UNIVERSIDAD POLITÉCNICA DE MADRID

DOCTORAL THESIS

**Distributed artificial neural network
architecture over wireless sensor network**

Author:
David ALEDO ORTEGA

Supervisor:
Dr. Félix MORENO
GONZÁLEZ

*A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Electrical and Electronic Engineering
in the*

Centro de Electrónica Industrial

December 18, 2019

Declaration of Authorship

I, David ALEDO ORTEGA, declare that this thesis titled, "Distributed artificial neural network architecture over wireless sensor network" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

Universidad Politécnica de Madrid

Abstract

Escuela Técnica Superior de Ingenieros Industriales

Doctor of Electrical and Electronic Engineering

Distributed artificial neural network architecture over wireless sensor network

by David ALEDO ORTEGA

The objective of this thesis is to study how to distribute an Artificial Neural Network (ANN) over a Wireless Sensor Network (WSN). On the one hand, ANNs are Artificial Intelligent (AI) algorithms inspired on our brains that exhibit a great amount of parallelism. The main advantage of the ANNs is that they can learn non-linear features and patterns from examples. On the other hand, WSNs are a technology with increasing interest due to the rise of the IoT (Internet of the Things). Nodes of a WSN usually are unattended and powered by batteries in remote or difficult-to-access places. Hence, their resources are limited in order to maximize their lifetime. However, more intelligent WSNs are demanded, so, to merge these two kinds of networks seems to be natural. This thesis is about how to do it in an efficient way.

In order to achieve a fast and energy-efficient implementation of the ANNs and due to the fact that they exhibit a great amount of parallelism, implementation into FPGAs was chosen. Microcontrollers and DSP processors may be too slow, and GPUs consume too much energy. An FPGA is a flexible intermediate option. Although there is already a lot of research about hardware implementation of ANNs into FPGAs, to the best of the authors' knowledge at the beginning of this thesis there was not any freely available IP (Intellectual Property). Thus, we opt for designing our own IP, making it highly configurable to allow its use in a great variety of applications.

The methodology used on this thesis was the following:

First, to choose an application example in order to be able to get measurable data on simulations and implementations. The initial application example was to compress images using auto-encoders. It was chosen because authors had previous works on low-power compressors for WSN to compare with. Auto-encoders are ANN architectures with as many outputs as inputs and a narrow hidden layer (i.e. with less neurons than the output layer). When an auto-encoder is trained through a supervised training algorithm the inputs are taken as the desired outputs, forcing the ANN to learn a compress representation of the inputs on the narrow hidden layer. However, due to finding a way to merge images from multiple sensors into a combined one to be learned, is difficult, and as a worthy real application was not found to justify it, this application was discarded. The second example application was car and pedestrian detection with low-cost radars. This application was chosen to compare results with the thesis of another colleague from the lab. However, the amount of collected examples was not enough to properly train the ANNs. Hence, finally the popular MNIST dataset of handwritten digit classification was chosen. In

order to adapt the MNIST dataset to a multi-sensor scenario, the images was divided into 4 quadrants and each quadrant assigned to a different node of the WSN. This emulates 4 sensors catching partial information or different points of view of the same event.

Second, to adjust and optimize the algorithms and the intended designs through high-level of abstraction simulations. Matlab scripts were used for optimization and off-line training of the hardware implementations of the ANNs. And SystemC programs to simulate the WSNs.

Then, to design, verify and implement the ANN on the FPGA. Two methodologies were used for this step: the classical RTL (Register Transfer Level) using VHDL and though HLS (High-Level Synthesis) using SystemC. Therefore two IPs were obtained: one in VHDL and another in SystemC.

As a result form designing and implementing the ANNs through two methodologies, an analysis about the configurability of both methodologies was extracted. The relevance of this analysis is, that it was found that, whereas high-level descriptions are more likely to be re-used, SystemC parameterization mechanisms are less robust and flexible than VHDL ones. A good parameterization is necessary to facilitate code re-use. The parameterization mechanisms of VHDL are the generics, and the SystemC ones are macros, templates and constructor arguments. Macros are limited to scalar values, constructor arguments are not synthesizables and templates have some limitations specified on the analysis. The challenge when parameterizing an ANN is to configure separately a configurable number of modules. In this case the layers need to be separately configured (number of neurons, activation function, etc.), and the number of layers should be a parameter too. On SystemC is easily doable a separate configuration of a fixed number of modules or a configurable insertion of exactly identical (i.e. with the same configuration) modules, but not both at the same time. Complex macros have been used to solve this problem. Whereas in VHDL, using generic arrays on for-generate statements can easily solve the challenge. Key points and hints for improvement of future HLS tools have been given.

Finally, the main result of this thesis is the developed framework for distributing ANNs over WSNs. This framework consists of the Matlab scripts to optimize and train the entire ANN needed, the SystemC programs to estimate the computation vs. communication energy consumptions of the WSN with the ANN distributed in order to balance it, and the IPs to implement the ANN fragments on the nodes with FPGAs.

Acknowledgements

I want to thank in first place to my advisor Félix Moreno, and to Benjamin Carrión Schäfer, who continued supporting me and collaborating with me after the short stay.

To my students who have helped me to do this thesis: Darío Viñarás, Eduarado Lasarte, José María Pardo (Chema) and Yi Zhang. And also to my students who haven't been part of this thesis but have helped me with other projects developed at CEI: Cristina Sanz (Bibi), Pablo Medrano and Cristina Fernández; without their help I would not have time enough to finish the thesis.

To Qu Xiaoyú for being the most adorable girlfriend, cheer me up, and make me a better person.

To all my family: my parents, my sister, my aunts and uncles, all my cousins, and my grandparents (although, they cannot see it). I am lucky to have such a good family.

I also want to thank my pencils and rubber, for allowing me correcting all my design mistakes.

Contents

Declaration of Authorship	iii
Abstract	v
Acknowledgements	vii
1 Introduction	1
1.1 Motivation	1
1.2 State-of-the-art	1
1.2.1 Artificial Neural Networks	2
Multi-Layer Perceptron	2
Backpropagation	3
A little bit of history	5
Deep Learning	6
1.2.2 Wireless Sensor Networks	10
Cookies	11
ZigBee	12
1.2.3 Design methods for FPGAs: Classic RTL and HLS	13
Classic RTL	14
HLS	16
SystemC	17
1.3 Objectives	18
1.4 Methodology and tools	19
2 Hardware implementations of ANNs	21
2.1 VHDL IP	21
2.1.1 Layer modules	22
2.1.2 Weight-and-bias memories	25
2.1.3 Activation functions	27
2.1.4 Operation	28
2.1.5 Migration from EDK to Vivado	28
2.2 SystemC HLS	29
2.2.1 Layer template function	30
2.2.2 Activation functions	32
2.2.3 Weight and bias memories	32
2.2.4 Explorer	32
Macro-Explorer	33
Micro-Explorer	33
3 Description of the simulations	39
3.1 Example applications	39
3.1.1 Image compression through auto-encoder	39
3.1.2 Detection of pedestrians or cars with low cost radars	41

3.1.3	Recognition of handwritten digits	41
3.2	Optimization of single ANN with Matlab	42
3.2.1	Activation functions	45
3.2.2	Co-simulation FIL	46
	System Generator	47
	HDL verifier	47
3.2.3	Classifier ensemble	48
3.2.4	Stacking DAEs	49
3.3	Distribution of ANN over WSN with SystemC	49
3.3.1	Node layers	50
	Sensor	50
	Power	50
	Communication	51
	Processor	53
3.3.2	Environment	54
4	Results	55
4.1	Simulations	55
4.1.1	Optimization of single ANN with Matlab	55
	First results: hierarchical auto-encoder with NTA	55
	Optimization of auto-encoder for image compression	55
	activation functions	58
4.1.2	Classifier ensemble	63
	Detection of pedestrians or cars with low cost radars	63
	Adaptation of the MNIST dataset	65
4.1.3	Stacking DAEs	67
4.1.4	Distribution of ANN over WSN with SystemC	73
	Communication layer measurements	73
	Experiment 1	74
4.2	Hardware implementation of ANNs	78
4.2.1	Validation of the VHDL IP	78
	RTL simulation	78
	Validation in FPGA	78
4.2.2	Validation of the SystemC IP	80
	SystemC high-level simulation	80
	Validation in FPGA	80
4.2.3	Easy to use	80
4.2.4	Resource utilization	83
4.2.5	Explorer	86
	Image compression though auto-encoder	86
	Classification of Handwriting ANN Results	89
4.3	Analysis of VHDL vs. SystemC	90
4.3.1	Parameterization in hardware design	91
4.3.2	VHDL parameterization	92
4.3.3	SystemC parameterization	92
4.3.4	Comparison	93
5	Conclusion, future work and summary of contributions	95
5.1	Conclusion	95
5.2	Related work	97
5.3	Future work	99

5.3.1	ANN IPs	99
5.3.2	WSN simulations	99
5.3.3	HLS tools	99
5.3.4	OS API for hardware	99
	Objectives	100
	Data types	100
	Dynamic memory	101
	Calls to OS	102
	Related work	102
5.4	Summary of contributions	103
5.5	Publications	103

List of Figures

1.1	Neuron model.	2
1.2	General 2 layer feedforward ANN.	3
1.3	Filters learned by a DBN.	7
1.4	Auto-encoder.	7
1.5	Process of stacking DAEs.	9
1.6	Comparison between popular deep ANNs.	10
1.7	Cookie node with camera sensor.	11
1.8	ZigBee message from SED to COO.	13
1.9	Y chart.	14
1.10	Template of synchronous process in VHDL.	15
1.11	Scheduling strategies: ASAP and ALAP.	17
1.12	Binding example.	17
1.13	Foor loop implementations.	18
2.1	VHDL for-generate statement.	23
2.2	Block diagram around a generic layer.	24
2.3	Weight and bias memories space address.	25
2.4	Fixed point format of layer outputs.	28
2.5	Macro to select the number of neurons of the layer n .	30
2.6	Declaration of the ANN's member function <i>layer</i> .	32
2.7	ANN macro-explorer algorithm.	34
2.8	Micro-explorer overview based on GA for HLS DSE	35
2.9	ANN micro-explorer algorithm (GA HLS DSE).	36
3.1	Hierarchical auto-encoder.	40
3.2	Outer layers in NTA.	40
3.3	Inner layers in NTA.	40
3.4	Deployment of a radar for measuring.	42
3.5	Matlab vectorization.	43
3.6	Examples of basic ANN Matlab functions.	44
3.7	Original and compressed image when training gets stuck.	45
3.8	Total squared error and learning rate of stuck training.	45
3.9	Modified saturated linear derivative.	46
3.10	Block diagram of the ANN IP connexions with System Generator.	47
4.1	Example of first Matlab result of auto-encoder for image compression.	55
4.2	Selection of the training parameters.	56
4.3	Selection of the sigmoid parameters.	57
4.4	Input images for Matlab simulations of single auto-encoder.	58
4.5	Results of the Matlab simulations of single auto-encoder.	59
4.6	Best results of the Matlab simulations of single auto-encoder.	60
4.7	PSNR vs. compress ratio for 8×8 input auto-encoders.	61
4.8	Results of circle parameterization with auto-encoder.	62

4.9	Example of overfitting on the training of first-stage MLP.	64
4.10	Accuracy vs. number of neurons on the second stage (classifier ensemble).	65
4.11	Learning parameters for DAE to pre-train the first hidden layer.	68
4.12	Learning parameters for DAE to pre-train the second hidden layer.	69
4.13	Learning parameters for the fine-tuning.	70
4.14	Zoom on the area with minimum MSE on the fine-tuning.	71
4.15	Corruption in Matlab with salt and pepper noise.	72
4.16	DAE train with salt and pepper noise.	72
4.17	DAE train with Gaussian noise.	73
4.18	DAE train on 2 hidden layer network.	73
4.19	Waveforms, measured and simulated, of a 1 byte message.	74
4.20	Waveforms, measured and simulated, of polling a APS acknowledge.	75
4.21	Waveforms, measured and simulated, of polling a 4 bytes message.	75
4.22	WSN distribution simulation between two events.	76
4.23	WSN distribution simulation zoom of an event.	77
4.24	SED1 power consumption on an event.	77
4.25	RTL training simulation in ISim.	78
4.26	Image coding with auto-encoder of the Lenna picture on the Virtex5.	79
4.27	Image coding with auto-encoder of the Lenna picture on ZYBO board (Zynq).	80
4.28	Block diagram in Vivado IP integrator with the VHDL IP.	81
4.29	Configuration window of the VHDL IP in Vivado.	81
4.30	Configuration of the VHDL IP on EDK.	82
4.31	Latency vs. number of DSPs.	85
4.32	Comparison of 2 and 4 layers configurations.	86
4.33	Results of auto-encoder macro-explorer with default options.	88
4.34	HLS DSE trade-off curve for 2 and 4 layer auto-encoder.	88
4.35	Accuracy and total exploration running time in seconds for 2-layer ANN. MNIST application.	89
4.36	Results of ANN macro-explorer with default options. MNIST application.	90
4.37	Activation function exploration results.	90
4.38	HLS DSE (micro-explorer) trade-off curve result (area vs. latency) for smallest 2-layer ANN.	91

List of Tables

2.1	IO ports of the ANN VHDL IP.	21
2.2	List of the VHDL ANN IP generics.	22
2.3	IO ports of the SystemC module ANN.	30
2.4	List of the SystemC ANN IP defines.	31
3.1	List of inputs and outputs of the Matlab ANN functions.	44
3.2	Message types.	52
4.1	4-layer architectures tested.	57
4.2	Percentage of saturated values by the saturated linear activation function.	60
4.3	Results of the detection with low-cost radars.	64
4.4	Accuracy of the first stage classification. First experiments.	66
4.5	Accuracy and adaptability of the second stage when the first stage has low accuracy.	66
4.6	Accuracy of the optimized first stage classification.	67
4.7	Accuracy and adaptability of the proposed algorithms for the second stage when the first stage has high accuracy (new experiments).	67
4.8	Total energy consumed by the nodes and mean power during 48h.	76
4.9	Resource utilization of the VHDL IP on KC7K325T using <i>ISE</i>	83
4.10	Resource utilization of a system with an auto-encoder on ZYBO. Synthesized with <i>Vivado</i>	84
4.11	Resource utilization of VHDL IP for “linear” and “siglut” hidden layer, synthesized with <i>ISE</i> and <i>Vivado</i>	84
4.12	Resource utilization of SystemC IP on ZYBO.	85
4.13	Explorer qualitative results: 2 layers vs. 4 layers.	87
5.1	Summary of pros and cons of VHDL and SystemC.	96

List of Abbreviations

ANN	Artificial Nneural Network
APS	APplication Support (sublayer)
ASIC	Aplication-Specific Integrated Circuit
CNN	Convolutional Neural Network
COO	COOrdintor (device)
DAE	Denoising Auto-Encoder
DBN	Deep Belief Network
DPR	Dynamic Partial Reconfiguration
DSE	Design Space Exploration
DSP	Digital Signal Processor
FFD	Full Function Device
FIL	FPGA-In-the-Loop
FPGA	Field-Programmable Gate Array
GA	Genetic Algorithm
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
HLS	Haigh-Level Synthesis
IoT	Internet of (the) Things
IP	Intellectual Property
LUT	Look-Up Table
LSB	Least Significant Bit
MAC	Media Access Control
MAC	Multiplier-ACcumulator or Multiply-and-ACcumulate
MLP	Multi-Layer Percept
MSE	Mean Square Error
NTA	Nested Training Algorithm
NWK	NetWorK (layer)
PCA	Principal Component Analysis
PCB	Printed Circuit Board
PHY	PHYsical (layer)
PSNR	Peak Signal-to-Noise Ratio
QoR	Quality of Results
RBM	Restricted Boltzmann Machine
RTL	Register-Transfer Level
SED	Sleepy End Device
SoC	System on Chip
VRC	Virtual Reconfigurable Circuit
WSN	Wairless Sensor Network

List of Symbols

$\vec{a}^k$	activation function input (layer k)
$\vec{b}^k$	bias of layer k
e	Euler number
e^2	MSE
$f()^k$	activation function (layer k)
f'_0	slope at the origin of function f
f_{max}	maximum or asymptotic maximum of function f
f_{min}	minimum or asymptotic minimum of function f
f_r	range of the output values of function f
$\vec{s}^k$	sensitivity of layer k
$\vec{t}$	target
$\vec{y}^k$	if $k=0$ input of the network, else output (layer k)
W^k	Weights of layer k
α	learning rate
ζ	error threshold for learning rate
η	learning rate increase factor
ρ	learning rate decrease factor

Chapter 1

Introduction

1.1 Motivation

Since I have studied ANNs (Artificial Neural Networks) in an optional subject of my degree, I have been fascinated by this interesting algorithm. I wanted to use it and implement it on hardware, because it exhibits a great amount of parallelism that can be exploited on hardware implementations. As I was unable to do it on my final degree project neither on my master thesis, and as stubborn student, I proposed to do it for my PhD thesis.

On the other hand, WSNs (Wireless Sensor Networks) are a technology with an increasing interest due to the rise of the IoT (Internet of the Things). More intelligent networks are demanded, so, to merge these two kind of networks seems natural. This Thesis is about how to do it in an efficient way.

An FPGA (Field-Programmable Gate Array) implementation has been chosen to exploit the parallelism of the ANNs in a power-efficient way with enough flexibility to allow different applications and different distribution schemes among the nodes of the WSN. Microcontrollers and DSPs (Digital Signal Processors) are usually power-efficient but they cannot exploit parallelism, thus they are slow; GPUs (Graphics Processing Units) are flexible but less power-efficient; and ASICs (Application-Specific Integrated Circuits) are the most power-efficient, however they have not any flexibility at all. An FPGA is in an intermediate option which offers different trade-offs between power, parallelism and flexibility.

Due to the fact that, at my short stay of the PhD, I learned SystemC and HLS (High-Level Synthesis), these topics have also been added to this doctoral thesis.

1.2 State-of-the-art

In the following subsections, state-of-the-art of the main topics are described to put in context the Thesis. After that, in the Objectives section (1.3), the specific objectives of the Thesis can be detailed. The main topics of this Thesis are ANNs, WSNs and design methods for FPGAs.

1.2.1 Artificial Neural Networks

ANNs are algorithms that belongs to AI, because they have the intelligent capability of learning, and are biologically inspired on a simple model of a neuron. ANNs are good at generalization and dimensional reduction.

There are different families/types of ANNs. Each family has different connections, activation functions, learning algorithms, or other particularities. Work on this Thesis implements MLPs (Multi-Layer Perceptrons), which has been one of the first ANN architectures developed, the most widely used on the past years, and simple enough to fit well on hardware.

Multi-Layer Perceptron

The mathematical model of a neuron is shown in (1.1) and Fig. 1.1. This model performs a weighted sum of the inputs and then, applies an activation function to the output. The activation function is usually a linear function, or a sort of saturation function like sigmoid functions.

$$a_i = \sum_{j=1}^M w_{ij} \cdot x_j + b_i , \quad (1.1a)$$

$$y_i = f(a_i) , \quad (1.1b)$$

with $i = 1, \dots, N$, where x_j are the inputs, w_{ij} the weights, b_i the bias, $f()$ the activation function, y_i the outputs, M the number of inputs, and N the number of neurons.

FIGURE 1.1: Neuron model.

The most popular sigmoid function used as activation function is:

$$f(x) = \frac{f_r}{1 + e^{-\frac{4f'_0}{f_r}x}} - f_{min} , \quad (1.2)$$

where f'_0 is the slope at the origin of this function, f_{max} is the asymptotic maximum, f_{min} is the asymptotic minimum and $f_r := f_{max} - f_{min}$ is the range of the output values.

One neuron alone can do very few things, but combined together, they can perform complicated tasks. Like in a brain, neurons are grouped by layers and connected in a way that outputs of a layer becomes inputs for the next layer. Certain

kind of ANNs called RNNs (Recurrent Neural Networks) allow feedback connections, although the implementations described on this Thesis only allow feedforward connections. Every neuron in a layer uses the same activation function, but different layers may use different activation functions. Figure 1.2 shows a general feedforward ANN architecture. The small circles represent the inputs of the network, and the big circle nodes performs (1.1).

Last layer is called output layer, since is the one who generates the network outputs. The other layers are usually called hidden layers.

FIGURE 1.2: General 2 layer feedforward ANN.

ANNs need to be trained. During the training phase, input examples are presented to the network, and a learning algorithm updates the neuron weights in order to complete the task or improve results. Regarding on the information provided along the training examples, the ANN learning can be classified as supervised, unsupervised, or reinforcement.

For *supervised* learning, the desired outputs of each training input example are needed. So, the learning algorithm can compare them with the obtained outputs from the ANN, and act accordingly on the weights. A feedforward architecture, like the ones implemented on this Thesis, together with a supervised learning is usually called MLP.

On *unsupervised* learning, no information at all is provided along input examples. Weights are updated following rules that depends on internal characteristics of the input data. Typical application of this is clustering, and the most representative ANNs of this kind are the SOMs (Self-Organizing Maps).

Reinforcement learning is when instead of having the desired outputs for an input example, there is only a measure of how good or bad the results are.

Backpropagation

The most popular supervised learning algorithm for MLP is backpropagation [RHW85]. It is an approximate steepest descent algorithm which uses the MSE (Mean Square Error) between the ANN output and the desired one (1.3) as the fitness function.

$$MSE = \frac{1}{N} \sum_{i=1}^N (t_i - y_i)^2, \quad (1.3)$$

where t_i is the desired output (target vector), and y_i is the current ANN output.

The neuron weights are updated using the following formula:

$$\Delta w_{ij} = -\alpha \cdot \frac{\partial e^2}{\partial w_{ij}}, \quad (1.4)$$

where e^2 is the MSE, and $\alpha \in [0, 1]$ is a scalar coefficient called learning rate which regulates how fast the ANN learns.

Before continuing deriving the backpropagation formulas, let's put previous equations on matrix notation and add superscripts to indicate to which layer the variables belong. Equations (1.1) and (1.3) become (1.5) and (1.6) respectively.

$$\vec{y}^k = f^k(W^k \cdot \vec{y}^{k-1} + \vec{b}^k), \quad (1.5)$$

$$e^2 = (\vec{t} - \vec{y}^L) \cdot (\vec{t} - \vec{y}^L), \quad (1.6)$$

where L is the number of layers, $k = 1, \dots, L$ indexes layers, $\vec{y}^0$ is the input of the network, and factor $1/N^L$ has been removed because it will be absorbed by the learning rate α .

The weight increments of the output layer are easy to obtain by deriving (1.4) using (1.6) and (1.5) to substitute the proper terms:

$$\Delta W^L = 2\alpha \dot{f}^L(\vec{a}^L) (\vec{t} - \vec{y}^L) \vec{y}^{L-1}. \quad (1.7)$$

However, to obtain the weight increments of the hidden layers, the chain rule must be applied:

$$\frac{\partial e^2}{\partial W^k} = \frac{\partial e^2}{\partial \vec{a}^k} \cdot \frac{\partial \vec{a}^k}{\partial W^k}. \quad (1.8)$$

In order to simplified maths, sensitivities are defined as

$$\vec{s}^k := \frac{\partial e^2}{\partial \vec{a}^k}. \quad (1.9)$$

Sensitivity of the output layer is again easy to get:

$$\vec{s}^L = -2 \dot{f}^L(\vec{a}^L) (\vec{t} - \vec{y}^L). \quad (1.10)$$

But for calculating sensitivities of the hidden layers, as mentioned before, the chain rule must be applied:

$$\vec{s}^k := \frac{\partial e^2}{\partial \vec{a}^k} = \frac{\partial e^2}{\partial \vec{a}^{k+1}} \cdot \frac{\partial \vec{a}^{k+1}}{\partial \vec{a}^k} = \vec{s}^{k+1} \cdot W^{k+1} \dot{f}^k(\vec{a}^k). \quad (1.11)$$

Therefore, sensitivities can be calculated from the output layer, using the next layer sensitivity (previously calculated) to the first layer. This is the fact about back-propagation takes its name: *error back-propagation*.

Then, the weight increments are obtained from (1.4), (1.8) and (1.12). Calculation of the bias increments is analogue. The only difference is the term (1.12b).

$$\frac{\partial \vec{a}^k}{\partial W^k} = \vec{y}^{k-1}, \quad (1.12a)$$

$$\frac{\partial \vec{a}^k}{\partial \vec{b}^k} = 1. \quad (1.12b)$$

The weight update is performed in batch mode. This means that weight increments for each training example are accumulated, and then applied after all the examples (an epoch). The learning rate is also updated at the end of an epoch, following these rules:

1. If the new MSE is less than the previous one, the learning rate is increased η times.
2. Else if the new MSE is greater than the previous one, but no more than a certain percentage ζ , the learning rate is maintained.
3. Else the new MSE is greater than the previous one plus a certain percentage of it (ζ), learning rate is decreased ρ times, and also weights from the previous epoch are recovered.

This is a variation of the standard backpropagation, called backpropagation with variable learning rate, which can be found on [HDB96].

If all initial weights and biases are zero, backpropagation algorithm does not evolve. Therefore, usually they are initialized by small random values.

A little bit of history

The first ANN models were published in 1943 by the neurobiologist Warren McCulloch and the statistician Walter Pitts [MP43]. A few years after, Donald Hebb develop his theories about learning. In the 50's Frank Rosenblatt developed the Perceptron. And later, Bernard Widrow and Marcian Hoff developed ADALINE (ADaptive LInear NEuron) and the two layers version MADALINE (Many ADALINE). These works are the basis of the ANNs.

In the 60's research on ANN decay due to the Marvin Minsky and Seymour Papert's study about Perceptron limitations [MP69]. The main problem was that the one layer Perceptron cannot learn how to solve the XOR problem (because it is not linearly separable), and the current learning techniques were unable to train Perceptrons with two or more layers.

The first one who proposed a steepest gradient descent algorithm to train ANNs was Paul Werbos in his doctoral thesis at 1974. However, this work passed unnoticed. Interest on ANNs had not arisen again until 1986, when David E. Rumelhart *et al.* [RHW85] made Backpropagation popular. With the Backpropagation learning algorithm was possible to train multiple layers with non-linear activation functions, and hence to solve the XOR problem.

Deep Learning

Deep learning is a tag for ANNs of *many* layers (Let's define later how many layers are *many*). Although the idea behind deep learning can be applied to other machine learning algorithms that can be divided into layers, ANN is the almost only one used for deep learning.

But before explaining what deep learning is about, I want to remark the following theorem (demonstrated here [HSW89]):

Theorem 1.1 *An ANN of two layers with enough hidden neurons can approximate any multivariate continuous function with arbitrary accuracy.*

Furthermore, when Backpropagation is used to train an ANN with a lot of layers a problem called *the gradient vanishing problem* appear. The cumulative backpropagated error signals decay (or explode) exponentially as the number of layer decrease (i.e. when is farther from the output layer). This phenomena is caused because the sensitivities are proportional to the derivative of the activation function and the sensitivity of the next layer (previously calculated by the same method on Backpropagation algorithm) (1.11). As sigmoid function is the most used activation function, and its derivative belongs to $[0,1]$, sensitivities decay on each step of the Backpropagation algorithm. Hence, weights of the input layers almost have not moved from its original random positions while weights of the output layers have already arrived to its optimal position. If the activation function has a derivative greater than 1, the sensitivities will explode, causing as well training problems. It was discovered by Sepp Hochreiter in his doctoral thesis [Hoc91].

Putting together theorem 1.1 and the gradient vanishing problem, the question is: Why bother using more than two layers?

First of all, regarding the theorem 1.1, *enough hidden neurons* may be too many and unpractical for some applications. Besides, a lot of hidden neurons cause extremely long training times.

But the key answer to the question is: by adding more layers can **more easily** be got more abstract features. For example, when processing images, first layers can easily detect dots and strokes, next layers can combine them to produce polygons, circles, etc... and so on, until they even can detect something as abstract as a cat! Google researchers achieved a deep ANN that learned unsupervisedly to identify cats on images from YouTube videos [Le+12]. Figure 1.3 shows how more abstract features are obtained layer by layer on an application for classification of handwritten digits [Erh+10].

To overcome the gradient vanishing problem massive computing is performed on GPUs. That is what I call: to solve the problem by brute force. Nevertheless, nowadays this is actually the most widely used method.

Although there are also methods that really solve the gradient vanishing problem. They are based on unsupervisedly pre-train the network in a layer-wise way. This means pre-training layer by layer from the input ones to the last ones. Doing so, general features from the input data are extracted, from low abstract ones like dots and strokes (on first layers) to others with higher level of abstraction (more abstract as more layers are added). In order to adapt the obtained ANN to the task to be solved, a fine-tuning supervised trained is carried out after the unsupervised pre-train. This last training adapts outer layers to focus on those features that have

FIGURE 1.3: Visualization of filters learned by a Deep Belief Network trained on InfiniteMNIST, after supervised fine-tuning; from left to right: units from the 1st, 2nd and 3rd layers, respectively [Erh+10].

interest to the task, but due to the gradient vanishing problem first layers are not modified essentially. Nevertheless, the origin points of the first layers are not longer random ones, they extract basic low-level features that are general enough for most of the imaginable tasks. The first one to come with this technique was Hinton, *et al.* [HOT06]. However, in this proposal the basic elements are not neurons like the described on section 1.2.1. Instead of, he uses RBMs (Restricted Boltzmann Machines) which are more complex computationally. Networks composed of RBMs are called DBNs (Deep Belief Networks) [Hin07].

The previous idea of layer-wise unsupervised pre-train was then used on MLPs. In order to train a layer unsupervisedly, the auto-encoder architecture is used. An auto-encoder is a MLP with as many outputs as inputs and a narrow hidden layer. The inputs are used as well as desired outputs, therefore there is no need for labeled training examples, and so, it can be considered unsupervised learning. As data is forced to pass through the narrow hidden layer, the auto-encoder has to learn a compressed representation of the data and how to recover them from the compressed representation. Figure 1.4 shows auto-encoder architecture.

FIGURE 1.4: Auto-encoder.

For deep learning, only the encoder part is needed. Decoder is discarded, and the encoder layer become a new layer of the deep ANN. Then, a new auto-encoder is training with the outputs of the staked previous encoders, to stack a new layer, and so on.

These basic auto-encoders are really good for data coding and compression, however they have a limitation on the number of neurons they can pre-train. If the new layer is desired to have equal or greater number of neurons than the previous one (e.g. to get more useful features) there is a risk of getting an identity mapping, i.e. an ANN which simply copy the input to the output. An identity mapping gets an optimum MSE of zero, however is totally undesirable because it does nothing. The objective is to get a more useful representation of the inputs, not the inputs themselves. A new representation of the inputs is more useful if it leads to better performance after the fine-tuning than random initial weights.

First attempts of avoiding identity mapping were to force sparse coding. It can be achieved by limiting the active neurons (neurons with output 1 or close to 1 against neurons with 0 output or close to it), or the amount of all outputs added together. However, the extra math complication that is added on the training to achieve it, do not worth the obtained results.

Pascal Vincent, Hugo Larochelle, *et al.* [Vin+10] came up with a simple idea which outperforms sparse coding and get results equal or better than DBNs: the DAE (Denoising Auto-Encoder). It consists on corrupting inputs with random noise before introducing them on an auto-encoder, but using the uncorrupted inputs as desired outputs. The main idea behind that is to find *a good representation that can be obtained robustly from a corrupted input and that will be useful for recovering the corresponding clean input*, it thus forces the learning of a far more clever mapping than the identity: one that extracts features useful for denoising [Vin+10]. Figure 1.5 shows the process of stacking DAEs to build a deep ANN. Notice that noise is only added on pre-train stage, once weights of the deep ANN has been initialized, there are no corruption neither of inputs nor between layers any more.

At this point, introduction of the state-of-the-art about ANN for this Thesis may finish here. Stacking DAEs is the technique which has been used on the Thesis. Although it is not the most novel technique on the-state-of-the-art of deep learning, it is sufficient for the ANNs tested on this Thesis, which are small due to the limited resources of the WSNs. However, now that we have already introduced the interesting topic of deep learning, let's continue to review the latest deep architectures.

Prior to introduce the newest deep architectures, it is necessary to point out a quite old one: the CNN (Convolutional Neural Network). It had success at its time for processing images, but has been omitted on propose until this point, because now it has become a fundamental basic layer to build the state-of-the-art deep ANNs.

The CNN architecture was proposed by Fukushima on the Neocognitron [Fuk80]. And then, Yann Le Cun used backpropagation to train it [LeC89; LeC+89; LeC+98] (which is almost identical to the nowadays basic CNN layers). [LeC+98] also introduced the MNIST database, which is the most used benchmark for machine learning. The CNN architecture has three kind of layers: convolutional, pooling and fully connected (the last ones are exactly the same as MLPs).

The convolutional layers, which are the most characteristic ones to the architecture, filter the input images (or whatever other N-dimensional inputs) with small patches of neurons. For example, 3×3 or 5×5 patches (or an equivalent N-dimensional one). The small neuron filters scan all the input, producing the so called feature maps. By doing that, the weights are *spatially* shared, reducing considerably the number of parameters to be learned.

FIGURE 1.5: Process of stacking DAEs.

Pooling layers reduce the number of data to the next layer. It summarizes a small region (typically a 2×2 patch) into one element by calculating the average, the maximum or a similar operation. Historically, average pooling had been used, although recently has been realized that max pooling performs better. In addition, downsampling along with the convolutional layers gives the CNN spatial invariance (i.e. a certain feature can be detected independently of its position), and a certain invariance to small perturbations like rotations or scales.

Those layers are successively repeated, and finally a fully connected network (i.e. a MLP) is added at the end to get the final result.

The most popular architectures are based on the winners of the ImageNet competition [Den+09; Rus+15]: the first deep CNN which achieved to outperform the other machine learning and image processing algorithms was AlexNet [KSH12] in 2012; in 2014 the winner GoogLeNet or Inception [Sze+15] and the second winner VGG [SZ14] become very important architectures; and the winner in 2015 ResNet [He+16]. Figure 1.6 from [CPC16] show a comparison between some of the most popular deep ANNs.

Another successful deep architecture is a RNN based on short-term and long-term memories, that has been used on speech recognition. However, this one is out of the scope of this Thesis introduction.

FIGURE 1.6: Comparison between popular deep ANNs. The vertical axis is the accuracy on the ImageNet competition benchmark, the horizontal axis the amount of operations for a single forward pass, and the size of the blobs is proportional to the number of network parameters [CPC16].

1.2.2 Wireless Sensor Networks

A WSN is (as its name suggest) a network of sensors which communicate themselves by wireless links. Ideally sensors should be invisible, unintrusive, eternal, inexpensive, etc. Unfortunately, with today technology it is impossible. But at least, they should be as less intrusive as possible, hence small; low-cost; and perdure as much as possible, hence low-power. Furthermore, these sensors in many cases will be placed on remote and/or difficult-to-access places, so they have to be autonomous too. That implies they cannot rely on any infrastructure (that is why they are wireless), and they will be powered by batteries in most cases.

To be powered by batteries, in turn, implies that sensor life-time is a key feature (maybe the most important one). Because change batteries from tens or hundreds of sensors every weak, month or few months is impractical.

Because low-power radios have a limited range, extra nodes are added to the network just for routing. It can be proved that to transmit a message on multiple hops of less transmission power may be more efficient than on one long hop [Rab+00].

In order to enhance life-time of the sensors, usually they are kept on an ultra low-power sleep mode or even off during most of the time. Waking them up only when it is necessary. Under this scenario, the most important parameters for prolonging the sensor life-time are: power consumption of sleep modes, frequency of awakening and **energy** consumed by tasks. Energy, needed for waking up is also important in relation with its frequency. It is important to notice that as a sensor node is in

a sleep mode most of the time, and the batteries have a certain amount of energy, the key point is the energy consumed by tasks and not the power consumed while they are being executed. This means that sometimes it is preferable to execute tasks at high-power but faster, if the energy consumed ($\text{power} \times \text{time}$) is smaller.

The communication network of a WSN is very dynamic. Because they are designed to use the minimum transmission power possible, changes on the environment makes wireless links changes (loosing ones while making another ones possible). Some nodes eventually will run out of energy and “die”, and others can be moved or being continuously moving. This makes network routing a challenge in WSNs. However, this challenge is out of the scope of this Thesis. There are already some techniques and protocols to solve this problem, but I will not discuss them here.

WSNs started with projects founded by the DARPA (Defense Advanced Research Projects Agency of the USA) in the 1990s. The first one WINS (Wireless Integrated Network Sensors) [PK00]. And then, the Smart Dust [KKP99] and Pico-Radio [Rab+00] projects from the University of California Berkeley, that envisioned “motes” of less than 1 cm^3 . Nowadays, WSNs are a subset of the IoT.

The most popular platform, at least for research purposes, is the TelosB from the Berkeley University of California [PSC05]. However, in this Thesis I made use of the *Cookies* platform developed in the Centro de Electrónica Industrial, which is described in the following subsection.

Cookies

Cookies [Por+06] is the WSN platform developed in the Centro de Electrónica Industrial. Its main contribution is the configurable hardware of the nodes, based on interchangeable layers. These layers are small PCBs (Printed Circuit Boards) that can be connected forming an stack. Figure 1.7 shows one cookie node.

FIGURE 1.7: Cookie node with camera sensor.

There is 4 main kinds of layers:

- Sensor layers: for the sensor or combination of sensors of the node.

- Power layers: for power supply and energy management of the node. Some these layers include batteries.
- Communication/network layers: For the communication modules of the node. Mainly wireless transceivers like ZigBee, WIFI, Bluetooth, ...
- Processing layers: for the main computations/program of the node. Different processing layers contains different microcontrollers and/or small FPGAs.

ZigBee

ZigBee is a communication protocol designed for WSN and similar communication scenarios (low-power, multi-hop, dynamically adaptable networks). It is build on top of the IEEE 802.15.4 standard [802], which defines the PHY (PHYsical) layer and MAC (Media Access Control) sublayer of the communication protocol stack. ZigBee [Zig] defines the NWK (NetWork) layer and the APS (APplication Support) sublayer.

The IEEE 802.15.4 differences two basic kinds of nodes: FFDs (Full Function Devices) and RFDs (Reduced Function Devices). As their names indicate, FFDs implement all the functionality, whereas RFDs are used as SEDs (Sleepy End Devices). A SED is in sleep mode most of its lifetime to minimize power consumption. It wakeup eventually (typically periodically, but sometimes as reaction of an external event) and communicates with its parent (which must be a FFD). A *parent* node should keep the messages for its *children* nodes a certain amount of time. When a SED wakes up, should poll its parent for pending messages.

A network is initiated by a FFD that receives the name of COO (coordinator). Each other FFD is called router. Although it is not mandatory, for simplicity, let's assume that the COO is also the gateway responsible for the communication outside the network.

Before sending every message (except MAC acknowledges and poll replies), the node should perform the CSMA-CA (Carrier Sense Multiple Access with Collision Avoidance) algorithm. It consist on listening the channel a random small number of small periods of time called backoffs. If the channel is quiet, the message can be sent. If there are interferences (e.g. another message from other node), the node should listen again for an increasing number of backoffs. In the rare occasions that after a long listening time, i.e. after the maximum number backoffs, the channel is not quiet, a failure is reported and the message is not sent.

Messages are acknowledged twice (at MAC and APS levels). The MAC acknowledge is sent just after receive a message by the next hop in the route. The channel is already free so CSMA-CA is not needed. The APS acknowledge is sent by the final destination as a data message for the MAC layer. Hence, an APS acknowledge produce MAC acknowledges back at every hop. Figure 1.8 shows the ZigBee sequence of a successful message from a SED to the COO through a router (FFD).

If a MAC acknowledge is not received the message is re-sent a maximum of 4 times (original try + 3 retries). It may either because the message is lost or because the MAC acknowledge is lost. To distinguish similar messages from copies of the same message, every message has a DSN (Data Sequence Number). Retries of a message uses the same DSN.

FIGURE 1.8: ZigBee message from SED to COO.

If a APS acknowledge is not received after a certain amount of time, calculated for allow the message hop forward and back enough times (it depends on the maximum network size), the message is re-sent a maximum of 3 times. In addition, although it is not part of the ZigBee standard, if no acknowledge is received (either MAC or APS) the ETRX2 firmware tries to send the message again 5 times at intervals of about 50 ms.

Broadcasts are sent 3 times, with the same DSN, and they are not acknowledged. The radius parameter of the broadcast indicates how many hops it should be transmitted. Every node that receives the broadcast first checks if it has been already received, if not it decreases the radius by 1 and retransmit the broadcast unless the radius become 0.

1.2.3 Design methods for FPGAs: Classic RTL and HLS

Although the HDLs (Hardware Description Languages) are the same for design ASICs (Application-Specific Integrated Circuits) and FPGAs, the code style may vary a little. A general HDL code can be synthesized on both FPGA and ASIC¹. However, a good optimized design is slight different if the target is an FPGA or an ASIC. The main difference is the technology, specially the basic construct elements.

¹In the case of full-custom ASIC, this synthesis is not the last step of the design flow. I.e. whereas for FPGAs and semi-custom ASICs the next steps are totally automated or heavily assisted, that is not the case for full-custom.

They are typically logic gates or standard cells for ASICs, whereas they are configurable LUTs (Look-Up-Tables) for FPGAs. This is not a big difference, but must be taken into account for some optimizations. Besides, recently, a grater difference has appeared: dynamical and partial reconfiguration on SRAM based FPGAs. I think this fact will radically change design methods for FPGAs in the near future, although it has not been applied to this work.

This subsection targets design methods for FPGAs because I have never designed an ASIC, and all my knowledge and experience is about FPGA design.

There are different level of abstraction for hardware design. Similarly to software design, in which levels of abstraction go from assembler, to program languages like C or Fortran, and then to object oriented programming (C++, Java, ...), etc. With low level of abstraction, better optimizations can be performed taking advantage of low level details of the particular technology involved. On the other hand, with high level of abstraction can be designed huge systems faster and with fewer lines of code. Daniel D. Gajski *et al.* in [Gaj+09] define the Y chart (Fig. 1.9), where there are represented different levels of abstraction for hardware design (circuit, logic, processor and system); and three design points of view (behavior, structure and physical).

FIGURE 1.9: Y chart [Gaj+09].

Classic RTL

Since the late 80s, RTL (Register-Transfer Level) has been the most used level of abstraction for hardware design. And VHDL and Verilog has been the most popular HDLs.

RTL can be placed in the Y chart between processor and logic levels, on behavior and structure domains. Basically, it consists on define precisely functionality and

signals at every clock cycle. Functionality and internal signals between clock cycles may be abstracted. Hence, it is not necessary to place every logic gate (or LUT). For example, an addition or multiplication operator (+ or * respectively) can be used to abstract some logic operations, with the condition that the result is clear at the end of the clock cycle. The elements who preserve a signal to the next clock cycle are synthesized as flip-flops. Several flip-flops together form a register, so this is the origin of the name for this level of abstraction.

In order to achieve a good synchronous design in VHDL, template of Fig. 1.10 is used to design concurrent elements. It is a process with only the clock signal in the sensitivity list. The first thing it does is to check for a rising edge event of the clock signal, and then checks for a synchronous reset. It is executed only if there is a rising edge event. If reset is active initializations are executed, else the logic and functionality described in the process.

```
process (clk)
  -- Variable declarations
begin
  if clk'event and clk = '1' then
    if rst = active_lvl then
      -- Reset initializations
    else
      -- Computations
      -- ...
    end if;
  end if;
end process;
```

FIGURE 1.10: Template of synchronous process in VHDL.

Apart from the common ones, like comment properly the code, here there is a list of good practices for hardware design with VHDL:

- Ports should be always of a standard type: `std_logic` or `std_logic_vector`.
- Parameterize the design as much as possible with generics. E.g. bit widths of ports. Usually it should be parameterized all aspects that adapts the design without changing its functionality. Although sometimes, alternative functionalities are also parameterized through boolean generics.

Design of a system starts with algorithm definitions. Their refinement, adaptation and optimization is performed at the highest level of abstraction, typically through mathematical tools like Matlab or high level programming languages like Python, C++, Java, etc. This can be called the algorithm level, and it is placed (on the Y chart) at the system level of the behavior domain. Once algorithms are defined and it is wanted to implement them on hardware, RTL codes are developed using VHDL or Verilog. Simulation tools at RTL are used to validate the design, and finally it is synthesized on an FPGA to be tested.

That is the classic RTL method. It is well consolidated, has been used for more than 30 years, and there are many experts and professional tools.

HLS

However, translation from algorithm descriptions (on high abstraction languages as mentioned before) to RTL HDLs, if it is done manually is a time-consuming and error-prone task. HLS (High-Level Synthesis) is the automation of part of this process. Hence, it is intended to eliminate the productivity gap between algorithm specification and RTL design. It starts on the behavior processor level (in the Y chart), with a behavioral description ideally untimed, and automatically generates an RTL code. Although, HLS with SystemC (see below) uses partially timed descriptions.

First generations of HLS tools used HDL for input sources, but they were a commercial failure due to the reasons explained in [MS09]:

- They had not enough QoR (Quality of Results) yet. They were a little bit unpredictable and there was not well understanding about the factors who influence synthesis. They produce reasonable results for data flow designs, however for designs with complex control systems results were not good enough.
- It was assumed to they be used by hardware designers (experts on RTL), which were willing to switch only if it provides great improvement on QoR for similar effort, or similar QoR with less effort and smooth learning curve. That was not the case at that time.
- For software engineers and algorithm designers, to learn a HDL was a hard task.

Present HLS tools uses C/C++ based languages, its technology has mature, and have been improved the QoR specially on control designs. However from all of the C/C++ based languages used for HLS, only SystemC is standard for HLS through the synthesizable subset [Syn]. Even being some HLS tools which accepts ANSI C++, like *Vivado HLS* and *CatapultC*, their synthesizable subsets and some C++ translations into hardware (data types, interfaces,...) are not standard. Each HLS tool that accepts pure C++ synthesizes it on its own *tool-dependent* way. Thus, only SystemC is portable among HLS tools.

HLS is not yet a silicon compiler who transforms whatever algorithm into a chip without manual refinement, although it almost eliminates this productivity gap. Guidance through optimization is needed, and translation from standard C/C++ algorithm descriptions into synthesizable C like descriptions is still needed. Nevertheless, this translation is more straightforward than translation from C/C++ to VHDL or Verilog, thus easier, faster, and less error-prone.

The main steps of HLS are allocation, scheduling and binding. During allocation, hardware resources that are necessary are identified, specified and listed. It is like creating a bag with the blocks that will be necessary. Then, during scheduling, the time at each operation should be performed is determined, such that no precedence constraint is violated. Figure 1.11 shows two basic scheduling strategies: ASAP (As Soon As Possible) and ALAP (As Late As Possible). Finally, during binding, operations are mapped to specific functional units (reusing them as necessary) and variables to registers. Figure 1.12 shows an example of a possible binding.

Furthermore, HLS can synthesize one description into different RTL architectures. For example, two of the main features of HLS to obtain different architectures are array mapping and loop unrolling (Fig. 1.13). Arrays can be synthesized

FIGURE 1.11: Scheduling strategies: ASAP and ALAP.

FIGURE 1.12: Binding example.

as RAM/ROM memory blocks, register benches, or even as shift registers or FIFOs. If there is not data dependences between loop iterations, a loop can be unrolled to achieve a parallel implementation with maximum performance but larger area. Alternatively, it can be kept its serial nature to allow reuse of the hardware described inside the loop, achieving minimum area. Besides, it can be partially unrolled by implementing some parallel copies of the loop's hardware, and reusing them serially to complete the remaining loop iterations. This shows very well how different trade-offs area versus performance can be achieved from a single loop.

HLS makes easier and faster DSE (Design Space Exploration), and allows its automation.

SystemC

SystemC is a C++ library to model hardware. It defines useful data types like arbitrary precision integers and logic vectors; mechanisms to explicitly describe

```
for  $i = 0$  to  $N$  do
```

```
    Outputs[i]  $\leftarrow$  Task( Inputs[i] )
```

```
end for
```

FIGURE 1.13: For loop and its rolled (left), partial unrolled (middle), and unrolled (right) implementations.

concurrency (modules, threads and methods); synchronization (wait function); and communications elements (signals, ports, interfaces, channels...).

The first version of SystemC was just another RTL language. It defined the simulation kernel, modules, signals and fixed point data types. However, as RTL languages VHDL and Verilog are more powerful. Nevertheless, current version of SystemC includes higher-level abstraction elements which make SystemC capable of target a wide range of abstraction levels. The SystemC's wide range of abstraction levels is very useful for simulation and validation. Whereas the unit under test is detailed on a low level of abstraction, other parts of the system can be kept on a high abstraction level to speed up the simulation.

Although all the C++ and SystemC features can be used on simulations for validation, for HLS the synthesizable subset [Syn] restricts the supported syntax. E.g. dynamic memory allocating is not allowed, as well as recursion or floating-point number representations.

SystemC descriptions has a slight lower abstraction level than C++ descriptions for HLS (at least than the Vivados's ones). C++ are truly untimed algorithm descriptions. No concurrency is explicitly described, so synchronization is not needed. Data dependencies and concurrency is extracted by the tool, thus designer should rely on the tool. SystemC allows designers explicitly describe concurrency and synchronization, giving more control on HLS at expenses of reducing slightly the abstraction level. This has its advantages and drawbacks.

1.3 Objectives

The main objective of this Thesis is to distribute an ANN over a WSN. By doing that, is expected to achieve the following derived secondary objectives:

- Reduce computational burden of each node.
- Improve fault tolerance, since an ANN when it loses neurons, it reduces its accuracy but still continue working.
- Improve security, for example, if a message is captured it cannot be understood without the trained ANN who generated the message.

In order to achieve it, the following secondary objectives are needed:

- An ANN low-power implementation, flexible enough to allow different applications be partitioned among the nodes of the WSN. At least, the following parameters should be configurable:
 - Number of layers.
 - Number of inputs of the network.
 - Number of neurons of each layer.
 - Activation function of each layer.
- To define a methodology and build a framework to test the effects of different distribution schemes for a particular application.

1.4 Methodology and tools

In order to achieve the main objectives listed on section 1.3, hardware implementations of ANNs must be carried out. First, a classic RTL design methodology has been used. Then, after the author short stay in the Polytechnic University of Hong Kong, an attempt of using HLS has been done.

Independently of the level of abstraction on which synthesis is performed, every design starts with software simulations to refine the algorithms. These simulations have been developed in Matlab, as exception of WSN simulations which has been carried out in SystemC. Matlab is both a language and a tool that provides the highest level of abstraction for the initial phases of the design process. Hence, the speed and mathematical power needed. It is important to notice that even at a high level of abstraction, training of an ANN is a quite time-consuming task. It took around 3 or 5 hours for each train of the proposed ANNs, and several has been done during optimization and parameter selection phases.

With the help of the Matlab simulations, ANNs algorithms have been adapted to the embedded hardware expected on a WSN's node. Then, optimizations and parameter selection have been performed through exhaustive simulations. Finally, it is obtained a functional golden reference and also weight and bias files for the hardware implementations with only feedforward stage.

First design and implementation of the ANN IP (Intellectual Property) in VHDL has been done with Xilinx's tools *ISE* and *EDK* (Embedded Development Kit). *ISE* has been used for design and development of the IP, and then *EDK* has been used to integrate it in a SoC (System on Chip) and the final implementation into an FPGA.

Due to the fact that during elaboration of the research, a new tool appeared for the newest FPGA families of Xilinx (*Vivado*), a porting of the IP to this new tool was necessary.

Simulations, design and HLS with SystemC has been performed with Xilinx's *Vivado HLS*.

In order to test and show results, an example application is needed. The first example application chosen was image compression through auto-encoder (section 3.1.1). Because of reasons explained on section 3.1.1, example application was changed to detection of pedestrians or cars with low cost radars (section 3.1.2). And the final example application has been handwritten digits recognition (section 3.1.3).

Chapter 2

Hardware implementations of ANNs

Although algorithm simulations have been carried out before hardware design, one of the objectives of the simulations is to adapt the algorithms to the hardware in an iterative process. Hence, an understanding of the designed hardware will help to better understand the description of the simulations. So, that is the reason because this chapter *hardware implementations of ANNs* comes before chapter 3 *Description of the simulations*.

2.1 VHDL IP

The objective of this IP is to describe an ANN as general as possible for reusing it on different applications. Only the feedforwards stage is desired to be accelerated on hardware. On the contrary, training is carried out off-line or with the help of a microprocessor, since it is a sequential task that is executed occasionally.

TABLE 2.1: IO ports of the ANN VHDL IP.

Port	Width	Direction	Description
reset	1	Input	Synchronous active high reset input
clk	1	Input	Clock input
run_in	1	Input	Start and input data validation
m_en	1	Input	Weight and bias memory enable ^a
m_we	1	Input	Weight and bias memory write enable ^a
inputs	NbitIn	Input	Input data ^b
wdata	NbitW	Input	Weight and bias memory write data ^a
addr	addr_l ^c	Input	Weight and bias memory address ^a
run_out	1	Output	Output data validation
rdata	NbitW	Output	Weight and bias memory read data ^a
outputs	NbitOut	Output	Output data ^b

^a BRAM interface.

^b Serial stream.

^c The address length is calculated with the formula on Eq. 2.1.

The IP is divided into two parts: a wrapper and the core. On the one hand, the core is the optimized ANN with the basic interface shown on table 2.1. On the other hand, a wrapper connects that basic interface of the core with the system bus interfaces. In this work, one wrapper connects to PLB v4.6 on *EDK* and another wrapper to AXI on *Vivado*.

Table 2.2 summarizes the parameters which configure the ANN IP. There are two kinds of parameters: those which affect the whole ANN (*Nlayer*, *NbitW*, *NumIn*, *NbitIn* and *NbitOut*), hence they are scalars; and those which affect to each layer separately (*NumN*, *l_type*, *f_type*, *LSbit* and *NbitO*), so they are vectors. *NbitW*, *NbitIn*, *LSbit*, *NbitO* and *NbitOut* configure fixed point data formats, thus they do not represent a big challenge. The rest of the parameters configure the ANN structure and functionality.

TABLE 2.2: List of the VHDL ANN IP generics.

Parameter	Type	Description
<i>Nlayer</i>	integer	Number of layers
<i>NbitW</i>	natural ^a	Bit width of weights and biases
<i>NumIn</i>	natural	Number of inputs to the network
<i>NbitIn</i>	natural	Bit width of the inputs
<i>NumN</i>	int_vector ^b	Number of neurons in each layer
<i>l_type</i>	string ^c	Layer type of each layer
<i>f_type</i>	string ^d	Activation function type of each layer
<i>LSbit</i>	int_vector ^b	LSB of the output of each layer
<i>NbitO</i>	int_vector ^b	Bit width of the outputs of each layer
<i>NbitOut</i>	natural	Bit width of the network output

^a *NbitW* should be a multiple of 8 due to memory alignment.

^b *int_vector* is an array of integers defined on the package *layers_pkg.vhd*.

^c *l_type* string must contain as layer types as number of layers, separated by spaces. Each layer type is a two character string "SP", "PS", or "PP".

^d *f_type* string must contain as activation function types as number of layers, separated by spaces. Each layer type is a six character string.

The IP described on this section is freely available at OpenCores.org [AM16].

2.1.1 Layer modules

The main structural blocks of the ANN are the layers of neurons. Due to their similarity, they are described as only one VHDL entity. Therefore, this entity need to be configurable enough to accept those parameters which affect each layer separately. The ability to instance the same module several times, each time with different configuration is achieved in VHDL by using generics. The generic parameters are constants in an entity, which values are given at the entity instantiation.

Instantiating blocks a configurable number of times, is done in VHDL by the `for-generate` statement. Figure 2.1 shows a VHDL code using a `for-generate` statement for instantiating one module a *Number_of_instances* times. The loop counter of the `for-generate` is a constant. For example, a loop counter *i* declared on a `for-generate` `for i in 0 to 2 generate`, is an integer constant named *i* with value 0 on the first iteration, another integer constant named as well *i* but with value 1 on the second iteration, and a integer constant named *i* with value 2 on the third iteration. The three iterations are concurrent, thus this is equivalent to copy and paste the code three times replacing *i* by the values 0, 1 or 2 as appropriate.

Thanks to the VHDL generics, the constant array types, and the fact that the `for-generate` loop counters are constant, each entity instance can be configured separately. At the Fig. 2.1 example, every instance of the entity *module* is configured with the same *global_parameter* and is connected to the same *clk* signal. But, each one is configured with a different element of the *array_parameter* and is connected to a different element of the *array_signal*.

```
Generate_label:
for i in 0 to Number_of_instances generate
  Instance_label:
    entity library.module
      generic map
      (
        parameter_a => array_parameter(i),
        parameter_b => global_parameter
      )
      port map
      (
        clk => clk,
        port_a => array_signal(i)
      );
end generate;
```

FIGURE 2.1: VHDL for-generate statement.

In order to give some flexibility on the ANN area-versus-performance trade-off, three kinds of layers have been designed: Serial-input Parallel-output (“SP”), Parallel-input Serial-output (“PS”), and Parallel-input Parallel-output (“PP”).

- The “SP” layer type is an array of MACs (Multiplier-Accumulators). Its resource utilization depends on the number of neurons, and its latency depends on the number of inputs. This is the most common implementation and is perfect for first processing layers, which receive inputs serially.
- The “PS” layer type implements one neuron that is reused to calculate all neurons of the layer. Its resource utilization depends on the number of inputs, and its latency depends on the number of neurons plus the base-2 logarithm of the number of inputs (because the adder tree). A drawback of this layer type is that there is not perfect mapping of the multipliers and the adder tree into embedded MACs like Xilinx’s DSP48Es. They have one multiplier and one adder, configurable to perform different operations like MAC (Multiply-and-Accumulate), but to combine all multipliers with the adder tree adders is not

possible, causing more DSPs utilization. Nevertheless, this layer type is good for output layers which need serial outputs.

- The “PP” layer type is the full parallel implementation of a layer, achieving the maximum performance at expenses of the largest resource utilization.

When serial-input and serial-output is needed, it is automatically accomplished by inserting a serializer after a “SP” layer or a parallelizer before a “PS” layer. A serializer is a shift register with parallel load, and a parallelizer is a shift register with parallel unload.

All neurons in a layer have the same activation function. The selected activation function is inserted wisely between layers. If a parallel output precedes a serial input, a serializer is inserted before a single activation function block. If a serial output precedes a parallel input, one activation function block is inserted before the parallelizer. Only when parallel output precedes parallel input an array of parallel activation function blocks is inserted. Figure 2.2 shows the configurable architecture around a generic layer l .

FIGURE 2.2: Block diagram around a generic layer l . If $l = 0$, $inputs$ and run_in are arriving to the layer (l) instead of $lidata(l-1)$ and $runI(l-1)$. If $l = N_{layer} - 1$, $outputs$ and run_out are leaving parallelizer or a single activation function instead of $lidata(l)$ and $runI(l)$. If the serializer is present, its outputs are $ladata(l)$ and $runA(l)$. If the parallelizer is present, its inputs are $ladata(l)$ and $runA(l)$.

Each block has its own control based on the propagation of a ready signal. The ANN IP core generates automatically as many of these blocks as the parameter $Nlayers$ specifies.

This high configurable architecture is achieved by programming it with the VHDL for-generate and if-generate statements. However, programming this degree of flexibility requires a lot of effort and it still has its limitations (each option has to be coded as a different submodule, etc.). Reaching the point where to write a new design for a new option is easier than make a more complex description.

2.1.2 Weight-and-bias memories

Every layer module has a RAM block (synthesized into BRAM on Xilinx's FPGAs) for storing its weights and biases. In the top module they are merged so they can be seen as a single large BRAM. Due to the high configurability of this IP, the weight-and-bias memories architecture is automatically generated based on the ANN parameters. However, although the address space of the weight-and-bias memories is seen as a unique BRAM space address through the BRAM interface, it is actually composed by several BRAMs and registers, with some addresses without physical implementation. It should be avoided to write or read these unimplemented addresses. Figure 2.3 shows the general scheme of the address space. Understanding the weight-and-bias memories architecture is a key point to properly use the ANN IP.

FIGURE 2.3: General scheme of the weight and bias memories space address.

The *total address* length is calculated by the formula:

$$addr_l = \log_2(Nlayer) + 1 + \max \left(\bigcup_{l=0}^{Nlayer} (\log_2(NumIn_l) + \log_2(NumN_l)) \right), \quad (2.1)$$

where:

- $addr_l$ is the *total address* length seen by the host;
- $Nlayer$ is the parameter which describe the number of layers in the ANN;
- $NumIn_l$ is the number of inputs of the layer l ;

- $NumIn_0 = NumIn$ (the ANN IP parameter which describe the number of inputs of the ANN);
- $NumIn_l = NumN_{l-1}$ when $l \neq 0$;
- $NumN_l$ is the number of neurons in the layer l . The combination of all $NumN_l$ forms the integer array parameter $NumN$. I.e they are each one of the elements of the integer array parameter $NumN$ of the IP.

The first $\log_2(Nlayer)$ bits, of the *total address* length, addresses the layer to be accessed. Starting from the input layer labeled 0, to the output layer labeled $Nlayer-1$. The layer number is coded with *little-endian* (i.e. `unsigned(addr_l-1 downto addr_l-log2(Nlayer))`).

The next bit of the weight-and-bias address selects between the weights ('0') or the biases ('1') of the layer. This bit is called *bias select*.

The remaining bits are used to address the weights or biases of the layer. The *layer RAM address* length used for each layer depends on the number of inputs and neurons on the layer. The *layer RAM address* length is defined as

$$lra_l = \log_2(NumIn_l) + \log_2(NumN_l) . \quad (2.2)$$

When *bias select* is asserted, a bias is addressed with `unsigned(bra_l-1 downto 0)`.

$$bra_l = \log_2(NumN_l) \quad (2.3)$$

is the bias RAM address length.

When the *bias select* is deasserted, a weight $w_{n,i}$ is addressed using the top part of the *layer RAM address* `unsigned(lra_l-1 downto wra_l)` to select neuron n , and the bottom part `unsigned(wra_l-1 downto 0)` to select input i .

$$wra_l = \log_2(NumIn_l) \quad (2.4)$$

is the *weight RAM address* length. Both n and i are coded with *little-endian* binary numbers.

Storage elements (BRAMs or registers) for each part of the memory structure depends on the layer type:

- On the "SP" layers, biases are stored on registers because they have to be accessed simultaneously from all MACs of the layer, whereas weights are stored on BRAMs. There is one BRAM per neuron, so all the n weights ($w_{n,i}$) for an input i are accessed simultaneously from all the MACs.
- As opposite, on "PS" layers biases are stored in a BRAM because they do not have to be accessed simultaneously. Weights are also stored on BRAMs, but there is one BRAM per input instead of per neuron. All the i weights ($w_{n,i}$) for a neuron n are accessed simultaneously from the neuron.
- On "PP" layers, as all the weights and biases have to be accessed simultaneously, all of them are stored in registers.

In order to correctly access to weight-and-bias memories from a host, when host code is written in C/C++, the best way is through 2-D arrays for weights and 1-D arrays for bias. Base addresses must be separated $\max(lra_l)$ spaces. The size of

the first dimension of weight arrays must be defined as the $NumIn_l$ next power of two. Although the first dimension of the 2-D weight arrays has to be defined as the $NumIn_l$ next power of two, unless $NumIn_l$ is a power of two, indexes between $NumIn_l$ and its next power of two must not be accessed. The reason for this way of defining weight arrays is because then, compiler will automatically jump to the correct address.

To help C/C++ programmers to access the weight-and-bias memories of the IP, a header file `ann.h` is provided with it. The user must edit the defines of the ANN parameters (base address of the IP, number of layers, neurons and inputs), and then, place a macro `Wyb(x)` on his code per layer (where x is the layer number, from 0 to $N_{layer} - 1$). This macro automatically declares and initializes to an appropriate address:

- `volatile int (*Wx)[NumNx][NumInx_b2]` (where $NumInx_b2$ is the next power of two of $NumIn_x$),
- `volatile int (*bx)[NumNx]`,
- `int (*NOT_EXISTx0)[...]`,
- and `int (*NOT_EXISTx1)[...]`.

The last two arrays, whose lengths have been omitted intentionally in order to do not complicate more this section, cover the unimplemented address space as an attempt to avoid compilers use those address, although it cannot be guaranteed.

2.1.3 Activation functions

The ANN IP has been designed to allow users to add easily their own activation functions. By default, the "linear" activation function, which actually is a saturated linear due to saturation on `lodata(l)`, and "siglut" and "siglu2" activation functions, that are sigmoid activation functions implemented as look-up-tables with different configurations, have been designed.

To add an user activation function, these steps should be followed:

1. Write the user activation function description into the template `af_template.vhd`. Change file and entity name accordingly, but do not modify the rest of the entity template. Add user description into the module architecture. The `af_sigmoid.vhd` file can be taken as an example.
2. Edit Structural architecture of `activation_function.vhd` file to instantiate the new user activation function inside an if-generate, like in its commented template. Choose a six characters tag for the new user activation function. This step allows the ANN IP to select the user activation function from the `f_type` parameter if the chosen tag is present on it.
3. Re-synthesize all the IP.
4. When configuring the ANN IP, the new activation function tag can be used for the `f_type` parameter.

2.1.4 Operation

Before sending input data to the ANN, weight-and-bias memories should be initialized through the BRAM port. Weight-and-bias memories can be written and read by a host through the BRAM interface.

Once the weights and biases are configured, data to the ANN can be sent serially with a validation bit (*run_in*). It should be noticed that the ANN IP does not check if the weights and biases have been initialized. If input data are sent and the *run_in* validation bit is asserted without weight and bias initialization, the IP will use the uninitialized data contained in the weight-and-bias memory blocks.

When inputs are provided at full speed (one input per clock cycle), output data is produced with a throughput of 1 clock cycle. The output validation bit *run_out* is asserted every clock cycle that a valid output data is produced. Latency of the IP depends on the number and type of layers.

Computations are performed in fixed point format in order to achieve good performance on a reasonable area. Format of the fixed point signals can be configured through their correspondent bit-width parameters and the *LSbit* array parameter. Internal layer registers and signals are sized to prevent overflow. However, to assure reasonable resource utilization, data between layers is re-sized. Independently of the layer type, results of the multiplications have a length of

$$mul_l = NbitW + NbitIn_l, \quad (2.5)$$

and final result of all the additions a length of

$$res_l = mul_l + \log_2(NumIn_l). \quad (2.6)$$

$NbitIn_0 = NbitIn$, and $NbitIn_l = NbitO_{(l-1)}$ when $l \neq 0$. After all layer computations, only $NbitO_l$ bits are sent to the next layer, starting the output LSB (Least Significant Bit) on the position defined by the $LSbit_l$ parameter. Overflow issues are avoided by saturating the result to the most positive or negative values when needed. However, underflow or excessive saturation issues may appear if the $NbitO$ and $LSbit$ parameters are not chosen carefully. Figure 2.4 shows the window of res_l that is selected as $ldata(l)$.

FIGURE 2.4: Fixed point format of layer outputs.

2.1.5 Migration from EDK to Vivado

It may look like it is as simple as changing the wrapper. However, the QoR obtained by Vivado was worse than the *ISE* ones. To achieve the same functionality *Vivado* uses one order of magnitude more resources than *ISE* (see section 4.2.4). The excessive resource utilization appear on the ANN IP core, not on the wrapper. Specifically, the problem is on the memory instantiation.

As it has been pointed out on [Xfo], *Vivado* has not the capability of synthesize adequately 3D RAMs yet. A 3D RAM is an array of RAMs. It is called like that because the VHDL type of it, is an array of arrays (the RAMs) of standard logic vectors (the data). Hence, is an array of 3 dimensions. Whereas the *ISE* tool is able to recognize it properly and synthesize this code as an optimized set of RAMs (BRAM or LUTRAM), *Vivado* does not recognize the RAMs and synthesize the code with an excessive number of LUTs and multiplexers. The functionality is the same (that is assured by the VHDL standard), however, the resource utilization is quite different.

Therefore, in order to migrate the VHDL code to *Vivado*, to change the 3D RAMs by for-generates, which instantiate 2D RAMs, was needed. The 2D RAMs are described with a code based on the *Vivado*'s template for a *byte-write read first RAM* (an extra read only port have been added to this template). So, they can be recognized by the synthesizer in *Vivado*.

Currently, only the sigmoid activation function implemented as a LUT still has the excessive-resource-utilization problem. This code needs to be re-written or new *Vivado*-friendly activation functions created.

To improve efficiency, AXI stream buses were used to connect the *inputs* and the *outputs* ports. They need to be connected through a DMA (Direct Memory Access) controller to a HP (High Performance) port of the PS (Processing Logic). And a full AXI bus was used to connect the weight and bias memories.

In addition, the AXI bus controllers provided by *Vivado* in its wrapper template do not work. Hence, an updated version by colleagues¹ of this Thesis' author has been used. It was modified also by the author of this Thesis to fit better with the ANN interface.

2.2 SystemC HLS

The SystemC source code files of the ANN are freely available as one of the benchmarks in the S2Cbench benchmark suite [CSM14; The15b]. They was used into a two-tier automatic explorer [CSAM17], capable of first find the smallest ANN configuration (for a given maximum error), and then perform the micro-architecture automatic DSE.

Table 2.3 shows the interface of the SystemC *module* that implements the ANN. The only differences with the VHDL implementation is the simplified weight and bias memories interface, and that inputs and outputs ports have become arrays. Array ports allow HLS to synthesize them as parallel ports, memories interfaces, FIFO interfaces, etc.

Parameters should be carefully decided to be macros or templates, taking into account if they will be used on a preprocessor if directive (`#if`) or they will configure parallel blocks separately. Besides, the absence of array of constants has been solved by defining macros like the one shown in Fig. 2.5. This kind of macros calculates which is the needed parameter. They can be used in those places which only a constant expression can be used like for example: array length definitions or as template values. However, they lead to excessive long expressions when replacing long arrays.

¹Alfonso Rodriguez and Javier Mora

TABLE 2.3: IO ports of the SystemC module ANN.

Port	Direction	Type	Description
rst	Input	bool	Synchronous active high reset input
clk	Input	bool	Clock input
run_in	Input	bool	Start and input data validation
inputs	Input	array ^a	Input data
wr	Input	bool	Weight and bias memory write start
wdata	Input	integer ^b	Weight and bias memory write data
run_out	Output	bool	Output data validation
outputs	Output	array ^c	Output data

^a `sc_int<NbitIn> [NUM_I]`

^b `sc_int<NbitW>`

^c `sc_int<NbitOut> [NUM_OUT]`

```
#define NumN(n) \
((n)==2 ? N_2 : ((n)==1 ? N_1 : ((n)==0 ? N_0 : -1)))
```

FIGURE 2.5: Macro to select the number of neurons of the layer n .

The table 2.4 shows the editable macros on the config.h file. Using these parameters, other parameters and macros are automatically calculated:

- NUM_OUT: number of outputs. It is equal to the number of neurons of the last layer.
- NUM_N(n): number of neurons of the layer $n-1$. NUM_N(0) $\equiv$ NUM_I.
- ORDER(n): Order of the nested loops in the layer n .
- MAX_N: maximum number of neurons.

There is a limit on the number of layers to 4. If more layers are needed, the hidden section of the config.h have to be edited. *Array* emulation macros and calculation of the MAX_N are generated using scalable elements. New scalable elements and their composition have to be added for the new layers, along with the *synth_layer_data* definitions (which are declarations of a template struct that defines the arrays for storing the weight and bias of its layer).

2.2.1 Layer template function

Layers have been implemented as one template function. Concurrence of layers could be explicitly expressed implementing them as *sc_module* or *sc_thread*. By implementing them as functions, concurrency will be deduced by the HLS tool. The layer function template is simply described with two nested for loops.

The layer template function is declared as in Fig. 2.6. It has two template arguments: a type argument for the weight and bias struct, and the layer number l .

A fixed number of parallel separately configured blocks (submodules or functions) should be instantiated one by one.

TABLE 2.4: List of the SystemC ANN IP defines.

Parameter	Example value	Description
TRAIN	- ^a	To allow the ANN to be trained
PRINTD	- ^a	To print extra data like weights and bias
NLAYER	2	Number of layers
NUM_I	49	Number of inputs
NUM_0	20	
NUM_1	10	Define number of neurons in each layer, and
NUM_2	1	comment unused layers
NUM_3	1	
ACTFUN_0	sigmoid	
ACTFUN_1	linear	Activation function definitions (comment unused
ACTFUN_2	sigmoid	layers)
ACTFUN_3	saturation	
ORDER_0	I_N	
ORDER_1	N_I	Order of the nested loops in each layer (comment
ORDER_2	I_N	unused layers): I_N outer loop iterates inputs and
ORDER_3	N_I	inner loop iterates neurons; N_I viceversa
NbitW	16	Bitwidth of the weights and bias
NbitIn	8	Bitwidth of ANN input
NbitBuff	12	Bitwidth of internal buffers between layers
NbitOut	8	Bitwidth of ANN output
trunc_out	12	Truncate these bits from the output of each layer
PRECISION	- ^a	Defines arbitrary precision types, if it is not defined standard C types are used

^a These macros are tags. They must be defined to take effect, or commented to avoid their effect.

For regular parallel structures, i.e. every block has the same configuration, a for statement can instantiate a configurable number of them. Because the for iterator is a variable, it can not be used to give values to template parameters or calculate them (like the layer number *l*). Nevertheless, it can be used to give values to inputs, calculate them, or select the outputs (from an array).

Parallel instantiation of separately configured blocks a configurable number of times has been addressed by authors through two methods. The first one relies on the explorer (or whatever other meta-program that can edit the source code) to write into the source code the configured number of calls to the layer function. The second method uses `#if` preprocessor statements to hide the non-needed calls. This second method does not need extra programs, but it is limited on the number of layers which can be configured. Because from the implemented maximum number of calls, the non-needed ones can be hidden, but can not be added new ones without editing the source code.

```
template <class Lt, int l>
void layer(Lt &Wyb);
```

FIGURE 2.6: Declaration of the ANN's member function *layer*.

2.2.2 Activation functions

Different activation functions are declared on `act_fun.h` and defined on `act_fun.cpp` files. They have a return value and a single argument of type `buff_t` (which is an `sc_int<NbitBuff>` type if the `PRECISION` macro is defined, else is an `int` type).

The used activation function is selected as `ACTFUN_0`, `ACTFUN_1`, etc. in a switch statement with the template argument `l` as selector. Then the `ACTFUN_0`, `ACTFUN_1`, etc. macros are replaced by the wished name of the activation function (as configured at `config.h`). There is the risk of the insertion of an undesired multiplexer (because the switch statement). Nevertheless, as the selector is a constant, hopefully the multiplexer will be removed at synthesis optimizations.

Other alternatives are to use function pointers and/or polymorphism of virtual functions. However, they are too complicate, and even more as they need to be determined statically (i.e. at synthesis time) in order to allow hardware optimizations.

2.2.3 Weight and bias memories

The focus of the high-level description and DSE falls in the ANN computation, hence weight and bias memories have been kept simple. SystemC allows to define them as arrays (2D arrays for weights and 1D arrays for bias). The pragmas applied to them will decide how they are implemented. With partition pragmas can be divided into separate RAMs, in such a way that all needed weights can be simultaneously read for different levels of parallelization.

They are controlled in a separate *clock thread*. When the *wr* port is asserted, writing on these memories will start. Correct values should be presented on the *wdata* port in order: first weights of layer 0 (first hidden layer) $W0(i, n)$, from $W0(0, 0), W0(0, 1) \dots W0(0, NUM_N(1)), W0(1, 0) \dots$ to $W(NUM_N(0), NUM_N(1))$ ²; then bias of layer 0; and then data of the next layer following the same order until the output layer.

2.2.4 Explorer

Based on the synthesizable ANN description in SystemC, Benjamin Carrión Schäfer³ developed a two-tier explorer [CSAM17]. It is called two-tier because it is composed of two main stages: a macro-exploration for optimizing the ANN architecture, and a micro-exploration for optimizing the hardware architecture of the selected ANN in the previous stage.

²Remember that $NUM_N(0) \equiv NUM_I$ (number of network's inputs) and $NUM_N(1) = NUM_0$ (number of neurons of layer 0)

³Advisor of this Thesis during the short stay in the Hong Kong Polytechnic University

Macro-Explorer

The goal of this first phase of the proposed two-tier exploration flow is to find the smallest possible ANN which can achieve the MSE smaller than a target error specified by the user ($TMSE$). Figure 2.7 shows the pseudo code of this phase. It is composed of four main steps:

Step 1: Input parameters and initialization. This first step reads all the input parameters, initializes the ANN data structure and initializes randomly the weights of the ANN. It then finds the minimum number of training iterations which lead to a valid solution. For this, the biggest ANN allowable is generated, compiled and simulated. If this ANN configuration does not lead to a $MSE < TMSE$, then the number of training iterations are incremented by NUM_{itr} and the process repeated. The main exploration step will use the number of training iterations (itr) found at this step as the starting point to find the smallest possible configuration in step 2.

Step 2: Binary search. This second step finds the very smallest possible ANN configuration based on the specified limitations. The initial ANN_{new} is the biggest ANN allowable, new ANNs are generated by decreasing or increasing the number of neurons by half of the previous ANN configuration, within the limitations. This process is repeated for the different activation functions available. In order to avoid having to re-explore each new ANN configuration for all the activation function combinations. Only when the configurations with MSE error is within a given interval of the $TMSE$, the activation functions are explored. This interval can be set by the user as an input parameter and is set by default to 5%.

Step 3: ANN compilation and training. This step is the main part of the macro-explorer. Each newly generated ANN generated is compiled (g++) and trained. The MSE is computed and the explorer exits if the error is smaller than the target $TMSE$ and is also smaller in number of neurons than any previous configuration.

Step 2 and 3 are repeated until either the smallest configuration which achieves the $TMSE$ is found or in the worst case, the initial ANN generated in step 1 is returned, as this guarantees that it meets the $TMSE$.

Step 4: QoR measurement. This last step computes the QoR (Quality of Results) of the obtained ANN by not only computing the MSE with respect to the input files used during the training, but also other files for cross-validation. Results are computed for these cross-validation files and stored in a log file.

The macro-explorer returns the smallest ANN configuration described in SystemC by a header file (config.h) which achieves $TMSE$. This ANN is passed to the micro-explorer, which will conduct a micro-architectural exploration of this particular ANN in order to get an area vs. performance trade-off curve.

Micro-Explorer

The micro-architectural explorer takes as input the ANN configuration returned by the macro-explorer and explores arrays, loops and functions by inserting different synthesis directives directly at the source code. Figure 2.9 shows the pseudo code in detail. The micro-explorer executes three main steps, as follows:

Inputs: *src*, *tb_src*, *lims*, *Num_itr*, *MaxN*, *TMSE*, *tar*, *test*
src: ANN synthesizable source codes in SystemC.
tb_src: Test bench codes in SystemC (to train the ANN).
lims: Limitations to the ANN (depends on the application).
Num_itr: Default number of training iterations.
MaxN: Maximum number of neurons allowed in a layer.
TMSE: Target MSE.
tar: Targets for supervised learning.
test: Targets for test QoR.

Outputs: $ANN_{min} = \{Layers, Num_itr\}$, *Log*
 ANN_{min} : Smallest ANN within given *lims*.
Layers: Number of neurons on each layer.
Num_itr: Number of training iterations.
Log: Log file.

```

/* Step 1: Read input parameters and initialize ANN */
1: weights  $\leftarrow$  random_init()
2: itr  $\leftarrow$  Num_itr
3: while (MSE > TMSE) or (itr < Max_itr) do
4:   ANN_new  $\leftarrow$  gen_new_ann(Layers, lims, MaxN)
5:   ann.exe  $\leftarrow$  compile_new_ann(ANN_new)
6:   MSE  $\leftarrow$  train_ann(ann.exe, itr)
7:   itr  $\leftarrow$  itr + Num_itr
8: end while

/* Step 2: Binary Search */
9: while (MSE > TMSE) and (MSE < MSE_prev) do
10:  ANN_new  $\leftarrow$  gen_new_ann(Layers, lims, MaxN)

/* Step 3: Compile and train */
11:  ann.exe  $\leftarrow$  compile_new_ann(ANN_new)
12:  MSE  $\leftarrow$  train_ann(ann.exe, itr)

13:  if (MSE_new < MSE_prev) then
14:    dec_neurons()
15:  else
16:    incr_neurons()
17:  end if
18:  MSE_prev  $\leftarrow$  MSE
19: end while

/* Step 4: Compute QoR and write log file */
20: log  $\leftarrow$  compute_QoR(ANN_new, tar, test)

```

FIGURE 2.7: ANN macro-explorer algorithm.

FIGURE 2.8: Micro-explorer overview based on GA for HLS DSE

Step1: Explorable operations extraction. In this first step the source code src is parsed in order to find explorable constructs. Each line src_i is checked for loops, array declaration and functions declaration. The operations are extracted and added to the exploration list OP_{list} , in case that one of these operations are found in the parsed line. This step continues until the entire src is fully parsed.

Step2: Genetic algorithm exploration. This is the main exploration step. Once all the explorable operations have been extracted from the source code, the micro-explorer continues by performing the actual exploration. Figure 2.8 shows a pictorial view of this step.

Because the search space is extremely large, a GA (Genetic Algorithm) meta-heuristic is used to generate new pragma configurations. GA has shown to be a very effective way to solve multi-objective optimization problems like this one. Krishnan et al. [KK06] already used GA for HLS DSE, but targeted global synthesis option.

Each explorable operation $OP \in OP_{exp}$ is represented as a gene to which a synthesis attribute (pragma) AT is assigned. The list of all genes build a chromosome Cr . Therefore, a gene $\Leftrightarrow OP$ and a $Cr \Leftrightarrow OP_{exp}$. In this work $OP_{exp} = \{\text{arrays, loops, func}\}$, where arrays = {register, expand, logic, RAM, ROM}, loop = {no, partial,

Inputs: $src, ATLib, Constr, GASP$

src : ANN synthesizable source codes in SystemC.

$ATLib$: Attribute library with synthesis directives.

$Constr$: HLS synthesis constraints (target freq., FPGA model, etc.).

$GASP$: GA search parameters ($N, r_c, r_m, Size, Limit$).

Output: $DList_{dom} = \{D_1 = (ATList), \dots, D_x = (ATList)\}$

$DList_{dom}$: Design List of dominating designs.

$ATList$: Attribute list of a design.

```

/* Step 1: Extract explorable operations from source code */
1: for all ( $src_i \in src$ ) do
2:   if (contains_explorable_operation( $S_i$ )) then
3:      $OP_{list} \leftarrow \text{extract\_operation}(src_i)$ 
4:   end if
5: end for

/* Step 2: Main GA exploration loop */
6: for  $x = 1$  to  $Size$  do
7:    $DList \leftarrow \text{generate\_initial\_population}(OP_{list}, N)$ 
8:    $count \leftarrow 0$ 
9:   while ( $count < Limit$ ) do
10:     $P_1 \leftarrow \text{random\_select\_parent}(DList)$ 
11:     $P_2 \leftarrow \text{random\_select\_parent}(DList)$ 
12:     $D_c \leftarrow \text{cross\_over}(P_1, P_2, r_c)$ 
13:     $\text{mutate}(D_c, r_m)$ 
14:    if (child_dominates_parent( $D_c, P_1$ )) then
15:       $P_1 \leftarrow D_c$ 
16:    else if (child_dominates_parent( $D_c, P_2$ )) then
17:       $P_2 \leftarrow D_c$ 
18:    else
19:       $count \leftarrow count + 1$ 
20:    end if
21:  end while
22: end for

/* Step 3: Select dominating designs */
23: for all ( $D_i \in DList$ ) do
24:   if (check_design_dominating( $D_i, DList$ )) then
25:      $DList_{dom} \leftarrow \text{add\_design}(D_i)$ 
26:   end if
27: end for

```

FIGURE 2.9: ANN micro-explorer algorithm (GA HLS DSE).

all, fold}, and func={goto, inline}. The genetic algorithm first step is the random generation of an initial population of N chromosomes. Then, each member in the population is paired with another randomly selected member from the population and is given a chance to produce an offspring using the crossover operator. The crossover operator selects randomly a cut-point, and combines the left half of one parent with the right half of the other. The crossover operator is only applied to a certain percentage, according to the crossover rate specified beforehand by the user, r_c . The offspring is then mutated, which involves randomly selecting one gene within the offspring chromosome and changing it to another random value. Only a certain percentage of the offspring produced are mutated, this proportion is determined by the mutation rate r_m (again passed by the user as an exploration parameter). At this point, the mutated offspring is synthesized calling the HLS tool and the area and latency of the synthesized design back-annotated. The newly generated offspring will replace one of the parents according with the following conditions:

1. If one of the parents is dominated by the offspring (i.e. is inferior to the offspring across all of the objectives), then the dominated parent is replaced by the offspring.
2. If the offspring improves on one or more of the best-so-far values, then the offspring replaces one of its parents (the parent to be replaced is randomly chosen).
3. If an identical copy of the offspring already exists within the population, then the offspring is discarded.

The algorithm continues until *Limit* number of child designs do not improve any of the parents. These same steps are repeated a certain number of times specified as an input to the explorer (*Size*).

The number of generations G , the size of the population N , the mutation rate r_m , the crossover rate r_c , *Limit* and *Size* are input parameters. Appropriate values for them were determined through experimentation. In this work they are set as: population $N=45$, number of generations $G=25$, crossover rate $r_c=0.7$, mutation rate $r_m=0.15$ *Limit*=10 and *Size*=20; as was empirically found that they produced good results and are similar to the ones used in [KK06].

Step3: Selecting dominating designs. The last step of the micro-explorer is to delete all dominated designs in order to return the Pareto-optimal designs trade-off curve. Each design $D_i \in DList$ is checked whether it is dominated or not. In case it is not dominated, it is added to the dominating design list $DList_{dom}$. This step returns only the Pareto-optimal designs of the area vs. performance exploration, where latency is used to measure performance.

Chapter 3

Description of the simulations

3.1 Example applications

In order to test algorithms and to show results, example applications are needed. As simulation codes are strongly related with the example application to be solved, the example applications used along this Thesis are explained on this section before the description of the simulations.

3.1.1 Image compression through auto-encoder

The first example application applied on this research is an auto-encoder for image compression. The auto-encoder architecture has been explained on section 1.2.1 and is shown on Fig. 1.4.

Like other image coding techniques, the whole image is divided into blocks, to which the coder is applied individually. The size of the blocks sets the number of inputs of the auto-encoder.

If all neurons in the network have linear activation functions and there is only one neuron in the hidden layer, the optimal solution for the hidden neuron is the first principal component of the inputs. The applied learning algorithm will try to converge to that. If there is more than one neuron in the hidden layer, the variance in each neuron will be balanced [CM88]; in contrast to PCA (Principal Component Analysis) where the variance decreases with the increase in the component.

There are another ways to use ANNs for image encoding [DH95], but the auto-encoder is by far the most widely implemented one because of its simplicity. Cottrel and Munro [CM88] were the first ones to use the auto-encoder to compress images. Other works about auto-encoders deal with different ways to improve their efficiency, e.g. [NCA96; AF97].

Namphol *et al.* present a hierarchical architecture and their own training shortcut in [NCA96]. The hierarchical auto-encoder is shown in Fig. 3.1. They call their training shortcut NTA (Nested Training Algorithm). It takes advantage of the hierarchical and symmetrical structure of this ANN by training first the outer layers (Fig. 3.2), and then the inner layers (Fig. 3.3).

However, in [AF97] is claimed that at least two layers are needed for a good mapping, while the training shortcut proposed by Namphol *et al.* could hinder the correct training of this mapping.

FIGURE 3.1: Hierarchical auto-encoder [NCA96].

FIGURE 3.2: Outer layers in NTA [NCA96].

FIGURE 3.3: Inner layers in NTA [NCA96].

This task was used as application example for the Matlab simulations of a single ANN and for testing first versions of the hardware implementations. However, due to find a way to merge images from multiple sensors into a combined one to be learned, is difficult, and as I did not find a worthy real application to justify it, this application was discarded.

3.1.2 Detection of pedestrians or cars with low cost radars

The task is to determine if the object moving in front of the radars is a pedestrian, a car, or whatever other thing (like an animal or the environment). This is a real application that can combine easily multiple radars. Besides, results of this application can be compared with ones from my college's Thesis [Vilfi], who applies it to adaptive classifier ensembles.

Since there were already some low cost radars in our laboratory from a previous project [Pér+14], they were re-used. When the radar signal surpasses a threshold, the software tool saves 512 samples in a second and the time of the event.

The first 128 points of the FFT (Fast Fourier Transform) of the 512 points signals are introduced as inputs to the ANN. Due to the fact that this is a classifier task instead of an auto-encoder, the ANN has an output per class to be identified. Here: car, pedestrian or environment. The output is expected to be 1 on the correct class associated neuron and -1 on the other neurons. Actually, outputs are numbers on $[-1, 1]$ range representing the confidence of the ANN for each class. The maximum of the outputs selects the classified class, and its value may give a clue about the accuracy of this output.

In order to increase the success ratio of the classification, more radars are used to catch the moving object from different angles. In this experiment, 3 radars were used. Another ANN takes as inputs the outputs of the 3 ANN, which classified the radar signals, and gives an improved classification. Other option, more realistic, is to have more radars along the road. However, that is more difficult to deploy to get the training examples.

In order to get tagged examples, 3 radars was deployed (like in Fig. 3.4) on the parking of my school. 3 degree students helped writing down the tags for every measure of each radar, supervised by two PhD students¹.

As shown on section 4.1.2, the labeled data collected from the radars was not enough to train properly the ANNs. Consequently, another example application was looked for.

3.1.3 Recognition of handwritten digits

The MNIST database of handwritten digits [Yan] is the most used benchmark for machine learning. It has 60 000 examples for training and 10 000 for test. The task is simple: to recognize a digit from an 28×28 pixels grayscale image of a handwritten digit. However, in order to fit it into the WSN multi-sensor scenario an non-realistic trick has been used. Images are divided into 4 quadrants of 14×14 pixels, emulating

¹Mónica Villaverde and myself

FIGURE 3.4: Deployment of a radar for measuring [PÁ17]. The low-cost radar is inside the circle.

four sensors catching partial information of an *event*. Although it is not a real application, it can show how the proposed algorithms and architectures work, with the advantage of having an enormous amount of labeled examples.

For the simulations of a single ANN, instead of dividing images into quadrants they are resized to 14×14 pixels by averaging every 4×4 pixel block. This is done to reduce the number of inputs from 784 to 196, hence to fit it in embedded hardware (in this case in a low-cost and low-power FPGA).

To solve this task, the output layer of an ANN must have 10 neurons (one per digit). The target is to achieve that the output of the neuron which corresponds with the correct digit values 1.0 and the rest neurons outputs value -1.0 .

3.2 Optimization of single ANN with Matlab

First tests of a single ANN (with the first example application) was carried out with the Matlab's Neural Network toolbox. However, this toolbox introduces by default pre- and post-processing functions that are difficult to remove. They are a problem when an auto-encoder is split into encoder and decoder, because they introduce distortion between them.

Therefore, Matlab scripts and functions have been developed for the simulations. They take into account limitations on the precision of the hardware. Although most

of the computations and intermediate variables are kept at double representation, because is the default type in Matlab. Some critical variables are rounded and/or saturated like on the hardware version.

The scripts and functions perform both feedforward and backpropagation algorithms. And of course, hardware like restrictions only apply to the feedforward stage.

The initial algorithm and parameters have been taken from the final degree project of Carlos Melchor [TT09], which is the backpropagation with variable learning rate described on section 1.2.1.

In order to speed up Matlab simulations the vectorization technique was applied [Vec]. Simulations that used to take around 3 hours, are completed in around 30 minutes after vectorization of the script. It consists on replace for loops by vector and matrix operations. For example, instead of apply (1.5) to each example, as training is executed in batch mode, Matlab can compute:

$$Y^k = f^k(W^k \cdot Y^{k-1} + [\vec{b}^k \dots \vec{b}^k]), \quad (3.1)$$

directly, where $Y^k = [\vec{y}^{k,0} \dots \vec{y}^{k,N_{ex}}]$ and N_{ex} is the number of training examples. Figure 3.5 shows the Matlab code transformation to vectorize this example. Vectorized scripts are faster because Matlab is very well optimized for vectorial computations. The for loop for layers cannot be vectorized because there are data dependencies between iterations.

```

for ex=1:Nex
    for k=1:Nlayer
        a{k} = W{k}*x{k} + b{k};
        % Dimensions: [NumN] = [NumN×NumIn]*[NumIn] + [NumN]
        % ...
    end
end
-----
for k=1:Nlayer
    A{k} = W{k}*X{k} + repmat(b{k},[1 Nex]);
    % Dim: [NumN×Nex]=[NumN×NumIn]*[NumIn×Nex]+[NumN×Nex]
    % ...
end

```

FIGURE 3.5: First, Matlab implementation of (1.5) with a loop, then vectorized.

To deal with the increasing complexity of these simulations, the scripts and functions have been structured. 10 basic functions for training and test of arbitrary ANNs have been developed. Their names start by `ann`, then suffixes determining the following features/functionalities, and end by `_par` to distinguish them from the prior versions without vectorization.

1. Precision: Software precision (no suffix) functions do not saturate or truncate any result. Although pure linear functions have not been implemented; instead, linear saturated between -1 and 1 was implemented. Inputs and targets should be on the range [-1,1]. Hardware precision (`_HW`) functions truncate

and saturate results and key intermediate results of the feedforward phase. Bit widths need to be provided, so input and target ranges depend on them.

2. Training (`_train`) functions carry out both feedforward and backpropagation phases alternately updating weight and bias, whereas test (`_test`) functions only performs the feedforward phase once.
3. Training functions with initial weights (`_iW`) start the Backpropagation algorithm with the given initial weights and bias instead from small random ones. Test functions always need trained weights and bias, hence this suffix is only applied to training functions.
4. Training functions with verification (`_ver`) performs a verification phase after every weight update. This phase is a test function applied to a verification set (i.e. a set of examples that have not been used on the training set). The verification performance is measured to know how good generalizes the ANN and to detect overfitting (see section 4.1.2).

Figure 3.6 shows three examples of basic functions. And Table 3.1 shows the inputs and outputs of the basic functions.

```
Out = ann_HW_test_par(I,W,b,actf,Hwbf);

[W,b,It,MSE,Lr] = ann_train_par(I,T,arch,actf,l_param);

[W,b,It,MSE,Lr,MSEv] = ...
ann_HW_train_iW_ver_par(I,T,Iv,Tv,W,b,actf,l_param,Hwbw);
```

FIGURE 3.6: Examples of basic ANN Matlab functions.

TABLE 3.1: List of inputs and outputs of the Matlab ANN functions.

Name	I/O	Type	Description
I	in	NumIn×N_ex	Inputs
T	in	NumOut×N_ex	Targets
Iv	in	NumIn×N_ex	Validation set inputs
Tv	in	NumOut×N_ex	Validation set targets
Out	out	NumOut×N_ex	Output of the network
W	out	Cell	Final weights of each layer
	in		Initial weights of each layer
b	out	Cell	Final bias of each layer
	in		Initial bias of each layer
arch	in	Struct	Architecture of the network
actf	in	Struct	Activation functions and derivatives
l_param	in	Struct	Parameters of the learning algorithm
Hwbw	in	Struct	Bit width of results
It	out	1×1	Number of iterations executed
MSE	out	It×1	MSE of each iteration
Lr	out	It×1	Learning rate of each iteration
MSEv	out	It×1	MSE of verification set of each iteration

3.2.1 Activation functions

During optimization of the auto-encoders for image compression, sometimes simulations got stuck like in the Figs. 3.7 and 3.8.

FIGURE 3.7: Left: Original image of Lenna. Right: Compressed image of Lenna when training got stuck.

FIGURE 3.8: Left: Total squared error (vertical) vs. epoch (horizontal). Right: Learning rate (vertical) vs. epoch (horizontal). Stuck training.

We discovered that problem was caused by saturation of the linear activation function. Due to the limited bit-width on hardware calculations, saturation was implemented to avoid catastrophic overflow problems. It was also introduced on the Matlab simulations to take that into account. If infinite precision is allowed in an MLP with at least two layers with linear activation functions, large weights can compensate very small inputs caused by small weights in a previous layer. Or vice versa, very small weights can compensate big inputs from a previous layer with large weights. If the derivative of the linear activation function is taken as constant 1, backpropagation algorithm can not control ranges of the weights. Training will fail when weights try to grow in excess causing saturation at output values. In order to avoid this problem, the constant derivative of 1 was replaced by a proper function able to stop the weight growing.

First, the derivative of the saturated linear function is not defined at the saturation points, because the function is not differentiable at these points. It must be defined as 0 at these points. However, that action alone is not very effective, and simulation still gets stuck frequently.

The proposed modified saturated linear derivative is:

$$f(x) = \begin{cases} 0 & , \quad x \leq -1 \text{ or } x \geq 1 \\ 0.0625, & x < -0.875 \text{ or } x > 0.875 \\ 0.125 & , \quad x < -0.75 \text{ or } x > 0.75 \\ 0.25 & , \quad x < -0.5 \text{ or } x > 0.5 \\ 0.5 & , \quad x < -0.375 \text{ or } x > 0.375 \\ 1 & , \quad \text{else} \end{cases} \quad (3.2)$$

It is shown in Fig. 3.9. It is very easy to implement with if/else sentences and it is fast to compute. This property is very useful for training on-line with the help of an embedded microprocessor.

FIGURE 3.9: Modified saturated linear derivative.

3.2.2 Co-simulation FIL

A co-simulation FIL (FPGA-In-the-Loop) is a simulation in which at least one of the blocks is executed in an FPGA connected to the host (in this case a PC with Matlab). It has been done with two different toolboxes: System Generator and HDL verifier. The former one is developed by Xilinx and includes blocks for particular features of their FPGAs. Its main disadvantage is that it can only use its own toolbox blocks for the executable (in the FPGA) part. The latter is provided by Matlab together with the HDL coder toolbox, and both can use most of the Simulink blocks and Matlab functions for the executable part.

The objective of FIL simulations in this Thesis is to speed up training of the already developed ANNs in VHDL. To allow training on-line, an extra output port for the hidden layers outputs is added to the ANN IP core described on section 2.1. The hidden layer outputs are needed for the backpropagation calculations (Eq.s (1.11) and (1.12a)).

The system running on the FPGA can be used as a Simulink module or as a *system object* in Matlab, which contains the functions to communicate with the FPGA. The latter has been chosen and integrated with the scripts of the previous simulations.

The *step* function sends all the inputs to the FPGA, executes 1 clock cycle of the FPGA and then reads all the outputs from the FPGA. If the inputs to the *step* function are vectors of length x , x clock cycles are executed on the FPGA. That is called burst mode.

The main purpose of the HDL verifier toolbox is to test the designed hardware on an environment simulated by Matlab or Simulink. This implies that all the inputs and outputs of the FPGA are communicated between the host PC and the FPGA at every clock cycle. There is not any real-time feature implemented. Therefore, even using Ethernet for the communication no speedup has been achieved. Moreover, the FPGA frequency is limited to 50MHz by the HDL verifier interface, which is smaller than the typical for the used FPGA (a Spartan 6 from Xilinx): 200MHz.

3.2.3 Classifier ensemble

Although the classifier ensemble architecture involves more than one ANN, it has been placed in this section because these simulations has been carried out with the Matlab functions described in this section and with variations of them.

A classifier ensemble is a system with several classifiers that give a result for the same object or event. In order to improve the accuracy of the system, an algorithm takes all these results (called hypothesis) and gives a final decision. Typically, the classifiers receive its inputs from different sensors that catch partial information or different point of view of the object or event to be classified.

In this work, the first stage classifiers are MLPs. In line with this Thesis, another MLP has been used as algorithm to gather the hypothesis of the first stage MLPs and give the final result (second stage). Furthermore, in collaboration with a college, the results has been compared against another 3 algorithms at the second stage: plurality majority voting, basic weighted majority voting and stochastic weighted majority voting [Vil+18; Vil+19].

Each first stage MLP is intended to be implemented on each sensor node catching data (and maybe distributed among other router nodes), and the second stage MLP is intended to be implemented on the cluster head (and maybe distributed among other router nodes too).

Because the algorithms proposed by Mónica Villaverde (basic weighted majority voting and stochastic weighted majority voting) are designed to be adaptive, the following perturbations were added to test the adaptability of the algorithms.

- Additive Gaussian noise to the inputs of one first stage MLP. The noise has mean 0 and different standard deviations.
- Salt and pepper noise to the inputs of one first stage MLP.
- Additive Gaussian noise to the outputs of one first stage MLP. The noise has mean 0 and different standard deviations.

- Absence of the outputs of one first stage MLP. For the case of an MLP on the second stage the outputs are replaced by -1, which means “no-idea”. For the voting algorithms, the outputs are replaced by a random assignment.

Perturbations on the inputs of one first stage MLP simulate corruption or malfunction of a sensor, and perturbations on the outputs of one first stage MLP simulate corruption or disconnection of the communication between nodes.

3.2.4 Stacking DAEs

This subsection describes the scripts and functions used to simulate the stacking DAEs technique explained on section 1.2.1. First, the *DAE_train* function has been developed to train a DAE and return the encoder layer (or layers) weight and bias. The tested parameters are:

- Noise corruption. Two noises with one parameter each one have been studied: salt and pepper with different percentage of corruption, and additive Gaussian with mean 0 and different standard deviations.
- Tied weights. A common practice in the state-of-the-art of stacking auto-encoders is to use for the decoder layers the same weights of the encoder layers. $W_e = W_d^T$, where W_e are the weights of the encoder layer and W_d are the weights decoder layer². That is called tied weights. It reduces the amount of parameters to be trained, reducing overfitting, and acts as a kind of regularization. However, it is not always beneficial. Thus, the two options has been tested (tied and untied weights).
- 2-layer or 4-layer auto-encoder. As shown on section 1.2.1, at least two layers are needed to achieve arbitrary accuracy. An auto-encoder has two layers. However, if we focus on the encoder part, two layers are needed to be able to learn whatever encoder function. Hence, besides the typical staking DAEs (explained on section 1.2.1 and shown on Fig. 1.5), stacking DAEs of 4 layers (2-layer encoders) has also been tested.

Then, using the previous function and the basic ANN functions, the function *staking_DAE_train* has been developed. It calls the *DAE_train* function iteratively to pre-train all layers except the output layer, and finally perform a fine-tune supervised training (using the previously obtained weights and bias as initial weights, and random initial weights and bias for the output layer).

Finally, a dedicated script for the tested application example is created. It generates the parameters, pre-process input data and targets into the correct format, and after calling the *staking_DAE_train* function saves the results and visualizes the relevant ones.

3.3 Distribution of ANN over WSN with SystemC

In order to determine the best way to divide the ANN, SystemC simulations have been carried out to estimate computation and communication energy consumptions.

²The T is the transpose operation.

These simulations are written with non-synthesizable SystemC. It provides constructions for modeling concurrency at different levels. Hence, it can be modeled the concurrency of the different elements of the node, and the concurrency among all the nodes of the network. In addition, it offers easy access to the simulation time in an event driven simulation with all the C++ flexibility.

Following the guidelines in [RGL09] master thesis, the functionality of the node is divided into 4 layers: sensor, power, communication and processor. Each layer has been modeled as a `SC_MODULE` that inherits from the corresponding template interface (`sensor_1.h`, `power_1.h`, `communication_1.h` or `processor_1`). These interfaces define how the functional layers communicates between them, but allow for the data types be decided *a posteriori*. Therefore, these codes can be easily adapted to re-use them in the future for other simulations.

In addition, a fifth layer has been envisioned: the actuator. Although it has not been implemented because it is not necessary for this simulation, communication to and from this layer has been added to the other interfaces.

Communications to or from outside the node are called to/from the environment.

3.3.1 Node layers

Sensor

The `sensor_1.h` interface has an input from environment for measures, an input from the processor for orders, the dead flag from the power layer, an output for data to the processor, and the current power output to the power layer.

An *ad hoc* sensor module is needed for each application example. It is responsible for get the input data, usually by reading them from the file or files of the application dataset. It may perform some kind of pre-processing, unless this pre-processing is part of the processor layer duties.

Power

The `power_1.h` interface has an input from each other layer to communicate their current power consumptions, and another input from the environment to simulate energy harvesting. It has an output (*soc*) to processor layer for monitoring the state-of-charge, and a broadcast output of bool type to tell every layer that there is not any energy left (dead flag).

There is an input port from processor layer that has been left commented. The author cannot think any relevant use for this port. However, maybe in the future may be useful to change behavior of the power supply based on processor orders. Note: the power layer does not simulate the power management; it does simulate the power supply like batteries, energy harvesters, etc. Changes on the power consumption should be coded in its corresponding layer.

For this simulation has been implemented a power layer called *basic_battery*. It has a parameter to set the initial amount of energy, then it calculates the remaining lifetime based on the current power consumption of the node and program an event

at that time. When power consumption of any layer changes, it recalculates the remaining lifetime and program a new event that discards the previous one. When the event occur, the dead flag is sent to all layers and the `SC_THREAD` that models the battery ends. A way to recharge the battery has not been modeled, nevertheless it can be easily added if needed.

Communication

The `communication_l.h` interface has a Rx input and a Tx output from/to environment, an input for orders from the processor layer, a data output to the processor layer, and the current power output for the power layer.

For this simulation has been simulated an ETRX2 transceiver module [Etra; R3x], which is the one used in the Cookies ZigBee 2.4MHz communication layer. It uses the ZigBee communication protocol stack [802; Zig].

As the mayor concerns are communication vs. computation energy consumption, establish and maintenance the network is not simulated. Hence, the NWK (network) layer of the ZigBee protocol stack has not been implemented. A SystemC thread simulates the PHY layer as a FSM (Finite State Machine) with `COM_OFF_st`, `IDLE_st`, `SLEEP_st`, `LISTENING_st` and `TRANSMITTING_st` states in which the proper power consumption is adjusted and data is transmitted and received through Tx and Rx channels at the corresponding states. Another SystemC thread controls the PHY layer. This last thread implements the MAC layer and a simplified version of the APS (application sublayer).

Two `SC_MODULES` has been designed: one for SEDs (`etrx2_SED`), and other for FFDs or COOs (`etrx2_v1`). The latter one implemented on two files: in `etrx_PHY.h` the `SC_MODULE` declaration and the PHY thread, and in `etrx2_FFD.h` the control thread and auxiliary functions.

At the beginning of each file, there is a list of parameter definitions for power consumptions, characteristic times, and other parameters of the ZigBee protocol stack algorithms. They may be edited to adapt the simulation to updates on the protocols, firmware, or even to adapt it to other communication modules with similar functionality. For those parameters which has not been found on the ETRX2 documentation [Etra; R3x; Etrb], measures has been taken (see section 4.1.4) to give them a value.

Both `SC_MODULES` have a mandatory constructor `unsigned int` parameter to give an address to the communication layer. Additionally, both have another `unsigned int` parameter called `paren_addr` which should be initialized on the elaboration stage of the simulation. This last parameter stores the parent address on a SED or a sink address on a FFD. FFDs have in addition the `child_list` class member, which is an *ad hoc* type for a dynamic list of children node addresses. The parameters described on this paragraph are not protected, hence they can be edited dynamically during the simulation if needed.

Messages sent to the Tx port and received from the Rx port, are of a `struct` type, called `basic_msg`, with the following fields:

- *data*: of a template type for the transmitted data.
- *length*: `unsigned char` for the number of bytes of the message.

- *address*: unsigned int for destination address.
- *src_addr*: unsigned int for source address.
- *DNS*: unsigned char for the data sequence number of the message.
- *msg_t*: of an enumeration type with the values of the table 3.2 for message type.

TABLE 3.2: Message types.

Value	Description
QUIET	There is not any message
DATA	Unicast with data
MACKACK	MAC acknowledge
APPACK	APS acknowledge
POLL_RQ	Poll request
POLL_REP_NODATA	Poll reply without pending messages
POLL_REP_DATA	Poll reply with pending messages
BCAST	Broadcast message

Orders received from the processor layer are of a `struct` type, called *simple_com_o*, with the same *data*, *length* and *address* fields that in *basic_msg*, and a field *o* of an enumeration type with the following values:

- *NONE_o*: is the default order to represent that there is not any issued order. It have not any effect.
- *SEND_o*: sends a unicast. It sends the data on field *data* to the address on field *address*.
- *LISTEN_o*: turns on the radio on listen mode and reports every message sent to the node (that corresponds with node's address).
- *STOP_L_o*: stops listening and turns off the radio. *LISTEN_o* and *STOP_L_o* are designed for being used only by FFDs (SEDs ignore these orders).
- *POLL_o*: polls data from the node's parent. FFDs ignore this order.
- *BCAST_o*: sends a broadcast. It broadcasts data on field *data* and uses the field *address* as radius.
- *MCAST_o*: sends a multi-cast. It has not been implemented yet.
- *CHANGE_PTX_o*: changes the transmission power. It has not been implemented yet.

As a curiosity and to prevent those who want to do similar simulations, due to a reason that I do not know yet, calls of function *rand* after *sc_start* call do not use the seed assigned on the elaboration phase. And all objects use a copy of the same pseudo-random sequence. Besides, function *time* gives the same value to every object when is called at the beginning of a *sc_thread* (at least when there are 4 objects). Therefore, in order to assign different seeds to each object, *srand* is called with the following value: `time(NULL) + (int) this`.

Processor

The `processor_l.h` interface has data input ports from sensor, power and communication layers; and output ports to sensor, communication and actuator layers for orders.

To carry out this simulation two simple *ad hoc* processor layers has been created: *pANN* and *pANN_s*, for FFDs and SEDs respectively. They take periodically inputs from the communication layer or the sensor layer (depending on the device kind); then they compute their fragment of the ANN; and finally they broadcast their output.

To get the values of the power consumption of the processor layer, which is mainly the FPGA consumption, there are three methods (for Xilinx FPGAs and SoCs). Through the XPE (Xilinx Power Estimator) spreadsheet, the ISE or Vivado integrated power analyzers, or measuring it. Although XPE is intended for pre-implementation phases of a project and less accurate than the integrated analyzers, it is used because is faster and easier. Integrated analyzers have some restrictions to get an accurate solution: VHDL cannot be used for post-implementation timing simulation, software running on processor system of a SoC is not taken into account, and others. Measurements on the real hardware are used only to check that simulations are accurate enough. It is impractical to measure every test case.

In addition, in order to make these simulations more re-usable for other WSN projects, another set of templates has been planed. These code templates will allow the C programs developed for the Cookies platform (using the Cookies' libraries [Muj+12]) to be used on the simulation directly or with minor modifications. However, they have not been finished and they have been left as a future project.

The key to allow the C programs to be used on the simulations is to implement an API (application interface) for the Cookies' libraries. The functions on this API are the responsible ones for call the `wait()` functions. And hence, advance the simulation time and allow the processes which model the hardware to be executed.

The most challenging part is to deal with interruptions. As time in a SystemC simulation does advance only at `wait()` functions calls, it is not possible to interrupt execution of a standard C function (here, the user program). Fortunately, the Cookie's libraries were designed to be used on a program structure that is easily hidden on user functions in a similar way like Arduino's `setup()` and `loop()`. These user functions will be:

- `CEI_user_configuration()`: it is executed once at the beginning of the simulation. As its name suggest it is used to configure the device.
- `CEI_user_radio()`: it is executed every time that is received an incoming message from the communication layer.
- `CEI_user_sensor()`: it is executed periodically (as configured). Its intention is to read sensors and/or send messages to the network.

These functions can be called inside SystemC processes (threads or methods).

3.3.2 Environment

The *environment* `SC_MODULE` works as the testbench of the simulation. It connects the communication layers as configured (see later), and provides triggers and/or data to the sensor layers.

Optionally, it prints the messages between nodes and the messages between processor and communication layers for debugging or tracking the simulation. This feature is activated by defining the `PRINT_MSGS` macro.

Environmental distortions like interferences, diffractions and reflexions have not been modeled yet. Only interferences between nodes of the network has been taken into account.

The total number of nodes must to be defined on the macro `Nnodes`. Then, in order to connect the nodes and form a network topology, two variable members are used:

- `connect_sizes`: unsigned int array of `Nnodes` size, to keep sizes of the `connect_list` arrays.
- `connect_list`: unsigned int pointer array of `Nnodes` size. Each pointer should point to a dynamic array (of size defined by the corresponding member of `connect_sizes`) which contains the nodes that have a direct connexion with the node.

The nodes should be numbered from 0 to `Nnodes-1` to identify them on the `connect_list`. This ordering may be independent from their addresses.

The elaboration segment for configure a simulation of a WSN with only 6 nodes (see 4.1.4) has around 250 lines. In order to elaborate a simulation with an acceptable number of nodes, these lines will explode. Hence, a GUI (Graphical User Interface) is necessary. With this GUI will be possible to place the nodes, connect graphically them, configure each layer/module of them, etc. However, to develop a GUI is out of the scope of this Thesis, so it will be a future project.

Chapter 4

Results

4.1 Simulations

4.1.1 Optimization of single ANN with Matlab

First results: hierarchical auto-encoder with NTA

The first results of the ANN simulations have been carried out with the Neural Network toolbox, and belongs to the author's Master thesis [AO16]. In appendix I of that document can be found all the images result of these simulations. Figure 4.1 shows an example of these results.

FIGURE 4.1: Left: original 512x512 pixels. Center: 4x4 pixel blocks through 4 *tansig* neurons trained with 16 *purelin* neurons decoder; $1/4$ compress ratio; 31.47 PSNR. Right: macroblocks of 4x4 previous blocks through 8 *tansig* neurons trained with 64 *purelin* neurons decoder by NTA; $1/4 \cdot 1/8$ compress ratio; 23.23 PSNR. [AO16].

The preprocessing and post-processing functions that the Neural Network toolbox adds to the ANNs, produces distortions when encoder is separated from decoder. That was one of the reasons that motivated to develop our own training Matlab scripts.

Optimization of auto-encoder for image compression

An exhaustive optimization of a single ANN using the Matlab scripts has been done as *Trabajo Fin de Grado* of a degree student [VM15], directed by this Thesis author.

The first step was to choose the best parameters for the learning algorithm. Three architectures of 2 layers ($16 \times 2 \times 16$, $64 \times 8 \times 64$ and $100 \times 12 \times 100$)¹ and other three architectures of 4 layers ($16 \times 8 \times 2 \times 8 \times 16$, $64 \times 32 \times 8 \times 32 \times 64$ and $100 \times 50 \times 12 \times 50 \times 100$)¹ were selected to test the training parameters. They represent architectures for the minimum input block (4×4 pixels), a typical input block (8×8 pixels) and the maximum input block (10×10 pixels) tested later. The parameters and their tested values are:

- ζ : 1.02, 1.05 and 1.07.
- η : 1.02, 1.05 and 1.07.
- ρ : 0.6, 0.8 and 0.95.

For each combination of architecture and set of training parameters has been measured 3 trainings. The stop condition has been set to happen when after 200 epochs the PSNR has not improved at least 0.2 dB. That stop condition is a fast one to save training time, hence the obtained PSNRs are low, but it is enough to select the training parameters. Linear saturated between -128 to 127 activation function was used on every layer with the proposed modifications on its derivative. Figure 4.2 shows the average PSNR of the three trainings per combination of architecture and parameters. From these results, $\zeta = 1.05$, $\eta = 1.02$ and $\rho = 0.8$ were chosen.

ρ	η	ζ	16x2	64x8	100x12	16x8x2	64x32x8	100x50x12	Average
0.6	1.02	1.02	26.629	27.025	25.990	26.499	26.665	26.118	26.488
0.6	1.02	1.05	26.464	26.099	25.295	26.379	26.079	26.389	26.117
0.6	1.02	1.07	26.482	26.880	25.511	26.137	26.138	25.816	26.161
0.6	1.05	1.02	26.633	26.962	26.004	26.508	26.657	26.375	26.523
0.6	1.05	1.05	26.625	27.089	26.201	26.513	26.669	26.392	26.582
0.6	1.05	1.07	25.402	26.899	26.104	25.764	26.474	26.343	26.164
0.6	1.07	1.02	26.621	26.936	25.958	26.482	26.659	26.377	26.505
0.6	1.07	1.05	26.613	27.094	25.961	26.425	26.640	26.406	26.523
0.6	1.07	1.07	26.619	26.854	25.961	26.490	26.597	26.444	26.491
0.8	1.02	1.02	26.628	27.051	26.052	26.472	26.555	26.392	26.525
0.8	1.02	1.05	26.615	27.149	26.287	26.496	26.586	26.562	26.616
0.8	1.02	1.07	26.601	27.035	26.261	25.741	26.591	26.636	26.478
0.8	1.05	1.02	26.596	27.028	26.025	26.482	26.549	26.108	26.465
0.8	1.05	1.05	26.599	27.064	26.209	26.459	26.577	26.189	26.516
0.8	1.05	1.07	26.606	27.110	25.462	26.477	26.554	26.312	26.420
0.8	1.07	1.02	26.606	27.073	26.077	26.459	26.513	26.493	26.537
0.8	1.07	1.05	26.589	27.127	26.316	26.476	26.553	25.984	26.507
0.8	1.07	1.07	26.600	27.155	26.512	26.487	26.553	26.109	26.569
0.95	1.02	1.02	26.582	20.413	26.038	26.385	26.782	26.643	25.474
0.95	1.02	1.05	26.630	26.960	26.332	26.269	26.611	26.326	26.521
0.95	1.02	1.07	24.576	26.651	25.889	26.122	26.046	25.996	25.880
0.95	1.05	1.02	9.921	21.047	21.469	15.503	14.135	8.930	15.167
0.95	1.05	1.05	9.921	21.011	25.911	15.459	14.028	8.930	15.876
0.95	1.05	1.07	9.921	20.815	25.881	15.431	13.951	8.930	15.822
0.95	1.07	1.02	9.921	15.752	14.128	14.267	12.529	8.969	12.594
0.95	1.07	1.05	9.921	16.091	16.714	14.267	12.529	8.969	13.082
0.95	1.07	1.07	9.921	16.528	16.714	14.267	12.529	8.930	13.148

FIGURE 4.2: Selection of the training parameters. Results are the PSNR of the obtained images respect to the original one. [VM15].

¹Architecture syntax is defined as NumIn $\times$ NumN hidden layer 1 $\times \dots \times$ NumOut

After selecting the training parameters, the best configuration to solve the task was chosen. Tested input blocks was 4×4 , 5×5 , 6×6 , 7×7 , 8×8 , 9×9 and 10×10 . The compression ratio was decided to be around $1/8$, hence tested 2-layer architectures was: $16 \times 2 \times 16$, $25 \times 3 \times 25$, $36 \times 4 \times 36$, $49 \times 6 \times 49$, $64 \times 8 \times 64$, $81 \times 10 \times 81$ and $100 \times 12 \times 100$; and 4-layer ones are shown in table 4.1.

TABLE 4.1: 4-layer architectures tested.

$16 \times 12 \times 2 \times 12 \times 16$	$16 \times 8 \times 2 \times 8 \times 16$	$16 \times 4 \times 2 \times 4 \times 16$
$25 \times 18 \times 3 \times 18 \times 25$	$25 \times 12 \times 3 \times 12 \times 25$	$25 \times 7 \times 3 \times 7 \times 25$
$36 \times 24 \times 4 \times 24 \times 36$	$36 \times 18 \times 4 \times 18 \times 36$	$36 \times 9 \times 4 \times 9 \times 36$
$49 \times 36 \times 6 \times 36 \times 49$	$49 \times 24 \times 6 \times 24 \times 49$	$49 \times 13 \times 6 \times 13 \times 49$
$64 \times 48 \times 8 \times 48 \times 64$	$64 \times 32 \times 8 \times 32 \times 64$	$64 \times 16 \times 8 \times 16 \times 64$
$81 \times 60 \times 10 \times 60 \times 81$	$81 \times 40 \times 10 \times 40 \times 81$	$81 \times 21 \times 10 \times 21 \times 81$
$100 \times 75 \times 12 \times 75 \times 100$	$100 \times 50 \times 12 \times 50 \times 100$	$100 \times 25 \times 12 \times 25 \times 100$

As activation functions, on 2-layer architectures, linear saturated between -128 to 127 was used always on output layers, and sigmoid (Eq. (1.2) with $f'_0 = 1$, $f_{min} = -128$ and $f_{max} = 127$) vs. linear saturated between -128 to 127 on bottleneck layers was tested. On 4-layer architectures, linear saturated between -128 to 127 was used always on output layers and bottleneck layers, and sigmoid functions (Eq. (1.2) with $f'_0 = 0.5$ on layer 1, $f'_0 = 2$ on layer 3, $f_{min} = -2048$ and $f_{max} = 2047$) vs. linear saturated between -2048 to 2047 on mapping layers (hidden layers 1 and 3) was tested. The parameters for the sigmoid functions have been chosen from the experiment shown in Fig. 4.3 with a $64 \times 32 \times 8 \times 32 \times 64$ network (400 epochs).

f'_0		$f_{max} = -f_{min}$			
$k = 1$	$k = 3$	256	512	1024	2048
0.5	0.5	22.4034	23.8037	23.5274	23.3473
0.5	1	22.3903	23.6571	24.3189	24.204
0.5	2	23.5671	24.1437	25.1319	25.1936
1	0.5	22.6609	23.2441	23.0817	22.8957
1	1	23.1485	24.322	23.2132	24.5143
1	2	23.3008	22.4995	24.7345	23.2979
2	0.5	21.4822	22.2484	22.1326	22.9082
2	1	22.5236	22.309	23.2079	22.3577
2	2	22.7816	21.4851	20.9819	19.7681

FIGURE 4.3: Selection of the sigmoid parameters for hidden layers 1 and 3. Results are PSNRs. [VM15].

Therefore, there were 56 combinations: (7 2-layer architectures + 21 4-layer architectures) $\times$ 2 activation functions. Each combination was trained 5 times. The stop condition was set to when after 200 epochs the PSNR has not improved at least 0.025 dB. At this stop condition the ANN has already learned the task. The input image (shown on Fig. 4.4) has 512×512 pixels and it was used for training. Then, the trained ANNs were tested with 13 images of 512×512 pixels (Fig. 4.4, the training input image was used also as test image).

Figure 4.5 shows the results: averages and confidence interval at 95% from each 5 trainings of the 13 test images.

From all of these results, a reduced set of networks was selected for further analysis, taking into account the following criteria:

FIGURE 4.4: Input images for Matlab simulations of single auto-encoder. Left: Training image. Right: Test images.

- Results with less than 0.5 dB of difference have neither statistical nor visual significant difference.
- Regarding architectures with similar performance, the easiest to implement should be chosen. Hence, as simpler it is and it has fewer neurons, as better the architecture is. Saturated linear activation functions are easier to implement than sigmoid ones.

Within these rules, 2-layer networks with sigmoid activation functions (group 2 on Fig. 4.5) are discarded, because they have similar results than ANNs with saturated linear activation functions. The selected best results are shown on Fig. 4.6. There is not a big visual difference among them. Hence, if resources are limited (like in most WSN nodes) 2-layer architectures with saturated linear activation functions are recommended. Although, if there are enough resources available, more complex architectures with slight better results may be used.

Previous experiments have a compression ratio around $1/8$, however as the compression ratio increases, difficulty of the training increases too. Figure 4.7 (left) shows the relation between PSNR and compression ratio for 2-layer architectures of 8×8 inputs. Figure 4.7 (right) demonstrates that results can be extended to other compression ratios.

activation functions

In order to test modifications to derivatives of the activation functions, another example application was used. It is an auto-encoders too, although with simpler but more non-linear task. It was chosen because image compression causes convergence problems when derivative of linear function is used, and furthermore because it turns to be a quite linear task. As is shown above on previous section, difference between PSNR of results from linear activation functions and non-linear activation functions is not statistically significant.

Table 4.2 shows percentages of saturated values by the saturated linear activation function (trained with the modified derivative). The results on this Table are the average of three 4-layer autoencoders ($64 \times 32 \times 8 \times 32 \times 64$) trained with the Lenna image and tested with the images of Fig. 4.4. The highest saturation percentage is 13.8%, therefore, good PSNRs can be obtained with pure linear activation functions.

FIGURE 4.5: Results (PSNR) of the Matlab simulations of single auto-encoder. Groups 1 and 2 are 2-layer architectures whereas 3 and 4 are 4-layer ones. Groups 1 and 3 uses saturated linear activation functions whereas 2 and 4 uses sigmoid ones. [VM15].

FIGURE 4.6: Best results of the Matlab simulations of single auto-encoder. Groups from Fig. 4.5. [VM15]

TABLE 4.2: Percentage of saturated values by the saturated linear activation function. Test images from Fig. 4.4.

	Training image												
Hidden	2.8	4.6	1.0	2.2	1.0	1.0	1.5	1.1	3.5	0.6	13.8	4.2	4.3
Narrow	2.0	3.5	0.8	1.6	0.8	0.8	1.4	1.2	4.9	0.6	9.4	3.5	3.1
Hidden	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Output	0.0	0.0	0.0	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.7	0.0	0.0
PSNR	25.4	25.5	29.4	29.2	26.7	30.0	28.1	26.6	30.4	28.6	28.0	28.3	24.4

In this section, a classic non-linear problem is presented to the ANN. It is a very simple circle parametrization and it falls into dimensionality reduction class, using the auto-encoder architecture. The results of the saturated linear activation function with the modified derivative training are compared with the results of the linear and sigmoid activation functions.

The objective of the circle parametrization is to show easily the limitation of the linear methods and what amount of non-linearity is needed to properly solve this kind of non-linear problems. When linear activations functions are applied, the auto-encoder neural network will perform a PCA. A linear method as PCA is not able to find a good solution to these non-linear problems.

The circle parametrization (Example 1 in [Kra91]) is an auto-encoder with 2 inputs and outputs, and one neuron in the narrow hidden layer. 6 neurons on mapping layers have been used, so the network has a $2 \times 6 \times 1 \times 6 \times 2$ architecture. The training set are 150 pairs (y_1, y_2) , generated randomly using an uniform distribution over the

FIGURE 4.7: PSNR vs. compress ratio for 8×8 input auto-encoders [VM15].

parameter θ (from 0 to 2π) in (4.1) and (4.2):

$$y_1 = 0.8 \sin(\theta) , \quad (4.1)$$

$$y_2 = 0.8 \cos(\theta) . \quad (4.2)$$

75 test points are randomly generated using the same procedure for each test. Figure 4.8 shows the results of some example tests. On the output layer and the narrow hidden layer, the saturated linear activation function (with its derivative defined as 0 on the saturating points) has been used always. Only on the mapping layers have been tested different activation functions and derivatives.

The linear solution, when the training points are perfectly uniform, is a diameter of the circle; which is the first principal component (Fig. 4.8a). As is shown in Fig. 4.8e, when the sigmoid activation function is used in the mapping layers, the solution although is not perfect, it can follow the circle.

The saturated linear activation function with its derivative (defined as 0 on saturating points) most of the times fails to converge or lead to solutions which are not significantly better than the linear solution (Fig. 4.8b).

Nevertheless, by applying the modified saturated linear derivative during the training, the network can easier take advantage of the non-linearities of the saturated linear activation function to learn better solutions (Fig. 4.8c). If the training is performed off-line and the sigmoid derivative is applied instead of the saturated linear derivative, results closer to the sigmoid ones are obtained even using the saturated linear activation function on the network (Fig. 4.8d).

Another convergence problem that sometimes appears related with the activation function is, when due to a zero derivative (or too small value), the weight increments become zero. That happens on the saturation points of the linear saturated function and when the absolute value of a_i is too big on the sigmoid function. And it causes the training getting stuck at local minima. The solution is to add a small value ϵ to the derivatives when they become zero (or too small). It produces small

FIGURE 4.8: Results of circle parameterization with auto-encoder. Test points are marked with “+” and auto-encoder outputs with “×”.

weight increments (instead of zero increments) which eventually will scape from the local minimum, without deforming the gradient in excess.

4.1.2 Classifier ensemble

Detection of pedestrians or cars with low cost radars

The best set of training examples caught with the 3 radars had only 128 entries and a great imbalance between people and cars (there are 4 times more people tags than car tags). 92 of the tagged examples belongs to pedestrians or group of pedestrians, 23 to cars and 13 to bikes. As they are not enough examples to train and test, 5 sets of 52 pedestrian signals taken randomly from the 92 ones, 13 cars taken randomly from the 23 ones and the 13 bikes have been used to train the ANNs. The 5 sets of the remaining 40 pedestrian signals, 10 cars and 0 bikes were used as test sets.

A big imbalance on the dataset produce problems on training. The ANN tends to classify most of the inputs as the dominant class and ignores the less frequent classes. To avoid this problem there are several techniques like oversampling (the low frequency classes), undersampling (the dominant classes), creation of synthetic examples and weighting the influence of the examples. I have weighted the error of each example inversely proportional to the class frequency on the dataset. Particularly, the squared error of each example is multiplied by the total number of examples of the training set (78) and divided by the number of examples of its class on the training set (52 persons, 13 cars and 13 bikes). Hence, as training is done in batch mode, on every epoch the addition of the error of all classes is balanced, avoiding the tendency to classify every input as the most abundant class.

The first-stage classifiers have 128 inputs (from the FFT), 4 hidden neurons and 3 output neurons. Each output neuron recognizes a class: pedestrian, car or bike. The parameters for the first-stage training are: 1500 iterations, $\zeta = 1.03$, $\eta = 1.04$, $\rho = 0.8$, random initial weights and bias with uniform distribution in the range $[-0.15, 0.15]$, initial learning rate 10^{-5} , and both layer's activation functions are sigmoid functions between -1 and 1, and origin slope 1. On the learning algorithm, the derivatives of the sigmoid functions have been saturated with a minimum value of 0.1.

The second-stage classifier ensemble has 9 inputs (3 from each radar), 12 hidden neurons, and 3 output neurons. The training parameters for the second stage are: 700 iterations, $\zeta = 1.03$, $\eta = 1.04$, $\rho = 0.8$, random initial weights and bias with uniform distribution in the range $[-0.15, 0.15]$, initial learning rate 10^{-5} , and both layer's activation functions are sigmoid functions between -1 and 1, and origin slope 1. On the learning algorithm, the derivatives of the sigmoid functions have been saturated with a minimum value of 0.1. It has been trained with the same training sets used to train the first-stage MLPs. So, the test sets can be used for testing the second stage too.

Table 4.3 shows the results on the 5 test sets and the comparatives with the plurality majority and the algorithms proposed in [Vil+18; Vil+19].

The lack of training examples causes overfitting problems on the ANN training. The overfitting problem is when whereas the training set scores improves by learning with precision each training example, the ANN loses generalization and the score on the test set decreases. It appears when there is more learning parameters

TABLE 4.3: Results in % of the detection with low-cost radars. H: hits, T: ties and M: miss.

	Test 1			Test 2			Test 3			Test 4			Test 5		
	H	T	M	H	T	M	H	T	M	H	T	M	H	T	M
Radar 1	82	0	18	76	0	24	86	0	14	78	0	22	84	0	16
Radar 2	78	0	22	84	0	16	80	0	20	84	0	16	82	0	18
Radar 3	94	0	6	86	0	14	86	0	14	84	0	16	88	0	12
ANN	97	0	3	92	0	8	90	0	10	88	0	12	87	0	13
Majority	94	0	6	90	4	6	94	0	6	90	4	6	92	6	2
Weighted	94	0	6	90	4	6	94	0	6	90	4	6	92	6	2
Stochastic W.	94	0	6	90	0	10	94	0	6	92	0	8	96	0	4

(i.e. weight and bias) than training examples, after some training epochs. As more learning parameters respect than training examples, as sooner (i.e. at earlier epoch) appears the problem. Figure 4.9 shows the evolution of the MSE of the train and test sets during a training of a first-stage MLP. Besides, it also causes the training of the first stage MLPs very unstable (getting very different results on each experiment). Hence, I have chosen 5 successful trainings (ones more than others) of the first-stage MLPs as examples. As can be shown on the 3 firsts rows of the Table 4.3 there is not a clear tendency. Therefore, these results are shown as they are instead of being averaged.

FIGURE 4.9: Example of overfitting on the training of first-stage MLP. MSE (vertical) over epoch (horizontal). The upper line (blue) is the test set and the lower line (orange) is the train set.

Adaptation of the MNIST dataset

Training of the first stage classifiers and preliminary results of the ensemble were carried out by one student, supervised by me, as his *Trabajo Fin de Grado* [PÁ17]. Figure 4.10 shows accuracy of the ensemble for different number of neurons on the hidden layer of the second stage and different combinations of activation functions (hidden layer and output layer). It can be seen that for more than 12 neurons, the accuracy does not increase with the number of neurons, hence 15 neurons has been chosen for the next experiments.

FIGURE 4.10: Accuracy vs. number of neurons (in the hidden layer) on the second stage (classifier ensemble) [PÁ17].

Table 4.4 shows the results of each quadrant into which is divided the input images, classified by a 2-layer MLP with 14×14 inputs, 40 hidden neurons, and 10 output neurons (each one recognizes a digit). The parameters for the first-stage training are: 1500 iterations, $\zeta = 1.03$, $\eta = 1.04$, $\rho = 0.9$, random initial weights and bias with uniform distribution in the range $[-0.15, 0.15]$, initial learning rate 10^{-5} , and both layer activation functions are saturated linear on -1 and +1, and slope 1. In order to avoid outliers, 3 sets of classifiers has been trained with 10000 different examples taken randomly from the *Train* set of the MNIST database. The rest of the signals which belong to the training set but were not used to train the first stage (I call them *Train⁻* sets) has been used to test the first stage and to train the second stage. As there are 3 *Train* sets for the first stage, there are also 3 *Train⁻* sets with 50000 examples. Therefore, values shown in this table are median of the three experiments and average of the three experiments for the global accuracy. It can be seen that the digit "5" can only be recognized effectively by quadrant 3.

Results of the Table 4.4 are not optimized. They were left with low accuracy on purpose in order to have more margin of improvement for the second stage.

Then, a MLP of 40 inputs (10 for each quadrant), 15 hidden neurons, and 10 outputs has been trained with the three *Train⁻* sets (which contains 50000 examples each one). The training parameters for the second stage are: 4000 iterations, $\zeta = 1.03$, $\eta = 1.04$, $\rho = 0.9$, random initial weights and bias with uniform distribution on range $[-0.15, 0.15]$, initial learning rate 10^{-5} , sigmoid activation function between -1 and 1, and origin slope 1, on the hidden layer; and saturated linear on -1 and +1 (slope 1) activation function on the output layer. Table 4.5 shows the results of the second

TABLE 4.4: Accuracy in % of the first stage classification. For each class is given the median of the three experiments (to avoid outliers), and for the global accuracy the mean (of the three experiments). First experiments from [Vil+18].

	Quadrant	0	1	2	3	4	5	6	7	8	9	Global
Train ⁻	1	61	91	65	71	63	12	57	77	56	59	62
	2	91	89	66	63	68	15	84	75	67	46	67
	3	87	93	48	44	61	64	82	74	43	60	66
	4	89	92	69	51	57	5	74	73	53	59	63
Test	1	65	93	67	73	65	11	62	75	58	60	64
	2	92	90	64	65	68	15	82	75	66	45	67
	3	87	93	50	49	63	61	82	72	40	62	66
	4	93	91	72	55	58	6	76	73	52	64	65

stage with and without the perturbations along with the results of the algorithms proposed in [Vil+18; Vil+19].

TABLE 4.5: Accuracy and adaptability of the second stage when the first stage has low accuracy ([Vil+18]).

	ANN		Majority			Basic weighted			Stochastic weighted	
	H	M	H	T	M	H	T	M	H	M
No perturbation	90.72	9.28	68.46	23.17	8.37	74.87	10.79	14.34	77.57	22.43
Im. Q1 G. 64	88.01	11.99	61.05	29.08	9.87	72.25	11.21	16.54	75.10	24.90
Im. Q3 G. 64	89.39	10.61	65.00	25.69	9.31	73.85	11.13	15.02	76.52	23.48
Link Q1 discon.	82.92	17.08	52.49	38.46	9.05	68.60	18.39	13.01	73.32	26.68
Link Q3 discon.	82.44	17.56	51.97	39.16	8.87	69.58	18.10	12.32	74.03	25.97
Link Q1 G. 0.5	86.07	13.93	63.11	28.05	8.84	72.90	11.75	15.35	76.02	23.98
Link Q3 G. 0.5	85.47	14.53	62.54	28.36	9.10	73.05	11.71	15.24	75.88	24.12
Link Q1 G. 1	75.36	24.64	57.72	33.32	8.96	71.73	11.67	16.61	74.64	25.36
Link Q3 G. 1	72.42	27.58	57.10	33.51	9.39	72.17	11.69	16.14	74.63	25.37

As mention before, previous results were obtained from a non-optimized first stage with low accuracy. Table 4.6 shows the results of an optimized first stage. The parameters for the optimized first-stage training are: 5000 iterations, $\zeta = 1.03$, $\eta = 1.04$, $\rho = 0.8$, random initial weights and bias with uniform distribution in the range $[-0.15, 0.15]$, initial learning rate 10^{-5} , sigmoid activation function between -1 and 1 (origin slope 1) on the hidden layer and saturated linear on -1 and +1 with slope 1 activation function on the output layer. To improve convergence of the learning algorithm, on the output layer, the activation function derivative has been replaced by the derivative of the sigmoid function, and saturated with a minimum value of 0.1. Note: in this case the sigmoid derivative of the hidden layer has not been saturated.

Table 4.7 shows results of the second stage when the first stage has been optimized. The training parameters are: 4000 iterations, $\zeta = 1.03$, $\eta = 1.04$, $\rho = 0.9$, random initial weights and bias with uniform distribution on range $[-0.15, 0.15]$, initial

TABLE 4.6: Accuracy in % of the optimized first stage classification. For each class is given the median of the three experiments (to avoid outliers), and for the global accuracy the mean (of the three experiments). From the new experiments [Vil+19].

	Quadrant	0	1	2	3	4	5	6	7	8	9	Global
Train ⁻	1	66	94	72	81	75	37	74	84	66	73	73
	2	94	93	80	70	71	52	90	91	80	76	80
	3	91	95	65	72	88	90	92	85	63	74	82
	4	91	94	85	57	68	43	80	78	67	70	74
Test	1	69	95	74	84	78	36	76	82	68	72	74
	2	94	94	80	74	71	49	88	92	80	76	80
	3	92	97	67	76	88	91	91	83	59	77	82
	4	93	94	85	61	71	43	80	77	64	74	75

TABLE 4.7: Accuracy and adaptability of the proposed algorithms for the second stage when the first stage has high accuracy (new experiments).

	ANN		Majority			Basic weighted			Stochastic weighted	
	H	M	H	T	M	H	T	M	H	M
No perturbation	96.21	3.79	84.14	11.68	4.18	87.36	5.62	7.02	89.22	10.78
Im. Q1 G. 64	95.46	4.54	81.48	13.75	4.77	86.25	5.78	7.97	88.14	11.86
Im. Q3 G. 64	95.54	4.46	81.88	13.43	4.69	86.26	6.07	7.67	88.23	11.77
Link Q1 discon.	94.01	5.99	70.91	24.18	4.91	83.64	8.52	7.84	86.08	13.92
Link Q3 discon.	92.60	7.40	65.81	28.41	5.79	82.10	10.67	7.23	85.07	14.93
Link Q1 G. 0.5	94.95	5.05	81.48	14.17	4.35	86.31	5.89	7.80	88.22	11.78
Link Q3 G. 0.5	95.20	4.80	82.06	13.71	4.23	86.53	6.16	7.31	88.41	11.59
Link Q1 G. 1	90.65	9.35	76.55	18.95	4.50	85.23	5.79	8.97	87.05	12.95
Link Q3 G. 1	91.38	8.62	75.22	19.99	4.79	84.51	7.04	8.45	86.74	13.26

learning rate 10^{-7} , sigmoid activation function between -1 and 1, and origin slope 1, on the hidden layer; and saturated linear on -1 and +1 (slope 1) activation function on the output layer. To improve convergence of the learning algorithm, on the output layer, the activation function derivative has been replaced by the derivative of the sigmoid function, and saturated with a minimum value of 0.1. Note: in this case the sigmoid derivative of the hidden layer has not been saturated.

4.1.3 Stacking DAEs

Simulations of stacking DAEs for the MNIST dataset were carried out in [Zha17] by a degree student directed by me.

First, the learning parameters (ζ , η and ρ) for the DAE training and the fine-tuning training have been adjusted. A sparse and non-equidistant set of values have been tested (more concentrated on areas we expected to have better results or more

gradient). Then, more concentrated points have been tested on the areas with minimum MSE. As the number of points to be evaluated is very big, every test has been repeated only 3 times and plotted the median of the obtained MSEs. The median is more robust against trainings that occasionally get stuck. Figure 4.11 shows the exploration on the learning of the first DAE layer, Fig. 4.12 shows the second DAE layer, Fig. 4.13 shows the fine-tuning and Fig. 4.14 shows the zoom on the area of interest for the fine-tuning.

FIGURE 4.11: Learning parameters for DAE to pre-train the first hidden layer. [Zha17].

On the experiments to adjust the learning parameters, the rest of parameters are: $196 \times 128 \times 36 \times 10$ architecture with sigmoid activation functions on the hidden layers and saturated linear activation function on the output layer; Gaussian noise with standard deviation of 0.2 and tied weights on DAEs; 1000 training examples and 500 test examples taken from the 1500 first examples from the MNIST train set; 300 iterations for the DAE pre-trainings and 1000 iterations for the fine-tuning.

Figure 4.16 shows the effect of the salt and pepper noise, with different percentages of corruption, on a 3 layers network with $196 \times Num.N_1 \times 64 \times 36 \times 10$ architecture

FIGURE 4.12: Learning parameters for DAE to pre-train the second hidden layer. [Zha17].

FIGURE 4.13: Learning parameters for the fine-tuning. [Zha17].

FIGURE 4.14: Zoom on the area with minimum MSE on the fine-tuning. [Zha17].

with sigmoid activation functions on the hidden layers and saturated linear activation function on the output layer. The parameters of the simulation are: $NumN_1$ takes 128, 160, 224 and 256; 5000 training examples and 3000 test examples taken from the 8000 first examples from the MNIST train set; 300 iterations for the DAE pre-trainings and 1000 iterations for the fine-tuning. The noise intensity variable (`noise.p`) is used on the Matlab code of Fig. 4.15 to generate different percentages of corruption. For every combination of $NumN_1$ and noise intensity 11 simulations were carried out and plotted the median of the failure (i.e. examples classified incorrectly) percentage.

```

Ic = I;                                ▷ Corrupted input ← input
for n=1:Nex
    Ic( ceil(NumIn*rand(round(NumIn*noise.p),1)),n ) = ...
        my_rand(noise.p);              ▷ Array of appropriate length of random -1 or 1
end

```

FIGURE 4.15: Corruption in Matlab with salt and pepper noise. For each example n , it corrupts the elements given by the array of length `NumIn*noise.p` (rounded) of random elements between 1 and $NumIn$. Notice that, as this array may have repeated elements, `noise.p` is not exactly the percentage of corruption.

FIGURE 4.16: DAE train with salt and pepper noise. [Zha17].

As a curiosity, the real percentage of corruption when `noise.p` = 1 tends to 63.21% for large input sizes (i.e. when the size of I_c tends to infinite). I got these data experimentally. Unfortunately I did not know the formula.

Figure 4.17 shows the effect of the Gaussian noise, with different standard deviations, on the same network than on the salt and pepper noise experiment and with the same parameters. The noise intensity variable in this figure is the standard deviation.

Figure 4.18 shows the same experiment that on Fig. 4.16 but reducing the number of hidden layers to 2 ($196 \times NumN_1 \times 36 \times 10$ architecture).

FIGURE 4.17: DAE train with Gaussian noise. [Zha17].

FIGURE 4.18: DAE train with salt and pepper noise on 2 hidden layer network. [Zha17].

4.1.4 Distribution of ANN over WSN with SystemC

Communication layer measurements

First, in order to verify and fully characterize the communication layer model for the Cookie layer with the ETRX2 transceiver, some measurements were performed.

To measure the power consumption of the Cookie node or layer, an auxiliary PCB with a shunt resistor, an instrumentation amplifier and necessary voltage regulators was made. Note: the voltage supplied to most of the Cookies nodes (including the layer with the ETRX2 transceiver) is 3 V, therefore the power consumption can be computed from the measurements.

Figure 4.19 shows: 4.19a the measured waveform in the oscilloscope, and 4.19b the simulated waveform, for sending a message of 1 byte (note that to the 1 byte message are added the necessary headers).

(A) Measured. Vertical divisions correspond to around 30 mV. (B) Simulated. Horizontal: time (ns) vs. vertical: power (μW).

FIGURE 4.19: Waveforms of a 1 byte message. It shows the wakeup time followed by listening for CSMA, transmission of the message and listening for the MAC acknowledge.

Figure 4.20 shows in the same way than Fig. 4.19 the waveforms for receiving the APK acknowledge of the previous message. In this case the process starts by the SED sending a poll request to the parent, which replies with the pending APS acknowledge message, and finalize by the SED sending the MAC acknowledge to the polled message (which results to be the APS acknowledge). And Figure 4.21 shows the polling of a 4 bytes message followed by the APS acknowledge.

Experiment 1

This simulation experiment consist on 6 nodes: 4 SED, 1 FFD, and the COO; performing the classifier ensemble described in 3.2.3 for the MNIST example application. Each SED node computes a $196 \times 40 \times 10$ MLP, and the FFD node computes a $40 \times 15 \times 10$ MLP. The COO only receives data from the FFD. The entire WSN is assumed well synchronized; all nodes spend most of the time in sleep mode to save energy and wake up every 5 minutes to process a new sensor input, i.e. a new MNIST image divided into 4 quadrants (each SED receives a quadrant).

Data of the communication layers is based on the ETRX2 datasheet and the previous measurements; and data of the processing layer is based on the static power consumption of an Artix FPGA, the latency of the ANN IP, and a rough estimation of the dynamic power consumption (because at this stage of the project, the power consumption of the microcontroller hosting the IP is unknown).

(A) Measured. Vertical divisions correspond to around 30 mW.

(B) Simulated. Horizontal: time (ns) vs. vertical: power (μ W).

FIGURE 4.20: Waveforms of polling a APS acknowledge. It shows the wakeup time followed by listening for CSMA, transmission of the poll request, listening for the APS acknowledge, transmission of a MAC acknowledge and a small processing time.

(A) Measured. Vertical divisions correspond to around 30 mW.

(B) Simulated. Horizontal: time (ns) vs. vertical: power (μ W).

FIGURE 4.21: Waveforms of polling a 4 bytes message. It shows the wakeup time, CSMA, transmission of the poll request, listening for a 4 bytes answer, transmission of a MAC acknowledge, some processing time, a second CSMA, transmission of the APS acknowledgement, listening for MAC acknowledge, and finally more processing time.

Table 4.8 shows the integrated total energy and the average power of the communication and processor layers of the simulated WSN during 48 hours. Pc stands for communication power and Pp for processing power (e.g. PcCOO is the communication power of the COO and PpSED is the processing power of the SED 1).

TABLE 4.8: Total energy consumed by the nodes and mean power during 48h.

Node	Total E (mJ)	Average P (μ W)
PcCOO	14.810	85.6
PcSED1	2.336	13.5
PcSED2	2.337	13.5
PcSED3	2.344	13.5
PcSED4	2.365	13.7
PcFFD	51.675	298.5
PpCOO	21.758	125.7
PpSED1	10.834	62.6
PpSED2	10.840	62.6
PpSED3	10.880	62.9
PpSED4	10.934	63.2
PpFFD	56.346	325.5

Figure 4.22 shows the power waveforms between two events. It can be seen that most of the time the nodes are in sleep mode. And Fig. 4.23 shows a zoom of the waveforms on an event. The FFD, as needs to collect the SEDs data, is the node with highest energy consumption.

FIGURE 4.22: WSN distribution simulation between two events.

The previous figures (extracted from the *GTKWave* tool) only show the shape of the waveforms proportional to their maximum value (which is 106.5 mW for the communication layers and 200 mW for the processor layers). Figure 4.24 shows accurately the simulated power waveforms from the SED 1. It can be seen that the static power consumption of the FPGA consumes the highest amount of energy of the node. The energy consumed during the actual computation of the ANN is very low (peak of processor power consumption around 300.02 s) because it is performed in a very short time.

FIGURE 4.23: WSN distribution simulation zoom of an event.

FIGURE 4.24: SED1 power consumption on an event.

4.2 Hardware implementation of ANNs

4.2.1 Validation of the VHDL IP

RTL simulation

The VHDL IP has been validated by RTL simulations with different VHDL test-benches. From small tests to validate incrementally each feature and submodule, to a big one to validate integration and full functionality. The last one can train the ANN by reading the examples from files. The first application example of image compression was implemented. However, it takes extraordinary long time, so I did not let the training to be finished. I just check that the signals were doing what they were designed to do, and that the first epochs of the training matched with the Matlab simulations. Figure 4.25 shows a fragment of this simulation on the *ISim* simulator (of the *ISE* tool).

FIGURE 4.25: RTL training simulation in ISim.

Validation in FPGA

After validation through RTL simulation, the first application example of image compression was implemented on the FPGA. Figure 4.26 shows the application running on a Virtex 5 FPGA (XC5VLX110T) of a XUPV5 Development Board, and Fig. 4.27 shows the application running on a Zynq SoC (XC7Z010) of a ZYBO. In both implementations, the weights were calculated on a off-line training by Matlab simulation, saved on an SD-card and written on the IP by the CPU (MicroBlaze or ARM respectively). The application project of the implementation on the Virtex 5 has been done with *ISE* and *EDK* tools, whereas the application project for the implementation on the ZYBO has been done with the *Vivado* tool.

FIGURE 4.26: Image coding with auto-encoder of the Lenna picture on the Virtex5. The left picture on the monitor is the original, and the right one has been processed by the auto-encoder.

FIGURE 4.27: Image coding with auto-encoder of the Lenna picture on ZYBO board (Zynq). The left picture on the monitor is the original, and the right one has been processed by the auto-encoder.

4.2.2 Validation of the SystemC IP

SystemC high-level simulation

The SystemC high-level simulations were used to check that the functionality matched with the Matlab simulations. The SystemC high-level simulations are also used by the macro-explorer to calculate the results for every set of parameters explored (see its results on section 4.2.5).

Validation in FPGA

Validation in the FPGA of the SystemC IP has been done, by a master student in the Hong Kong Polytechnic University supervised by my short stay advisor, as one of the examples in the *AS2Bench* [The15a; XCCS17]. Terasic's DE1-SoC reconfigurable computing board, which contains an Altera Cyclone V SoC.

4.2.3 Easy to use

Thanks to the parameterization performed, the IPs are easy to configure for different applications.

The VHDL IP can be easily added and configured in *EDK* and *Vivado* IP integrator. Figure 4.30 shows the configuration of the IP in *EDK*. The opened project is the system loaded in the Virtex 5 of the Fig. 4.26. Further, it can be seen how the BRAM port for the weight and bias memories (*PORTA*) is connected to a BRAM controller

(*bram_cntlr_0*) which is connected to the memory data bus (*dlmb*) of the MicroBlaze, and the slave PLB port (*SPLB*) to the generic peripheral bus (*mb_plb*).

Figure 4.28 shows a simplified example block diagram in *Vivado* IP integrator and Fig. 4.29 a configuration window of the IP in *Vivado*.

FIGURE 4.28: Block diagram in *Vivado* IP integrator with the VHDL IP (*ann_0*).

FIGURE 4.29: Configuration window of the VHDL IP in *Vivado*.

FIGURE 4.30: Configuration of the VHDL IP on EDK for the example in Virtex 5. The red circle shows the ANN IP (labeled *layers_0*).

4.2.4 Resource utilization

Table 4.9 shows the resource utilization of different configurations of the VHDL IP implemented on a Kintex 7 with *ISE*. The hidden layers activation functions are “siglut”, whereas the output layers uses “linear” ones. The configurations are designed as $NumIn \times NumN(0) \times NumN(1) \dots$. Layer type of the first layers is “SP”, of the output layers is “PS”, and of the middle hidden layer (for the 3 layers versions) is shown “SP” and “PS” cases. Rest of the parameters are *NbitW* 16, *NbitIn* 8, *LSbit* 12 for all layers, *NbitO* 12 for hidden layers and 8 for output layers, and *NbitOut* 8.

TABLE 4.9: Resource utilization of the VHDL IP on KC7K325T using *ISE*. The ANN rows shows the resource utilization of the whole IP, whereas its following rows only of the specific layer.

		Slices	Reg.	LUTs	DSPs	LUTRAM	BRAM
196×128×10	ANN	10452	4801	28108	288	18448	128
	SP	8481	3488	23018	128	16384	0
	PS	1962	1312	5080	160	2064	0
196×64×10	ANN	5348	2464	14053	144	9232	64
	SP	4312	1760	11531	64	8192	0
	PS	1026	703	2512	80	1040	0
196×32×10	ANN	2709	1295	7047	72	4624	32
	SP	2176	896	5764	32	4096	0
	PS	525	398	1273	40	528	0
196×128×32×10	ANN	103116	6216	28265	200	18960	33
	SP	8265	3488	23020	128	16384	0
	SP	1192	895	3078	32	2048	0
	PS	521	398	1269	40	528	0
	ANN	11337	5613	29427	328	18976	129
	SP	8595	3488	23021	128	16384	0
	PS	2096	1317	5084	160	2064	0
	PS	549	398	1264	40	528	0
	ANN	5092	3177	13619	100	8976	17
	SP	4281	1760	11528	64	8192	0
	SP	389	462	1002	16	512	0
	PS	262	241	643	20	272	0
196×64×16×10	ANN	5663	2908	14731	164	9504	65
	SP	4339	1760	11538	64	8192	0
	PS	1006	706	2513	80	1040	0
	PS	259	241	639	20	272	0

Table 4.10 shows the resource utilization of the VHDL IP configured for the example shown in Fig. 4.27 on the subsection 4.2.1, which has been implemented with the *Vivado* tool. This is a little optimized ANN that can be fitted onto a low-cost SoC. It has been configured with the following parameters: *Nlayer* 2, *NbitW* 16, *NumIn* 64, *NbitIn* 8, *NumN* {8 64}, *l_type* “SP PS”, *f_type* “linear linear”, *LSbit* {12 12}, *NbitO* {12 8}, and *NbitOut* 8.

TABLE 4.10: Resource utilization of a system with an auto-encoder on ZYBO. Synthesized with *Vivado*.

	System (PL)	ANN IP
Slice LUTs (17600)	6252	444
- as Logic (17600)	5861	444
- as Memory (6000)	391	0
Slice Registers (35200)	8299	481
BRAM tile (60)	14.5	8.5
DSP48E1 (80)	16	16

The table 4.11 shows how different tools (*ISE* and *Vivado*) synthesize the same VHDL codes. The parameters for this example (that performs the MNIST classification) are *Nlayer* 2, *NbitW* 16, *NumIn* 49, *NbitIn* 8, *NumN* {20 10}, *l_type* “SP PS”, *LSbit* {12 12}, *NbitO* {12 8}, and *NbitOut* 8. The design with “linear” hidden layer has *f_type* “linear linear” and the design with “siglut” hidden layer has *f_type* “siglut linear”. All of them has been synthesized for a Zynq XC7Z010, which is the SoC in the ZYBO development board. It can be seen that even being the two tools from the same vendor, *Vivado* uses far more resources than *ISE* when a “siglut” layer is present. The functionality that both results have is exactly the same, because that is what the language (VHDL) assures. QoR depends on the synthesis tool. In this case the reason is that *Vivado* does not recognize the sigmoid look-up-table as a ROM, and uses a great amount of LUTs, registers and muxes to implement the same functionality.

TABLE 4.11: Resource utilization of VHDL IP for “linear” and “siglut” hidden layer, synthesized with *ISE* and *Vivado*.

	“linear” ISE			“linear” Vivado		
	ANN	SP	PS	ANN	SP	PS
Slice LUTs (17600)	2080	1267	804	1305	519	786
- as Logic (17600)	1104	627	468	969	519	450
- as Memory (6000)	976	640	336	336	0	336
Slice Registers (35200)	871	590	281	1425	628	797
F7 Muxes (8800)				64	64	0
F8 Muxes (4400)				16	16	0
BRAM (60)	0	0	0	5	5	0
DSP48E1 (80)	45	20	25	40	20	20

	“siglut” ISE			“siglut” Vivado		
	ANN	SP	PS	ANN	SP	PS
Slice LUTs (17600)	2067	1245	812	10780	9979	801
- as Logic (17600)	1091	605	476	10444	9979	465
- as Memory (6000)	976	640	336	336	0	336
Slice Registers (35200)	852	570	281	1606	808	797
F7 Muxes (8800)				3316	3284	32
F8 Muxes (4400)				1092	1076	16
BRAM (60)	20	0	0	5	5	0
DSP48E1 (80)	45	20	25	40	20	20

Table 4.12 shows an example of DSE carried out on *Vivado* HLS for the same

configuration as the example of Table 4.11 with “siglut” hidden layer. It has been added the solution produced by ISE for the VHDL IP for compare them. It can be seen how for the same configuration, HLS can produce different solutions for the trade-off resources vs. latency, whereas RTL synthesis produces a single optimized fixed architecture. It must be noticed that the VHDL IP uses streaming ports for the ANN data, so input and output buffers are not necessary. On the contrary, the SystemC codes describes such buffers to allow generation of different input and output interfaces. Reading and writing these buffers takes around 60 clock cycles. Hence, the VHDL IP solution is equivalent to the *unroll complete* version. Solutions marked with * may present timing issues.

TABLE 4.12: Resource utilization of SystemC IP on ZYBO.

	Lat.	Slices	Reg.	LUTs	DSPs	BRAM
unroll complete	140	1577	3758	3600	40	5
unroll factor 10	199	1202	3283	2995	20	15
unroll factor 5	321	1078	3387	2393	10	10
unroll factor 4*	376	949	2841	2437	8	9
unroll factor 2*	1176	930	2803	1679	4	9
VHDL IP (ISE)	68		852	2067	45	20

Figure 4.31 shows other DSE for another two ANNs. In this case, inputs are $14 \times 14 = 196$ pixel images, and they have 3 layers. The middle layer kind is changed to get extra pareto-optimal architectures.

FIGURE 4.31: Latency in clocks (horizontal) vs. number of DSPs (vertical). Note: extra latency of $196 + 10$ has been added to VHDL designs to compensate SystemC buffers.

4.2.5 Explorer

Different qualitative and quantitative experiments have been conducted in order to test the effectiveness of the exploration flow on two of the example applications: image compression through auto-encoder and classification of handwritten digits. These results have been obtained mainly by my advisor on my short stay in the Hong Kong Polytechnic University (with my assistance) and published in [CSAM17].

In both cases the experiments have been run on an Intel Xeon2 processor running at 2.4 GHz machine with 8 GBytes of RAM running Linux Fedora version 20.0. The HLS tool used is *CyberWorkBench* v.5.4 [NEC14] from NEC. The target parameters are: an Altera Cyclone V (5CSEMA) as target technology, 100MHz frequency, and the exploration target is set to area vs. latency (throughput can also be selected).

Image compression through auto-encoder

Table 4.13 shows the qualitative results of the marco-explorer for the auto-encoder ANN. The macro-explorer is executed using the following parameters: 100 maximum number of neurons per layer, 3.0 and 1.25 maximum and minimum compression rate between input layer and hidden layer respectively, 2000 initial training iterations, 2000 increment of iterations, and target $MSE=10$. This implies an extremely low target error, as this experiment intends to analyze the impact of the QoR vs. running time. This first set of experiments were conducted first targeting an ANN configuration of 2 layers and re-run for a configuration of 4 layers. The experiments have been executed 3 times using different random seeds, so the averages of the results are reported.

From Table 4.13 it can be observed that the PSNR is good, with average values between ~ 31 and ~ 36 for the 2 layer configuration and ~ 26 and ~ 29 for the 4-layer configuration. The best configurations are $16 \times 10 \times 16$ for the 2-layer, and $100 \times 82 \times 50 \times 82 \times 100$ for the 4-layer auto-encoder.

Figure 4.32 (left) and (center) shows the configuration of lowest MSE and highest $PSNR$ found by the macro-explorer for the 2 and 4-layer auto-encoders for different number of training iterations (Num_itr). It can be observed that as shown in Table 4.13, the 2-layer auto-encoder always achieves better results for any number of Num_itr . Figure 4.32 (right) also shows the total running time taken by the macro-explorer. As expected the 4-layer auto-encoder takes longer.

FIGURE 4.32: Comparison of 2 and 4 layers configurations: MSE min, PSNR max and running time (s) vs. number of iterations.

TABLE 4.13: Explorer qualitative results: 2 layers vs. 4 layers.

PSNR	Testing Images									
	2 layers ($16 \times 10 \times 16$)					4 layers ($100 \times 82 \times 50 \times 82 \times 100$)				
Train

38.14	29.73	35.78	34.00	33.02	30.88	24.16	31.04	26.27	25.89
39.04	38.14	36.58	38.26	39.38	28.56	24.83	27.96	25.68	24.76
28.69	27.63	36.37	30.60	33.53	28.61	24.06	30.72	27.34	29.10
39.62	32.71	36.39	38.14	38.11	28.08	23.60	28.85	25.81	24.66
36.18	27.08	35.97	32.84	36.18	29.79	24.15	29.83	27.11	29.70
Avg.	36.33	31.06	36.22	34.77	36.04	29.19	24.16	29.68	26.68	26.82

Figure 4.33 (a) and (b) show the main results MSE and $PSNR$, respectively, for each of the different configurations executed for the 2 layer ANN vs. the number of training iterations (2000, 4000, 6000 and 8000). It can be observed that the QoR is a function of the ANN compression ratio. For ANN configurations with high compression ratio (e.g. $36 \times 8 \times 36$, $64 \times 2 \times 64$ or $100 \times 2 \times 100$) the MSE is always larger than 100, while the $PSNR$ lower than 18. The training duration also has an impact on the QoR, achieving better results with longer training, although as show in a later experiment, the best results are not obtained with the longest number of iterations (8000 in this case), but with 6000.

Figure 4.33 (c) characterizes the running time of the macro-explorer for each of the generated configurations. In this case, in order to fully characterize the method, instead of the binary search used to find the smallest ANN configuration, all possible configurations are generated and shown in this figure. It can be observed how the ANNs with larger number of neurons require longer running times. Also, as expected, the number of training iterations significantly impact the running time.

Figure 4.34 shows the HLS DSE exploration results (micro-explorer) for the smallest 2 and 4 layer ANN found by the macro-explorer reaching the specified target MSE . For the 2-layer ANN the trade-off curve is composed of 6 dominating designs with unique area vs. latency found by the explorer. The exploration running time took 3814 seconds, while the the 4-layer ANN trade-off curve is composed of 7 dominating designs. The exploration running time in this case took 6191 seconds. It can be observed the 2-layer designs require less area than the 4-layer configuration as expected for all designs on the trade-off curve, which also require larger latencies.

The results were initially surprising because it was expected that the 4-layer ANN would achieve better results than the 2-layer version. Further investigation

FIGURE 4.33: Results of auto-encoder macro-explorer with default options: (a) MSE (b) PSNR (c) Running time in seconds.

FIGURE 4.34: HLS DSE (micro-explorer) trade-off (area vs. latency) curve for 2 and 4 layer auto-encoder.

on this issue revealed that the main reasons for this was due the linear activation function used.

Classification of Handwriting ANN Results

In this case the input and output layers of the ANN are constant and hence do not need to be explored, but the ANN is more complicated as it involves different type of activation functions. In particular the explorer can choose between: sigmoid, saturating, and linear activation functions. Thus, in this case the macro-explorer does not only explore the number of neurons of the hidden layer, but also the activation functions of each layer.

As shown in the previous section, the 2-layer ANN in this case also leads to better results, thus only the results of the 2 layer exploration are shown here in order to simplify the experimental results. The results shown here, further validates the observation that the QoR is dependant on the number of training iterations, although in this case by very little: 67%, 69%, 71% and 72% accuracy for the 2000, 4000, 6000 and 8000 iterations case respectively. At the same time the total running time grows linearly with it, clearly showing the trade-off between running time and QoR.

FIGURE 4.35: Accuracy and total exploration running time in seconds for 2-layer ANN. MNIST application.

Figure 4.36 shows a closer look to the quality of results as indicated by the accuracy and the running time when the number of training iterations is set to 2000, 4000, 6000 and 8000. It is interesting to observe that in this case contrary to the auto-encoder, the QoR is not influenced much by the training iterations, while the running time grows considerably. for ANNs with small number of hidden layers the number of iterations have a higher impact on the QoR, while for bigger ones the results in all cases seem to converge around 70%. Thus, it can be concluded that for this type of ANN 2000 training iterations is enough.

Figure 4.37 shows the result in terms of accuracy when the activation function is explored for 2-layer ANN found by the macro-explorer (in this case $196 \times 76 \times 10$). It can be observed that the activation function does impact the QoR. All the combinations of the activation functions are show in the figure. It can be observed that only three configurations, all of them with a sigmoid activation function, reach correct classifications of 70% or over. In particular saturation-sigmoid, sigmoid-saturation

FIGURE 4.36: Results of ANN macro-explorer with default options: (left) Running time in seconds (right) Accuracy. MNIST application.

and sigmoid-sigmoid. This shows that it is important to include also the activation function as input parameter in the explorer.

FIGURE 4.37: Activation function exploration results.

Finally, figure 4.38 shows the final trade-off curve of the dominating micro-architectures found for the best 2-layer configuration (when a threshold of correct classification is set to 70%). In this case $196 \times 76 \times 10$ with a sigmoid and saturation activation function for the two respective layers. The fastest micro-architectures mainly correspond to the mapping of the arrays to registers, while the slowest ones when the arrays are mapped to RAM.

4.3 Analysis of VHDL vs. SystemC

After my experience describing ANNs in both VHDL and SystemC, analysis and conclusions have been extracted. Particularly, for the parameterization and configuration mechanisms that both languages offer. Because it has been found that, even being this more important for higher level of abstractions, the VHDL mechanisms

FIGURE 4.38: HLS DSE (micro-explorer) trade-off curve result (area vs. latency) for smallest 2-layer ANN.

(the *generics*) are simpler but more flexible than the SystemC mechanisms. This analysis by itself is another result of this Thesis and has been published in [ACSM19].

In order to avoid confusions and clarify terminology, let's define *to configure* as to give values to a set of *parameters* to obtain a particularization from a general model or description. A *parameter* as a variable which can be set (make it constant before synthesis) to modify an aspect of the model, generally a parameter determines a specific aspect of the model but not its general nature. And *to parameterize* as to extract, select, and determine the parameters of a model.

Besides the parameterization analysis, using the tool *cloc* [Dan] I obtained the code lines of both designs excluding blank lines and comments. The VHDL IP kernel has 1620 lines whereas the SystemC IP has only 594 lines. This shows (as expected), how with the higher level of abstraction in SystemC can be achieved an ANN design with fewer code lines.

4.3.1 Parameterization in hardware design

In order to show the relevance of this analysis, let's perform a brief introduction on the issue:

To improve efficiency of hardware design, reuse of modules is a common practice. Those modules are the well-known IPs. Further, in order to allow and make easier the code reuse, two features should be enhanced: portability and configurability. Portability is ensured by using standard languages, in this case VHDL [Vhd; Std] and SystemC [Sys; Syn]. Then, mayor discussion on this section is about configurability.

When abstraction level increases code reusability becomes more important because they are more general, therefore they are more likely to be reused. Furthermore, since high-level abstraction descriptions do not specify micro-architectures, they can be more easily reused for different area versus performance trade-off requirements. As opposite, Ad-Hoc optimizations are described in a low abstraction

level in order to take advantage of the low-level details. Hence, these descriptions are difficult to reuse.

Good configurability allows also making faster changes on the code (if the needed changes are already parameterized).

Structure of ANNs makes them a challenge to be parameterized on HDLs. Because they are composed of layers of neurons, and each one of this layers may have different configurations. And the number of layers is needed to be a parameter too. Thus, parameterization of this structure, made of a configurable number of submodules which are needed to be set independently is a challenge as will be shown on next subsections.

4.3.2 VHDL parameterization

As explained on chapter 2 section 2.1.1, structure of ANNs can be easily parameterized by generics, constant array types and for-generate loops.

Another interesting feature of VHDL and its RTL synthesis tools is the calculation of constants. In order to optimize hardware designs, every value which can be known at synthesis time can save all the logic needed for its computation. Results of operations with constants are treated as constants as well, even when functions are applied (as long as these functions do not use any run-time value). As opposite to software compilers, which leave these computations to run time when they become complex and always that a function is applied. Because usually on software for increasing its flexibility, a slight bigger object code is preferred. This is true for earlier versions of C++ than C++11. However, even in C++11 and newer versions, synthesis-time constant calculation must be specified on the code through the key word `constexpr`. Thus, already existing functions (like library ones) can not be used for this purpose.

One of the main drawbacks of RTL synthesis against HLS, is the fixed architecture. On RTL abstraction every clock cycle data calculation or movement is detailed. Therefore, trade-offs between area and performance have to be decided *a priori* and coded explicitly. This means that have to be decided which calculations will be parallelized and which ones will be serialized to reuse resources. If a future application needs another trade off, old code can not be reused and have to be rewritten.

4.3.3 SystemC parameterization

However, configurability is a weak point of SystemC. Due to the fact that SystemC is a C++ library, its parameterization mechanisms are defined by the ANSI C++ standard. Although it might seem contradictory, because C++ is well-known as one of the most flexible languages, the C++ standard was created for software development (which has different characteristics and needs than hardware development). Although SystemC is a good adaptation of a software language to a hardware description language, which has successfully addressed most of these differences, parameterization still has some issues related with this fact.

The ANSI C++ parameterization mechanisms which can be used to parameterize a module in SystemC are: macros, templates, and constructor parameters.

- **Macros:** with the `#define` preprocessor directive can be defined scalar global parameters. An advantage of macro parameters is they can be used on `#if` preprocessor directives. Like VHDL if-generate statements, `#if` can be used to skip from synthesis those parts of the code which may produce errors for the selected configuration.
- **Templates:** the C++ template parameters can be configured independently for different instances of the same module. It seems that C++ templates are the best option to parametrize modules like VHDL generics do. However, they have some limitations explained below.
- **Constructor parameters:** as SystemC modules are classes, they have constructor functions which may have parameters. The problem of constructor parameters (as any function parameters) is that from the C++ point of view they are variables known at run time. For this reason, constructor parameters are not synthesizable [Syn].

The main drawback of the C++ templates is they cannot interact with the preprocessor. They define parameters known at compiler time that can be used as constants and to perform some compiler-time optimizations, but they cannot be used to take decisions at synthesis time. This means to use C/C++ macros on `#if` preprocessor statements, like VHDL generics are used on if-generate and for-generate statements. It is important to notice that there is not a `#for` preprocessor statement neither something similar in C++. If templates are used on C conditional statements (if, switch, etc.), compilers and HLS tools might try to process code blocks that never will be active for the selected configuration. Some C++ compilers and HLS tools are smart enough to avoid processing these blocks of code. However, this feature is not a requirement, so there is not guarantee of these code blocks will not cause any compiler/synthesis error in any tool.

Another important limitation is that array elements or string literals cannot be used to give value to template parameters.

Besides, there is also some limitations on the data types which templates can accept. Arrays and user defined types are tricky to add properly. Here [templatetype] is shown how to add this kind of template parameters, but this syntax is complex and it is not sure if every HLS tool will synthesize it correctly.

Furthermore, C++ has not arrays of constants. A C++ *const* array is a read-only address of an array, but its elements are treated as variables (known only at run time) even if they are addressed by constant indexes.

4.3.4 Comparison

The main differences between software and hardware development that affects parametrization are preprocessor features, parameter types, and when decisions are taken. The C++ preprocessor is quite limited, avoiding iterations expressly. Iterations are loops or recursions. Although C++ define templates, as it was shown above, they are not always enough. Function parameters are known in run time. Typically in software, most of the decisions are taken at run time, providing software programs huge flexibility at expenses of slight bigger object code (which most of the times this increase on the code is neglected). On the other hand, on hardware design every optimization that can be done on synthesis time has a great impact.

Summarizing and putting together analysis of VHDL and SystemC ANN descriptions, the main points of improvement for future HLS tools are:

Calculation of constant values during synthesis time: whereas VHDL synthesis tools can calculate constant values even applying functions, SystemC only can use constant expressions. In VHDL, the result of a function which has been called with constant values and do not use internally any run-time data, is treated as well as a constant. Allowing to use this result in such places where only a constant can be used, like length definition of an array, and to perform optimizations. All these computations are skipped from run time, hence no extra logic is inferred. However, C and C++ functions cannot distinguish constant parameters known at compiler-time from variable parameters only known at run-time, so they treated all of them as variable values, carrying out all computations at run time.

Parameter types: VHDL only uses one kind of parameter: the generics. They can be of whatever type, including array and user defined types, and they can be used on if-generate and for-generate statements (which are processed on synthesis time). As opposite, SystemC uses two synthesizable kinds of parameters: macros which can interact with the preprocessor but are limited to global scalars or string literals; and templates that can be set independently for different instances of the same element, but they have the following issues:

- They cannot interact with the preprocessor, so they cannot be used to take decisions at synthesis time. If they are used on C conditional statements (if, switch, etc.), compilers and HLS tools may try to process the code blocks that never will be active for the selected configuration, so there is not guarantee of these code blocks will not cause any compiler/synthesis error in any tool.
- Although they can define more parameters types, arrays and user defined types still cannot be guaranteed that every HLS tool will synthesize them correctly. Besides, array elements or string literals cannot be used to give value to template parameters.

Further, C++ does not have constant array of constants, neither anything equivalent to a *const* of an array type in VHDL. A C++ *const* array is a read-only address of an array, but its elements are treated as variables (known only at run time) even if they are addressed by constant indexes.

For-generate: There is not a `#for` preprocessor statement neither anything similar in C++. Because C/C++ preprocessor avoids iterations expressly. The VHDL for-generate statement is very useful for instantiating parallel modules of the same type without having to copy and paste code. Furthermore, it allows configuring the number of copies. Furthermore in addition, combining it with the generic arrays, each module copy can be set with different parameter configurations.

Chapter 5

Conclusion, future work and summary of contributions

5.1 Conclusion

During this PhD thesis has been explored different methods, algorithms and techniques to implement ANNs into embedded FPGAs that may be found into WSN nodes (e.g. in the Cookies platform).

Chapter 4 gives enough details of the experiments for replicability.

The following conclusions have been extracted from this Thesis:

- The best (or at least good enough) combination of parameters of the ANNs to solve the proposed application examples have been found.
- It has been studied the effect of linear, saturated linear and sigmoid activation functions to solve non-linear problems, and what is more interesting: the effect of modifying the derivative of the activation function in the backpropagation learning algorithm. Whereas the sigmoid activation function solves better the non-linearities, the saturated linear activation function with the modified derivative or the sigmoid derivative achieves acceptable solutions using a cheaper hardware. Replacing or adding small values to the derivatives when they become zero, helps to solve specific application convergence problems. However, the later technique is only applied when these convergence problems appears.
- One option to distribute an ANN over a WSN for classification applications is as a classifier ensemble. It produces larger ANNs on the sensor nodes, although they may be decomposed using other techniques. Nevertheless, it has been proved that is an effective method and it is easy to train. Compared against the other ensemble methods tested in [Vil+18; Vil+19], the ANNs achieve the highest accuracies; and although they are less adaptive (without re-training), because they lose proportionally more accuracy when perturbations are present, they still retain the highest accuracies under most of the perturbations.
- Regarding the stacking of DEAs, it has been shown that salt and pepper noise works better for the MNIST application (and the ANN tested sizes) than Gaussian noise. Besides, for an unknown reason, the effect of the noise is stronger in the 2-hidden-layer network than in the 3-hidden-layer one.

- The explored FIL methods (System Generator and HDL verifier) did not accelerate nor training neither simulation of the ANN. Therefore, they are only suited for verification proposes.
- Through the SystemC simulation of the WSN, the trade-offs between communication and processing consumptions can be detected, and hence, new opportunities for optimization. For example, in the example 1 of section 4.1.4, the static power of the FPGA represent the highest energy consumption whereas the energy consumed during the actual computation of the ANN is very low. Using a separate low-power microprocessor for node management and keeping the FPGA off during longer time may be studied. Although, turning on and FPGA configuration times and energy consumption must be carefully taken into account.
- The result source codes of both IPs (VHDL and SystemC) are freely available on [AM16] (VHDL) and [The15b] (SystemC). They efficiently implement small MLPs on the FPGAs and can be configured for different applications.
- The two-tier explorer can save a lot of time selecting the ANN parameters and configuring the micro-architecture for the SystemC IP.

Regarding the analysis of VHDL vs. SystemC, Table 5.1 summarizes pros and cons of VHDL and SystemC. As result of this analysis, I have concluded that SystemC parameterization mechanisms are more complex and less flexible than VHDL's ones.

TABLE 5.1: Summary of pros and cons of VHDL and SystemC.

	Pros	Cons
VHDL	<ul style="list-style-type: none"> - Ad-Hoc optimizations can take advantage of low-level details. - Flexible and efficient parameterization. 	<ul style="list-style-type: none"> - More code lines and design time. - Bigger gap between algorithm design and RTL design. - Fixed architecture. Re-design needed for different area vs performance trade-offs.
SystemC	<ul style="list-style-type: none"> - Less code lines and design time. - Smaller gap between algorithm design and HLS design. - Easier and faster DSE. 	<ul style="list-style-type: none"> - Parameterization mechanisms (inherited from C++) are inefficient and not flexible for hardware synthesis.

As traditional RTL synthesis tools can address some of the SystemC HLS weak points, I hope that future HLS tools will address properly these issues too. It is contradictory that low-level of abstractions tools can parametrize properly complex designs like ANN, but the high-level of abstraction ones cannot, precisely because parameterization becomes more important when the level of abstraction rises.

5.2 Related work

A large variety of ANN hardware implementations have been published targeting FPGAs, ASICs, or both [MS10; OR06; ZS03]. In most of the related work, the ANN is described at the RTL, using either VHDL or Verilog. In more recent publications use HLS to construct the ANNs. Some examples include [NTV91; Ort+06; Sma+14; Far+07; THGG07; Sud+16; Bet+17; LTA16], of which [Let+04; CH05] make use of SystemC.

Higher levels of abstractions have been gradually introduced in the hardware design flow in order to deal with the increasing productivity gap (gap between transistors available to designers and transistors that designers can design and verify). C-based design is one of them and is becoming mainstream, mainly because the two major FPGA vendors (Altera and Xilinx) have released their own HLS tools and have made them widely available to academia and industry. One early example in the use of HLS languages for ANNs is [NTV91]. Here, the authors extended the ANSI-C syntax to allow designers to explicitly specify the parallelism in the code using e.g. `PAR` statements. They called this new C subset nC. More recent examples using SystemC as input language includes [Let+04]. Ortigosa *et al.* [Ort+06] performed a comparative study between an ANN implementation at the RTL using VHDL and a higher abstraction level using Handel-C (ANSI-C subset for HW implementation). They report a design productivity increase of a factor of 10× when using it. In [Sma+14] a cell network model, originally coded in Matlab, is mapped onto an FPGA chip using ANSI-C as input language synthesizing it with Xilinx Vivado HLS.

Achieving a high level of configurability at the RTL is not complicated, an example is [GUZ11]. Here, an ANN is parameterized in a similar way using generics and generate statements. The VHDL IP of this Thesis includes three kinds of layers in order to make it even more flexible in terms of area vs. performance trade-off. In [Bah+14], shows a parameterizable VHDL IP to perform a DSE with their own tool.

In [Eij+14], shows a parameterizable SystemC-TML code for DSE and simulation, however it has to be converted into synthesizable SystemC for the selected configuration. In [Kre17] successfully parameterize layers for HLS using templates, however it need to instantiate each layer separately with their own scalar parameters, *i.e.* the number of layers is not a parameter. Hence, the code of the IP must be modified if a different number of layers is needed. Another solution is to have a tool that parses an even higher-level description, such as graphical based one or using Caffe like the re-Vision stack from Xilinx [Rev]. This approach is useful when the tool is also used for training the ANN or carrying an automated DSE like in the explorer. This last approach is gaining popularity as it fully abstracts the details of the ANN away from the user. Other examples include [HR17; ZCP17; Sha+16; VB16; Gua+17], which propose easy to use frameworks to generate ANNs. Nevertheless, to have a parameterizable ready-to-use IP is convenient. For example, for inserting it directly on the FPGA programming tool like Vivado without needing to call an external tool to re-write the IP code every time that it is reconfigured. Besides, it allows to make changes or add new functionality directly into the source code, and reuse the code in other projects. On the other hand, frameworks obscure the source code or make it inaccessible.

The contributions of this Thesis' designs are not only the improved level of configurability of the ANN, but also the analysis and comparison of both design

methodologies; and more important, the suggestions to improve future HLS tools, as a result of this analysis.

The HLS ANN models allow the automatic generation of different ANN configurations from the same model (DSE). Related work in ANN DSE can be decomposed into macro and micro-architectural exploration, where macro-architecture refers to the ANN configuration, while micro-architecture refers to the hardware implementation architecture of a particular configuration.

Regarding macro-architectural exploration Farrugia *et al.* [Far+07] presented a dataflow based Architecture Algorithm Adequation (AAA) methodology in order to find the best compromise between the processing power and functionality requirement of each processor element (PE). Torres *et al.* [THGG07] developed a software platform called NNetWARE-Builder, which takes as inputs graphically-designed neural networks and automatically compiles them onto FPGA devices with third party synthesis tools.

With regards to micro-architectural ANN DSE, Chtourou *et al.* [CH05] proposed an explorer which automatically modifies the global synthesis options of the HLS achieving ANNs with different area vs. performance vs. power implementations. Other research which particularly targets HLS DSE can be classified by the exploration method: predictive or synthesis based. In the first category (predictive based), the proposed methods generate a set of designs and then derive a predictive model to avoid having to re-synthesize every new configuration (which is the most time-consuming part of the DSE). Examples of predictive-based HLS DSE are [LC13] and [CSW13]. In [LC13], the authors proposed a learning-based method based on transductive experimental design techniques, while in [CSW13] the predictive model is based on a regression tree learner. With regards to the second category of HLS DSE methods, evolutionary multi-objective meta-heuristics have been proposed and shown good results [HKR07; CSTW09]. Thus, in the proposed explorer a GA method is used, which also leads to good results. It is different from previous works as it integrates both macro and micro-exploration and it is fully automated. It also directly maps any ANN configuration onto a configurable SoC FPGA, allowing it to be executed in hardware fully automatically.

In summary, there is a lot of research in the area of hardware implementations of ANNs. This Thesis is complementary to the previous results as it not only describes in detailed how to design highly flexible ANNs at the behavioral and RTL, but also compares both approaches and finally exposes some of the limitations in current methodologies, making suggestions on how to relieve these.

Regarding the WSN simulator in SystemC, it was based on [RGL09] which was inspired on [FV09; Man; Lev+03; Fek+07; Ril03; Oul+05; TLP05; Pol+04; Che+05; Ahm+05; Bal+05]. There are a great variety of WSN simulators, which differ on their generality or platform-specific, operating systems, language and/or graphical inputs, versatility, etc... The proposed one is specific for the Cookies platform, but since it is a very flexible platform with different kinds of processors and communication layers, the SystemC language input and the selected structure allows to model all the variety presented by the Cookies platform.

5.3 Future work

These sections provide some research lines, that will be performed in the future or that may be followed by the author, his colleagues or maybe the readers of this Thesis.

Regarding:

5.3.1 ANN IPs

New activation functions and layer types can be added to both the VHDL and SystemC IPs. Specially interesting will be add CNN layers and pooling layers, since they have become very popular in deep learning for image processing applications. Future technology will allow to effectively introduce more neurons and deeper ANNs in WSNs.

5.3.2 WSN simulations

As mentioned in section 3.3.1, the template functions to execute C programs for the Cookies platform using their libraries, and the GUI for the SystemC simulations of WSNs have been left as a future projects. Besides, new models for each layer can be added (for different sensors, other transceivers, more complex power layers, etc.).

5.3.3 HLS tools

Regarding the preprocessor issues, one possible solution may be to allow a processing stage during synthesis. As SystemC define three phases for the simulation: elaboration, execution, and cleanup; elaboration phase could be used for synthesis-time calculations. Elaboration phase is every statement executed prior to the `sc_start()` call. If synthesis is divided in two phases, the first one can compute C++ statements in a less restrictive way to get a set of values known at the end of this phase, let us call them post-elaboration constants, and to program the system structure instantiating modules using if and for statements. Note that this may imply to allow creation of dynamic objects during this phase. This also open the possibility of using constructor parameters.

5.3.4 OS API for hardware

Although the future line described here is also related to HLS tools, it is described in a separate subsection because it is a project on its own.

The aim of this research proposal is to achieve synthesize almost whatever C++ source code in order to allow incremental refinement for HLS. The first version, for sure, will not be an optimized hardware. Nevertheless, refinement towards optimization will be able to be made step by step, until it is fast or small enough for the application. To achieve this, the restrictions to the synthesizable subsets of C++ should be eliminated. They are (mainly) data types, dynamic allocations and calls to the OS (Operating System). In this proposal are pointed some feasible objectives to

synthesize these non-synthesizable parts of C++. One of the key ideas is to create a synthesis tool capable of “link” calls to the OS (with POSIX interface) to the services provided by ReconOS [Agn+14].

Using HLS, there is still the need for a manual translation from the highest level descriptions to the synthesizable subsets. Although this translation is more straightforward, it disables incremental refinement. Even in the best case in which the highest level description is a C++ one, data types must be changed, and any use of dynamic memory must be replaced by equivalent static storage, along another adjustments to the synthesizable subset.

The first synthesized version of an algorithm in C++ downloaded from the Internet, for sure will not be an optimized hardware. And it may not fit on most of the available FPGAs on current the market. Nevertheless, refinement towards optimization will be able to be made step by step, until it is fast or small enough for the application. A correct functional first version will be of a great value. Reducing drastically the debugging process that is needed, with current HLS, after changing at once all the non-synthesizable data types, dynamic memory allocations, OS calls and other refinements.

Of course, the synthesizability restrictions have a purpose: floating-point data types are inefficient for hardware implementations, dynamic memory allocation is not allowed because hardware does not volatilize or is created suddenly... really? On SRAM based FPGAs, by partial and dynamic reconfiguration, dynamic objects can be synthesized. On ASICs, a reconfigurable logic can be added. Besides, this is not the only solution (see subsection 5.3.4).

Calls to the OS are not supported by synthesis tools because there is not such a thing as an OS for hardware... Actually, there are! like ReconOS for example (see section 5.3.4 for others).

Objectives

The main objective (as introduced) is to achieve synthesize almost whatever C++ source code. Hence, to reduce, or even eliminate if it is possible, the restrictions for the synthesizable subsets of C++. The main restrictions are:

- Data types.
- Dynamic memory.
- Calls to OS.

On the following subsections there are more specific objectives regarding these restrictions and discussion about how to obtain them.

Data types

Data types, including custom types and classes, may represent data or functionality. When they represent just data, it is maybe the easiest restriction to eliminate. In this case, enough bits should be reserved for its storage and signals/bus lengths.

When arithmetic operations on them are needed, they can be decomposed into operations on basic types. Although not all basic data types are efficient for hardware implementation (e.g. floating-point types), they can actually be implemented on hardware. Some HLS tools can synthesize floating-point data types, for example Vivado HLS [Hri12].

- **Sub-objective 1:** Add library for non-synthesizable types (e.g. floating-point) operations regardless of how inefficient or big are these implementations.

The state *the data may represent functionality* means that object code (CPU instructions) are stored in memory, so they can be accessed as data. Some data types store or point to instructions like pointers to functions and classes with methods. For software execution, get data or execute the desired functionality can be reduced to access the appropriate memory address.

C++ is so flexible that allows a program to mesh up with jumps (e.g. *go to*), function pointers, virtual functions, etc... At the end it will simply get an address and place the memory content pointed by it to a data register or the program pointer register (along with other operations and context switching). However, to synthesize such programs special care must be taken to identify which data types are just data and hence reserve enough space for them, or they represent functionality, so they will be synthesized as logic and produce a hardware module.

- **Sub-objective 2:** For those types which are not synthesizable yet, define new rules to properly synthesize them as data (signals, registers and/or memory positions) or as functional logic.

Dynamic memory

Dynamic memory allocation is not allowed by the HLS synthesis tools. Nevertheless, there are already solutions to this restriction in the state-of-the-art (see section 5.3.4). These solutions can be summarized into: the use of a heap storage and virtual memory facilities for data, and reconfiguration for functionality. Heap storage and virtual memory can be added to the synthesized hardware by providing these services by an OS. Therefore, this sub-objective is included in the sub-objective 4.

There is two main kinds of reconfiguration: VRCs (Virtual Reconfigurable Circuits) and DPR (Dynamic Partial Reconfiguration). Both have their advantages and drawbacks, but the author of this proposal believes that for big functionality (big modules), DPR should be the solution. A short of VCR for polymorphism is covered by the patent [VC13], and the author of this proposal has not found yet any publication for fine grain DPR applied to synthesize *malloc* calls. DPR on ReconOS is a coarse grain DPR for thread allocation.

Summarizing: synthesize dynamic allocations is feasible. Decision about implementing a dynamic functionality as VCR or DPR may be fixed or restricted, automated, or decided by user (e.g. by using pragmas).

- **Sub-objective 3:** Use reconfiguration to synthesize dynamic allocations of functionality.

Calls to OS

That is the key of this proposed project. ReconOS already provide some OS services to the hardware threads. Therefore, the next step is to include in the synthesis tool a way to “link” calls to OS functions to the ReconOS services. The synthesis tool should recognize the OS API (ideally a POSIX API) and map it into the corresponding ReconOS structure.

To achieve compliance with POSIX, it may be needed to add support to new OS services that have not been implemented yet in ReconOS. Some standard OS services may be totally unnecessary for hardware threads. In such cases, that functionality should be ignored or provided an alternative functionality, in order to allow its synthesis without crashing or malfunction.

Add to the synthesis tool a way to recognize POSIX API and map these functions to the ReconOS resources. Add new or alternative functionality to the ReconOS services to allow POSIX compliance for hardware threads if it is necessary.

- **Sub-objective 4:** Add to the synthesis tool a way to recognize POSIX API and map these functions to the ReconOS resources. Add new or alternative functionality to the ReconOS services to allow POSIX compliance for hardware threads if it is necessary.

Related work

In addition to ReconOS [Agn+14], there are other OS supporting hardware accelerators. BORPH [SB08] from Berkeley is a very complete example, however, unfortunately it look like it has poor maintenance on the last years. In the University of Ottawa they propose [VAG08; GA04]. Another examples with a quite complete interfaces for the hardware threads is [And+08; Asi+17]. In [VP17] it is proposed a light version. These other examples are more focused on the reconfiguration [VF14; Pag+17], and this one on communications and I/O abstractions [Fle+14].

There are some works that uses Java instead of C++ [Poz+05; BCS14; WK02], hence they introduce a virtual machine and an intermediate format, which is very useful for portability. However, state-of-the-art HLS tools takes C++ no Java. [WK02] is an old proposal but very interesting because they also propose metrics for OS targeting reconfigurable systems.

Although there are already some OS to support reconfigurable hardware accelerators, this proposal differ on the purpose: to achieve full C++ stabilizability, and in the need of a synthesis tool that integrates the OS services into the HLS flow.

Regarding synthesis of memory allocations of data the state-of-the-art solutions are based on the use of virtual memory [Dia+15; Des+14; SSDM00]. In [Coo+09] they found a way to get the maximum heap size needed. In [WBC16] they use a technique called separation logic to find dynamic data structures, then they optimize access to them in loops and re-writes the source code to fit into the synthesizable subset. In [Zha+16] they optimize typical dynamic data structures by dividing their implementation in access modules and mutator modules.

5.4 Summary of contributions

- The configurable and freely available ANN IPs (both VHDL and SystemC versions).
- The two-tier explorer for the SystemC ANN IP.
- The Matlab scripts and functions to simulate and train ANNs.
- The SystemC code for WSN simulations.
- The analysis of VHDL vs. SystemC.

5.5 Publications

The following are the publications published during this Thesis:

- Transactions: Aledo, David; Carrion Schafer, Benjamin; Moreno, Félix. "VHDL vs. SystemC: Design of Highly Parameterizable Artificial Neural Networks". In *IEICE Transactions on Information and Systems*, March 2019 [[ACSM19](#)].
- Journal: Villaverde, Mónica; Aledo, David; Pérez, David; Moreno, Félix. "On the adaptability of ensemble methods for distributed classification systems: A comparative analysis". In *International Journal of Distributed Sensor Networks*, July 2019 [[Vil+19](#)].
- Conference: Carrion Schafer, Benjamin; Aledo, David; Moreno, Félix. "Application Specific Behavioral Synthesis Design Space Exploration: Artificial Neural Networks. A Case Study". In *2017 Euromicro Conference on Digital System Design (DSD)*, August 2017 [[CSAM17](#)].
- Conference: Villaverde, Mónica; Aledo, David; Pérez, David; Moreno, Félix. "A comparison of adaptive and non-adaptive ensemble methods for classification systems". In *17th Mexican International Conference on Artificial Intelligence (MICAI 2018)*, October 2018 [[Vil+18](#)].
- Software: Aledo, David; Moreno, Félix. "ANN". In *OpenCores*, June 2016 [[AM16](#)].
- Software: The Hong Kong Polytechnic University, DARClab. "S2CBench v1.1, ann". 2015 [[The15b](#)].
- Degree thesis: Viñarás Molinero, Darío. "Estudio de la arquitectura de redes neuronales artificiales para compresión de imágenes". From *Universidad Politécnica de Madrid*, February 2015 [[VM15](#)].
- Degree thesis: Lasarte Zapata, Eduardo. "Entrenamiento de una red neuronal hardware desde Matlab (hardware in the loop)". From *Universidad Politécnica de Madrid*, February 2017 [[LZ17](#)].
- Degree thesis: Pardo Álvarez, José María. "Aplicación de algoritmos de redes neuronales distribuidas en la clasificación de números manuscritos". From *Universidad Politécnica de Madrid*, February 2017 [[PÁ17](#)].

- Degree thesis: Zhang, Yi. “Optimization of Multi-Parameters Deep Learning Neural Networks”. From Northwestern Polytechnical University, June 2017 [Zha17].

Bibliography

- [802] *IEEE Standard for Low-Rate Wireless Networks*. Apr. 2016. DOI: [10.1109/IEEESTD.2016.7460875](https://doi.org/10.1109/IEEESTD.2016.7460875).
- [ACSM19] David Aledo, Benjamin Carrion Schafer, and Félix Moreno. "VHDL vs. SystemC: Design of Highly Parameterizable Artificial Neural Networks". In: *IEICE Transactions on Information and Systems* E102.D.3 (Mar. 2019), pp. 512–521. DOI: [10.1587/transinf.2018EDP7142](https://doi.org/10.1587/transinf.2018EDP7142).
- [AF97] Osama A. Ahmed and M.M. Fahmy. "Application of multi-layer neural networks to image compression". In: *Circuits and Systems, 1997. ISCAS '97., Proceedings of 1997 IEEE International Symposium on*. Vol. 2. June 1997, 1273–1276 vol.2. DOI: [10.1109/ISCAS.1997.622069](https://doi.org/10.1109/ISCAS.1997.622069).
- [Agn+14] Andreas Agne et al. "ReconOS: An Operating System Approach for Reconfigurable Computing". In: *IEEE Micro* 34.1 (Jan. 2014), pp. 60–71. ISSN: 0272-1732. DOI: [10.1109/MM.2013.110](https://doi.org/10.1109/MM.2013.110).
- [AM16] David Aledo and Félix Moreno. ANN. OpenCores. June 2016. URL: http://opencores.org/project,artificial_neural_network.
- [And+08] David Andrews et al. "Achieving Programming Model Abstractions for Reconfigurable Computing". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16.1 (Jan. 2008), pp. 34–44. ISSN: 1063-8210. DOI: [10.1109/TVLSI.2007.912106](https://doi.org/10.1109/TVLSI.2007.912106).
- [AO16] David Aledo Ortega. *Compresión de imágenes para redes inalámbricas. Optimización en consumo energético*. In Spanish. Editorial Academica Española, Oct. 2016. ISBN: 3841751954.
- [Asi+17] Mikhail Asiatici et al. "Virtualized Execution Runtime for FPGA Accelerators in the Cloud". In: *IEEE Access* 5 (Feb. 2017), pp. 1900–1910. DOI: [10.1109/ACCESS.2017.2661582](https://doi.org/10.1109/ACCESS.2017.2661582).
- [Bah+14] Mohammadreza Baharani et al. "High-level design space exploration of locally linear neuro-fuzzy models for embedded systems". In: *Fuzzy Sets and Systems* 253.Supplement C (2014), pp. 44 –63. ISSN: 0165-0114. DOI: [10.1016/j.fss.2013.12.006](https://doi.org/10.1016/j.fss.2013.12.006).
- [Bal+05] Philip Baldwin et al. *Visualsense: visual modeling for wireless and sensor network systems*. Tech. rep. Version 5.0. UCB ERL Memorandum UCB/ERL M05/25. July 2005.
- [BCS14] David F. Bacon, Perry Cheng, and Sunil Shukla. "Parallel Real-time Garbage Collection of Multiple Heaps in Reconfigurable Hardware". In: *Proceedings of the 2014 International Symposium on Memory Management*. Edinburgh, United Kingdom, June 2014, pp. 117–127. ISBN: 978-1-4503-2921-7. DOI: [10.1145/2602988.2602996](https://doi.org/10.1145/2602988.2602996).

- [Bet+17] M. Bettoni et al. "A Convolutional Neural Network Fully Implemented on FPGA for Embedded Platforms". In: *2017 New Generation of CAS (NGCAS)*. Sept. 2017, pp. 49–52. DOI: [10.1109/NGCAS.2017.16](https://doi.org/10.1109/NGCAS.2017.16).
- [CH05] S. Chtourou and O. Hammami. "Design space exploration of SystemC SOM implementation". In: *Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on*. Dec. 2005, pp. 159–169.
- [Che+05] Gilbert Chen et al. "Sense: A Wireless Sensor Network Simulator". In: *Advances in Pervasive Computing and Networking*. Ed. by Boleslaw K. Szymanski and Bülent Yener. Springer US, 2005, pp. 249–267. ISBN: 978-0-387-23466-3. DOI: [10.1007/0-387-23466-7_13](https://doi.org/10.1007/0-387-23466-7_13).
- [CM88] *Principal components analysis of images via back propagation*. Vol. 1001. 1988, pp. 1070–1077. DOI: [10.1117/12.969060](https://doi.org/10.1117/12.969060).
- [Coo+09] B. Cook et al. "Finding heap-bounds for hardware synthesis". In: *Formal Methods in Computer-Aided Design*. Nov. 2009, pp. 205–212. DOI: [10.1109/FMCAD.2009.5351120](https://doi.org/10.1109/FMCAD.2009.5351120).
- [CPC16] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. *An analysis of deep neural network models for practical applications*. 2016. arXiv: [1605.07678v2](https://arxiv.org/abs/1605.07678v2) [cs.CV].
- [CSAM17] Benjamin Carrion Schafer, David Aledo, and Félix Moreno. "Application Specific Behavioral Synthesis Design Space Exploration: Artificial Neural Networks. A Case Study". In: *2017 Euromicro Conference on Digital System Design (DSD)*. Aug. 2017, pp. 129–136. DOI: [10.1109/DSD.2017.56](https://doi.org/10.1109/DSD.2017.56).
- [CSM14] Benjamin Carrion Schafer and Anushree Mahapatra. "S2CBench: Synthesizable SystemC Benchmark Suite for High-Level Synthesis". In: *IEEE Embedded Systems Letters* 6.3 (Sept. 2014), pp. 53–56. ISSN: 1943-0663. DOI: [10.1109/LES.2014.2320556](https://doi.org/10.1109/LES.2014.2320556).
- [CSTW09] B. Carrion Schafer, T. Takenaka, and K. Wakabayashi. "Adaptive simulated annealer for high level synthesis design space exploration". In: *VLSI-DAT*. 2009, pp. 106–109.
- [CSW13] B. Carrion Schafer and K. Wakabayashi. "Machine learning high level synthesis design space exploration". In: *IET Computers & Digital Techniques* 6.3 (2013), pp. 153–159.
- [Dan] Al Danial. *Count Lines of Code (cloc)*. URL: <https://github.com/AIDanial/cloc>.
- [Den+09] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2009, pp. 248–255. DOI: [10.1109/CVPR.2009.5206848](https://doi.org/10.1109/CVPR.2009.5206848).
- [Des+14] Ghada Dessouky et al. "Adaptive Dynamic On-chip Memory Management for FPGA-based reconfigurable architectures". In: *24th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2014, pp. 1–8. DOI: [10.1109/FPL.2014.6927471](https://doi.org/10.1109/FPL.2014.6927471).
- [DH95] R.D. Dony and Simon Haykin. "Neural network approaches to image compression". In: *Proceedings of the IEEE* 83.2 (Feb. 1995), pp. 288–303. ISSN: 0018-9219. DOI: [10.1109/5.364461](https://doi.org/10.1109/5.364461).

- [Dia+15] Dionysios Diamantopoulos et al. "Dynamic Memory Management in Vivado-HLS for Scalable Many-Accelerator Architectures". In: *Applied Reconfigurable Computing*. Ed. by Kentaro Sano et al. Cham: Springer International Publishing, Mar. 2015, pp. 117–128. ISBN: 978-3-319-16214-0. DOI: [10.1007/978-3-319-16214-0_10](https://doi.org/10.1007/978-3-319-16214-0_10).
- [Eij+14] M. van Eijk et al. "ESL design of customizable real-time neuron networks". In: *2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) Proceedings*. Oct. 2014, pp. 671–674. DOI: [10.1109/BioCAS.2014.6981815](https://doi.org/10.1109/BioCAS.2014.6981815).
- [Erh+10] Dumitru Erhan et al. "Why does unsupervised pre-training help deep learning?" In: *Journal of Machine Learning Research* 11 (Feb. 2010), pp. 625–660.
- [Etra] ETRX2 and ETRX2HR ZigBee® modules. Product Manual 1.10. Telegesis. 2016. URL: <https://www.silabs.com/documents/public/application-notes/TG-ETRX2-PM-001-110.pdf>.
- [Etrb] TG-APP-ETRX2Power-100. Application Note. Telegesis, 2007.
- [Far+07] N. Farrugia et al. "A parallel face detection system implemented on FPGA". In: *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*. May 2007, pp. 3704–3707.
- [Fle+14] Kermin Fleming et al. "The LEAP FPGA operating system". In: *24th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2014, pp. 1–8. DOI: [10.1109/FPL.2014.6927488](https://doi.org/10.1109/FPL.2014.6927488).
- [Fuk80] Kunihiro Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36.4 (Apr. 1980), pp. 193–202. ISSN: 1432-0770. DOI: [10.1007/BF00344251](https://doi.org/10.1007/BF00344251).
- [FV09] K. Fall and K. Varadhan, eds. *The ns Manual*. 2009. URL: http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf.
- [GA04] Voicu Groza and Rami Abielmona. "What next? A hardware operating system?" In: *Proceedings of the 21st IEEE Instrumentation and Measurement Technology Conference*. Vol. 2. May 2004, 1496–1501 Vol.2. DOI: [10.1109/IMTC.2004.1351349](https://doi.org/10.1109/IMTC.2004.1351349).
- [Gaj+09] Daniel D Gajski et al. *Embedded system design: modeling, synthesis and verification*. Springer Science & Business Media, 2009.
- [Gua+17] Y. Guan et al. "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates". In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 2017, pp. 152–159. DOI: [10.1109/FCCM.2017.25](https://doi.org/10.1109/FCCM.2017.25).
- [GUZ11] A. Gomperts, A. Ukil, and F. Zurfluh. "Development and Implementation of Parameterized FPGA-Based General Purpose Neural Networks for Online Applications". In: *IEEE Transactions on Industrial Informatics* 7.1 (Feb. 2011), pp. 78–89. ISSN: 1551-3203. DOI: [10.1109/TII.2010.2085006](https://doi.org/10.1109/TII.2010.2085006).
- [HDB96] Martin T. Hagan, Howard B. Demuth, and Mark Beale. *Neural Network Design*. Boston, MA, USA: PWS Publishing Co., 1996. ISBN: 0-534-94332-2.

- [He+16] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 770–778. DOI: [10.1109/CVPR.2016.90](https://doi.org/10.1109/CVPR.2016.90).
- [Hin07] Geoffrey E. Hinton. "Learning multiple layers of representation". In: *Trends in Cognitive Sciences* 11.10 (2007), pp. 428–434. ISSN: 1364-6613. DOI: [10.1016/j.tics.2007.09.004](https://doi.org/10.1016/j.tics.2007.09.004).
- [HKR07] M. Holzer, B. Knerr, and M. Rupp. "Design space exploration with evolutionary multi-objective optimisation". In: *Industrial Embedded Systems, 2007. SIES '07. International Symposium on*. July 2007, pp. 126–133.
- [Hoc91] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen". PhD thesis. Institut für informatik, Technische Universität München, 1991.
- [HOT06] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. "A Fast Learning Algorithm for Deep Belief Nets". In: *Neural Computation* 18.7 (2006). PMID: 16764513, pp. 1527–1554. DOI: [10.1162/neco.2006.18.7.1527](https://doi.org/10.1162/neco.2006.18.7.1527).
- [HR17] M. K. Hamdan and D.T. Rover. "VHDL Generator for a high performance convolutional neural network FPGA-based accelerator". In: *ReConFig*. Dec. 2017, pp. 1–7.
- [Hri12] James Hrica. *Floating-point design with Vivado HLS*. Application note XAPP599. Xilinx, Sept. 2012.
- [HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: [10.1016/0893-6080\(89\)90020-8](https://doi.org/10.1016/0893-6080(89)90020-8).
- [KK06] V. Krishnan and S. Katkoori. "A genetic algorithm for the design space exploration of datapaths during high-level synthesis". In: *Evolutionary Computation, IEEE Transactions on* 10.3 (June 2006), pp. 213–229.
- [KKP99] J. M. Kahn, R. H. Katz, and K. S. J. Pister. "Next Century Challenges: Mobile Networking for "Smart Dust"". In: *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*. MobiCom '99. Seattle, Washington, USA: ACM, 1999, pp. 271–278. ISBN: 1-58113-142-9. DOI: [10.1145/313451.313558](https://doi.org/10.1145/313451.313558).
- [Kra91] Mark A. Kramer. "Nonlinear principal component analysis using autoassociative neural networks". In: *AIChE Journal* 37.2 (1991), pp. 233–243. ISSN: 1547-5905. DOI: [10.1002/aic.690370209](https://doi.org/10.1002/aic.690370209).
- [Kre17] Edward J. Kreinar. "RFNoC Neural Network Library using Vivado HLS". In: *Proceedings of the GNU Radio Conference* 2.1 (2017), p. 7.
- [KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems* 25. 2012, pp. 1097–1105. URL: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.
- [LC13] Hung-Yi Liu and L.P. Carloni. "On learning-based methods for design-space exploration with High-Level Synthesis". In: *DAC*. 2013, pp. 1–7.
- [Le+12] Quoc Le et al. "Building high-level features using large scale unsupervised learning". In: *International Conference in Machine Learning*. 2012.

- [LeC+89] Yann LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: [10.1162/neco.1989.1.4.541](https://doi.org/10.1162/neco.1989.1.4.541).
- [LeC+98] Y. LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 0018-9219. DOI: [10.1109/5.726791](https://doi.org/10.1109/5.726791).
- [LeC89] Yann LeCun. *Generalization and Network Design Strategies*. Tech. rep. CRG-TR-89-4. Department of Computer Science University of Toronto, June 1989.
- [Let+04] Djones Lettnin et al. "Synthesis of embedded SystemC design: a case study of digital neural networks". In: *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*. Vol. 3. IEEE Computer Society, Feb. 2004, 248–253 Vol.3. DOI: [10.1109/DATE.2004.1269239](https://doi.org/10.1109/DATE.2004.1269239).
- [Lev+03] Philip Levis et al. "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications". In: *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*. SenSys '03. Los Angeles, California, USA: ACM, Nov. 2003, pp. 126–137. ISBN: 1-58113-707-9. DOI: [10.1145/958491.958506](https://doi.org/10.1145/958491.958506).
- [LTA16] Griffin Lacey, Graham W Taylor, and Shawki Areibi. "Deep learning on fpgas: Past, present, and future". In: *arXiv preprint arXiv:1602.04283* (2016).
- [LZ17] Eduardo Lasarte Zapata. "Entrenamiento de una red neuronal hardware desde Matlab (hardware in the loop)". Spanish. Degree thesis. Feb. 2017.
- [Man] *Mannasim Framework Classes Manual*. The Manna Research Group. Feb. 2006. URL: <http://www.mannasim.dcc.ufmg.br/download/mannasim-classesmanual.pdf>.
- [MP43] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 1522-9602. DOI: [10.1007/BF02478259](https://doi.org/10.1007/BF02478259).
- [MP69] Marvin Minsky and Seymour Papert. *Perceptrons: an introduction to computational geometry*. MIT press, 1969.
- [MS09] G. Martin and G. Smith. "High-Level Synthesis: Past, Present, and Future". In: *IEEE Design Test of Computers* 26.4 (July 2009), pp. 18–25. ISSN: 0740-7475. DOI: [10.1109/MDT.2009.83](https://doi.org/10.1109/MDT.2009.83).
- [MS10] Janardan Misra and Indranil Saha. "Artificial neural networks in hardware: A survey of two decades of progress". In: *Neurocomputing* 74.1–3 (2010). Artificial Brains, pp. 239–255. ISSN: 0925-2312. DOI: [10.1016/j.neucom.2010.03.021](https://doi.org/10.1016/j.neucom.2010.03.021).
- [Muj+12] Gabriel Mujica et al. "Hardware-software integration platform for a WSN testbed based on cookies nodes". In: *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*. Oct. 2012, pp. 6013–6018. DOI: [10.1109/IECON.2012.6389099](https://doi.org/10.1109/IECON.2012.6389099).
- [NCA96] A. Namphol, S.H. Chin, and M. Arozullah. "Image compression with a hierarchical neural network". In: *Aerospace and Electronic Systems, IEEE Transactions on* 32.1 (Jan. 1996), pp. 326–338. ISSN: 0018-9251. DOI: [10.1109/7.481272](https://doi.org/10.1109/7.481272).

- [NEC14] NEC. *CyberWorkBench*. Tech. rep. 2014.
- [NTV91] M. Nigri, P. Treleaven, and M. Vellasco. "Silicon compilation of neural networks". In: *CompEuro '91. Advanced Computer Technology, Reliable Systems and Applications. 5th Annual European Computer Conference. Proceedings*. May 1991, pp. 541–546.
- [OR06] Amos R Omondi and Jagath Chandana Rajapakse, eds. *FPGA implementations of neural networks*. Springer, 2006. DOI: [10.1007/0-387-28487-7](https://doi.org/10.1007/0-387-28487-7).
- [Ort+06] E.M. Ortigosa et al. "Hardware description of multi-layer perceptrons with different abstraction levels". In: *Microprocessors and Microsystems* 30.7 (2006), pp. 435–444.
- [Pag+17] Marco Pagani et al. "A Linux-based support for developing real-time applications on heterogeneous platforms with dynamic FPGA reconfiguration". In: *30th IEEE International System-on-Chip Conference (SOCC)*. Sept. 2017, pp. 96–101. DOI: [10.1109/SOCC.2017.8226015](https://doi.org/10.1109/SOCC.2017.8226015).
- [PK00] G. J. Pottie and W. J. Kaiser. "Wireless Integrated Network Sensors". In: *Commun. ACM* 43.5 (May 2000), pp. 51–58. ISSN: 0001-0782. DOI: [10.1145/332833.332838](https://doi.org/10.1145/332833.332838).
- [Por+06] Jorge Portilla et al. "A Modular Architecture for Nodes in Wireless Sensor Networks". In: *Journal of Universal Computer Science* 12.3 (Mar. 28, 2006), pp. 328–339. DOI: [10.3217/jucs-012-03-0328](https://doi.org/10.3217/jucs-012-03-0328).
- [Poz+05] Laura Pozzi et al. "Enabling unrestricted automated synthesis of portable hardware accelerators for virtual machines". In: *Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*. Sept. 2005, pp. 243–248. DOI: [10.1145/1084834.1084896](https://doi.org/10.1145/1084834.1084896).
- [PSC05] J. Polastre, R. Szewczyk, and D. Culler. "Telos: enabling ultra-low power wireless research". In: *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks*, 2005. Apr. 2005, pp. 364–369. DOI: [10.1109/IPSN.2005.1440950](https://doi.org/10.1109/IPSN.2005.1440950).
- [PÁ17] José María Pardo Álvarez. "Aplicación de algoritmos de redes neuronales distribuidas en la clasificación de números manuscritos". Spanish. Degree thesis. Feb. 2017.
- [Pér+14] David Pérez et al. "Low-cost radar-based target identification prototype using an expert system". In: *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. July 2014, pp. 54–59. DOI: [10.1109/INDIN.2014.6945483](https://doi.org/10.1109/INDIN.2014.6945483).
- [R3x] *ETRX2 and ETRX357 wireless mesh networking modules R3xx firmware*. User Guide 2.00. Telegesis. 2016. URL: <https://www.silabs.com/documents/public/user-guides/TG-ETRXn-UG-0512.pdf>.
- [Rab+00] J. M. Rabaey et al. "PicoRadio supports ad hoc ultra-low power wireless networking". In: *Computer* 33.7 (July 2000), pp. 42–48. ISSN: 0018-9162. DOI: [10.1109/2.869369](https://doi.org/10.1109/2.869369).
- [Rev] *reVision Zone*. Xilinx. URL: <https://www.xilinx.com/products/design-tools/embedded-vision-zone.html>.

- [RGL09] Víctor Julián Roselló Gómez-Lobo. “Modelado para simulación de redes de sensores inalámbricas predespliegue basado en VisualSense”. MA thesis. Universidad Politécnica de Madrid, Sept. 2009. URL: <http://oa.upm.es/14295/>.
- [RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. DTIC Document, 1985.
- [Ril03] George F. Riley. “The Georgia Tech Network Simulator”. In: *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*. MoMeTools ’03. Karlsruhe, Germany: ACM, 2003, pp. 5–12. ISBN: 1-58113-748-6. DOI: [10.1145/944773.944775](https://doi.org/10.1145/944773.944775).
- [Rus+15] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision* 115.3 (Dec. 2015), pp. 211–252. ISSN: 1573-1405. DOI: [10.1007/s11263-015-0816-y](https://doi.org/10.1007/s11263-015-0816-y).
- [SB08] Hayden Kwok-Hay So and Robert Brodersen. “A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers Using BORPH”. In: *ACM Transactions Embedded Computing Systems (TECS)* 7.2 (Jan. 2008), 14:1–14:28. ISSN: 1539-9087. DOI: [10.1145/1331331.1331338](https://doi.org/10.1145/1331331.1331338).
- [Sha+16] H. Sharma et al. “From high-level deep neural models to FPGAs”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–12. DOI: [10.1109/MICRO.2016.7783720](https://doi.org/10.1109/MICRO.2016.7783720).
- [Sma+14] Georgios Smaragdos et al. “FPGA-based biophysically-meaningful modeling of olivocerebellar neurons”. In: *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. FPGA ’14. ACM, 2014, pp. 89–98.
- [SSDM00] Luc Séméria, Koichi Sato, and Giovanni De Micheli. “Resolution of dynamic memory allocation and pointers for the behavioral synthesis from C”. In: *Proceedings Design, Automation and Test in Europe Conference and Exhibition*. Mar. 2000, pp. 312–319. DOI: [10.1109/DATE.2000.840289](https://doi.org/10.1109/DATE.2000.840289).
- [Std] *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*. 2004. DOI: [10.1109/IEEESTD.2004.94802](https://doi.org/10.1109/IEEESTD.2004.94802).
- [Sud+16] Naveen Suda et al. “Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks”. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’16. Monterey, California, USA: ACM, 2016, pp. 16–25. ISBN: 978-1-4503-3856-1. DOI: [10.1145/2847263.2847276](https://doi.org/10.1145/2847263.2847276).
- [Syn] *SystemC Synthesizable Subset Version 1.4.7*. 2016.
- [Sys] *IEEE Standard for Standard SystemC Language Reference Manual*. Jan. 2012. DOI: [10.1109/IEEESTD.2012.6134619](https://doi.org/10.1109/IEEESTD.2012.6134619).
- [SZ14] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. arXiv: [1409.1556](https://arxiv.org/abs/1409.1556) [cs.CV].
- [Sze+15] Christian Szegedy et al. “Going deeper with convolutions”. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 1–9. DOI: [10.1109/CVPR.2015.7298594](https://doi.org/10.1109/CVPR.2015.7298594).

- [THGG07] Cesar Torres-Huitzil, Bernard Girau, and Adrien Gauffriau. "Hardware/software codesign for embedded implementation of neural networks". In: *Proceedings of the 3rd International Conference on Reconfigurable Computing: Architectures, Tools and Applications*. ARC'07. Mangaratiba, Brazil: Springer-Verlag, 2007, pp. 167–178. ISBN: 978-3-540-71430-9.
- [TLP05] B. L. Titzer, D. K. Lee, and J. Palsberg. "Avrora: scalable sensor network simulation with precise timing". In: *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks*, 2005. Apr. 2005, pp. 477–482. DOI: [10.1109/IPSN.2005.1440978](https://doi.org/10.1109/IPSN.2005.1440978).
- [TT09] Carlos Melchor Terleira Trazábal. "Integración del algoritmo backpropagation en un procesador hardware embebido para entrenamiento online". Degree thesis. Apr. 2009.
- [VAG08] Ricardo Villalobos, Rami Abielmona, and Voicu Groza. "A Bridging Layer for Run-Time Reconfigurable Hardware Operating Systems". In: *IEEE Instrumentation and Measurement Technology Conference*. May 2008, pp. 624–629. DOI: [10.1109/IMTC.2008.4547112](https://doi.org/10.1109/IMTC.2008.4547112).
- [VB16] S. I. Venieris and C. S. Bouganis. "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs". In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. May 2016, pp. 40–47. DOI: [10.1109/FCCM.2016.22](https://doi.org/10.1109/FCCM.2016.22).
- [VC13] David Van Canpenhout. "Supporting dynamic aspects of polymorphism in high-level synthesis of integrated circuit designs". US Grant US8458630B1. June 2013.
- [Vec] *Vectorization*. MathWorks. URL: https://es.mathworks.com/help/matlab/matlab_prog/vectorization.html.
- [VF14] Kizheppatt Vipin and Suhaib A. Fahmy. "DyRACT: A partial reconfiguration enabled accelerator and test platform". In: *24th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2014, pp. 1–7. DOI: [10.1109/FPL.2014.6927507](https://doi.org/10.1109/FPL.2014.6927507).
- [Vhd] *IEEE Standard VHDL Language Reference Manual*. Jan. 2009. DOI: [10.1109/IEEESTD.2009.4772740](https://doi.org/10.1109/IEEESTD.2009.4772740).
- [Vil+18] Mónica Villaverde et al. "A comparison of adaptive and non-adaptive ensemble methods for classification systems". In: *17th Mexican International Conference on Artificial Intelligence (MICAI 2018), In Press for the MICAI proceedings on the Polytechnic Open Library International Bulletin of Information Technology and Science (POLIBITS)*. Guadalajara, Mexico, Oct. 2018.
- [Vil+19] Mónica Villaverde et al. "On the adaptability of ensemble methods for distributed classification systems: A comparative analysis". In: *International Journal of Distributed Sensor Networks* 15.7 (July 2019). DOI: [10.1177/1550147719865505](https://doi.org/10.1177/1550147719865505).
- [Vilfi] Mónica Villaverde. "Cooperative Learning Model based on Multi-Agent Architecture for Embedded Intelligent Systems". PhD thesis. Universidad Politécnica de Madrid, unfinished.

- [Vin+10] Pascal Vincent et al. "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion". In: *Journal of Machine Learning Research* 11 (Dec. 2010), pp. 3371–3408.
- [VM15] Darío Viñarás Molinero. "Estudio de la arquitectura de redes neuronales artificiales para compresión de imágenes". Spanish. Degree thesis. Feb. 2015.
- [VP17] Charalempos Vatsolakis and Dionisios Pnevmatikatos. "RACOS: Transparent access and virtualization of reconfigurable hardware accelerators". In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. July 2017, pp. 11–19. DOI: [10.1109/SAMOS.2017.8344606](https://doi.org/10.1109/SAMOS.2017.8344606).
- [WBC16] Felix J. Winterstein, Samuel R. Bayliss, and George A. Constantinides. "Separation Logic for High-Level Synthesis". In: *ACM Transaction on Reconfigurable Technology and Systems* 9.2 (Feb. 2016), 10:1–10:23. ISSN: 1936-7406. DOI: [10.1145/2836169](https://doi.org/10.1145/2836169). URL: <http://doi.acm.org/10.1145/2836169>.
- [WK02] Grant Brian Wigley and David Andrew Kearney. "Research issues in operating systems for reconfigurable computing". In: *Proceedings of the International Conference on Engineering of Reconfigurable System and Algorithms (ERSA)*. June 2002, pp. 10–16.
- [XCCS17] Siyuan Xu, Jianqi Chen, and Benjamin Carrion Schafer. "HW/SW co-design experimental framework using configurable SoCs". In: *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Dec. 2017, pp. 1–6. DOI: [10.1109/RECONFIG.2017.8279801](https://doi.org/10.1109/RECONFIG.2017.8279801).
- [Xfo] Too sparse RAM. Xilinx community forums. 2016. URL: <https://forums.xilinx.com/t5/Synthesis/Too-sparse-RAM/td-p/709486>.
- [ZCP17] H. Zeng, C. Zhang, and V. Prasanna. "Fast generation of high throughput customized deep learning accelerators using FPGAs". In: *ReConFig*. Dec. 2017, pp. 1–7.
- [Zha+16] Ritchie Zhao et al. "Improving High-level Synthesis with Decoupled Data Structure Optimization". In: *Proceedings of the 53rd Annual Design Automation Conference (DAC)*. Austin, Texas, June 2016, pp. 1–6. ISBN: 978-1-4503-4236-0. DOI: [10.1145/2897937.2898030](https://doi.org/10.1145/2897937.2898030).
- [Zha17] Yi Zhang. "Optimization of Multi-Parameters Deep Learning Neural Networks". Degree thesis. June 2017.
- [Zig] ZigBee specification. ZigBee Alliance, Aug. 2015. URL: <http://www.zigbee.org>.
- [ZS03] J. Zhu and P. Sutton. "FPGA implementations of neural networks: A survey of a decade of progress". In: *Field Programmable Logic and Application*. Vol. 2778. Springer Berlin Heidelberg, 2003, pp. 1062–1066.
- [Ahm+05] Ahmed Sobeih et al. "J-Sim: a simulation environment for wireless sensor networks". In: *38th Annual Simulation Symposium*. Apr. 2005, pp. 175–187. DOI: [10.1109/ANSS.2005.27](https://doi.org/10.1109/ANSS.2005.27).
- [Fek+07] S. P. Fekete et al. "Shawn: The fast, highly customizable sensor network simulator". In: *2007 Fourth International Conference on Networked Sensing Systems*. June 2007, pp. 299–299. DOI: [10.1109/INSS.2007.4297441](https://doi.org/10.1109/INSS.2007.4297441).

- [Oul+05] E. Ould-Ahmed-Vall et al. "Simulation of large-scale sensor networks using GTSNetS". In: *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Sept. 2005, pp. 211–218. DOI: [10.1109/MASCOTS.2005.65](https://doi.org/10.1109/MASCOTS.2005.65).
- [Pol+04] J. Polley et al. "ATEMU: a fine-grained sensor network simulator". In: *2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004*. Oct. 2004, pp. 145–152. DOI: [10.1109/SAHCN.2004.1381912](https://doi.org/10.1109/SAHCN.2004.1381912).
- [The15a] The Hong Kong Polytechnic University, DARClab. *AS2CBench v.1.0*. 2015. URL: <http://sourceforge.net/projects/as2cbench/>.
- [The15b] The Hong Kong Polytechnic University, DARClab. *S2CBench v.1.1*. 2015. URL: <http://sourceforge.net/projects/s2cbench/>.
- [Yan] Yann LeCun, Corinna Cortes, Christopher J.C. Burges. *The MNIST database of handwritten digits*. URL: <http://yann.lecun.com/exdb/mnist/index.html>.