and the Scenario those represent. Moreover, it is applicable when this dataset of RTSs and Scenarios is kept to a relatively small size.

On the contrary, dynamic implementation with the use of an artificial neural network is by far more elastic, in terms that we have developed techniques to reduce the –already hardware expensive– produced neural network. Apart from the reduced cost, it also offers the luxury of predicting undescribed situations which resemblance other situations that have been used to train the network. This attribute

is significant, whereas it is also challenging to develop reliable training techniques so as our design will benefit from this attribute at the maximum rate.

We will specify the theoretical steps involved within these implementations and in Chapter 4 the case study will provide with those arithmetical results which are useful to perform comparisons.

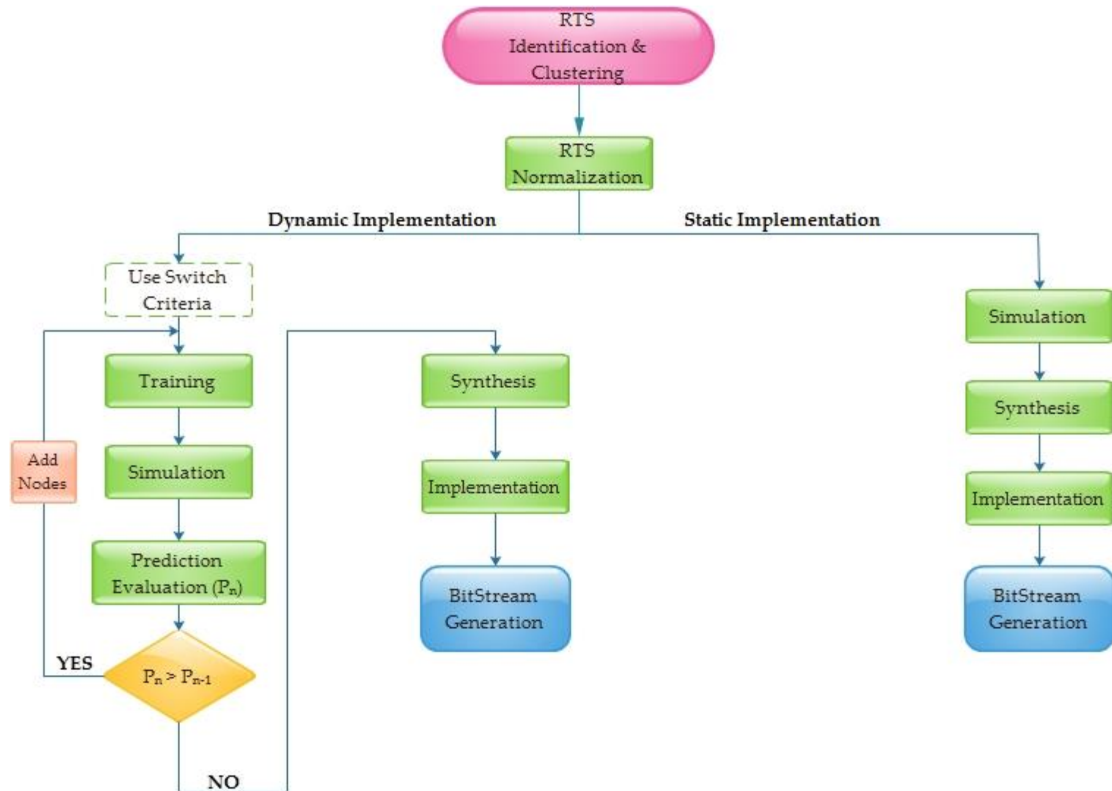


Figure 3.1 Flowchart of the proposed Methodology

3.2.2 Static Implementation

Our study concentrates on implementing a detection scheme using artificial neural network. In order to compare our main implementation with another functional one, we developed a static implementation which is consisted of the following steps:

- **RTS Identification & Clustering**

This step is common for both implementations. The extraction of RTSs out of an actual system specification and its clustering to form a limited number of Scenarios is part of System Scenarios methodology, which has been presented in Chapter 1. It is actually a demanding task which presupposes a total awareness of the parameters of the system we are going to describe. After extracting the RTS and Scenarios values, we need to present them in a proper format, which will allow us to handle them in a systematic way.

➢ **RTS Normalization**

Normalization regarding the current implementation refers to a form of compression for RTS values. It might seem insignificant, but it is actually a critical step. Scenario selection is made by traversing an array that is consisted of concatenated RTS values. If the length of that parameter exceeds a critical value, the complexity this array introduces, becomes a restraining factor, thus it may become nearly impossible for the synthesizer to implement it properly.

➢ **Simulation**

Simulation of the implementation is performed by using a testbench which is produced at the same time that the code of the detector is produced, so it is adapted to the existing parameters. If simulation finishes with zero errors, we can proceed to the next step.

➢ **Synthesis, Implementation & Bitstream Generation**

These steps, as well as Simulation, are performed within the proper Software environment. During our study, we used Xilinx ISE software to perform the current steps. The final product is the code which will be used to instantiate the respective FPGA platform.

3.2.3 Dynamic Implementation

Our main effort is towards an implementation that enables the use of neural networks. The current methodology is based on the experimental results as presented in literature and more analytically in [1000] that artificial neural networks problems match a unique number (or small range of numbers) of hidden layer nodes, to maximize their performance and avoid unwanted overtraining and over-generalization. Thus, taken this into consideration, we developed techniques for improving the performance of a neural network detector, so the next steps present the methodology that we used in order to achieve this improvement.

➢ **RTS Identification & Clustering**

This step has already been described. It is identical to that of the static implementation.

➢ **RTS Normalization**

Normalization of input variables is essential to neural networks. The values of these RTS parameters that were extracted during the RTS identification stage, need to follow that rule. The reason why we should normalize input has been explained in the previous sub-chapter and is effective in our designed neural network too.

- **Use Switch Criteria**

This step is optional. It enables a more sophisticated method of classifying, which is ruled by specific criteria, varying amongst different Scenarios. We can use this self-designed setting in order to reduce the amount of times that computations need to take place, as we can take advantage of the information provided by the criteria we hold and force the neural network to run only when it is necessary.

- **Training**

Training of the neural network is performed through a software platform, in our case MatLab. Our dataset is separated in three fragments: training, validation and testing. We use only the training fragment, which by the rules should be the largest of the three to train the network. There are various parameters that can affect the results of training. Two of the most significant factors are 1) the size of the network (the size of hidden layer should be adequate to store the non-linear relationships between input and output, but not too large, in order to prevent network from overfitting or overtraining) and 2) the complexity of the problem (whereas this factor is not measurable, it has an immense impact on the performance of training).

- **Simulation**

Evaluation of our design can be achieved through Simulation. There are two possible causes for errors during Simulation. In this critical stage, we will use the fragment of the dataset which is unknown for the network, since we did not use it during training, in order to evaluate the number of cases the network provides correct output.

- **Prediction Evaluation (P_n)**

Out of the cases presented to the network, there is a small fragment that is unknown for it as it has never been trained with these values. The percentage of accurate predictions on this fragment provides the desired outcome, which is the prediction ability of the network.

- **$P_n > P_{n-1}$**

This is the stage of decision. If the current percentage of prediction is larger than the previous measurement, we should continue the process by adding some nodes to the implementation and repeating the stages from the beginning. It is indication that there is still room for improvement for our network. If the percentage is lower though, our network is saturated, so we should seek the optimal solution in our exact previous instantiation, with fewer hidden nodes.

- **Synthesis, Implementation & Bitstream Generation**

These steps are identical to those of static implementation and form the pure technical part of the methodology.

3.2.4 Neural Networks Builder

Based on the options described previously in this Chapter, we have an outline for the project we want to build. But going deeper into its details, it is easily noticeable that the aspects of the structure are so many, and there is also a different approach matching each case. The solution on this scale of variation is to create a generator, which will describe Neural Networks in VHDL language based on the given dataset and user choices. This generator was developed in MatLab language taking into consideration the most important design aspects. Finally, a GUI was developed to provide convenience in handling the different parameters, and the set of MatLab files was compressed into a single MatLab application named "Build Neural Networks". We present the GUI environment followed by a brief explanation for each option in Appendix B.

3.3 Anatomy of the Design

3.3.1 Project Hierarchy

The produced files from MatLab App, combined together, form the Project of the Hybrid Neural Network. The network has a top-down hierarchy, so we will examine each file's underlying logic and design aspects, starting backwards, from the small, independent modules lying inside larger modules, to the bigger and more complex ones. The Project Hierarchy for the basic architecture is shown in Figure 3.2

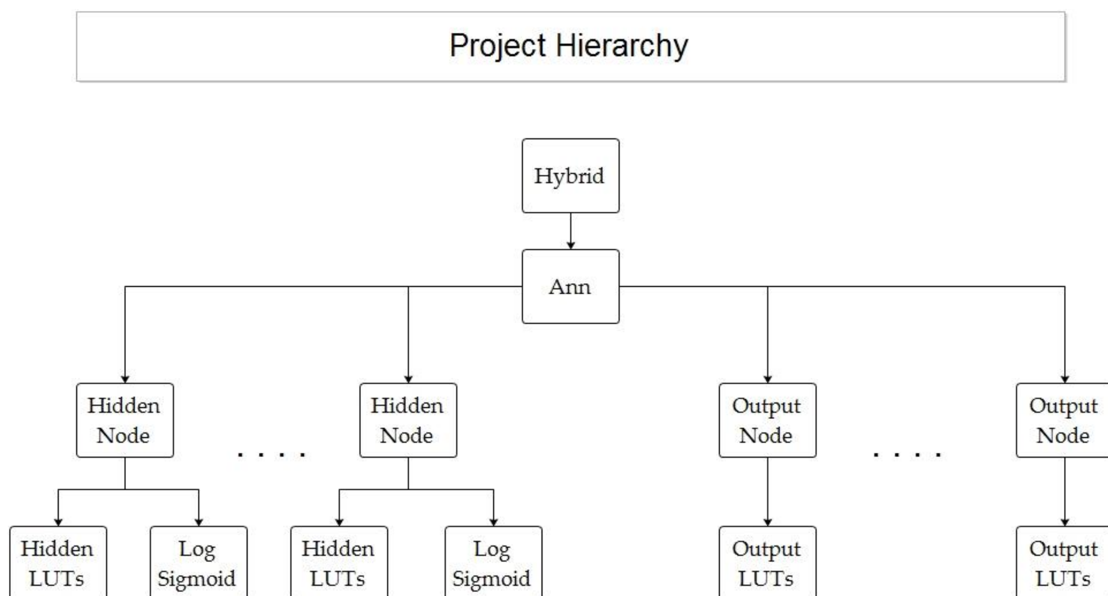


Figure 3.2 Schematic Depiction of Project Hierarchy

Two Library files, which are not depicted in the Figure, are also used in the Project. Fixed_float_types_c.vhdl and fixed_pkg_c.vhdl [33] contain various type definitions and functions regarding fixed point type representation.

Log_sigmoid, which is the implementation of activation function in Hardware, and Ann, the implementation of Artificial Neural Network, constitute the Templates of the Project, since they do not contain any data dependable on the specific dataset. The rest of VHDL files are directy dependent both on the dataset values and on user choices about neural architecture, bits precision, etc.
The rest of this chapter is dedicated to provide extensive analysis of each individual file from both functional and structural perspective. For better understanding, we cite pieces of code along with comments and explanations. Schematics are also used to project the functions and interactions between different modules. Details about schematics are provided in Appendix C.

3.4.2 Neural Library Module

This file forms a package, which is called "neural_library". This package is practically a library that contains constants, user-defined types, component declarations and functions. The usefulness of this library is that it contains, in concentrated form, elements that are used throughout the entire design. Each of the other files includes this package, by adding the line "use work.neural_library.all;" in the declaration part of the code, so they are capable to use any of the types, functions or constants contained. In other words, these are the global variables of the design.

The application produces the following Constants, which provide information about the basics of the design. Apart from *N_INPUTS*, *N_HIDDEN* and *N_OUTPUTS* which have an obvious significance, the next ones define the length of decimal numbers used. *UPPER_LIMIT* stands for the length of integer part, while *DOWN_LIMIT* stands for the length of fraction part with a negative sign. *N_BITS* is then calculated: *UPPER_LIMIT* +*DOWN_LIMIT* + Sign. The level of precision that determines those lengths is user-defined.

```
CONSTANT N_INPUTS     : integer := 5;
CONSTANT N_HIDDEN     : integer := 60;
CONSTANT N_OUTPUTS    : integer := 4;
CONSTANT N_BITS       : integer := 20;
CONSTANT UPPER_LIMIT  : integer := 10;
CONSTANT DOWN_LIMIT   : integer := -9;
```

Since we have the task to run a Neural Network, operations with decimal numbers are inevitable. For this purpose we name a new type (*fixedX)* which is a flexible type of signed fixed – point type. When we make an assignment of fixed – point number we need to specify the length of integer and fraction part. These values are already determined by the constants *UPPER_LIMIT* and *DOWN_LIMIT*. FixedX_vector is an array of fixedX numbers. Since we have frequent parallel multiplications and

additions, it is convenient to declare these values as an array, rather than separately. Finally, we define zero constant, which is an array of '0' bits.

```
subtype fixedX        IS sfixed(10 downto -9);
type fixedX_vector    IS array (INTEGER RANGE <>) OF fixedX;
CONSTANT zero         : fixedX  := (others => '0');
```

The next part of library holds the auto generated types that define the possible states of Finite State Machines. Not only each node's function is controlled by a unique FSM, but there is also a FSM to control the function of the neural network. *Hidden_node_modes* type describes the FSM that hidden nodes use. The number of states provides us with information about the number of Clock Cycles required until a hidden node completes its tasks, since they are executed sequentially. The final state is *activation_function,* where the node output is generated and sent as input to output_nodes. Respectively, *output_node_modes* type describes the FSM of output nodes. The states are *idle*, *multiply*, *accumulate*, but we have emitted *activation_function* state. The reasons for that will be explained in the appropriate chapter.

Ann_modes type represents a Finite State Machine that controls the computational row of the Neural Network. Details are provided later.

```
type hidden_node_modes IS (
      idle,
      multiply,
      accumulate_1,
      accumulate_2,
      accumulate_3,
      activation_function
  );

type output_node_modes IS (
      idle,
      multiply,
      accumulate_1,
      accumulate_2,
      accumulate_3,
      accumulate_4,
      accumulate_5,
      accumulate_6
  );

type ann_modes is (
  idle,
  run,
  run_next,
  turn_off_output
);

type node_modes is (
  idle,
  run
);
```

The following types are used to hold the values of inputs (*input_vector*, *ann_input_vector*) and the intermediate values produced by the hidden nodes (*hidden_vector*). Since our inputs are of different lengths, it would be a waste of valuable Input Ports in our targeted hardware platform, to use an array of same – length elements, so we use this record type, where each input is given exactly the amount of bits required.
Ann_input_vector also contains the values of input, but after they have been compressed. It can be seen by the cited code, that the length of inputs is significantly smaller in this case. The purpose for this compression will be discussed later.

Hidden_vector is the type we need in order to hold the intermediate values produced after the hidden nodes have completed processing, the so – called hidden output. The size of the array is the number of hidden nodes (60 in this case).

```
type hidden_vector IS array (1 TO 60) OF
STD_LOGIC_VECTOR(7 downto 0);

type input_vector IS
  record
    one    : STD_LOGIC_VECTOR(7 downto 0);
    two    : STD_LOGIC_VECTOR(4 downto 0);
    three    : STD_LOGIC_VECTOR(2 downto 0);
    four    : STD_LOGIC_VECTOR(7 downto 0);
    five    : STD_LOGIC_VECTOR(1 downto 0);
  end record;

type ann_input_vector IS
  record
    one    : STD_LOGIC_VECTOR(2 downto 0);
    two    : STD_LOGIC_VECTOR(3 downto 0);
    three    : STD_LOGIC_VECTOR(1 downto 0);
    four    : STD_LOGIC_VECTOR(1 downto 0);
    five    : STD_LOGIC_VECTOR(0 downto 0);
  end record;
```

Weights and biases are the structural elements of an artificial neural network. They are the stored "knowledge" of the machine. All operations involve the use of input values, weights and biases. The application stores the values of biases after the training, converts them in binary representation and prints them in the library as a *fixedX_vector* constant. Weight values are provided to the network in a different way, in which we will also refer to.

```
CONSTANT hidden_bias : fixedX_vector(1 to 60) := (

                        "11111001110100111000",

                        "00000001101110100111",

                        "00000001000010110011",

                        "00000100001011000110",

                        "00000001000010100010",

                        "11111110010101010110",
                        .
                        .
                        .

                        "00000001001001011010");
CONSTANT output_bias : fixedX_vector(1 to 4) := (

                        "11111111100111100011",

                        "11111101011000011110",

                        "00000010010101011110",

                        "00000101001111011001");
```

These types and constants shown next, are exclusively used during Simulation process. We will make a brief reference, as they are not directly used for the design.

N_EXAMPLES shows the size of the dataset we made use of. *Latency* constant stores the number of Clock Cycles in which the final implementation will perform and is presented for informative reasons.
Input_bitvector and *ann_input_bitvector* contain exactly the same information as *input_vector* and *ann_input_vector,* but we use BIT type instead of STD_LOGIC, because STD_LOGIC type cannot perform specific operations we need during the Simulation. *Vector_length* shows the length of a vector, which is the outcome of the concatenation of all inputs. This vector will be used to form the complementary LUTs that "fix" the errors of the Network.

Counters, *MATRIXES* and *ERROR_MATRIXES* pertain to measure the errors and hold the data of the network when such an error is produced.

```
CONSTANT N_EXAMPLES : integer := 2560;
CONSTANT latency    : integer := 20;

type input_bitvector IS
  record
    one    : BIT_VECTOR(7 downto 0);
    two    : BIT_VECTOR(4 downto 0);
    three   : BIT_VECTOR(2 downto 0);
    four   : BIT_VECTOR(7 downto 0);
    five   : BIT_VECTOR(1 downto 0);
  end record;

type ann_input_bitvector IS
  record
    one    : BIT_VECTOR(2 downto 0);
    two    : BIT_VECTOR(3 downto 0);
    three   : BIT_VECTOR(1 downto 0);
    four   : BIT_VECTOR(1 downto 0);
    five   : BIT_VECTOR(0 downto 0);
  end record;

CONSTANT vector_length   : integer := 12;
type counters        IS ARRAY(1 to 4) OF INTEGER;
type MATRIXES        IS ARRAY(1 to 1000) OF
  BIT_VECTOR(vector_length-1 downto 0);
type ERROR_MATRIXES IS ARRAY(1 to 4) OF MATRIXES;
```

There are also various Component Declarations in neural_library package, but it would be meaningless to refer to extensively, as they are simple copies of their respective Entities. Finally, two simple functions, *to_sl* and *to_slv* convert BIT types to STD_LOGIC and BIT_VECTOR types to STD_LOGIC_VECTOR. They are also exclusively used during simulation.

3.4.3 Log Sigmoid Module

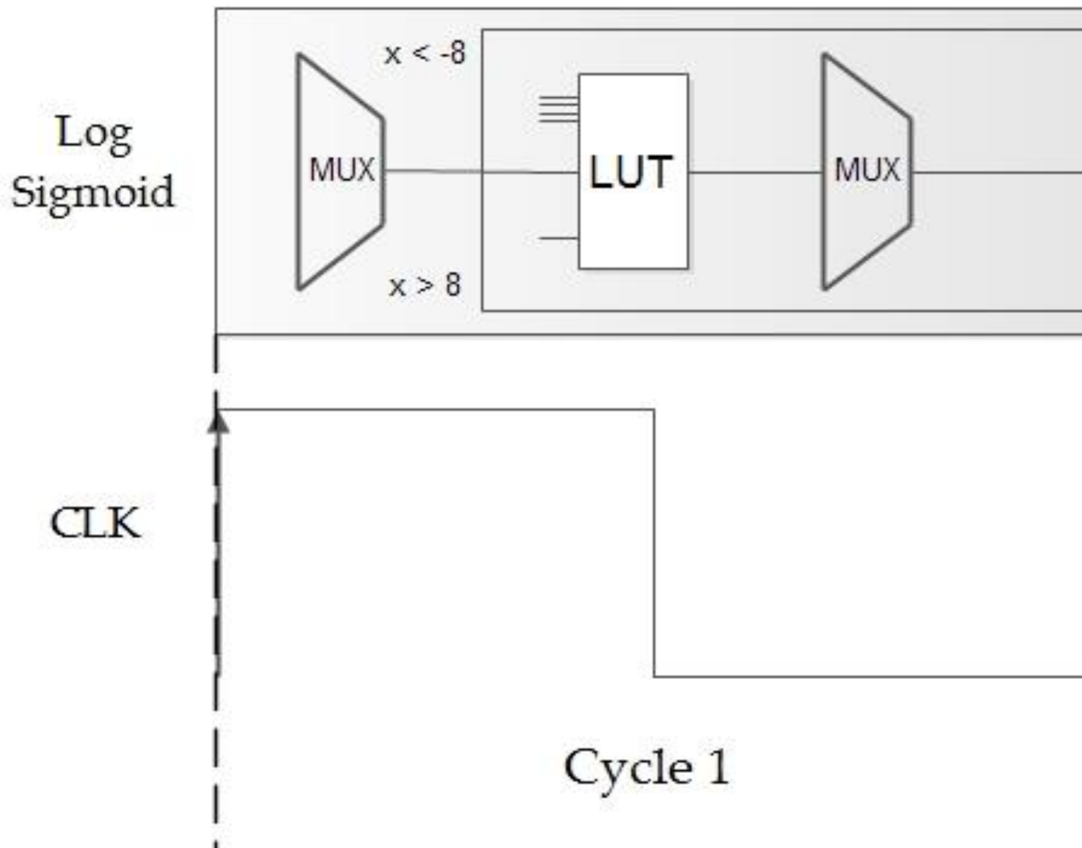


Figure 3.3 Log Sigmoid Module

The Schematic in Fig. 3.3 depicts the behavioral flow of the log_sigmoid module, which performs the activation function. Logistic sigmoid function is the following : $f(x) = \frac{1}{1+e^{-x}}$, and its plot is shown next

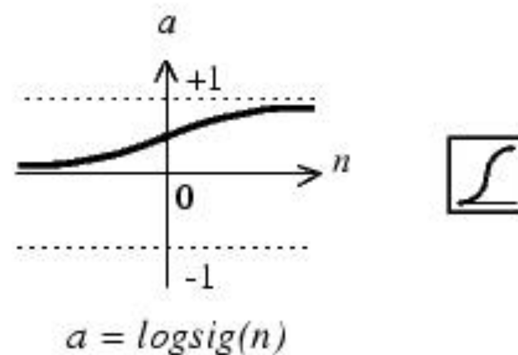


Figure 3.1 Log-Sigmoid Transfer Function

Since we need to use the specific function in a hardware implementation, we should find ways to avoid the "expensive" in terms of logic and time consuming operations of division and exponential. Among many implementations of logsig function found in literature, we ended up in the most suitable for our purpose, which is described in [19].

This design takes advantage of two basic attributes of the function:

- $f(x) =$ practically 0 if $x \le -8$, and $f(x) =$ practically 1, if $x \ge 8$.
- It is a symmetrical function, $f(-x) = 1 - f(x)$.

Of course, it is a clocked function which begins when the enable bit that arrives from the node FSM is '1'.

```
PROCESS(CLK) IS

BEGIN
  IF (CLK'event) AND (CLK = '1') THEN
    IF (enable = '1') THEN
```

Next lines of code perform comparisons of input with -8 and 8. Instead of simply using comparing signs (< , >) which would enable substractors in the underlying hardware circuit, we use some clever logic.

- *Temp1* variable is '1' only when all bits from the sign bit downto the 4th Least Significant bit of the Integer part are '1', since it uses 2's complement to represent negative numbers. Yet, in that case input is no smaller than -8.
- *Temp2* variable is '0' only when all bits from the sign bit down to the 4th Least Significant bit of the Integer part are '0'. Yet, in that case input is no greater than 8.

If input meets one of these conditions, output is given and process is terminated. If not, the process goes to second computational stage.

```
temp1 := '1';
temp2 := '0';
FOR i IN 3 to UPPER_LIMIT LOOP
  temp1 := temp1 AND input(i);
  temp2 := temp2 OR input(i);
END LOOP;

smaller_than_8 := temp1;
greater_than_8 := temp2;

IF (smaller_than_8 = '0' AND   input(UPPER_LIMIT) = '1')
    THEN output <= "00000000";
ELSIF (greater_than_8 = '1' AND input(UPPER_LIMIT) = '0')
    THEN output <= "10000000";
```

We examine the sign bit (*input(UPPER_LIMIT)*) to see if input is positive. If yes, we use variable x to store the 3 LSB of integer part and 3 MSB of fraction part of input, which will be used in the next computational stage. If input is negative we use *minus_input* variable to store its absolute value. We will compute the output of –x as we take advantage of the function's attribute $f(-x) = 1 - f(x)$.

```
ELSE
   IF (input(UPPER_LIMIT) = '0') THEN
     FOR i IN 5 downto 0 LOOP
        x(i) := input(i-3);
     END LOOP;
   ELSIF (input(UPPER_LIMIT) = '1') THEN
      minus_input := - input;
      FOR i IN 5 downto 0 LOOP
         x(i) := minus_input(i-3);
      END LOOP;
   END IF;
```

The next stage is a two-level AND-OR gate implementation as described in [19], which stores in y variable a 8 - bit STD_LOGIC_VECTOR. Those bits consist of the sign bit and 7 bits of the fractional part of the output. As a result, we have a quantized form of the function's output. The final computational stage is a condition that determines the output.

Again, we should check if input is positive. If yes, y variable drives the output, but if not, we should subtract y variable from 1, store the result in *mid* variable, and then convert the result in STD_LOGIC_VECTOR type.

```
pre_output :=  to_ufixed(y,0,-7);
CASE input(UPPER_LIMIT) IS
  WHEN '0' =>
    output <=  y;
  WHEN '1' =>
    mid := resize(ONE - pre_output,0,-7);
    output <= to_slv(mid);
  WHEN OTHERS => null;
END CASE;
```

3.4.4 Hidden LUTs Module

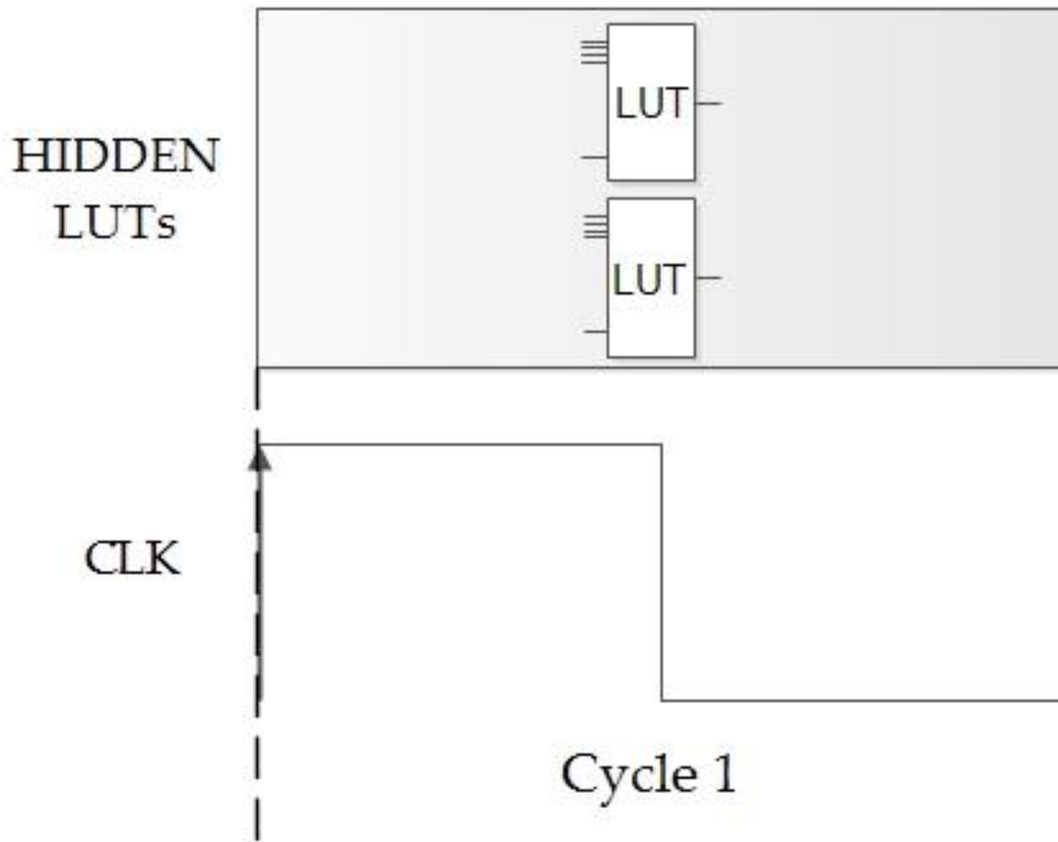


Figure 3.4 Hidden LUT Module

This module is a concentration of Look up Tables. The word “hidden” indicates that hidden nodes make use of these LUTs. But what exactly is their function? We already mentioned that weights and biases are the structural elements of an artificial neural network. Bias values are declared in the neural_library package, however we did not encounter weight values declaration until now. The reason why this is happening is a major aspect of our selected architecture.

The first operations when a node is enabled are parallel multiplications. Every input coming from the previous layer is being multiplied by its respective weight. The outcome of these multiplications is then accumulated and sent to the activation function to produce the output value of the node. This is described by relationship: $output = f(\sum(weight \times input) + bias)$, where f() is the activation function.

The problem that arises from these specifications is that we should use a rather large number of multipliers, even for medium – sized networks. The example network given, with 5 Input Nodes, 60 Hidden Nodes and 4 Output Nodes would require 5x60 + 60x4 = 540 multipliers. Modern FPGAs have no problem to support this amount of logic, but apart from the Area restrictions in our chip, which is always a

parameter that matters, when designing a Project, the use of multipliers – unless pipelined - will slow down our design, since multiplications require significantly more time than additions.

There are however two facts that allow us to follow another path. The first is always a fact regardless the characteristics of our dataset: Since training takes place only once, the weights produced are going to be constant numbers. There is no condition that will change their values. So, the first operand of multiplications is a constant, thus the complexity is reduced, since we multiply number x constant. The second fact, which allows us to completely emit multipliers from the design comes from the observation of the dataset: If every single input has a relatively small number of possible values, then multiplications are further more simplified, so as to being capable to take the form of a Look Up Table.

The application prints separate entities for every node. The input as we see next, is of *ann_input_vector* type. As we have already described, the values stored using this type have originated from the compression of actual input values. The fact that we do not use inputs, but a compressed form of the latter, is an additional way to save valuable resources in our design. There is also a bit that controls their function. When *Enable* = '1', hidden_LUTS are activated.

```
ENTITY hiddenNode_1 IS
  PORT (
        CLK                : IN  STD_LOGIC;
        input              : IN  ann_input_vector;
        Enable             : IN  STD_LOGIC;
        Lut_output         : OUT fixedX_vector(1 to 5)
  );
END ENTITY;
```

While LUTs are implemented using case statements, there are as many LUTs in every node as the number of inputs in the design. Possible outputs have been calculated within "Build_Neural" application, and then reformed to signed binary numbers of already given specifications. Moreover, there are Comment Lines next to the LUTs that show the physical meaning for every selection.

3.4.5 Output LUTs Module

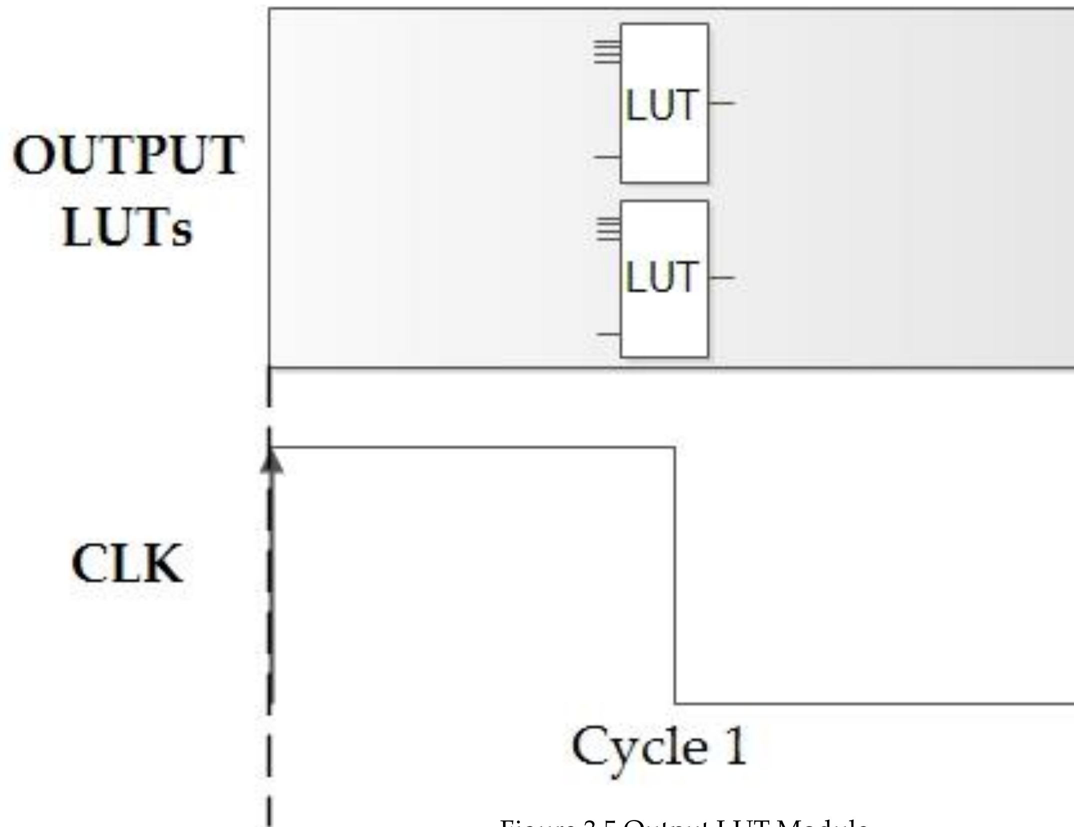


Figure 3.5 Output LUT Module

Obviously, output_LUTS are the equivalent modules for output nodes, as hidden_LUTS are for hidden nodes. There is only a slight difference between the two modules. We previously examined the module performing the activation function and we mention again that the output of a hidden node is also the input for an output node. This value in in the range [0, 1] as we know, because this is the range of logsig function. Moreover, after experimenting, we decided to use seven bits to hold the fraction part of this value. The criterion on this decision is that it provides the best trade-off balance, between error propagation and activation function size-complexity.

So, provided the values of possible hidden outputs are fixed, we know the exact size of multiplication output LUTs. The size of these tables is 129 positions, $2^7 + 1$. They are also implemented as separate entities, each one dedicated to its respective node. Case statements implement parallel LUTs within every node.

3.4.6 Hidden Node Module

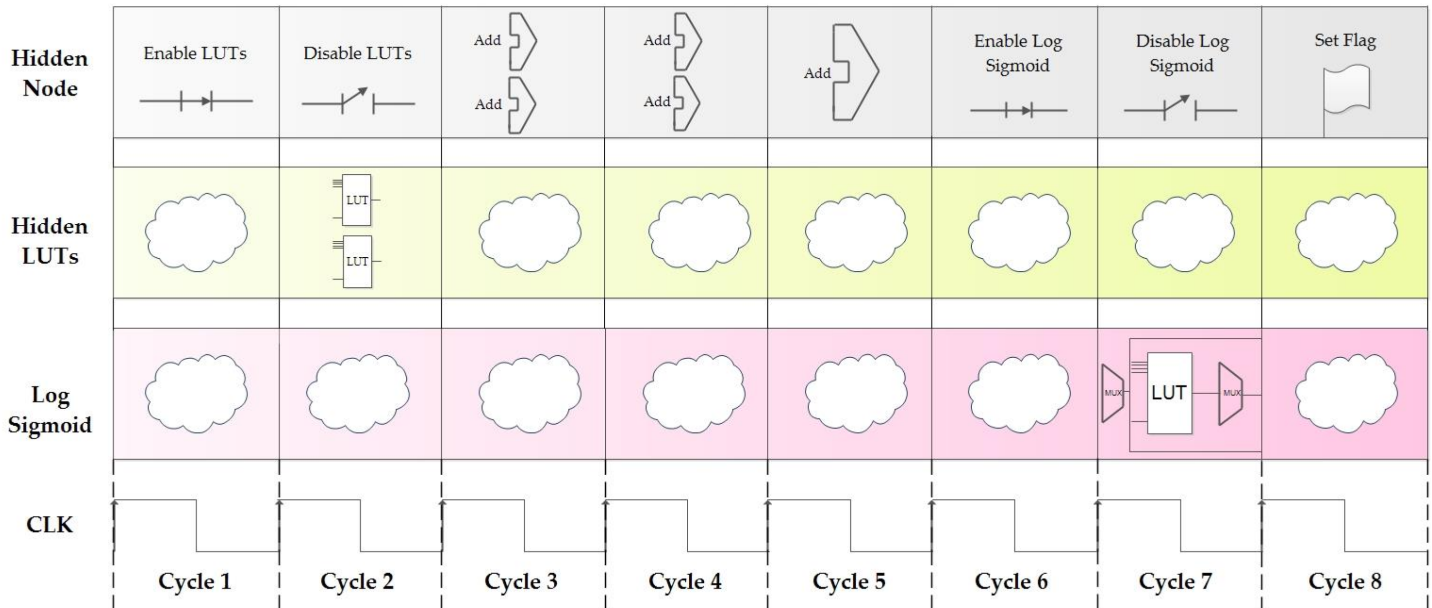


Figure 3.6 Hidden LUT module

Hidden nodes along with output nodes are the processing units of the Network. Every hidden node module should instantiate a log_sigmoid submodule, which will be enabled at the final stages of computation, to evaluate the result of the node output. Moreover, a single hidden node instantiates the appropriate hidden_LUT entity, which holds for the particular node, the weight values.

The following cited piece of code shows the entity of a hidden node module. Two generic values show the number of inputs in the network ("*Num_Inputs*") and the serial number of the specific hidden node ("*Position*"). Since all hidden nodes share the same code, the differentiation made by the latter parameter is essential as will be seen next, for linking every hidden node with the according hidden_LUT. "*Node_en*" input acts as a switch, which is handled by the central ANN FSM. "*Node_mode*" is also a parameter provided by ANN FSM and defines the exact action that the node will perform. "*Node_flag*" is an indicator that the specific node has completed all stages of computation and is ready to accept new inputs, while "*node_output*" is the provided result of these computations and in terms of artificial neural networks it is the hidden output of the network. This output is the result of the log_sigmoid function, its range of values is [-1, 1] so it is always an 8 – bit parameter, with 7 out of 8 bits expressing its decimal part.

```
ENTITY hidden_node is
      GENERIC (
            Num_Inputs : INTEGER := 4;
            Position   : INTEGER
      );
      PORT (
            input        : IN ann_input_vector;
            node_en      : IN STD_LOGIC;
            node_mode    : IN node_modes;
            CLK          : IN STD_LOGIC;
            node_flag    : BUFFER STD_LOGIC :='1';
            node_output : OUT STD_LOGIC_VECTOR(7 downto 0)
   );
```

Every node is linked to a unique hidden_LUT module which performs the multiplication as described in previous sub-chapter. The following code shows how this link is achieved for the hidden node with serial number 1 and the parameters passed to the log_sigmoid module as well as the parameter that is returned from log_sigmoid module (“*weight_x_input*”) which is used in the next stages of computation.

```
node1: IF (Position = 1) GENERATE
    lut1: hiddenNode_1 port map(
                                CLK,
                                input,
                                LUT_enable,
                                weight_x_input
                            );
END GENERATE node1;
```

Every hidden node integrates a log_sigmoid sub-module. The instantiation is shown next, with the parameters “*sig_input*” and “*sig_enable*” that are passed to the sub-module representing the input and control bit, respectively. “*Node_output*” is the result of the log_sigmoid function, which happens to be the later stage of computation within the node. Thus, the specific signal will be the output parameter of the node.

```
sigmoid_0 : log_sigmoid port map(
                                 sig_input,
                                 sig_enable,
                                 CLK,
                                 node_output
                               );
```

Main control functions of the Artificial Neural Networks are performed in the ann module, while node modules consist of the main processing units. The FSM of the ann module can possibly set the nodes in two possible modes: either “*run*” or “*idle*”(this setting can be expanded to include a “*learning*” mode).

“*Run*” mode along with the “*node_en*” bit sets the node to read data from its input and perform sequentially the functions that is designed to. “*Idle*” mode sets the node

to an idle state, which is the state that declares at the current time the node does not perform any computations.

The following figure depicts the possible states of a hidden node, and the incoming modes set by the Ann FSM.

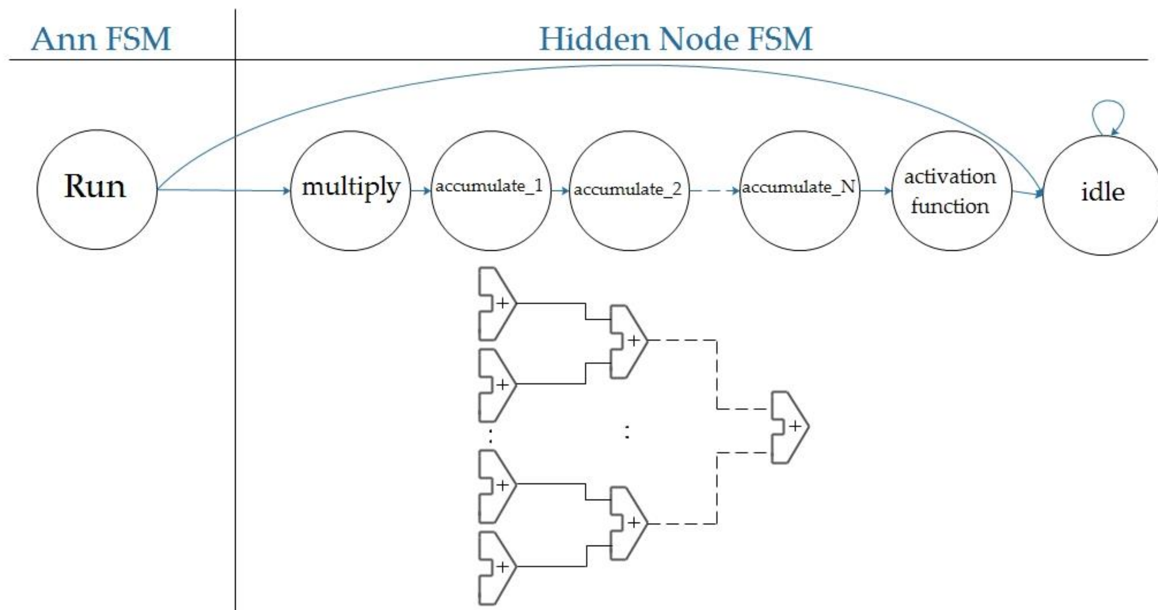


Figure 3.7 Hidden Node FSM

There is a distinction between hidden node mode and hidden node state. This choice intends to provide stability to our design, as the node state is isolated from the incoming node mode signals, so an unexpected change in the value of hidden node mode will not affect the current action of the node until it finishes and transits into idle state. The first actions when the incoming "*node_mode*" signal's value is "*run*" are described by VHDL in the cited piece of code below and are namely:

- Unsetting the "*node_flag*" so it will be unavailable from the main ann module to be assigned new tasks until it finishes current computations.
- Setting the "*LUT_enable*" bit, that is, switching on and transferring control to the hidden_LUT module.
- Set the node FSM to its initial state, which is "*multiply*".

In case of an incoming "*idle*" signal the actions performed are:

- Setting the "*node_flag*" so it is ready to accept new tasks.
- Zero the intermediate accumulators used during the computational stages

```
WHEN idle =>
  CASE node_mode IS
    WHEN run =>
      node_flag  <= '0';
      LUT_enable <= '1';
      node_state <= multiply;
    WHEN idle =>
      node_flag        <= '1';
      temp_accumulator <= (others => zero);
```

The slowest part of the design is the part of additions. A number of operands which is equal to the number of inputs from the previous layer need to be summed, and the sum will be used as input to the log_sigmoid function in order to provide the final output of the node. The scheme of additions is of critical importance, because if we choose adders with many operands this will eventually be the bottleneck of our system and will have negative effect in timing performance. Eventually, we choose to add operands in pairs of two, thus the formed adder tree will have a depth of $ceil(log_2(N_inputs))$. The cited code shows an example of the described adder tree, which in this case handles 6 inputs, so the depth of the tree will be $ceil(log_2(6)) = 3$, that means that 3 Hardware cycles are required to produce the final sum. The code of adder tree is followed by the final state of the node, which switches on the log_sigmoid sub-module and sets the node into an idle state.

```
WHEN accumulate_1 =>
  temp_accumulator(1) <= resize(weight_x_input(1) + weight_x_input(2),11,-8);
  temp_accumulator(2) <= resize(weight_x_input(3) + weight_x_input(4),11,-8);
  temp_accumulator(3) <= resize(weight_x_input(5) + weight_x_input(6),11,-8);
  temp_accumulator(4) <= resize(bias + weight_x_input(7),11,-8);
  node_state <= accumulate_2;
WHEN accumulate_2 =>
  temp_accumulator(5) <= resize(temp_accumulator(1) + temp_accumulator(2),11,-8);
  temp_accumulator(6) <= resize(temp_accumulator(3) + temp_accumulator(4),11,-8);
  node_state <= accumulate_3;
WHEN accumulate_3 =>
  temp_accumulator(7) <= resize(temp_accumulator(5) + temp_accumulator(6),11,-8);
  node_state <= activation_function;
WHEN activation_function =>
  sig_input     <= temp_accumulator(7);
  sig_enable    <= '1';
  node_state    <= idle;
```

3.4.7 Output Node Module

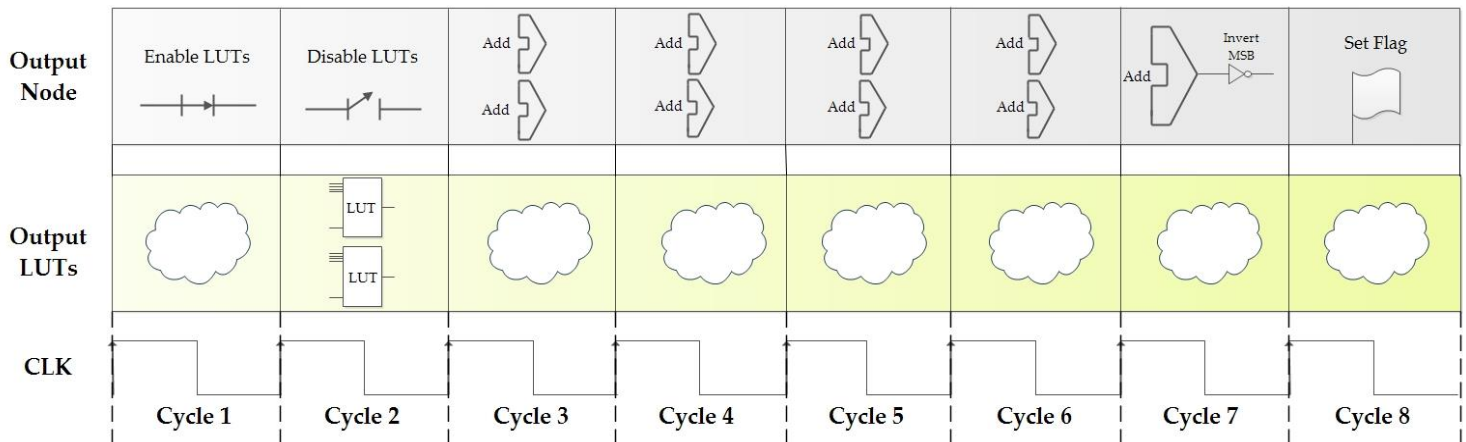


Figure 3.8 Output Node module

Output nodes are almost identical to hidden nodes. Slight modifications are made to enhance timing performance. The entity of output node holds exactly the same signals as this of hidden node, except for the output signal, which in that case of is the final output of the neural network and is only a single bit per each node. On the same way as hidden nodes, every output node is linked to an output_LUT module, in which we have already referred to as an indirect way to perform multiplications. This link is based on the logic that we used to link hidden nodes to their respective LUT modules, and is shown below.

```
node1: IF (Position = 1) GENERATE
    lut1: outputNode_1 port map(
                                CLK,
                                input,
                                LUT_enable,
                                weight_x_input
                            );
END GENERATE node1;
```

The modification made is in the design of the output_node FSM. Since we decided that the neural network will use binary logic to show its output, every output node is obliged to 2 possible values, 0 or 1. However, the final result is provided by the log_sigmoid function, which gives as output continuous values in the range [0,1]. In order to save valuable hardware resources we performed a logical leap by observing closely the graph of log-sigmoid function and specifically its result for input with zero value. It is logsig(0) = 0.5, logsig(0-) < 0.5 and logsig(0+) > 0.5. This attribute allows us to set this value as threshold and force all negative values to give output '*0*' and all positive values '*1*'. Based on this thinking, we emitted the log_sigmoid module, thus the output is provided by determining the sign of the final adder. The FSM of the output node is controlled by the Ann FSM and it shows great resemblance with that of hidden node.

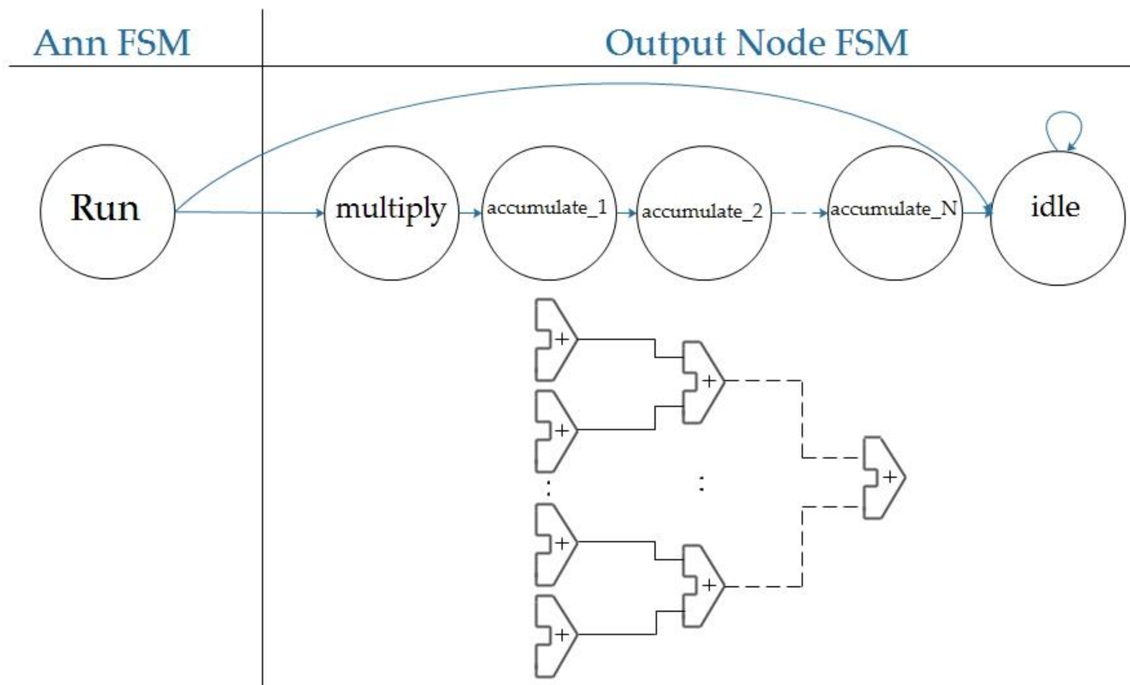


Figure 3.9 Output Node FSM

The final action of the output node has been already described, and the code that carries out this action is given below. The assignments made in this piece of code namely:

- The final adder of the tree adder scheme. Notice that this assignment refers to a variable and not a signal. This differentiation allows us to use the value of the variable within the same Clock Cycle.
- Invert the most significant bit of the variable, which in our selection of signed numbers depicts the sign of the variable. If the result of the final adder is positive we should drive the output to the value '*1*' whereas value '*0*' should be given if negative. Thus, inversion is appropriate.
- Set the "*node_flag*" so it is ready to accept new tasks.
- Set the node FSM to idle state.

```
WHEN accumulate_6 =>
  final_accumulator := resize(temp_accumulator(58) + temp_accumulator(59),6,-8);
  node_output <= NOT final_accumulator(UPPER_LIMIT);
  node_flag   <= '1';
  node_state  <= idle;
```

3.4.8 Ann Module

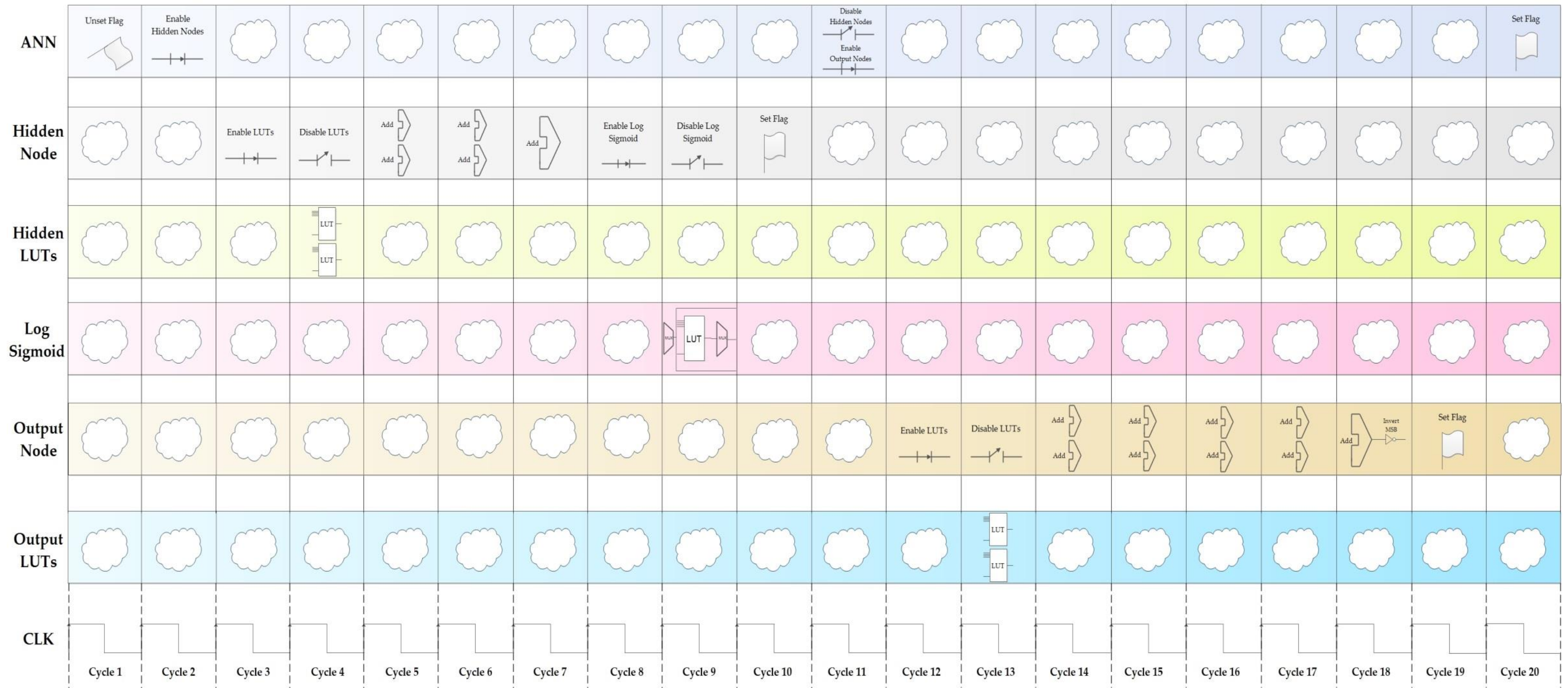


Figure 3.10 Ann Module

Ann module is the top module of the Neural Network design. There are few computations performed by this module; its purpose is to instantiate the number of hidden and output nodes and distribute tasks to them whenever is necessary. The entity of the module contains some generic values which determine the size of the Ann. We briefly name the rest signals. "*Input*" is provided as a vector of fixed numbers, "*ann_mode*" is an input signal that handles the function of the network, "*Enable*" bit acts as a switch, while "*reset*" is useful to recall the network to its initial state when it has already proceed to computational stages and is necessary to stop. "*Output*" is a vector of bits, sized equal to the Number of outputs which eventually shows the number of selected Scenario in binary representation.

```
ENTITY ann IS
  GENERIC (
      N_I : INTEGER := N_INPUTS;
      N_H : INTEGER := N_HIDDEN;
      N_O : INTEGER := N_OUTPUTS
   );

  PORT (
      input            : IN  ann_input_vector ;
      ann_mode         : IN  ann_modes;
      CLK              : IN  STD_LOGIC;
      Enable           : IN  STD_LOGIC :='0';
      reset            : IN  STD_LOGIC :='0';
      Ready            : OUT STD_LOGIC :='1';
      output           : OUT STD_LOGIC_VECTOR(1 to N_OUTPUTS)
  );
END ENTITY ann;
```

The number of nodes is passed as a generic value, so the ann module should be able to read these numbers and instantiate as many nodes. The signals that are used to handle the functions of hidden nodes are the following:

- "*Hidden_layer_en*" : A vector of bits; each of them enables a specific hidden node. We can either simultaneously enable all hidden nodes, or in a more sophisticated approach, make a selection of the nodes we enable.
- "*Hidden_layer_mode*" : This signal defines the state of the hidden nodes.

```
hidden_layer : FOR i IN 1 to N_H GENERATE
   hidden_nodes : hidden_node generic map( N_I,
                                           i
                                         )
                                port    map(
                                           input,
                                           hidden_layer_en(i-1),
                                           hidden_layer_mode,
                                           CLK,
                                           hidden_node_flag(i-1),
                                           hidden_output(i)
                                         );
  END GENERATE hidden_layer;
```

In the same way ann module creates and instantiates the appropriate number of output nodes. It also uses handling signals as the ones mentioned previously.

The ann code consists of three processes, the main process that controls all main functions and two sub-processes. These sub-processes behave like large AND gates that combine the "flag" signals from all the hidden and output nodes. The output of these gates becomes '1' only when all the nodes' flag either on the hidden or the output layer are set to '1', so it is an indication that we can proceed to the next stage of computations.

```
and_gate1: PROCESS(hidden_node_flag) IS

variable temp : STD_LOGIC;

BEGIN
  temp := '1';
  FOR i IN hidden_node_flag'range LOOP
    temp := temp AND hidden_node_flag(i);
  END LOOP;
  hidden_layer_flag <= temp;

END PROCESS;
```

The most secure way to see how these sub-processes are useful is to cite the code at the beginning of the main process and see how the signals that are the products of these sub-processes are used. We can descriptively name the condition that must be met to enter the main code of the process which is actually the function of the Ann FSM :

1. CLK signal is in its positive edge.
2. "*Enable*" signal is set to 1.
3. "*Reset*" signal is set to 0.
4. Both "*Hidden_layer_mode*" and "*Output_layer_mode*" signals are set to idle. This practically means that control has not been transferred neither to hidden nor to output nodes.
5. Both "*Hidden_layer_flag*" and "*Output_layer_flag*" as they are produced by the large AND gates are set to 1.

```
fsm: process(CLK) IS
BEGIN
  IF (CLK = '1' AND CLK'EVENT) THEN
    IF (Enable = '1') THEN
      IF (reset = '1') THEN
        ann_state <= idle;
        Ready     <= '1';
      ELSE
        IF (hidden_layer_mode /= idle OR output_layer_mode /= idle) THEN
              hidden_layer_mode <= idle;
              output_layer_mode <= idle;
        ELSIF (hidden_layer_flag = '1' AND output_layer_flag = '1') THEN
```

The next fragment of code describes the function of the ann FSM, which has the responsibility to share tasks and collect the results. It behaves as a regulator that assures a safe processing flow. "*Run*" state sets the enable signals of the hidden nodes and orders their respective FSMs to start computations, whereas "*run_next*" state switches off hidden nodes while switching on output nodes.

```
CASE ann_state is
   WHEN run =>
     hidden_layer_mode <= run;
     hidden_layer_en <= (others => '1');
     ann_state <= run_next;
   WHEN run_next
     hidden_layer_en <= (others => '0');
     output_layer_mode <= run;
     output_layer_en <= (others => '1');
     ann_state <= turn_off_output;
   WHEN turn_off_output =>
     output_layer_en <= (others =>'0');
     Ready      <= '1';
     ann_state <= idle;
```

The next schematic depicts the ann FSM and its interaction with nodes FSMs.

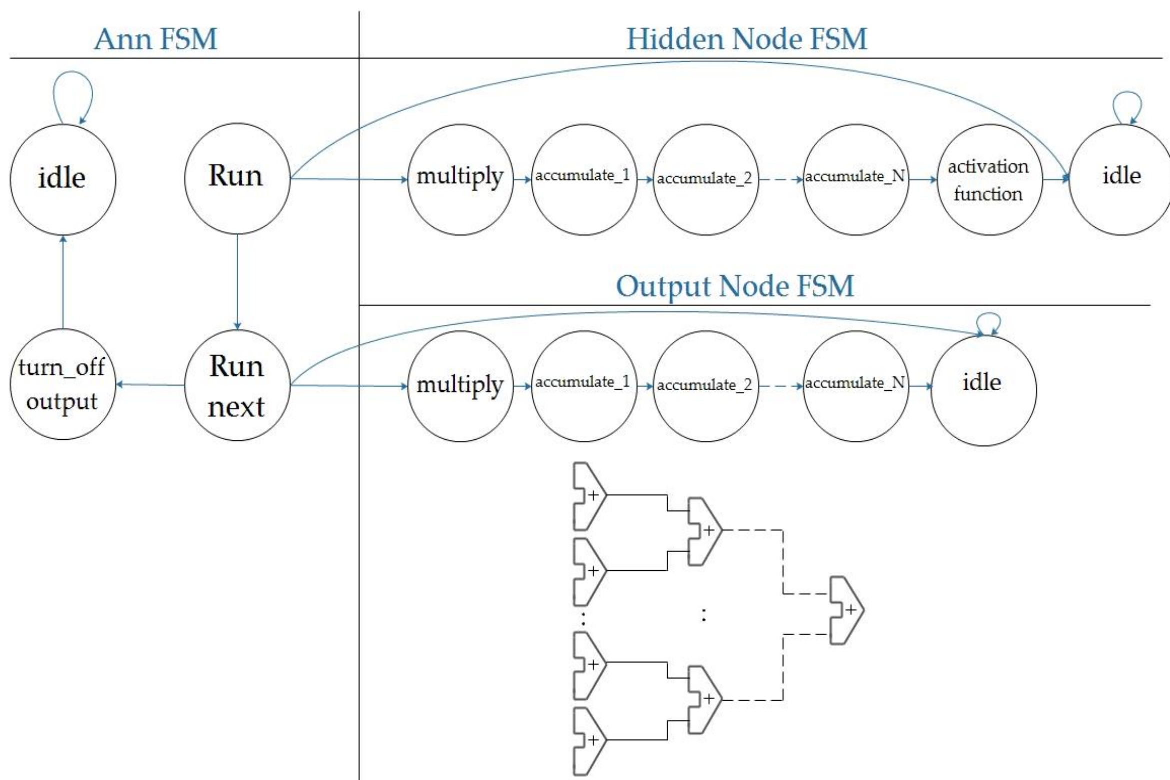


Figure 3.11 Ann FSM

3.4.9 Hybrid Module

Figure 3.12 Hybrid Module

Under specific circumstances, it is useful to use a complementary LUT to hold some of the Scenario values. In that case, neural module will be used as a Component into the Hybrid module, which becomes the top module. There are four stages involved in this module, the first two are common for all cases, and the last two are complementary, only one out of them will be selected on a specific run of the module. The first stage is named "*read_input*" and it performs the following :

- It uses one register per input, which acts as a sensor. In every Clock Cycle this register is compared to the input, and if it does not locate a change in at least one input, it drives the output to the last computed result. If it does locate a change, it proceeds to the next actions.
- It performs a sort of compression for the new input located. This compression is very useful in cases where input has a large value, so it needs a significant amount of bits to be represented. Afterwards only as many bits as needed to represent the possible states of this value are used. Since multiplications are performed using LUTs, the literal values of inputs are not directly needed, to the contrary we can use their "symbolic" values, which are the result of the compression.
- Based on the signals that show whether a change in each input was located, it enables the Ann module or keeps reading input.

The cited code describes the functions of sensor and compression for one single input.

```
WHEN read_input =>
  IF (hold_input.one /= input.one) THEN
    hold_input.one <= input.one;
    input_flag(1)        := '1';
    CASE input.one IS
      WHEN "01"   => new_input.one <= "0";
      WHEN "10"   => new_input.one <= "1";
      WHEN others => NULL;
    END CASE;
  ELSE
    input_flag(1)   := '0';
  END IF;
```

The decision at the end of the stage, which determines whether a change in Scenario is possible based on the combination of inputs, so it will be needed to enable Ann module.

```
flag := input_flag(1) OR input_flag(2) OR input_flag(3) OR input_flag(4);
  IF (flag = '1') THEN
    enable       <= '1';
    ann_function <= run;
    stage        <= correct;
    output_flag  <= '0';
  ELSE
    enable       <= '0';
    ann_function <= idle;
    output_flag  <= '1';
  END IF;
```

The second stage is a memory-like structure which is used to store an array of Scenarios. These scenarios are selected when their respective combined RTSs are given as the incoming address in this structure and bypass the structure of neural network. For shortcut reasons, we only show the instantiation of the variable that acts as address and the first values of the LUT.

```
WHEN correct =>
  LUT_decision := '1';
  test_vector := new_input.one & new_input.two & new_input.three & new_input.four;
  CASE test_vector IS
    WHEN "0000001110011100000"  =>  LUT_output <= "000000001";
    WHEN "0000000001111000000"  =>  LUT_output <= "000000001";
    WHEN "0101000100011000000"  =>  LUT_output <= "000000001";
    WHEN "0000001110011000000"  =>  LUT_output <= "000000001";
    WHEN "0000001001111000000"  =>  LUT_output <= "000000001";
    WHEN "0000001110010100000"  =>  LUT_output <= "000000001";
    WHEN "0000000001110000000"  =>  LUT_output <= "000000001";
```

The third stage of execution is selected only in certain cases; when the combination of inputs is not contained in the complementary LUT we described, our system follows this execution, which is depicted in the Schematic as *Execution 1*. The program stalls until the output of the ann is provided, and then it drives this signal to the hybrid output. It also sets the flag of this module, a sign that calls for further action, and moves back into the first stage.

```
WHEN drive_ann =>
  IF (ann_ready = '1') THEN
    output        <= ann_output;
    reg_output    <= ann_output;
    output_flag   <= '1';
    enable        <= '0';
    stage         <= read_input;
  END IF;
```

The fourth stage is the alternative and it describes the case when the specific Scenario coded by the current inputs forms a register in the memory-like component, so the function of ann module is not necessary and *Execution 2* shown by the schematic takes place.

```
WHEN drive_LUTS =>
  output            <= LUT_output;
  reg_output        <= LUT_output;
  ann_stop          <= '0';
  output_flag       <= '1';
  enable            <= '0';
  stage             <= read_input;
```

Chapter 4 Case Study

In the context of the current case study, we describe the extraction of RTSs from a wireless system based on the study made in [20], followed by the experimental results regarding detection implementation.

4.1 System Modeling

Antennas Signal Power We consider an uplink Wireless transmission channel of a MIMO-OFDM system based on the IEEE 802.11ac communication protocol [34]. The transmission data rate, for which we can achieve a successful transmission, is defined by the bandwidth, the capacity and the noise on the channel. A fundamental trade-off exists between Bit-Error-Rate (BER), which is correlated with the provided QoS, and antenna signal power. A potential run-time reconfiguration manager can adjust the signal power and the memory subsystem to the running situation. The scheduler selects the energy optimal configuration scheme (number of spatial streams, bandwidth, modulation and coding (MC) schemes) which respect the running constrains, based on the targeted communication standard (WLAN 802.11ac) characterization [34]. More precisely, the scheduler chooses the communication scheme, which requires the minimum SNR for the current data rate requirements under given conditions of external distortion. This presupposes that the scheduler has perfect updated knowledge of the channel condition and the application deadlines. The antenna signal power is adjusted to give the required data rate.

The aforementioned fundamental bound between signal power and data rate under specific noise conditions is mathematically expressed by the Shannon–Hartley theorem:

$$C = B\log_2\left(1+\frac{S}{N}\right) \qquad (1)$$

,where C is the channel capacity in bits per second; B is the bandwidth of the channel in hertz; S is the average received signal power over the bandwidth, measured in Watt; N is the average noise or interference power over the bandwidth, measured in Watt; and S/N is the signal-to-noise ratio (SNR).

This equation shows that a theoretical minimum SNR exists for achieving a target capacity with specific available channel bandwidth. The minimum SNR for a specific level of noise defines the minimum required signal power for an error-free transmission. For example, if the available bandwidth is B_w the theoretical minimum SNR for a transmission with bit-rate C_b without errors is:

$$SNR \le 2^{\frac{C_b}{B_w}} - 1 \qquad (2)$$

The average signal power, S, can be written as $S=E_bC$, where E_b is the average energy per bit. The average noise power, N, can also be redefined as, $N=N_0B$, where N_0 is the noise power (Watts/Hz). The Shannon–Hartley theorem can be written in the form:

$$\frac{C}{B}=\log_2\left(1+\frac{E_bC}{N_0B}\right) \qquad (3)$$

The ratio C/B represents the bandwidth efficiency of the system in bits/second/Hz. Knowing the SNR levels, we can characterize the total signal power efficiency of every configuration (minimum Signal Power) to achieve the targeted capacity. If the configuration supports multiple antennas (multiple spatial streams) the total signal power is estimated as the sum of the signal of each antenna.

The theoretical minimum SNR for an error-free transmission is impossible to reach in practice. The modulation schemes define how close to this theoretical SNRmin the transmission can be. Every modulation scheme is characterized by a minimum SNR that allows the demodulation of the transmitted symbols without errors. Knowing the minimum SNR for every modulation scheme (MS), we can define the minimum Signal Power for every MS for specific levels of noise. The equation that defines the symbol error probability (Ps) for every MS, with respect to SNR is the following [35]:

$$P_{s,M\text{-}ary}=\left[\left(\frac{M-1}{M}\right)\right]\bullet erfc\left(\left(\frac{3}{(M^2-1)}\right)\bullet\frac{E_{Saver}}{N_0}\right)^{\frac{1}{2}} \qquad \mathbf{(4)}$$

M is the number of symbols used, Es the average received signal power, N_0 the average noise signal power and erfc is the complementary error function.

The graphical expression of this equation for the modulation schemes of the 802.11ac is presented in Figure 4.1. Channel coding improves the SNR by a factor R [18]. So the curves can be normalized for equal energy per information bit (pre-coding) bearing in mind that the energy per transmitted bit is less than the energy per information bit by a factor equal to the code rate R. The graphical expression of the symbol error probability for the modulation and coding (MC) schemes of the 802.11ac can be found in Figure 4.2.

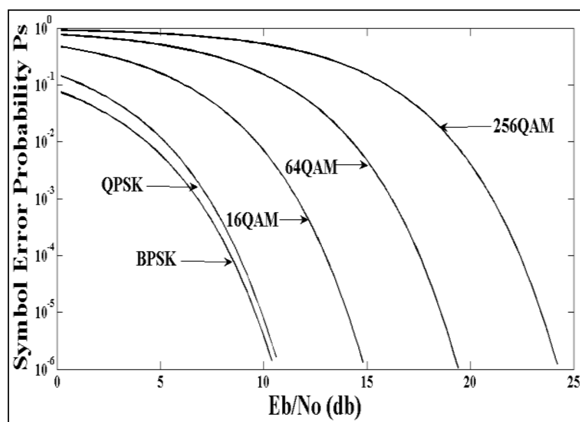


Figure 4.1 Symbol Error Probabilities for 802.11ac Modulation Schemes

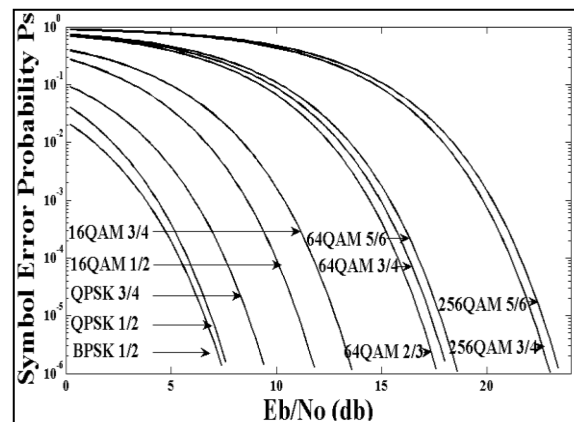


Figure 4.2 Symbol Error Probabilities for 802.11ac Coding & Mod Schemes

In this context, every system scenario RTS is characterized by a two-dimensional cost 1) the total signal power and 2) the bit error rate (BER). The signal power is inversely proportional to the symbol error probability and correspondingly to bit error probability as shown in Figure 5.1 and Figure 5.2. Each RTS is characterized by a curve in the two-dimensional space of total signal power. This curve is derived by the respective curve at Figure 3 that corresponds at the MCs of the RTS. Based on the bits-per-symbol of MCs (BPSK: 1bps, QPSK: 2bps, etc.), the short guard interval (SGI) and the noise level of the RTS, the Ps (symbol error probability) to SNR curve can be transformed to BER to Signal Power curve.

Besides the above-mentioned technical analysis the most unstable parameter for a transmission is the user profile, e.g., the distance between receiver and transmitter, the existence of other communication channels or others sources of distortion. These are factors that influence the channel transmission and are directly influenced by the user behavior. For example, if the user moves in a saturated spectrum area or in a noisy environment high communication channel interference is expected. Correspondingly, if the user changes position very rapidly, (for example, driving a car) this has impact on the normal demodulation of the transmitted signal (Doppler Effect).

Memory Banks The SNR level and the changing environment on the wireless channel also affects the memory requirements. In more detail, the conditions of the channel determine the coding and modulation scheme needed for a successful communication and, consequently the required data rate. The coding phase transforms an m-bit data string into an n-bit string in order to be encoded, when the given coding rate is m/n. The modulation phase conveys a varying number of bit streams together, based on the chosen modulation. The data rate constraint defines the storage and transmission requirements for the data. As a result, the memory footprint depends on the data rate of the channel and is dynamic for a changing environment. Energy consumption on the memory subsystem depends on the number of accesses and the energy per access, which are different based on the size and the type of memory.

The observation that the memory requirements at run-time vary significantly due to dynamic variations on the transmission channel and the protocol, is exploited through use of system scenarios. Instead of defining the memory requirements for the worst-case data rate and tuning the system according to this, system scenarios are generated for different situations. The combination of the coding and the modulation parameters define the data rate for each RTS. The data rate is the identification variable and the cost factor is its memory footprint. Based on the cost factor, the different memory footprints are clustered into scenarios. The clustering of RTSs is based both on their distance on the memory size axis and the frequency of their occurrence. The key feature needed in the platform architecture is the ability to efficiently support different memory sizes that correspond to the system scenarios generated by the methodology. Execution of different system scenarios then leads to different energy costs, as each configuration of the platform results in a specific

memory energy consumption. The dynamic memory platform is achieved by organizing the memory area in a varying number of banks that can be switched between different energy states.

4.2 CASE STUDY (I)

Our development platform is the Xilinx Virtex 6 XC6XCX75T platform [21]. Since our implementation is not directed exclusively to the specific platform, but it is designed to have general applicability, we only mention the basic characteristics of the platform, shown in Table 4.1. It is not the latest design, but it is large enough to fit the current implementation. It is worth to mention that the current platform also holds special hardware blocks, as shown in Figure 4.3. And it handles arithmetic operations using a number of special blocks named DSP48E1s [22], 864 in total. The latter could be a presumptive constraint for the multiplication operations on neural networks, so we use optimized architecture to overcome this potential problem.

Device	Slice Registers	Slice LUTs	Bonded IOBs
Virtex - 6 XC6VCX75T	708,480	354,240	720

Table 4.1. Main Specifications of the Virtex 6 – XC6VCX75T (Package FF484)

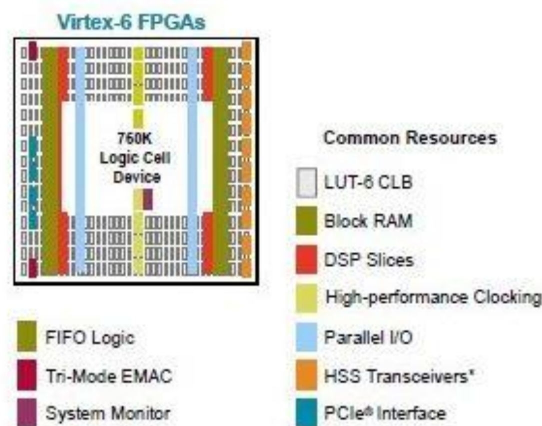
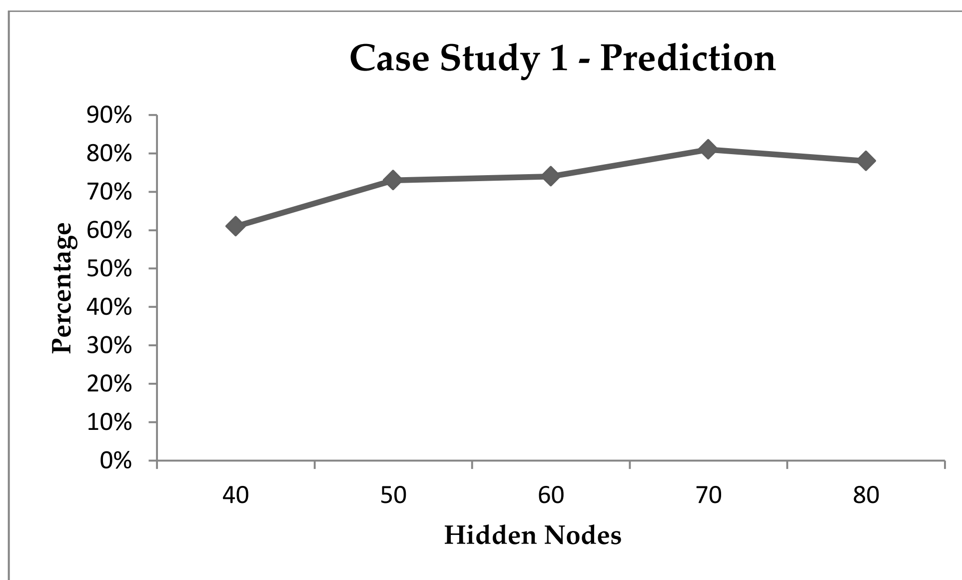


Figure 4.3 Virtex–6 Blocks

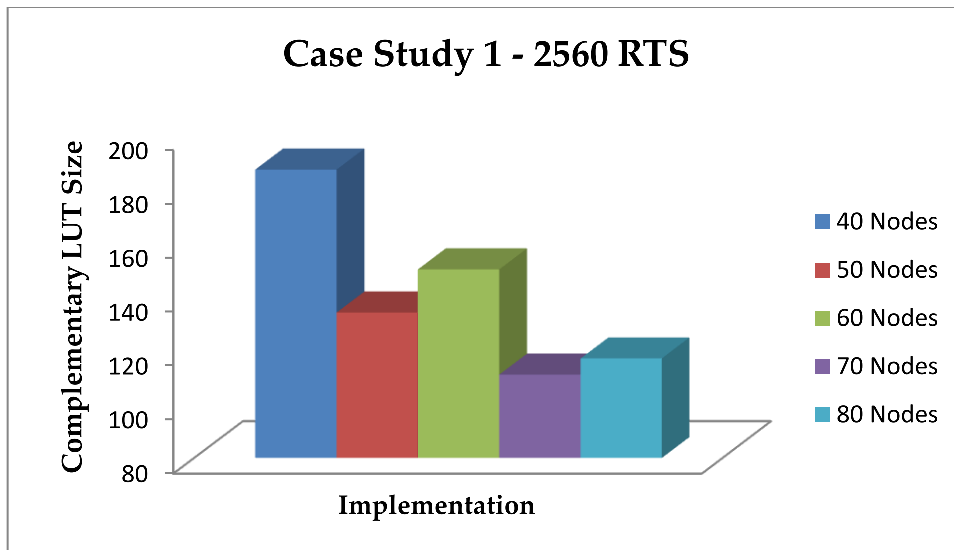
We extracted results for two different Clustering options (2560 and 5120 RTS) for a series of dynamic implementations following the methodology steps, and for the static implementation as well. The most important metrics of Synthesis, Implementation and Simulation stages are depicted in the following pages.

Implementation (# Hidden Nodes)	QoS Metric	Hardware Metrics			
	Prediction (Perc.)	Latency (Cycles)	Frequency (MHz)	Slice LUTs (Utilization)	Power (W)
40	61%	22	290,43	4%	7,43
50	73%	22	286,04	5%	7,41
60	74%	22	250,62	6%	7,43
70	81%	23	222,52	8%	7,43
80	78%	23	222,42	9%	7,45
Static	-	2	417,88	< 1%	7,3

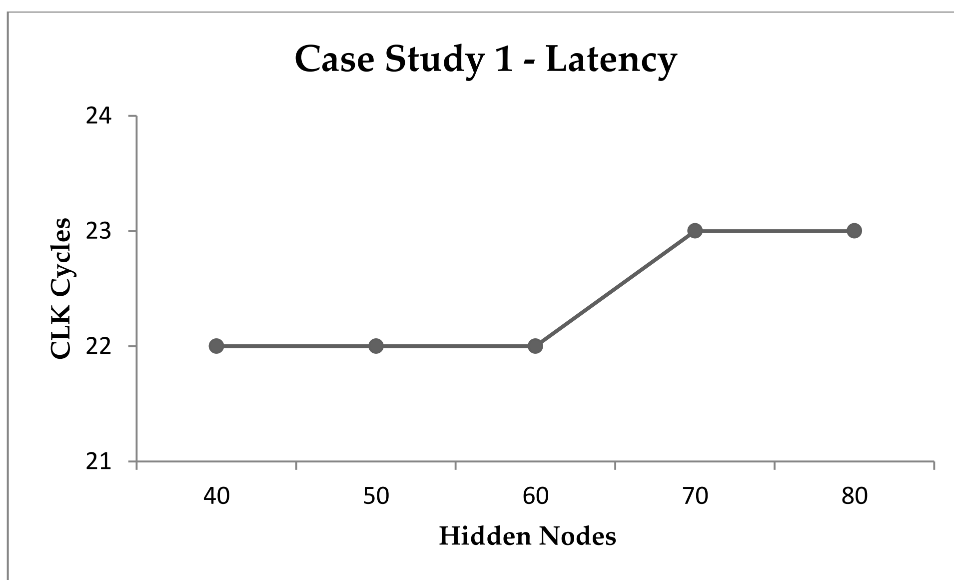
It can be easily noticed, that in terms of Hardware cost, the Static Implementation is superior than the implementations with the use of Neural Networks. However, its dynamic ability is non – existent, since it can only detect the Scenarios it has been trained of. Each metric is shown separate in the next diagrams.



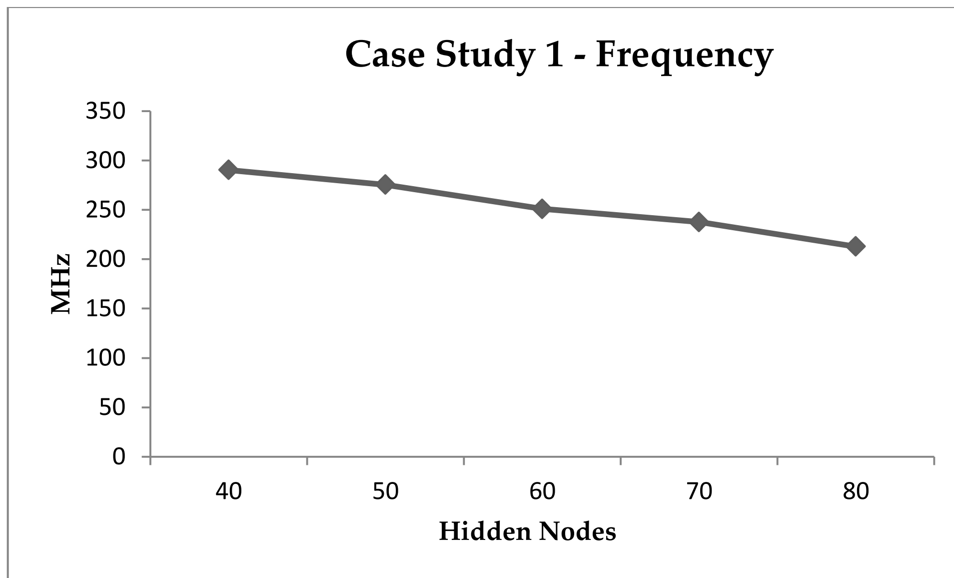
The best-tuned neural network is the one with 70 Nodes. From that stage on, additional hidden nodes do not provide with more prediction capacity.



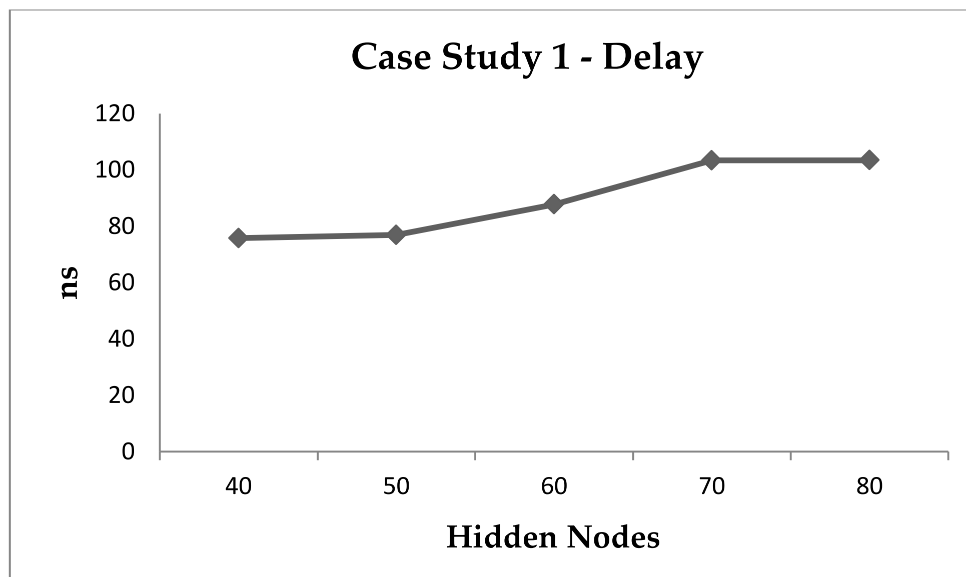
The size of Complementary LUTs in each implementation is an indication of the effectiveness of the training. Complementary LUT is the structure that is instantiated during Simulation and has as many entries as the number of error cases. If we had not been using cross – validation technique for better generalization results, we would expect that with the increase of hidden nodes, a reduction of the size of Complementary LUT. But since we use validation, 'early stopping' prevents the network from overfitting to the known data. It is worth to notice that the implementation with the best prediction capacity is the one with the smallest Complementary LUT size.



Our designs are structured in such way that latency is directly dependent only on the size of hidden layer, because at this stage they infer a tree adder, the length of which defines the total latency in CLK cycles. An increase in latency by one CLK cycle is noticed at the transition from the implementation with 60 nodes to that with 70 nodes.



The implementations are synchronized at a certain frequency. Additional nodes do not affect the critical path in terms of additional logic, but the complexity of the circuit becomes higher, so it is more difficult for the tools that do the placement in the FPGA platform. The gradual reduction of frequency is due to rooting delays.

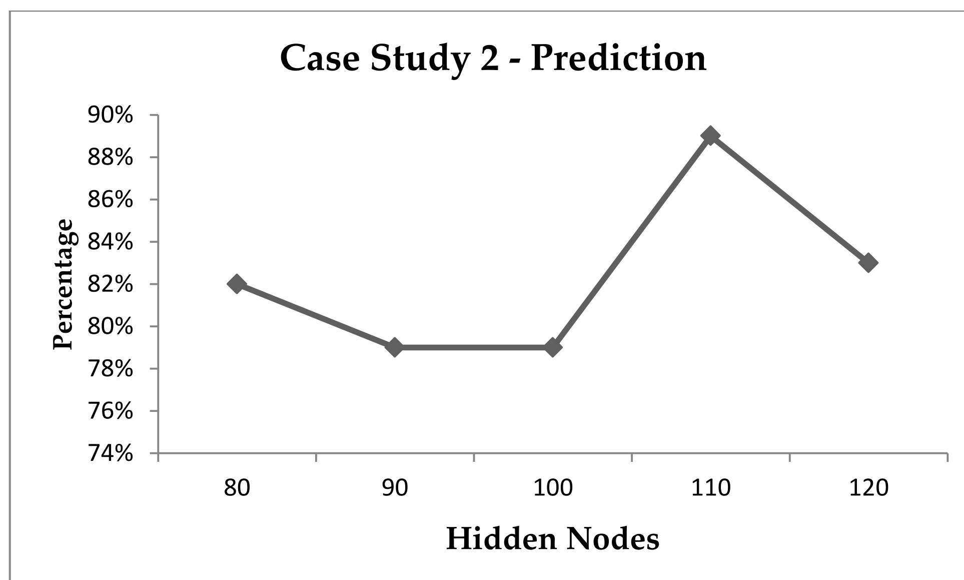


Finally, we can see the total delay of each implementation, the response time from the moment that RTSs are given to the input stage, to the output stage, where the number of Scenario is produced. The slowest implementation needs 103 ns, but it is worth to notice, that our customization offers shortcuts for the extraction of scenarios, it is therefore feasible to complete the process in significantly lower fragment of time.

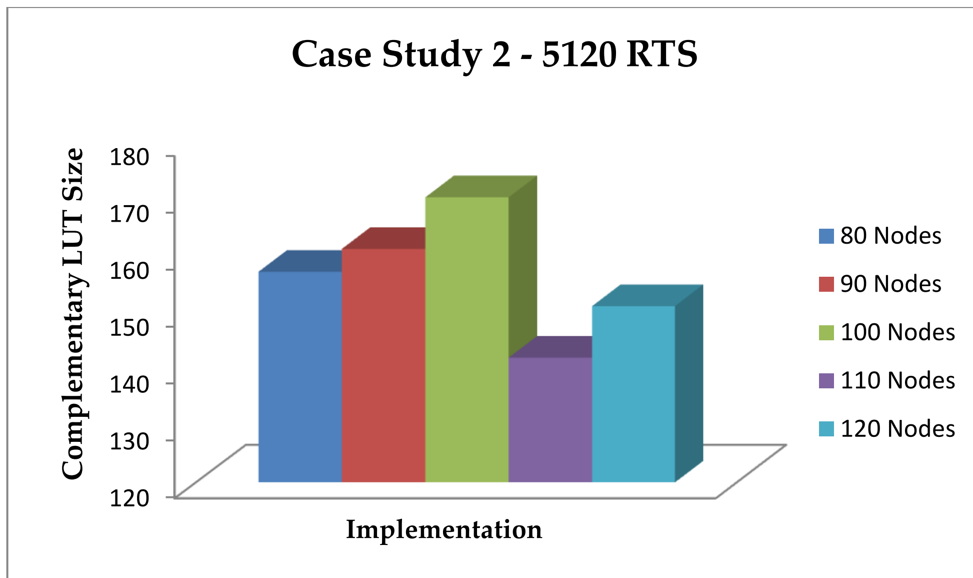
The results for the Clustering with 5120 RTS can be seen next:

4.3 Case Study (II)

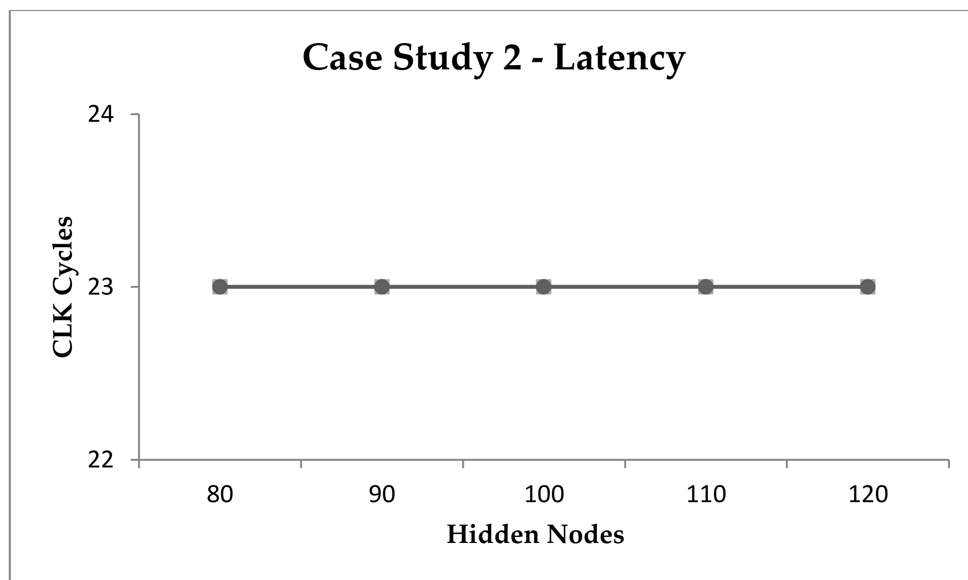
Implementation (# Hidden Nodes)	QoS Metric	Hardware Metrics			
	Prediction (Perc.)	Latency (Cycles)	Frequency (MHz)	Slice LUTs (Utilization)	Power (W)
80	82%	23	197,12	11%	7,47
90	79%	23	178,83	13%	7,49
100	79%	23	171,23	14%	7,49
110	89%	23	166,86	17%	7,48
120	83%	23	166,78	18%	7,5
Static	-	2	348,79	1%	7,34



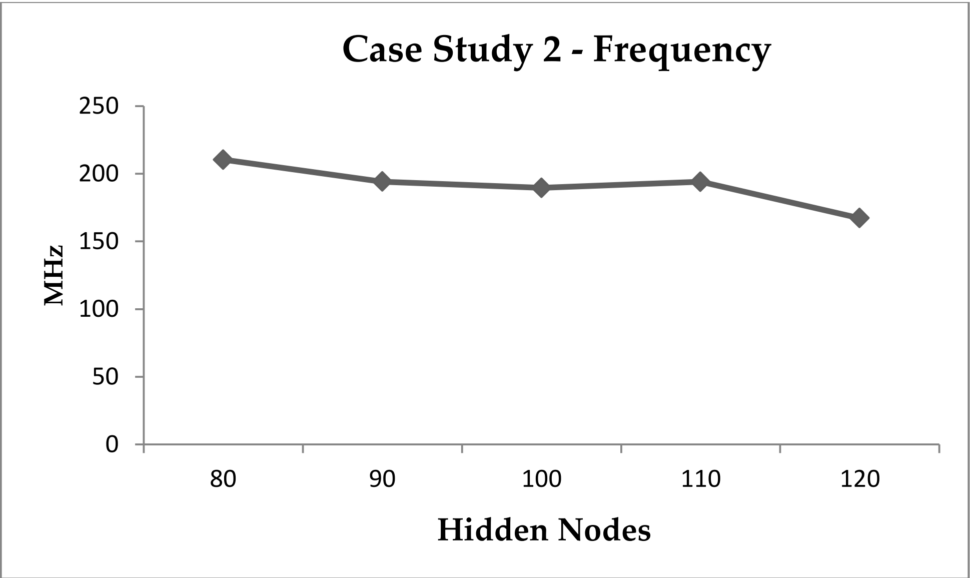
Starting with 80 hidden nodes, gradually we increase the size of hidden layer, until the network's dynamic ability is saturated. This point was discovered for the implementation with 110 nodes.



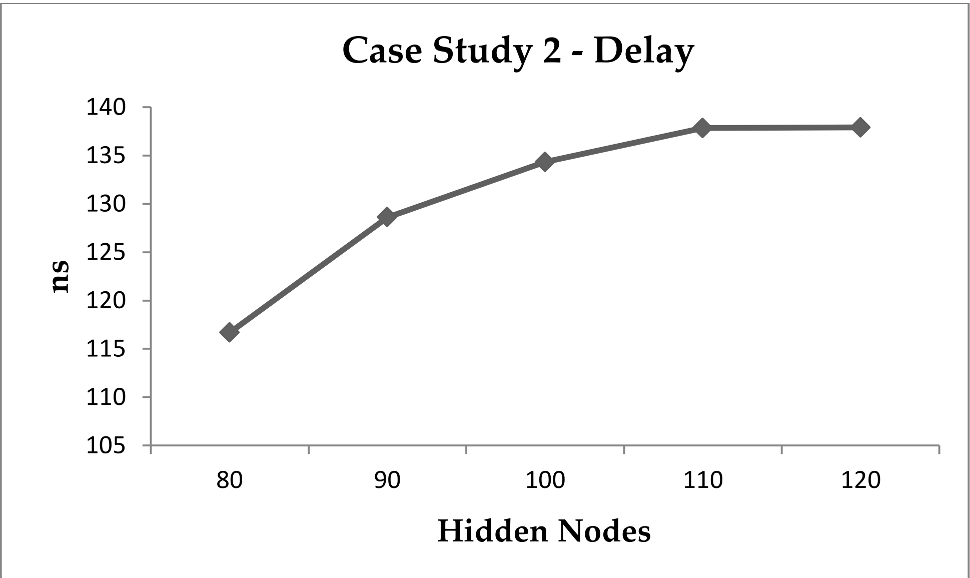
Again, we notice that the implementation with the best prediction capacity, is also the one with the smallest size of Complementary LUT. The combination of these factors makes the current implementation in both ways the most suitable.



Migrating from the implementation with 80 Nodes to that with 120 Nodes does not add a single CLK Cycle. There is balance in this metric.



Additional nodes result in reduction of operating frequency, as we had seen on the previous Case Study. The added size and complexity of the circuit is the reason for this deterioration on frequency performance.



Finally, the total delay is increasing gradually, so the performance in terms of timing cost is deteriorating. This result is expected, as the number of CLK cycles remains stable, while frequency reduces.

Chapter 5 Conclusions & Future Work

Our work was focused towards developing a non-existing implementation for Scenario Detection, which apart from the usual process of responding to a combination of inputs and providing the suitable output, will also have dynamic ability, that is to predict during unknown situations the Scenario to be implemented. This was achieved with high level of success, as we developed highly accurate Neural Networks, with prediction ability up to 90%. The systematic way by which a methodology to extract the optimal solution in terms of efficiency, and the scientific documentation that this methodology was based on, suggests that it is a rather reliable solution.

Besides the numerous interventions that targeted in optimizing the implementation in terms of efficiency and cost, there is still room for improvement from the technical point of view. Some ideas would be:

- Ternary adders could be added in place of the existent which add only two operands. Since the stage of additions is the most time consuming part of the design, the reduction of the stages of the tree adder that would be inferred with the use of ternary adders, would significantly reduce the latency of the implementation, without affecting the critical path, that is the operating frequency.
- The implementation has been designed with minimum levels of logic at each CLK Cycle, so the final Timing delay in the Critical path is due to routing delays. The solution to reduce routing cost in the Hardware would be an analytical floorplanning, which is performed with the use of the Software tools that provide us all the suitable tools.

There are also many ways of customization using the existent implementation. One such customization could be an energy – saving solution which would enable at each stage, only the neurons that would be necessary for each case. Even better, we could create the ANN the way we create it at the moment, and with the use of genetic algorithms we could reduce the number of neurons that do not eventually participate in computations. This solution is towards a more compact implementation, with less hardware footprint and better possibilities to be embedded into a small chip.

Finally, the biggest challenge would be to create a system which would be instantiated as an artificial neural challenge, but its parameters for training would be given only during run-time. Thus, the system should have the capability to perform on-chip training, and periodically evolve, depending on the scale of the different inputs it will encounter. Perhaps the structure of Cascade Correlation Networks is the most suitable to perform the specific task. Anywise, on - chip training is extremely demanding, since constraints in hardware devices would reduce the

wanted precision and therefore undermine the capability of the network for proper training.

References

[1] N. Zompakis, “Development of a Systematic Methodology for Dynamic Resource Management for Embedded Systems,” NTUA, 2014.

[2] “www.wirelessmotivation.org.” .

[3] S. V. Georghitta, *Dealing with Dynamism in Embedded System Design , Application Scenarios*. 2007.

[4] W. B. Frakes, “Software reuse research: status and future,” *IEEE Trans. Softw. Eng.*, vol. 31, no. 7, pp. 529–536, Jul. 2005.

[5] A. Sangiovanni-Vincentelli and G. Martin, “Platform-based design and software design methodology for embedded systems,” *IEEE Des. Test Comput.*, vol. 18, no. 6, pp. 23–33, 2001.

[6] Y.-T. S. Li and S. Malik, “Performance analysis of embedded software using implicit path enumeration,” in *Proceedings of the 32nd ACM/IEEE conference on Design automation conference - DAC ’95*, 1995, pp. 456–461.

[7] S. V. Gheorghita, F. Vandeputte, K. De Bosschere, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, and F. Catthoor, “System-scenario-based design of dynamic embedded systems,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, pp. 1–45, Jan. 2009.

[8] S. Gheorghita, T. Basten, and H. Corporaal, “Application Scenarios in Streaming-Oriented Embedded System Design,” in *2006 International Symposium on System-on-Chip*, 2006, pp. 1–4.

[9] S. Lee, K. Choi, and S. Yoo, “An intra-task dynamic voltage scaling method for SoC design with hierarchical FSM and synchronous dataflow model,” in *Proceedings of the 2002 international symposium on Low power electronics and design - ISLPED ’02*, 2002, p. 84.

[10] T. Sherwood, S. Sair, and B. Calder, “Phase tracking and prediction,” *ACM SIGARCH Comput. Archit. News*, vol. 31, no. 2, p. 336, May 2003.

[11] D. H. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 248–259.

[12] L. Fausett, *Fundamentals of Neural Networks: architectures, algorithms, and applications*. Melbourne: Prentice-Hall, 1994.

[13] R. Lange, "Design of a Generic Neural Network FPGA-Implementation," Chemnitz University of Technology, 2005.

[14] J.-P. Deschamps, G. J. A. Bioul, and G. D. Sutter, *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems*. Wiley, 2006.

[15] J.-P. Deschamps, G. J. A. Bioul, and G. D. Sutter, *Synthesis of Arithmetic Circuits*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2005.

[16] J. Zhu and P. Sutton, *Field Programmable Logic and Application*, vol. 2778. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[17] M. H. Beale, M. T. Hagan, and H. B. Demuth, "Neural Network Toolbox ™ User's Guide R 2014 a," 2014.

[18] A. Elisseefl and F. Lyon, "Size of multilayer networks for exact learning: analytic approach," in *Advances in Neural Information Processing Systems*, 1996.

[19] M. T. Tommiska, "Efficient digital implementation of the sigmoid function for reprogrammable logic," *IEE Proc. - Comput. Digit. Tech.*, vol. 150, no. 6, p. 403, 2003.

[20] N. Zompakis, I. Filippopoulos, P. G. Kjeldsberg, F. Catthoor, and D. Soudris, "Systematic Exploration of Power-Aware Scenarios for IEEE 802.11ac WLAN Systems," in *2014 17th Euromicro Conference on Digital System Design*, 2014, pp. 28–35.

[21] Xilinx, "Virtex-6 FPGA Data Sheet," vol. 152, pp. 1–65, 2014.

[22] Xilinx, "DSP48E1 Slice User Guide," vol. 369, pp. 1–52, 2011.

[23] B. M. W. Hao Yu, *Intelligent Systems*. CRC Press, 2011, pp. 12–1,12–16.

[24] S. Haykin, Neural networks : a comprehensive foundation, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 1999.

[25] Scherer, Andreas: Neuronale Netze: Grundlagen und Anwendungen. Braunschweig: Vieweg, 1997

[26] Rosenblatt, Frank: The Perceptron: A probabilistic model for information storage and organization in the brain. In: Psychological Review 65 (1958), no.6, p. 386–408

[27] Werbos, Paul J.: Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences, Harvard University, Diss., 1974

[28] Hagan, Martin T. ; Demuth, Howard B. ; Beale, Mark: Neural Network Design. Boston : PWS Publishing Company, 1996

[29] Hopfield, J. J.: Neural Networks and Physical Systems with Emergent Collective Computational Abilities. In: Proceedings of the National Academy of Sciences, 1982, p. 2554–2558

[30] Fahlman, S. E. ; Lebiere, C.: The Cascade-Correlation Learning Architecture. In: Touretzky, D. S. (Ed.): Advances in Neural Information Processing Systems vol. 2. Denver 1989 : Morgan Kaufmann, San Mateo, 1990, p. 524–532

[31] Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43:59–69, 1982.

[32] T. Kohonen. The self-organizing map. *Neurocomputing*, 21(1-3):1–6, 1998.

[33] D. Bishop, "Fixed point package." [Online]. Available: http://www.eda.org/vhdl-200x/vhdl-200x-ft/packages/fixed_pkg.vhd

[34] "IEEE 802.11ac, 2011. Specification frame work for ac: IEEE 802.1109/0992r21,"

[35] Proakis, "Digital Communication Systems," *4th Ed., McGraw Hill*, 2001.

[36] Muhammad Imran TAJ "Network on chip based Multiprocessor System on Chip for Wireless Software Defined and Cognitive Radios", PhD Thesis, Université Paris-Est, ESIEE Paris, Laboratoire LIGM Feb, 2012.

[37] J. Mitola, "The Software Radio," IEEE National Telesystems Conference, 1992.

[38] P. Marchal, C. Wong, A. Prayati, N. Cossement, F. Catthoor, R. Lauwereins, D. Verkest, "Impact of task-level concurrency transformations on the MPEG4 IM1 player for weakly parallel processor platforms", on Compilers and Operating Systems for Low Power (COLP'00) in conjunction with Intnl. Conf. on Parallel Arch. And Compilation Techniques (PACT), Philadelphia PN, Oct. 2000.

Appendix A

A.1 Introduction

The Levenberg–Marquardt algorithm which was independently developed by Kenneth Levenberg and Donald Marquardt, provides a numerical solution to the problem of minimizing a nonlinear function. It is fast and has stable convergence. In the artificial neural-networks field, this algorithm is suitable for training small- and medium-sized problems.

Many other methods have already been developed for neural-networks training. The steepest descent algorithm, also known as the error backpropagation (EBP) algorithm, dispersed the dark clouds on the field of artificial neural networks and could be regarded as one of the most significant breakthroughs for training neural networks. Many improvements have been made to EBP, but these improvements are relatively minor. The EBP algorithm is still widely used today; however, it is also known as an inefficient algorithm because of its slow convergence. There are two main reasons for the slow convergence: the first reason is that its step sizes should be adequate to the gradients). Logically, small step sizes should be taken where the gradient is steep so as not to rattle out of the required minima (because of oscillation). So, if the step size is a constant, it needs to be chosen small. Then, in the place where the gradient is gentle, the training process would be very slow. The second reason is that the curvature of the error surface may not be the same in all directions, such as the Rosenbrock function, so the classic "error valley" problem [28] may exist and may result in the slow convergence.

The slow convergence of the steepest descent method can be greatly improved by the Gauss–Newton algorithm [28]. Using second-order derivatives of error function to "naturally" evaluate the curvature of error surface, The Gauss–Newton algorithm can find proper step sizes for each direction and converge very fast; especially, if the error function has a quadratic surface, it can converge directly in the first iteration. But this improvement only happens when the quadratic approximation of error function is reasonable. Otherwise, the Gauss–Newton algorithm would be mostly divergent.

The Levenberg–Marquardt algorithm blends the steepest descent method and the Gauss–Newton algorithm. Fortunately, it inherits the speed advantage of the Gauss–Newton algorithm and the stability of the steepest descent method. It's more robust than the Gauss–Newton algorithm, because in many cases it can converge well even if the error surface is much more complex than the quadratic situation. Although the Levenberg–Marquardt algorithm tends to be a bit slower than Gauss–Newton algorithm (in convergent situation), it converges much faster than the steepest descent method.

The basic idea of the Levenberg–Marquardt algorithm is that it performs a combined training process: around the area with complex curvature, the Levenberg–Marquardt algorithm switches to the steepest descent algorithm, until the local curvature is proper to make a quadratic approximation; then it approximately becomes the Gauss–Newton algorithm, which can speed up the convergence significantly.

A.2 Algorithm Derivation

In this part, the derivation of the Levenberg–Marquardt algorithm will be presented in four parts: (1) steepest descent algorithm, (2) Newton's method, (3) Gauss–Newton's algorithm, and (4) Levenberg– Marquardt algorithm.
Before the derivation, let us introduce some commonly used indices:

- p is the index of patterns, from 1 to P, where P is the number of patterns.
- m is the index of outputs, from 1 to M, where M is the number of outputs.
- i and j are the indices of weights, from 1 to N, where N is the number of weights.
- k is the index of iterations.

Other indices will be explained in related places.
Sum square error (SSE) is defined to evaluate the training process. For all training patterns and network outputs, it is calculated by

$$E(x,w) = \frac{1}{2}\sum_{p=1}^{P}\sum_{m=1}^{M} e_{p,m}^2 \tag{A.1}$$

where

x is the input vector
w is the weight vector
$e_{p,m}$ is the training error at output m when applying pattern p and it is defined as:

$$e_{p,m} = d_{p,m} - o_{p,m} \tag{A.2}$$

where

d is the desired output vector
o is the actual output vector

A.2.1 Steepest Descent Algorithm

The steepest descent algorithm is a first-order algorithm. It uses the first-order derivative of total error function to find the minima in error space. Normally, gradient g is defined as the first-order derivative of total error function (A.1):

$$g = \frac{\partial E(x,w)}{\partial w} = \left[\frac{\partial E}{\partial w_1} \quad \frac{\partial E}{\partial w_2} \quad \cdots \quad \frac{\partial E}{\partial w_N}\right]^T \tag{A.3}$$

With the definition of gradient g in (A.3), the update rule of the steepest descent algorithm could be written as

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha \mathbf{g}_k \tag{A.4}$$

where α is the learning constant (step size).
The training process of the steepest descent algorithm is asymptotic convergence. Around the solution, all the elements of gradient vector would be very small and there would be a very tiny weight change.

A.2.2 Newton's Method

Newton's method assumes that all the gradient components g1, g2, …, gN are functions of weights and all weights are linearly independent:

$$\begin{cases} g_1 = F_1(w_1, w_2 \cdots w_N) \\ g_2 = F_2(w_1, w_2 \cdots w_N) \\ \cdots \\ g_N = F_N(w_1, w_2 \cdots w_N) \end{cases} \tag{A.5}$$

where F1,F2, …, FN are nonlinear relationships between weights and related gradient components.
Unfold each g_i (i = 1, 2,…, N) in Equations A.5 by Taylor series and take the first-order approximation:

$$\begin{cases} g_1 \approx g_{1,0} + \frac{\partial g_1}{\partial w_1}\Delta w_1 + \frac{\partial g_1}{\partial w_2}\Delta w_2 + \cdots + \frac{\partial g_1}{\partial w_N}\Delta w_N \\ g_2 \approx g_{2,0} + \frac{\partial g_2}{\partial w_1}\Delta w_1 + \frac{\partial g_2}{\partial w_2}\Delta w_2 + \cdots + \frac{\partial g_2}{\partial w_N}\Delta w_N \\ \cdots \\ g_N \approx g_{N,0} + \frac{\partial g_N}{\partial w_1}\Delta w_1 + \frac{\partial g_N}{\partial w_2}\Delta w_2 + \cdots + \frac{\partial g_N}{\partial w_N}\Delta w_N \end{cases} \tag{A.6}$$

By combining the definition of gradient vector g in (A.3), it could be determined that

$$\frac{\partial g_i}{\partial w_j} = \frac{\partial \left(\frac{\partial E}{\partial w_j} \right)}{\partial w_j} = \frac{\partial^2 E}{\partial w_i \partial w_j} \tag{A.7}$$

By inserting Equation A.7 to A.6:

$$\begin{cases} g_1 \approx g_{1,0} + \dfrac{\partial^2 E}{\partial w_1^2}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_1 \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_1 \partial w_N}\Delta w_N \\ g_2 \approx g_{2,0} + \dfrac{\partial^2 E}{\partial w_2 \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_2^2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_2 \partial w_N}\Delta w_N \\ \cdots \\ g_N \approx g_{N,0} + \dfrac{\partial^2 E}{\partial w_N \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_N \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_N^2}\Delta w_N \end{cases} \quad (A.8)$$

Comparing with the steepest descent method, the second-order derivatives of the total error function need to be calculated for each component of gradient vector. In order to get the minima of total error function E, each element of the gradient vector should be zero. Therefore, left sides of the Equations A.8 are all zero, then

$$\begin{cases} 0 \approx g_{1,0} + \dfrac{\partial^2 E}{\partial w_1^2}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_1 \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_1 \partial w_N}\Delta w_N \\ 0 \approx g_{2,0} + \dfrac{\partial^2 E}{\partial w_2 \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_2^2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_2 \partial w_N}\Delta w_N \\ \cdots \\ 0 \approx g_{N,0} + \dfrac{\partial^2 E}{\partial w_N \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_N \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_N^2}\Delta w_N \end{cases} \quad (A.9)$$

By combining Equation A.3 with A.9

$$\begin{cases} -\dfrac{\partial E}{\partial w_1} = -g_{1,0} \approx \dfrac{\partial^2 E}{\partial w_1^2}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_1 \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_1 \partial w_N}\Delta w_N \\ -\dfrac{\partial E}{\partial w_2} = -g_{2,0} \approx \dfrac{\partial^2 E}{\partial w_2 \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_2^2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_2 \partial w_N}\Delta w_N \\ \cdots \\ -\dfrac{\partial E}{\partial w_N} = -g_{N,0} \approx \dfrac{\partial^2 E}{\partial w_N \partial w_1}\Delta w_1 + \dfrac{\partial^2 E}{\partial w_N \partial w_2}\Delta w_2 + \cdots + \dfrac{\partial^2 E}{\partial w_N^2}\Delta w_N \end{cases} \quad (A.10)$$

There are N equations for N parameters so that all Δwi can be calculated. With the solutions, the weight space can be updated iteratively.

Equations A.10 can be also written in matrix form

$$\begin{bmatrix} -g_1 \\ -g_2 \\ \cdots \\ -g_N \end{bmatrix} = \begin{bmatrix} -\frac{\partial E}{\partial w_1} \\ -\frac{\partial E}{\partial w_2} \\ \cdots \\ -\frac{\partial E}{\partial w_N} \end{bmatrix} = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_N} \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 E}{\partial w_N \partial w_1} & \frac{\partial^2 E}{\partial w_N \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_N^2} \end{bmatrix} \times \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \cdots \\ \Delta w_N \end{bmatrix} \quad \text{(A.11)}$$

where the square matrix is Hessian matrix:

$$\boldsymbol{H} = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_N} \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 E}{\partial w_N \partial w_1} & \frac{\partial^2 E}{\partial w_N \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_N^2} \end{bmatrix} \quad \text{(A.12)}$$

By combining Equations A.3 and A.12 with Equation A.11

$$-\boldsymbol{g} = \boldsymbol{H} \Delta \boldsymbol{w} \quad \text{(A.13)}$$

So $$\Delta \boldsymbol{w} = -\boldsymbol{H}^{-1} \boldsymbol{g} \quad \text{(A.14)}$$

Therefore, the update rule for Newton's method is

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \boldsymbol{H}_k^{-1} \boldsymbol{g}_k \quad \text{(A.15)}$$

As the second-order derivatives of total error function, Hessian matrix H gives the proper evaluation on the change of gradient vector. By comparing Equations A.4 and A.15, one may notice that well-matched step sizes are given by the inverted Hessian matrix.

A.2.3 Gauss – Newton Algorithm

If Newton's method is applied for weight updating, in order to get Hessian matrix H, the second-order derivatives of total error function have to be calculated and it could

be very complicated. In order to simplify the calculating process, Jacobian matrix J is introduced as

$$J = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w_1} & \frac{\partial e_{1,1}}{\partial w_2} & \cdots & \frac{\partial e_{1,1}}{\partial w_N} \\ \frac{\partial e_{1,2}}{\partial w_1} & \frac{\partial e_{1,2}}{\partial w_2} & \cdots & \frac{\partial e_{1,2}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{1,M}}{\partial w_1} & \frac{\partial e_{1,M}}{\partial w_2} & \cdots & \frac{\partial e_{1,M}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{P,1}}{\partial w_1} & \frac{\partial e_{P,1}}{\partial w_2} & \cdots & \frac{\partial e_{P,1}}{\partial w_N} \\ \frac{\partial e_{P,2}}{\partial w_1} & \frac{\partial e_{P,2}}{\partial w_2} & \cdots & \frac{\partial e_{P,2}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{P,M}}{\partial w_1} & \frac{\partial e_{P,M}}{\partial w_2} & \cdots & \frac{\partial e_{P,M}}{\partial w_N} \end{bmatrix} \tag{A.16}$$

By integrating Equations A.1 and A.3, the elements of gradient vector can be calculated as

$$g_i = \frac{\partial E}{\partial w_i} = \frac{\partial \left(\frac{1}{2} \sum_{p=1}^{P} \sum_{m=1}^{M} e_{p,m}^2 \right)}{\partial w_i} = \sum_{p=1}^{P} \sum_{m=1}^{M} \left(\frac{\partial e_{p,m}}{\partial w_i} e_{p,m} \right) \tag{A.17}$$

Combining Equations A.16 and A.17, the relationship between Jacobian matrix J and gradient vector g would be

$$g = Je \tag{A.18}$$

where error vector e has the form

$$e = \begin{bmatrix} e_{1,1} \\ e_{1,2} \\ \cdots \\ e_{1,M} \\ \cdots \\ e_{P,1} \\ e_{P,2} \\ \cdots \\ e_{P,M} \end{bmatrix} \tag{A.19}$$

Inserting Equation A.1 into A.12, the element at i_{th} row and j_{th} column of Hessian matrix can be calculated as

$$h_{i,j} = \frac{\partial^2 E}{\partial w_i \partial w_j} = \frac{\partial^2 \left(\frac{1}{2} \sum_{p=1}^{P} \sum_{m=1}^{M} e_{p,m}^2 \right)}{\partial w_i \partial w_j} = \sum_{p=1}^{P} \sum_{m=1}^{M} \frac{\partial e_{p,m}}{\partial w_i} \frac{\partial e_{p,m}}{\partial w_j} + S_{i,j} \qquad \text{(A.20)}$$

where $S_{i,j}$ is equal to

$$S_{i,j} = \sum_{p=1}^{P} \sum_{m=1}^{M} \frac{\partial^2 e_{p,m}}{\partial w_i \partial w_j} e_{p,m} \qquad \text{(A.21)}$$

As the basic assumption of Newton's method is that $S_{i,j}$ is closed to zero [29], the relationship between Hessian matrix H and Jacobian matrix J can be rewritten as

$$H \approx J^T J \qquad \text{(A.22)}$$

By combining Equations A.15, A.18, and A.22, the update rule of the Gauss–Newton algorithm is presented as

$$w_{k+1} = w_k - \left(J_k^T J_k \right)^{-1} J_k e_k \qquad \text{(A.23)}$$

Obviously, the advantage of the Gauss–Newton algorithm over the standard Newton's method (Equation A.15) is that the former does not require the calculation of second-order derivatives of the total error function, by introducing Jacobian matrix **J** instead. However, the Gauss–Newton algorithm still faces the same convergent problem like the Newton algorithm for complex error space optimization. Mathematically, the problem can be interpreted as the matrix $\mathbf{J^T J}$ may not be invertible.

A.2.4 Levenberg – Marquadt Algorithm

In order to make sure that the approximated Hessian matrix $\mathbf{J^T J}$ is invertible, Levenberg–Marquardt algorithm introduces another approximation to Hessian matrix:

$$H \approx J^T J + \mu I \qquad \text{(A.24)}$$

where

μ is always positive, called combination coefficient

I is the identity matrix

From Equation A.24, one may notice that the elements on the main diagonal of the approximated Hessian matrix will be larger than zero. Therefore, with this approximation (Equation A.24), it can be sure that matrix **H** is always invertible.

By combining Equations A.23 and A.24, the update rule of Levenberg–Marquardt algorithm can be presented as

$$w_{k+1} = w_k - \left(J_k^T J_k + \mu I\right)^{-1} J_k e_k \quad \text{(A.25)}$$

As the combination of the steepest descent algorithm and the Gauss–Newton algorithm, the Levenberg–Marquardt algorithm switches between the two algorithms during the training process. When the combination coefficient μ is very small (nearly zero), Equation A.25 is approaching to Equation A.23 and Gauss–Newton algorithm is used. When combination coefficient μ is very large, Equation A.25 approximates to Equation A.4 and the steepest descent method is used.

If the combination coefficient μ in Equation A.25 is very big, it can be interpreted as the learning coefficient in the steepest descent method (A.4):

$$\alpha = \frac{1}{\mu} \quad \text{(A.26)}$$

Appendix B

This section intends to provide information about the Neural Network Tool that was developed in MatLab environment for the purposes of our study.

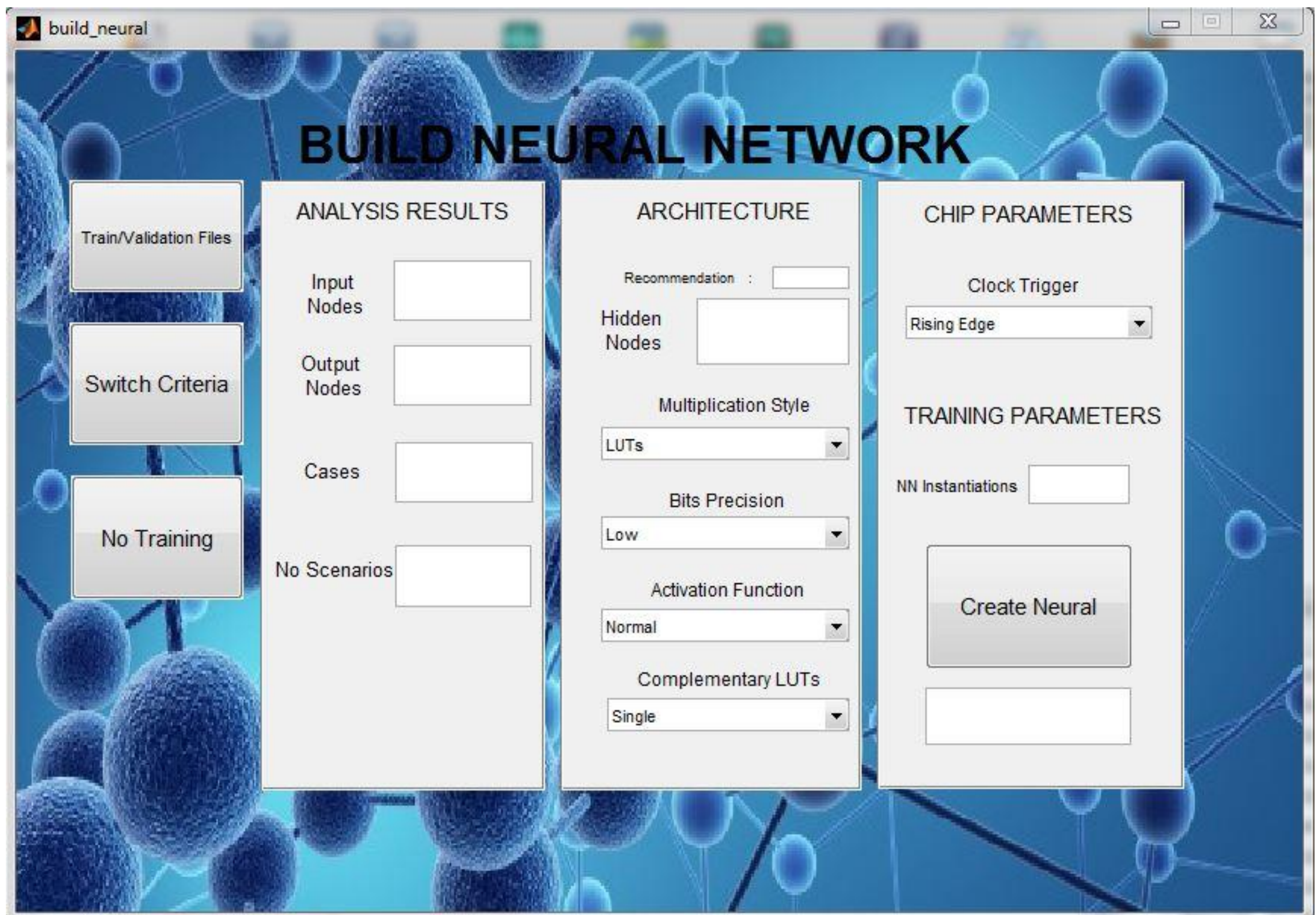


Figure B.1 Neural Network tool

Train/Validation Files

Firstly, we have to insert our input files, which should be .dat files and their format should be strictly arranged.

Training File contains the part of the dataset that we want to be engaged to the training procedure, while Validation File contains the part used to evaluate the performance of our network and its ability to generalize.

Every column represents an input, while every row represents a combination of inputs and their output. The last column stands for the output values. MatLab will accept as a delimiter a comma, a semicolon, or even a space for the separation of the values. We should note here a design restriction, which is that input values from the

dataset should be positive integer numbers, and output values should be continuous integer numbers beginning with 1.

Switch Criteria

"*Switch Criteria*" button is an optional choice. It enables a more sophisticated method of classifying, which is ruled by specific criteria, varying amongst different Scenarios. Application requests a .dat file which should keep to the following rules :

- The file should have as many set of Criteria as the number of Scenarios.
- A number specifies the number of Criteria for each set and is followed by as many lines it defines.
- 0s and 1s are used to represent changes in Inputs. Thus, line *1;0;0;1;0* implies that when the first and fourth input do not change, a change in Output is never triggered.

No Training

"*No Training Values*" button is another optional choice. It is useful in cases of having extreme values in our dataset that could slow down convergence while training the dataset, or even affect negatively in the performance of the Network. These values are given as a .dat file following the rules already explained, however it should be noted that those values should not be erased from the original dataset; they are just copied to a new file. Also, for reasons explained later, this option cannot be combined along with "*Separate*" Complementary LUTs.

Analysis Results

Underlying functions process input files and provide information about the impending Neural Network that is going to be built. *Input Nodes*, *Output Nodes*, *No Scenarios* and Cases, which is a measurement of the size of the dataset. Also, there is a box titled "*Recommendation*" which makes use of an algorithm that estimates the least number of Hidden Neurons that would provide the maximum efficiency. It is worth to mention that this is just a rough estimation; the decision about the size of a Neural Network is more like a trial – and – error process.

Multiplication Style

This option is of major significance. If it is feasible, we could choose to bypass the costly multiplications, and use LUTs instead, in a manner that we will explain later in this chapter. If the use of multipliers is necessary, though, we choose the respective option in this pop – up menu.

Bits Precision

When implementing an Artificial Neural Network in FPGA, precision is one of the most important aspects. The desired level of precision is handled as an input parameter, and throughout Simulation of the final circuit, the user could define the impact in his own design, and balance the tradeoff between less logic and better performance.

Activation Function

There are two possible options regarding Activation Function. The underlying implementations perform the same task, whereas '*Extended*' form provides better precision, because it covers twice as many cases as '*Normal*' form, thus using more logic. If there is requirement for a very – high precision implementation, then the 'Extended' form of Activation Function should be chosen. We should also note that the letter could have a slightly negative impact in Frequency.

Complementary LUTs

This option indicates the style by which the Neural Networks transforms into a Hybrid Network; that is embedding complementary LUTs in the stage of Simulation, which perform the task of covering the cases where the Neural Network itself is unable to provide the correct output.

- "Single" option infers the implementation of a large LUT, which contains all the cases of miscalculation. When this LUT is enabled, the function of NN is not triggered, and output is provided by the LUT.
- "Multiple" option infers as many LUTs as the output bits, each one holding the inputs that cause an error in this specific bit. When enabled, it does not bypass the function of NN, just inverts the erroneous bits.

Clock Trigger

By the option "*Clock Trigger*" we are allowed to define the CLK edge which triggers our final circuit. It is included to provide some flexibility for the final design.

Training Parameters

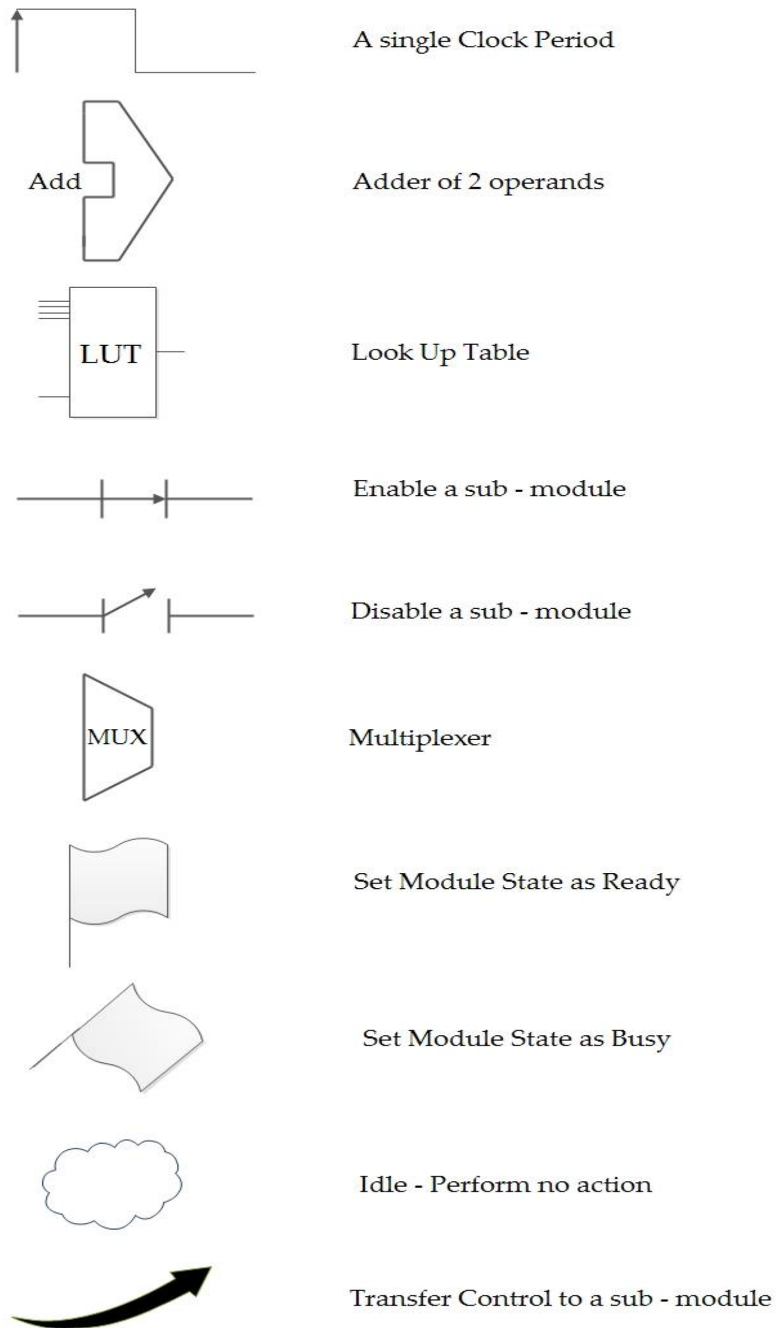
This parameter is not directly linked with the VHDL implementation, its purpose is to simplify the stage of training Neural Networks. We can determine the exact number of NN instantiations trained in our System. Out of these instances, the one with the better performance will be selected and converted into RTL description.

Create Neural

The final action when all settings have been fixed is to press "*Create Neural*" button, which will pass the chosen parameters to the built – in MatLab toolbox in order to train our Neural Network. A message appears indicating the state of the Application. While the Neural Network is being trained and afterwards evaluating the results and producing the appropriate files, the state is set to "*Processing*". If this procedure is completed without errors, the state is set to "*Successf*ul".

Appendix C

The following legend intends to define the simplified shapes used for the structural and functional analysis of the modules. Each rectangle represents a single module, while shapes imply the following functions :



* Two units of the same kind imply a parallel array of such units.