

TESE DE DOUTORADO Nº 322

**RESILIENT TRAINING OF NEURAL NETWORK CLASSIFIERS WITH
APPROXIMATE COMPUTING TECHNIQUES FOR A
HARDWARE-OPTIMIZED IMPLEMENTATION**

Vitor Angelo Maria Ferreira Torres

DATA DA DEFESA: 16/12/2019

Universidade Federal de Minas Gerais

Escola de Engenharia

Programa de Pós-Graduação em Engenharia Elétrica

**RESILIENT TRAINING OF NEURAL NETWORK CLASSIFIERS
WITH APPROXIMATE COMPUTING TECHNIQUES FOR A
HARDWARE-OPTIMIZED IMPLEMENTATION**

Vitor Angelo Maria Ferreira Torres

Tese de Doutorado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito para obtenção do Título de Doutor em Engenharia Elétrica.

Orientador: Prof. Frank Sill Torres

Belo Horizonte - MG

Dezembro de 2019

T693r

Torres, Vitor Angelo Maria Ferreira.

Resilient training of neural network classifiers with approximate computing techniques for a hardware-optimized implementation [recurso eletrônico] /Vitor Angelo Maria Ferreira. - 2019.

1 recurso online (xiii, 134 f. : il., color.) : pdf.

Orientador: Frank Sill Torres.

Tese (doutorado) - Universidade Federal de Minas Gerais, Escola de Engenharia.

Apêndices: f.109-121.

Bibliografia: f.99-108.

Exigências do sistema: Adobe Acrobat Reader.

1. Engenharia Elétrica - Teses. 2. Redes neurais (Computação) - Teses. 3. Computação aproximativa – Teses. I. Torres, Frank Sill. II. Universidade Federal de Minas Gerais. Escola de Engenharia. III. Título.

CDU: 621.3(043)

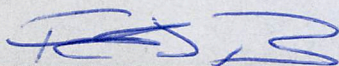
**"Resilient Training of Neural Network Classifiers with
Approximate Computing Techniques for a Hardware-optimized
Implementation"**

Vitor Angelo Maria Ferreira Torres

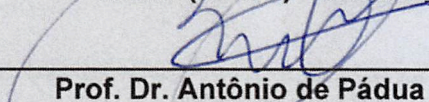
Tese de Doutorado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito para obtenção do grau de Doutor em Engenharia Elétrica.

Aprovada em 16 de dezembro de 2019.

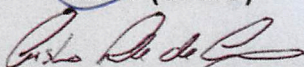
Por:



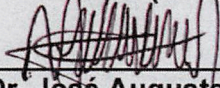
Prof. Dr. Frank Sill Torres
DELT (UFMG) - Orientador



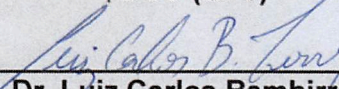
Prof. Dr. Antônio de Pádua Braga
DELT (UFMG)



Prof. Dr. Cristiano Leite de Castro
DEE (UFMG)



Prof. Dr. José Augusto Miranda Nacif
DCC (UFV)



Prof. Dr. Luiz Carlos Bambirra Torres
DCS (UFOP)

Vitor A. M. Ferreira Torres

**Resilient Training of Neural Network Classifiers
with Approximate Computing Techniques for
a Hardware-optimized Implementation**

Brazil

December 2019

Vitor A. M. Ferreira Torres

**Resilient Training of Neural Network Classifiers
with Approximate Computing Techniques for
a Hardware-optimized Implementation**

Text revised according to the comments received from the Evaluation Committee designated by the Graduate Program in Electrical Engineering as a partial requirement for the Electrical Engineering PhD Degree (Microelectronics and Microsystems).

Universidade Federal de Minas Gerais – UFMG

Graduate Program in Electrical Engineering

Supervisor: Frank Sill Torres

Brazil

December 2019

*This thesis is dedicated to all of those, my family members and dear friends,
whose full support and trust were crucial to help with all the difficult decisions
along the way which resulted in the contributions proposed with this work.*

Acknowledgements

I'd like to thank my brother Luiz and godfather Paulo who were the first ones to understand the reasons behind the decisions which lead to this work and provide their immediate, full and unconditional support for this undertaking.

Gratitude also goes to my mother Ana and all the other teachers in my wonderful family (Julita, Angela, Fernando, Luiza and Mônica) for passing to me this beautiful passion for education, firstly as a motivated student, later as a curious self-taught technician who became an Engineer and finally as an Educator.

My dear friends Leonardo Torres and Leandro Collares should also know how important all the long, frequent, sincere and open conversations were to help me keep the confidence to pursue this project.

My supervisor Prof. Frank Sill Torres provided the first spark which started this journey, by organizing the seminars which caught my attention to the interesting open problems in the reliability field. His patience in the years that followed, giving me directions and providing careful reviews were also crucial to allow my focused attention to the most important issues.

The professors at the CEFET-MG Electrical Engineering Graduation Course were also very understanding colleagues, which gave me full support while I had to share my time between my duties as a teacher and researcher. After this phase, the CNPq financial support was also very important.

Finally, I'd like to thank the free software community for decades of excellent quality tools and a development process which values openness and early collaboration as fundamental and successful attitudes.

“Although this may seem a paradox, all exact science is dominated by the idea of approximation. When a man tells you that he knows the exact truth about anything, you are safe in inferring that he is an inexact man.”
(Bertrand Russell in “The Scientific Outlook”, 1931)

Abstract

As Machine Learning applications drastically increase their demand for optimized implementations, both in embedded environments and in high-end parallel processing platforms, the industry and research community have been responding with different approaches to provide the required solutions. This work presents approximations to arithmetic operations and mathematical functions that, associated with adaptive Artificial Neural Networks training methods and an automatic precision adjustment mechanism, provide reliable and efficient implementations of classifiers, without depending on mixed operations with higher precision or complex rounding methods that are commonly proposed only with highly redundant datasets and large networks. This research investigates Approximate Computing concepts that simplify the design of classifier training accelerators based on hardware with Application Specific Integrated Circuits or Field Programmable Gate Arrays (FPGAs). The goal was not to find the optimal simplifications for each problem but to build a method, based on currently available technology, that can be used as reliably as one implemented with precise operations and standard training algorithms. Reducing the number of *bits* in the Floating Point (FP) format from 32 to 16 has an immediate effect of dividing by half the memory requirements and is a commonly used technique. By not using mixed precision and performing further simplifications to the smaller format, this thesis reduces the implementation complexity of the FP software emulation by $\approx 53\%$. Exponentiation and division by square root operations are also simplified, without requiring Look-Up Tables and with implicit interpolation. A preliminary migration of the design to an FPGA has confirmed that the area optimizations are also relevant in this environment, even when compared to other optimized implementation which lack the mechanism to adapt the FP representation range. A logical resource reduction of $\approx 64\%$ is achieved when compared to mixed-precision approaches.

Keywords: Approximate Computing. Artificial Neural Networks. Hardware Implementation.

Resumo

À medida em que aplicações de Aprendizado de Máquinas aumentam drasticamente sua demanda por implementações otimizadas, tanto em ambientes embarcadas quanto em plataformas de processamento paralelo de alto desempenho, a indústria e a comunidade de pesquisa têm respondido com diferentes propostas para prover as soluções requeridas. Esse trabalho apresenta aproximações em operações aritméticas e funções matemáticas que, associadas a métodos adaptativos para treinamento de Redes Neurais Artificiais e um mecanismo automático de ajuste de precisão, proporcionam implementações confiáveis e eficientes de classificadores, sem a dependência de algumas operações com maior precisão ou métodos complexos de arredondamento, que são frequentemente propostos somente com conjuntos de treinamento redundantes e grandes redes. Essa pesquisa investiga conceitos de Computação Aproximativa que simplificam o projeto de aceleradores para o treinamento de classificadores implementados em *hardware* com Circuitos Integrados de Aplicação Específica ou *Field Programmable Gate Arrays* (FPGA). O objetivo não era encontrar as simplificações ótimas para cada problema mas construir um método, baseado em tecnologia atualmente disponível, que possa ser usado de forma tão confiável quanto um implementado com operações precisas e métodos de treinamento padrão. A redução do número de *bits* no formato de Ponto Flutuante (PF) de 32 para 16 tem efeito imediato na divisão pela metade dos requisitos de memória e é uma técnica comumente usada. Por não utilizar parcialmente operações precisas e propor outras modificações no menor formato, essa tese reduz a complexidade de implementação da emulação de PF em *software* por $\approx 53\%$. Operações de exponenciação e divisão pela raiz quadrada também são simplificadas, sem requerer Look-Up Tables e com interpolação implícita. Uma migração preliminar do projeto para uma FPGA confirmou que as otimizações de área também são relevantes nesse ambiente, mesmo quando comparadas com outra implementação otimizada que não provê o mecanismo para adaptação da faixa de representação do PF. Uma redução de recursos lógicos de $\approx 64\%$ é obtida quando comparada com soluções parciais (*mixed-precision*).

Palavras-chave: Computação Aproximativa. Redes Neurais Artificiais. Implementação em Hardware.

List of Figures

Figure 1 – A simple ANN with two layers of neurons	7
Figure 2 – Graphical representation of <i>back-propagation</i>	8
Figure 3 – Three-class SVM/RBF classifier with large margins	19
Figure 4 – Number of datasets used for concept validation.	26
Figure 5 – Bit representation of the FP16 format.	31
Figure 6 – Effect of subnormal numbers on the lowest exponent	31
Figure 7 – Accumulation error due to limited precision	34
Figure 8 – Activation Functions Comparison	36
Figure 9 – Stepwise approximation of $\tanh(x)$, with interpolation.	37
Figure 10 – Training Process Comparison between half and double precision FP representations for the MNIST and Soybean datasets	43
Figure 11 – Training Process Comparison between half and double precision FP representations for the Breast Cancer and Thyroid datasets	43
Figure 12 – Average and maximum absolute values for weight adjustments	47
Figure 13 – Bit layout of the FP16 format	51
Figure 14 – FP16 exponentiation approximation	53
Figure 15 – Approximations for powers of 2	54
Figure 16 – Reciprocal Square Root Approximations	56
Figure 17 – Division by Square Root Approximation	57
Figure 18 – Division by Square Root Approximation: 2 Newton-Raphson steps . . .	57
Figure 19 – Training Process Comparison between standard IEEE and approximate FP16 representations for the MNIST and Soybean datasets	59
Figure 20 – Training Process Comparison between standard IEEE and approximate FP16 representations for the Breast Cancer and Thyroid datasets . . .	60
Figure 21 – Breast Cancer without outliers: comparison of the effect of the momen- tum term using a double precision FP64 as baseline	63
Figure 22 – Thyroid in full-batch mode: comparison of the effect of the momentum term using a double precision FP64 as baseline	64
Figure 23 – MNIST dataset: comparing the RMSProp adaptation implemented with FP16 approximations and a double precision FP64 as baseline (positive values on the right mean better “ approx. ” performance)	67
Figure 24 – Modified RMSProp training with the approximate FP16 using different exponent bias values for variable in Group 3	69
Figure 25 – FP bias distribution among neurons (end of training)	72
Figure 26 – FP bias: feed-forward and back-propagation	72
Figure 27 – Comparisons between different FP formats	73

Figure 28 – Final performance comparisons with the MNIST Dataset	74
Figure 29 – Final performance comparisons with the Breast Cancer Dataset	74
Figure 30 – Final performance comparisons with the Gene Dataset	75
Figure 31 – Final performance comparisons with the Thyroid Dataset	75
Figure 32 – Final performance comparisons with the Soybean Dataset	76
Figure 33 – Performance influence of the Dynamic bias method	77
Figure 34 – Performance influence of the Dynamic bias method	77
Figure 35 – Performance influence of the Dynamic bias method	77
Figure 36 – Performance influence of the Dynamic bias method	78
Figure 37 – All positive values representable in the posit16 format	78
Figure 38 – Internal fields in the generic Posit format with fixed total <i>bits</i> . Source: (GUSTAFSON; YONEMOTO, 2017), unchanged	79
Figure 39 – Relative increase in training times (compared to the fp32fp32 reference). .	82
Figure 40 – Soybean accuracy in ARM Cortex A53.	83
Figure 41 – Zynq subsystem block diagram with two memory blocks.	89
Figure 42 – LUTs and FFs allocated for each memory block added to the bus. Each separate AXI slave memory block could implement an ANN layer. . . .	89
Figure 43 – Inputs and outputs of the input layer and the first hidden one.	90
Figure 44 – Inputs and outputs of the second hidden layer and the output one. . .	91
Figure 45 – Performance comparisons with the Abalone Dataset	111
Figure 46 – Performance comparisons with the Abalone (18-9 variation) Dataset . .	111
Figure 47 – Performance comparisons with the Breast Cancer Dataset	112
Figure 48 – Performance comparisons with the Car Dataset	112
Figure 49 – Performance comparisons with the Diabetes Dataset	112
Figure 50 – Performance comparisons with the Euthyroid Dataset	113
Figure 51 – Performance comparisons with the German Dataset	113
Figure 52 – Performance comparisons with the Glass Dataset	113
Figure 53 – Performance comparisons with the Heart Dataset	114
Figure 54 – Performance comparisons with the Ionosphere Dataset	114
Figure 55 – Performance comparisons with the Satimage Dataset	114
Figure 56 – Performance comparisons with the Segmentation Dataset	115
Figure 57 – Performance comparisons with the Vehicle Dataset	115
Figure 58 – Performance comparisons with the Vowel Dataset	115
Figure 59 – Performance comparisons with the Yeast Dataset	116
Figure 60 – Performance comparisons with the Yeast (9-1 variation) Dataset	116

List of Tables

Table 1 – Approximation Type	28
Table 2 – Approximation Method	29
Table 3 – Characteristics of the benchmark datasets	42
Table 4 – FPGA resource comparison after synthesis using the Vivado HLS	50
Table 5 – High Level Synthesis of Complex Functions	58
Table 6 – Different ranges provided by larger exponent biases	68
Table 7 – Training times (in seconds, with 99% confidence intervals) in the Broad- com BCM2837 SoC. The mixed precision approach is presented in fp32fp16 , while in the others all variables use the same representa- tions (FP16 or FP32)	82
Table 8 – Memory usage reduction (from fp32fp32 to fp32fp16) and dataset sizes	83
Table 9 – Comparison of FPGA resources for the VFloat based FPU with two- operations (multiplication and addition) the respective approximated implementations.	86
Table 10 – Comparison of FPGA resouces after implementation	91
Table 11 – Comparison of FPGA resouces after implementation	91

List of abbreviations and acronyms

AC	Approximate Computing
ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
ASIC	Application Specific Integrated Circuits
CPU	Central Processing Unit
CNN	Convolutional Neural Network
DNN	Deep Neural Network
DPP	Direct Parallel Perceptrons
DSP	Digital Signal Processor
ELU	Exponential Linear Units
FMA	Fused Multiply Add
FMAC	Fused Multiply Accumulate
FF	Flip-Flop
FP	Floating Point
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GCC	GNU Compiler Collection
GD	Gradient Descent
GPU	Graphics Processing Unit
HLD	High Level Design
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things

LDA	Linear Discriminant Analysis
LLSE	Linear Least Squares Estimation
LNS	Logarithmic Number System
LSB	Least Significant Bit (or Byte)
LUT	Look-Up Table
MAC	Multiply and Accumulate
ML	Machine Learning
MLP	Multi Layer Perceptron
MSB	Most Significant Bit (or Byte)
NaN	Not a Number
PE	Processing Element
RAM	Random Access Memory
RBF	Radial Basis Function
ReLU	Rectified Linear Unit
RLS	Recursive Least Squares
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SIMD	Single Instruction Multiple Data
SVM	Support Vector Machine
TNR	True Negative Ratio
TPM	Tensor Processing Unit
TPR	True Positive Ratio
VLSI	Very Large Scale Integration
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language

Contents

1	INTRODUCTION	1
2	HISTORICAL BACKGROUND	6
2.1	Introduction to Artificial Neural Networks	6
2.2	Digital Hardware Implementations	9
2.3	Analog and Neuromorphic Implementations	10
3	RELATED WORK	12
3.1	Neural Networks	13
3.2	Support Vector Machines	19
3.3	Generic Approximate Computing	22
3.4	Comparisons and Limitations of Current Techniques	25
4	ANN IMPLEMENTATIONS	30
4.1	Floating Point Review	30
4.1.1	Standard Floating Representation	30
4.1.2	Standard Floating Point Operations	32
4.2	Mathematical Operations	33
4.2.1	Small Values Accumulation	33
4.2.2	Deep Learning	35
4.2.3	Activation Functions	36
4.3	Baseline Tests with standard IEEE FP (64 vs 16 <i>bits</i>)	38
5	SIMPLIFICATIONS AND APPROXIMATIONS	45
5.1	Floating Point Simplifications	45
5.1.1	Removal of Infinities and NaNs	45
5.1.2	Restrict Format to Normalized Numbers	46
5.1.3	Preliminary Implementation of FP arithmetic	48
5.2	Math Operations Approximations	50
5.2.1	Exponentiation and Activation Functions	51
5.2.2	Reciprocal Square Root	52
5.2.3	Division by Square Root	56
5.2.4	Preliminary Implementation of Complex Functions	58
5.3	Comparative Tests: Standard IEEE FP16 vs Approximated	58
6	RESILIENT TRAINING OF ANNS	61
6.1	Adaptive Training Mechanisms	62

6.1.1	Gradient Descent with Momentum	62
6.1.2	iRProp-	63
6.1.3	RMSPProp	65
6.2	Exploring Different FP16 Ranges	67
6.3	Dynamic FP16 Bias Adjustments	70
6.4	Experiments with Dynamic FP16 Bias	73
6.5	Experiments with Fixed FP16 Bias	76
6.6	Comparison with Posit (16 <i>bits</i>)	78
7	PRELIMINARY OPTIMIZED IMPLEMENTATIONS	81
7.1	Exponential Approximation with Hardware IEEE FP16	81
7.2	Implementation of Approximated FP16 Operations in FPGA	84
7.2.1	Preliminary Floating Point Unit	85
7.2.2	Communication between CPU and FPGA	87
7.2.3	Structure for a multi-layer ANN in FPGA	90
8	CONCLUSIONS AND FUTURE DIRECTIONS	92
8.1	Analysis of the current status	92
8.2	Future Improvements	94
8.2.1	Training Methods	94
8.2.2	Regularization	95
8.3	FPGA Implementation	96
8.4	Conclusion and Final Remarks	97
8.5	Contributions	98
	BIBLIOGRAPHY	99
	APPENDIX	109
	APPENDIX A – ADDITIONAL VALIDATION	110
	APPENDIX B – COMPARISON WITH POSIT	117

1 Introduction

Machine Learning implementations have been closely related to hardware since their early proposals. Artificial Neural Networks (ANNs), for example, were not only inspired by biological models but also presented very early in efficient hardware implementation proposals which later became known as “Neuromorphic Computing” (MEAD, 1990). Despite the high efficiency shown in these approaches, the hardware limits predicted at that time have been surpassed and the following years showed an impressive increase in computer performance. This evolution probably influenced the decrease in the interest by hardware optimized implementations. In this time period, the ML (Machine Learning) field matured and not only new methods were proposed, like SVM (Support Vector Machines) (CORTES; VAPNIK, 1995) and Random Forests (HO, 1995), but the researchers put great effort in reducing the need of parameterization, which also indirectly helped to optimize hardware implementations.

After decades of fast performance growth, digital circuits implementations began to show their limits, and the processing speed increase had to rely on other techniques than denser ICs (Integrated Circuits) and higher clock rates. The level of miniaturization and voltage scaling reached a point at which reliability became a real and practical issue (AGARWAL et al., 2000). Parallelization became more present and beneficial for many ML methods, however, this approach also has its limits (ESMAEILZADEH et al., 2011): power dissipation is a concern in many scenarios, not only in the embedded and energy constrained platforms, but also in large processing clusters or powerful GPUs (Graphics Processing Unit). The AC (Approximate Computing) paradigm - “deterministic designs that produce imprecise results” (HAN; ORSHANSKY, 2013) - appeared as a technique not only to be aware of the reliability problems but to explore them to obtain efficiency gains (AGRAWAL et al., 2016).

Concerns with unreliable hardware date back to the first computing devices (NEUMANN, 1956), and by that time the expression “Stochastic Computing” also became popular. Conversely, AC does not assume stochastic behavior of the hardware but studies how quantifiable and controllable simplifications can impact the computation precision, even if they are analyzed statistically. Applications with redundant and/or noisy inputs or with no single “correct” output (or one that is not guaranteed to be found) are usually good candidates for AC. Inherent to ML applications, these characteristics have attracted the attention of the research community interested in the trade-off between precision and efficient resource usage provided by AC techniques.

Efficiency is frequently a fundamental concern in computing methods implementa-

tions. From embedded platforms, with power, processing speed and storage limitations, to large scale clusters or tightly coupled massively parallel processing units, optimizations are often required. These two extremes (WU et al., 2011; REED; DONGARRA, 2015) in hardware architectures are highly relevant to ML: smart sensors, ubiquitous cell phones, wearables, Internet of Things (IoT), main stream Big Data applications and high performance computer vision platforms.

Embedded platforms may execute previously trained and static models, which may be optimized (pruned and compressed) before the final implementations. This is possible since the inference phase can be very resilient and can work well with noise (due to its generalization capabilities) and reduced precision. Conversely, the training process may suffer if executed on these simplified structures. Since the previously mentioned applications frequently require Incremental Learning (GEPPERTH; HAMMER, 2016), and the dependency on remote processing and storage is not always possible, there is an active interest in researching the training process under approximations in order to provide more efficient implementations.

Training process optimization is also relevant to the other extreme of hardware platforms. If Incremental Learning is not required on the final device, extensive optimizations for training are frequently neglected since an efficient process in this case is only relevant during development. Despite this, since Deep Learning (LECUN; BENGIO; HINTON, 2015) increased its popularity, the size of the models also reached unprecedented scales. Networks with dozens of layers, thousands of nodes and billions of parameters became common to solve increasingly complex problems, using vast amounts of data. Training times of days or weeks were needed even when using powerful hardware platforms like a GPU server cluster (COATES et al., 2013) or a CPU (Central Processing Unit) cluster with 1000 machines (16000 cores) (LE, 2013). High level Frameworks like “Caffe” (JIA et al., 2014): also became popular, since they usually hide the high performance back-ends programming details.

The technological barriers in IC design and the increasing commercial interest in ML (in both extremes just mentioned) probably played an important role in reviving the interest on efficient and simplified hardware implementations. Recent announcements from Qualcomm (Zeroth Processor - 2013), IBM (MEROLLA et al., 2014) (TrueNorth - 2014), NVIDIA (Tesla P100 - 2016), Google (JOUPII et al., 2017) (Tensor Processing Unit - 2016), Apple (Neural Engine - 2017) and also from Intel (Nervana and Movidius - 2017) confirm the importance of commercial implementations of such solutions. This scenario of efforts to improve efficiency of ML applications using approximated techniques or optimized hardware architectures, implemented in several levels, is the main driver of the work presented here.

Alternative binary value representation formats, like fixed point and LNS (Loga-

rithmic Number System), are being currently explored by the ML research community. A considerable part of the work consists in evaluating the effect of reduced bit-width on the performance with benchmark datasets or specific applications. These efforts are justifiable considering that simpler hardware to perform arithmetic operations allows more efficient specialized implementations. Bit-width is also highly relevant, since it has direct impact on data movement and storage. It should be noted that non-standard formats or bit-widths may represent difficulties when these customized systems are interfaced with standard ones. Additionally, compressing the representations to find the minimal requirements for a maximum acceptable degradation in specific datasets is a different paradigm when compared to the current confidence in the standard floating points reliability for any problems. Even in these cases, some simple measures are required to avoid numerical issues.

The vast majority of ANN research and application is performed at a high abstraction level, using the corresponding programming languages and software libraries. Single and double precision IEEE (Institute of Electrical and Electronics Engineers) FPs (Floating Points) are reliable enough so that researchers are able to focus on algorithmic issues, with rare and simple concerns for numerical problems. Conversely, half precision FPs require more attention and mixed precision training is becoming the norm for resource efficient and optimized implementations.

The main objective of this research is to develop a mechanism which unifies the numerical representation used in ANN training implementations in a more efficient format. The two main motivations of this approach are that it allows the solution to provide a single and simpler type of arithmetic unit, which results in even more efficient solutions, and results in lower memory usage when compared to the mixed precision approach. Additionally to the representation, this thesis also investigates further simplifications to the FP arithmetic, adapts and proposes higher level mathematical operations approximations and finally an automatic precision adjustment during the training process which allows the use of the method without extra care due to the lower precision. The goal is to perform the training with the same methods, the same network structures and hyper-parameters used in precise baselines. The approach is not to find the lowest acceptable precision, which varies for each specific application and training algorithm, but to define an efficient approximate implementation which can be used as reliably as the reference ones.

The work detailed in the following chapters explores modifications to the standard half precision FP that simplify its implementation in software (when using low-end embedded processors, without an FPU - Floating Point Unit) and in hardware (with fixed implementations or in FPGAs - Field Programmable Gate Arrays). This is achieved with only minor and optional modifications to the standard binary representation. The standard format is already being supported by some hardware platforms (e.g. NVIDIA and ARM)

and compilers (e.g. GCC) and some of the presented results also apply to the standard IEEE FP format. Approximate implementations of more complex mathematical functions are also adapted to the 16 *bits* FP format, as simple replacements for LUT (Look-Up Table) based activation function implementations. An automatic precision adjustment mechanism was crucial to avoid using custom training algorithms. Another important note is that the network configurations (number of layers, nodes and connections) used are the same minimal ones employed in other references. It is not uncommon to see larger networks in articles analyzing low precision training, which partially defeats the purpose of the simplifications.

The main contributions of this thesis are distributed in the following manner:

Chapter 3 Contributes a critical view on the recent AC research applied to ML, showing that the most aggressive approximations are focused on a small number of similar problems based on large datasets. Generic, automatic and fully approximated proposals are rarer in the literature.

Chapter 4 Details the modifications implemented in an ANN training library to transform it in a simulated and easily modifiable environment to investigate operations with approximated representations. The same library can be compiled with efficient native numeric types or with emulated modules which provide the basic operations.

Chapter 5 Contributes adaptations of mathematical functions approximations to a limited FP format and a new approximation which merges a division operation with a square root calculation. These approximations behave like stepwise simplifications with implicit interpolation and require considerable less resources than their precise counterparts.

Chapter 6 Achieves resilient training of ANNs using a standard adaptive algorithm by contributing a fully automatic and resource efficient range adjustment mechanism, applied to the back-propagation operations. Equivalent classification performance is achieved by quickly reacting to overflows but gradually shifting the representation range to fit smaller numbers.

Chapter 7 Presents two implementations for viability analyses of the proposed methods. The first one uses a general purpose processor and one of the function approximations. The other provides a preliminary comparison of FPGA operations to a reference from the literature with relevant gains when compared to the mixed-precision approaches. The scalability of a multi-layer structure with several FPUs, including the bus communication, is also analyzed.

Regarding the remaining content, Chapter 2 surveys the related research with a short historical background on ANNs and their early optimized implementations, which preceded

a period of diminished research interest in ANNs and their hardware implementations. Final remarks are outlined in Chapter 8, which reviews the main achievements and challenges of the current work and analyzes some improvement possibilities and next steps regarding the complete hardware implementation.

As appropriate and expected, this text gave preference to non-personal language, which may be sometimes not clear regarding the source of the information. Great effort was put in making the uses of expressions like “this work”, “this thesis”, “this research”, “the experiments” and “the proposed method” to clearly refer to actions performed or decisions taken by the author, which takes full and sole responsibility for them. Wherever similar constructions refer to other research, persons or actions, explicit references are made in order to avoid the confusion which normally arises in these situations. Conversely, the use of the plural in the first person was employed only where the reader is invited to join a reasoning process or in clearly marked quotes.

2 Historical Background

In order to exemplify how efficient implementations of ANNs have been long related to hardware, the following sections briefly review the fundamental concepts and some historical research on the topic. A short introduction to the common aspects of this ML technique is presented in the first section, without detailing the several architectural differences and training methods. Some examples of early hardware implementations are divided into digital and neuromorphic approaches in the two subsequent sections. Back in (TRELEAVEN; PACHECO; VELLASCO, 1989) this division was still unclear, since digital implementations appeared more promising for practical purposes. Despite that, it did not take a long time for an increase in the popularity of the neuromorphic designs. These two sections briefly cover the period which preceded a decrease in the interest for optimized ANN hardware implementations.

2.1 Introduction to Artificial Neural Networks

The biological inspiration for ANNs (ROSENBLATT, 1958; MCCULLOCH; PITTS, 1943) points to the first direction in which AC could improve the efficiency of these networks. The basic artificial neuron model consists of several inputs, each one applying a certain weight to its respective input signal, contributing to the activation of the neuron's output according to a specific function. This is clearly an intrinsically analog model and several examples in Chapter 3 will show its resilience to approximations. Figure 1 depicts a simple two layer ANN with two neurons (N_{11} and N_{12}) connected to the inputs and one (N_{21}) providing the output. Each arrow pointing to a neuron represents a connection which multiplies an input signal by a specific weight. In multi-layer topologies, the neuron outputs of the previous layer(s) usually become the inputs of the next one.

At each layer, the output o_k of the neuron k with m inputs is obtained according to the Equation 2.1, in which x_j represents the input signal j and w_{kj} its respective weight. A bias term (b_k), independent from the input signals, is included to shift the argument of the activation function φ . In practice, the bias is normally implemented as a weight which is always connected to a fixed value (e.g. 1). The complete system (with all neurons, layers and connections) represents a model which intends to correctly infer, within an acceptable error margin, one or more outputs for inputs which were not previously known.

$$o_k = \varphi \left(b_k + \sum_{j=1}^m w_{kj} x_j \right) \quad (2.1)$$

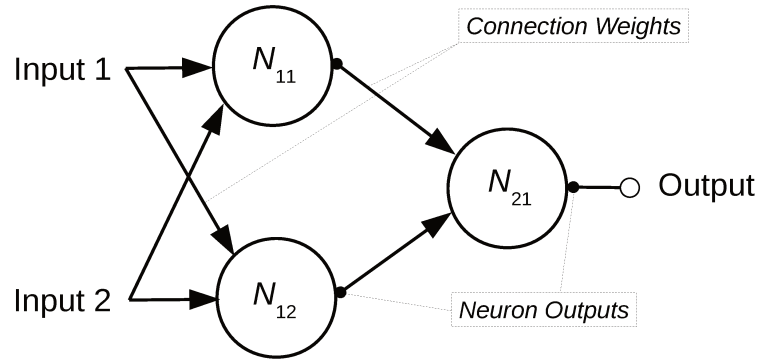


Figure 1 – A simple ANN with two layers of neurons

In order to build this model, a structure with enough capacity, which is directly related to its complexity, has to be defined and its parameters (weights and biases) have to be found. The supervised training technique, which basically consists of adjusting the input weights and bias based on known input examples with their respective outputs, had its first breakthrough in (WILLIAMS; HINTON, 1986) and a practical framework was presented in (RUMELHART et al., 1986), causing the concept of *back-propagation* to gain considerable popularity. The term refers to the fact that during the training stage, after the inputs are “forward-propagated” to the outputs, the inference error (based on a loss function ¹) is propagated backwards and the contribution of each connection to the output error determines how its weight should be adjusted.

To guide the weights and biases adjustments, the whole network is analyzed as a sequence of differentiable functions. Even considering that the activation functions φ are non linear, like the logistic function detailed in Equation 2.2, their derivatives are known, and frequently can be conveniently calculated using the actual value of the function, like exemplified in Equation 2.3.

$$\varphi(z) = \frac{1}{1 + e^{-z}} \quad (2.2)$$

$$\frac{d\varphi(z)}{dz} = \varphi(z)(1 - \varphi(z)) \quad (2.3)$$

¹ A loss function is defined to measure how far the network is from the ideal model, considering the whole training set. The training objective is to minimize the distance to this error-free situation, without losing the generalization capability.

Figure 2 illustrates this process of calculating parameter adjustments by applying the chain-rule of derivatives of every operation (sum-of-products, activation functions and loss function) from each known training example. Using the chain rule, from the output to the input, like indicated in Equation 2.4, it is possible to estimate how small changes in each specific weight w_{kj} would affect the loss function E . This is the fundamental principle of Gradient Descent (GD) optimization, detailed in Equation 2.5. This method makes progressively smaller adjustments $\Delta\theta_n$ in each parameter θ_n , by applying a small fraction (determined by η) of the gradient ∇_θ , which guides the direction and intensity of each adjustment to reduce the loss function E (hence the negative sign).

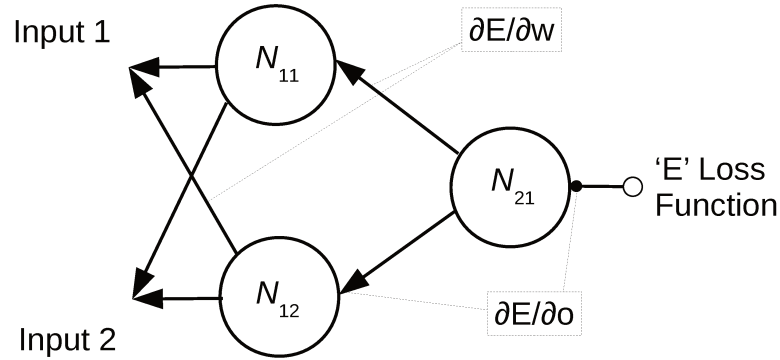


Figure 2 – Graphical representation of *back-propagation*

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial w_{kj}} \quad (2.4)$$

$$\Delta\theta_n = -\eta \nabla_\theta E(\theta_n; x; y) \quad (2.5a)$$

$$\theta_{n+1} = \theta_n + \Delta\theta_n \quad (2.5b)$$

There are several variations of the main idea behind *back-propagation*, but three common aspects are highly relevant when approximations and reduced precision are considered. Firstly, although each training example inference and its error propagation are performed separately, the weight updates may not be executed in every step. This accumulation process, which may happen for the whole dataset or partial batches, may loose contributions of small magnitudes when they are accumulated with larger values. Secondly, the theoretical background for this type of optimization (GD) usually relies on precise definitions of the activation functions and their derivatives. Approximated

versions of these equations may harm the optimization process (e.g. causing instabilities). Finally, as the training process evolves, the error magnitudes and the respective parameter adjustments tend to have very low values. This fact may cause the learning process to stop prematurely if these small adjustments are not considered with enough precision.

This work will assume familiarity with multilayer ANN topologies and their main training methods. For further information, the reader is referred to (HAYKIN, 1998) (Chapters 1 to 4) for a comprehensive explanation of ANNs architectures and training methods. An excellent historical overview of ANNs from the first implementations to the current state-of-the-art approaches is provided by (SCHMIDHUBER, 2015).

2.2 Digital Hardware Implementations

Due to their simplicity and good performance on certain applications (e.g. change detection), Weightless Neural Networks are still actively used and researched, despite their introduction more than half a century ago, in (BLEDSE; BROWNING, 1959) and later in (ALEKSANDER, 1966). Instead of the common weighted-sum approach, the nodes use a LUT based technique, normally realized with regular Random Access Memory (RAM). WISARD (ALEKSANDER; THOMAS; BOWDEN, 1984), one of the first practical examples, is completely implemented with standard digital hardware components. The training consists of simply storing “ones” on memory positions directly accessed by black-and-white image pixels, which are randomly connected to the address lines of several nodes. In the inference phase, the amount of nodes recovering the *bit* “one” from the memory gives an indication of the confidence in the image similarity. Despite its simple implementation, this method may achieve a high resilience to noise, specially if noiseless training is possible. Although not relevant in the short term, quantum computing may also boost the research interest in this type of ANN.

The ANN size, the way the nodes are connected and the definition of the activation functions are structural decisions that inform the design of such ML systems. These and other parameters vary considerably according to the application. Consequently, any problem-independent hardware implementation proposal needs to be given such flexibility. FPGAs have this capability and modern devices have enough resources to implement complete systems for several applications. The intrinsic parallelism of ANNs is also an incentive for efficient FPGAs implementations. One of the earliest attempts (BOTROS; ABDUL-AZIZ, 1994) presented a solution which required two FPGAs and an external memory to implement a single node. In this case the training phase was implemented entirely in software, but more recent examples (ÇAVUŞLU et al., 2011) provide an almost complete hardware solution. Even years before this implementation, the book (ZHU; SUTTON, 2003) surveyed successful examples corroborating the increase in FPGAs

complexity as a crucial requirement to successful ANN implementations. The recent trend towards general purpose CPUs integrated with FPGAs may also provide interesting opportunities for hybrid solutions that explore FPGA parallelism and CPU flexibility.

Even in a period when the available integration technologies did not provide enough capability to implement complex digital ANNs, several solutions were proposed. The survey (DIAS; ANTUNES; MOTA, 2004) analyzed recent hardware implementations with emphasis on the commercially available solutions, including some from large companies (e.g. Philips, Hitachi, Siemens and IBM). Approximately half the components presented in the paper were considered optimized multiprocessor chips and the vast majority could implement no more than 64 nodes. While some of the examples could not even hold the connections internally, two of them could work with 262,144 weights. Precision was also not high, since most of the products worked with 16 *bits* or less.

Both FPGA and fixed ASIC implementations of ANNs can be made denser if arithmetic units and data paths are simplified. In order to evaluate the impacts of such hardware optimizations, the study (SAVICH; MOUSSA; AREIBI, 2007) compared the performance of MLPs trained with *back-propagation* in several fixed and floating point number formats. The presented trade-off results between accuracy and resources apply only to FPGAs, since they provide internal DSP modules. Conversely, the impact of numeric precision on accuracy could be applied to any other similar form of implementation. In addition to the different representation formats, LUT based activation functions were also tested. The classic exclusive OR (XOR) problem was employed as benchmark using the standard configuration of two nodes in the hidden layer and one in the output. From the 15 fixed point formats and 10 floating point ones, from 16 to 22 *bits*, a fixed point with 4 integer and 13 fractional *bits* was considered as the optimal configuration. Using this format as reference the authors also analyzed how resource usage grows for more complex two layer networks. The amount of slices used varied fairly linearly with the number of weights.

2.3 Analog and Neuromorphic Implementations

A more direct analog modeling of the neuron behavior is an orthogonal approach for hardware ANN implementations. In (HOLLER et al., 1989), Intel presented an integrated component that offered 64 fully interconnected neurons to 64 fully connected inputs. The connections were implemented with a non-volatile and electrically modifiable memory which provided analog storage of the weights using a “Floating Gate” technology, previously proposed. Inputs and outputs were also analog, differential and with compatible levels, allowing multi-layer implementations using interconnected components. The product of inputs by their respective weights, the summation and the sigmoid activation function

were implemented in analog circuits as well. The learning process, however, was performed off-chip. The original article was very detailed but did not evaluate a final application. A few years latter, the effects of radiation exposure (CASTRO; SWEET, 1993) and component variations (CASTRO; TAM; HOLLER, 1993) were evaluated, with satisfactory results. The survey mentioned in Section 2.2 also considered the chip as the only viable analog one available at the time, comparable only with another more specialized and neuromorphic proposal.

The paper (VALLE, 2002) mentions three motivations for analog VLSI (Very Large Scale Integration) implementation of ANNs: analog processing is one of the reasons behind biological NNs computational power; the physics of the silicon devices can efficiently implement computation of analog signals; units very similar to biological neurons can be realized. The authors provide an extensive analysis of practical aspects of this type of implementations, focusing on the critical issues related to the supervised learning process. It is argued that *weight perturbation* is a more promising strategy than *back-propagation* for analog circuits and four parallel methods are presented. The work proceeds with a detailed analysis of related research leading to a conclusion that defends the feasibility and good performance of such solutions for small scale networks.

3 Related Work

After a relatively long period of diminished interest in ANNs and their hardware implementations, the research in this area regained momentum. The increase in ANN achievements was in part related to higher availability of training data and some breakthroughs (HINTON; SALAKHUTDINOV, 2006) in Deep Learning training, which demands great computing power. This higher demand kept pushing the digital hardware performance limits, which was also facing growth rate limitations and exploring new optimized techniques. The current chapter separates this recent phase from the historic proposals and presents a review which covers some of the most relevant contributions related to digital hardware implementations. The survey provided by this chapter tries to fill an apparent gap in the literature reviewing the recent trend of applying AC techniques to ML, with this form of organization and classification.

The following sections group the surveyed research initially by the two most popular ML techniques, which share many characteristics and explore the AC methodology. The third group of publications analyses approaches that are more generic (apply to more than one ML method) or are related to the mathematical methods that are commonly used in the ML field. The final section summarizes the gaps found in current research that justify the relevance of the contributions presented in the following chapters.

In order to better understand the following review and its scope, some categorization should be provided. As previously mentioned, AC explicitly excludes a related field, Stochastic Computing, which analyzes the effect of non-deterministic hardware behavior. Algorithmic optimizations in ML methods are also excluded if they do not use some type of approximation (e.g. trading of mathematical functions precision for efficiency). It should be also noted that the feature selection or extraction processes, which considerably simplify the input space, are separate research fields and are not considered as approximations in the context presented here. Also excluded from this collection of studies are the mere customizations of specific methods to certain platforms, like DSPs (Digital Signal Processors) or FPGAs without implying some explicit form of approximation.

Two of the most important aspects of AC are identifying which parts of the methods or data are good candidates for approximation and determining the compromise between the level of simplifications and quality. If this analysis is performed at design time and all approximations are defined in a fixed manner, this can be considered a *static* approach. Conversely, if either the promising method parts or the simplification levels are determined or adjusted at runtime, a *dynamic* approach is used. Specifically in ML, it is common to find small parts of the method, like kernels, which represent a large fraction of the

processing time, being good candidates for further analysis regarding the approximation consequences.

Optimizing a part of the method which is not heavily used or selecting an approximation level which compromises the output quality beyond a certain tolerable amount should be avoided. There are several examples in the literature that show methods (not specific to ML) to statically analyze a generic code or algorithm and obtain these optimization parameters. If this process is performed without the designer intervention, it is considered an *automatic* approach. This is the opposite of a *guided* optimization, which demands knowledge from the designer to decide where and how to approximate operations and data, but also obviously provides more control of the process.

Several options are present in the literature, not only related to ML, to implement the computing approximations. One of the most common approaches is the *data representation*, which includes not only less precise floating or fixed point formats but also different numerical techniques like pure integer operations or logarithmic number systems. Another option is *memory reduction* by approximation, which may not significantly affect execution time but may considerably reduce resource usage and data communication. Elementary *mathematical operations* or more *complex functions* may also be the objects of approximations trading result exactness for processing time gains. It is also worth mentioning that *simplified algorithms* may also obtain advantages by approximation, when early termination, skipped steps or simpler models, for example, are used to anticipate part of the results with less quality.

3.1 Neural Networks

Parallel Perceptrons (PPs) are a very resource efficient neural networks realization, with simple threshold activations and binary outputs. As in SVMs, the classification margin is maximized and the original training method is based on an iterative optimization scheme. “Direct Parallel Perceptrons” (DPPs) are proposed with a new algorithm in (FERNANDEZ-DELGADO et al., 2011) which analytically calculates the network weights, without any iterative search or parameter-based optimization process, and with computational complexity varying linearly with the input dimensions. This is possible due to a linear approximation of the error function. Online training is also very efficient because the method is incremental and does not have to be repeated for the entire dataset, as in SVMs. The proposal is compared to Adaboost and Bagging of MLP networks, to the original PPs, LDA and also two SVM implementations, all applied to several benchmark problems. For datasets with two classes, the average accuracy of DPPs is 2.8% lower than the best reference method but higher than 6 out of 9 competitors. Considering C/C++ implementations of 3 reference methods, DPPs are more than 400 times faster than the

fastest one.

The work presented in (COURBARIAUX; BENGIO; DAVID, 2014) evaluates three benchmark datasets with deep neural networks (specifically Maxout networks) using three different number representations: floating point, fixed point and dynamic fixed point. The authors focus on the multiplication operations, implementing the accumulators with single precision (32 *bits*) floating point. This approach results from an analysis showing that accumulator precision has low impact on FPGA implementation costs. Simple fixed point is considered harmful because activations, parameters and gradients have very different ranges. Gradients are specially critical, since they slowly diminish this range during training. With a simple dynamic adaption of fixed point to different ranges, the authors find no statistically relevant accuracy reduction with a precision down to 10 *bits*. This reduced precision is used not only for running the networks but also for the training process.

The authors of (ZHANG et al., 2015b) aim at achieving considerable energy savings in artificial neural networks by applying approximation in data representation, computation and memory accesses. The framework analyzes the impact that neurons (with any topologies) have on the output quality, sorts them according to a score, and specifically adjusts the approximations. Memory Access Skipping is implemented by simply avoiding certain non-critical neurons (by not reading their respective weights). Precision Scaling is obtained by reducing the data representation to 4 *bits* or by using an adjustable *bit*-width approximate multiplier (down to 18 *bits*). An iterative optimization heuristics to select candidate neurons and update their approximation is presented. The experimental results are obtained by simulation of the real hardware in 45nm implementation and show energy benefits from 34% to 51% with less than 5% of quality loss.

Another confirmation of the resilience of ANNs to less precise data representation is shown in (GUPTA et al., 2015), in which the authors implement deep networks with 16 *bits* fixed-point number representations and stochastic rounding. This topology provides similar results to 32 *bits* floating-point reference implementations in terms of classification accuracy. It is argued that the use of stochastic rounding reduced by half the number of *bits* required for fixed point training, compared to the latest known work of approximate data representation in deep networks. During the analysis of common benchmark datasets, it is mentioned that a mixed-precision approach, with higher precision fixed-point operations at the end of the training process, is also a promising strategy. Furthermore, a proof of concept FPGA implementation is also presented. This optimized version achieves 37 $G_{ops}/s/W$ indicating a considerable improvement over CPU and GPU implementations (in the range of 1 – 5 $G_{ops}/s/W$).

The paper (LIN; TALATHI, 2016) cites three recent works which have shown that stochastic rounding is a good strategy to improve stability of the training process in deep neural networks under limited numeric precision. The main contribution of this work

is a theoretical insight into the root cause of such stability problems. Complementary techniques to stochastic rounding are also presented. Starting from a “perfect” activation function, the authors show how the effective operation (which is not differentiable) introduces quantization errors. Such errors affect the gradient computation especially in the case of deep networks, because each level propagates the previous errors and introduces others. Three proposals are presented to the problem: low precision weights and full precision activations during training, fine-tuning only the top layers since gradient errors build up and a bottom-to-top iterative fine-tuning scheme which applies approximations progressively. Results show quality improvement by the three proposals, with similar results (less than 0.5% difference) among them and to the reference floating point. The best performance was achieved with 8 *bits* weights and 16 *bits* activations, which, if reduced to 8 *bits*, cause only a 2% drop in accuracy.

The work (KIM; SMARAGDIS, 2016) presents a neural network architecture that implements weight parameters, bias terms, input, and intermediate hidden layer output signals as binary values. These conditions apply only to the feed-forward phase (which involves just XNOR operations and *bit* counting), since the training process evolves from real-valued operations to the convergence to the final binary results. A weight compression technique is described as the first step of the training process (which has to handle real-valued inputs) and the final step involves the BNN with a “Noisy Backpropagation” implementation. The normal sigmoid activation function is also replaced by a simple sign operation. The method is compared with a 64 *bits* floating point equivalent ANN, with negligible accuracy loss for a hand-written digit recognition database. Resource savings are not analyzed in this publication.

The technical report (HINTON et al., 2012) is cited as the basis or inspiration for several ANN based approximated implementations - like (WAN et al., 2013) and (SRIVASTAVA et al., 2014) - and binary networks (with 1 bit precision weights) as presented in (COURBARIAUX; BENGIO; DAVID, 2015) and (GOODFELLOW et al., 2013). The main idea is not related to AC by the authors, and presented as an effective method to trim the complexity of the ANN during training to avoid over-fitting. Despite this, when compared to other proposals in this review, the method devised by Hinton et al. can be considered as an approximation, since randomly selected (or less relevant) connections are dropped and disregarded from calculations during training. Obviously this would not have an efficiency impact (at least in implemented circuits) on hardware based topologies, but still serves as another confirmation of the resilience of these algorithms. Since the removal of nodes reduces the learning capacity of an ANN, it can be used to implement more complex network topologies that dynamically adapt to problems that are easier to solve. This technique may provide implicit regularization, as proposed in (ANGUITA; BONI; RIDELLA, 2003b).

The analysis presented in (HASHEMI et al., 2016) is based on a broad range of numerical representations applied to ANNs in both inputs and network parameters: floating point (as reference), fixed point (4 different *bit* widths), powers of two (replacing multiplications by *bit* shifts) and binary weight nets. The analysis is based on several benchmark datasets and evaluates the compromise between accuracy and hardware implementation metrics, with techniques that use some of the saved resources to increase the ANN size and improve performance. Results show a wide range of approximation parameters with negligible degradation in performance, but binary nets achieved the best results: the highest energy savings (94%) with the best accuracy (higher than the floating point reference). Conversely, powers of two and 8 *bits* fixed point implementations were more consistent among different datasets.

A method which dynamically prunes redundant connections from a trained ANN is presented in (HAN et al., 2015). Since external memory accesses are shown to be the most relevant operations when energy consumption is considered, reducing the number of network weights is an important optimization. The reason the authors consider the proposal as a dynamic one is that the three phases (training, pruning and fine tuning) can be repeated iteratively. The results for two datasets achieve $9\times$ and $13\times$ compression ratio, without loss of accuracy. A comparison with 6 other pruning methods is provided and the effects of regularization parameters are also analyzed. Further developments, which include weight sharing and Huffman coding, are proposed in (HAN; MAO; DALLY, 2015). With these improvements, the compression ratios reach $35\times$ in one dataset and $49\times$ on another.

The *Flexpoint* format is proposed in (KÖSTER et al., 2017). The precision is a compromise between fixed point and floating point formats, since the exponent is shared among all values within the same tensor. This simplification achieves both memory requirements reduction (which also affects the communication traffic) and computations complexity savings. Other interesting improvement is the automatic statistical analysis of the mantissas, which tries to anticipate overflows and adjust the tensor exponents. The authors compare the approach to single precision floating point references using three popular deep learning computer vision benchmark datasets with convolutional and generative adversarial networks. The results show equivalent accuracy performance when compared to single precision floating point implementations and superior to half precision floating point in some cases.

In (LI et al., 2017) the authors reinforce the tendency verified in this survey: “results in this area are largely experimental”. Initially the authors analyze and compare the convergence of BinaryConnect (BC) and Stochastic Rounding (SR) for convex problems. Both methods are covered in this work. Despite the fact that the measures of convergence presented are not directly comparable, in conditions normally found in the final optimization

stages for ANNs, BC would perform better than SR. The comparison is then extended without the convexity assumptions, analyzing SR as a Markov Chain. This method is found to have a limitation: at the end of the learning process, as the weights adjustments diminish, it becomes less likely to perform finer minimizations to the loss function. Considering this characteristic, increasing the batch size would improve the performance of SR. The experimental section compares the approximations methods using Adam as the baseline optimizer with popular computer vision benchmarks. The results support the claims of the presented theoretical study.

Support for standard half-precision floating point format (with 16 *bits*) is becoming easily available in GPUs, ASICs and general use CPUs. Three techniques to use this format with deep neural networks are proposed in (MICIKEVICIUS et al., 2018), without modifications to hyper-parameters and with equivalent accuracy to precise formats. The first one does not provide the full memory usage benefits of the lower precision since it uses single precision (with 32 *bits*) format to store a “master copy” of weights. The second technique scales the loss function to avoid losing the small but significant weight adjustments. Finally, a mixed precision dot-product arithmetic is presented with half-precision arguments and single-precision results. With the three techniques enabled, the authors trained 13 models which matched the baseline accuracy in most cases and even presented slightly improved results in some of them. This effect is also found in other methods and is commonly associated to regularizing effects of lower precision.

Another mixed precision training setup is described in (DAS et al., 2018). Like in the *Flexpoint* approach, an exponent is shared between values in the same tensor. Conversely, unlike many related methods, the authors present a solution which is simpler to implement in general purpose hardware, using integer operations, instead of relying on specialized hardware. The main part of the optimization are the FMA (Fused Multiply and Accumulate) operations with 16 *bits* integer arguments and 32 *bits* integer output. The practice of storing precise representation of weights for training calculations is also used. For this reason, the conversion of integer accumulations to floating point must be performed. Results show equivalent or slightly better accuracies and almost 50% savings in computation when compared to a reference single precision floating point baseline.

Mixed precision is also the recommended technique in (DRUMOND et al., 2018). The authors propose a hybrid Block Floating Point (BFP), sharing the exponent among all values in the same tensor, and applying the format to all dot product operations. All the remaining operations are performed with regular floating points. Stochastic rounding is used for mantissa truncation, with the advantages already mentioned in this Section. Besides three image classification datasets, frequently used to test AC techniques, the article also reports equivalent performance for language modeling tasks. A proof-of-concept implemented in FPGA achieves a throughput improvement of 8.5 times, when compared

to FP16 multiply-and-add units implemented in the same FPGA.

A deep ANN architecture, implemented only with integer operations for inference and training is proposed in (WU et al., 2018). The authors criticize the fact that even in very strongly approximated ANN implementations, most SGD based methods accumulate the weight gradients with higher precision. This overhead is even worse in adaptive methods, which store at least one extra parameter for each weight. Addressing these issues, the main features may be summarized as: an efficient quantization function with stochastic rounding and a modified weight initialization to avoid zeroed initial values. Details of how the quantization is applied to weights, activations, errors and gradients are also provided. No adaptive terms (like the ones present in Adam, RMSProp or Momentum based SGD) are included, no batch normalization is used in the optimization process and no explicit regularization is included in most cases. Softmax is also avoided in models with fewer outputs. Four classic computer vision benchmark datasets are used to evaluate the techniques and compare them to eight other methods with approximations. The authors also conduct experiments testing different quantization parameters, increasing the back-propagation precision. With the strongest approximation (maximum of 8 *bits*) the method achieved state of the art performance in two datasets, and similar accuracy in a third one. The fourth problem resulted in considerably worse results with this approximation level.

Even historic architectures, like WISARD (ALEKSANDER; THOMAS; BOWDEN, 1984), mentioned in Section 2.2, are still being revisited. In 2019, the 27th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning featured a special section on Weightless Neural Networks (WNN). These networks are so simple that the only arithmetic required is the comparison between unsigned integers. The generalization does not come from small changes in input values, but from random binary connections (the “weight” is either 1 or 0). One of the presented approaches in the mentioned conference was evolved and published in the paper (TORRES et al., 2020). The authors merge a simplified edge detection mechanism (based on pre-defined patterns) to deforestation detection WNNs (which also act as binarization circuits for several *bits* in parallel). Since training is not required on the target application, the framework automatically generates VHDL code for trained WNNs, connected to the edge detector, and also synthesize them unattended, allowing the complexity of the resulting FPGA implementation to be part of the optimization process. Very simple logic is achievable for each block of 3×3 pixels, due to the sparsity of the trained networks and the optimization capabilities of the synthesis tool.

3.2 Support Vector Machines

The SVM is a supervised learning method initially proposed (BOSER; GUYON; VAPNIK, 1992; CORTES; VAPNIK, 1995) as an algorithm to optimize the separation hyperplane of binary classifiers. The method can be also used for multi-class classification or regression and has become one of the de-facto standards in the machine learning research. The use of *kernels* to map the input to feature spaces is of special interest for AC, as shown in the analysis of several papers in this section. Figure 3, generated with LIBSVM (CHANG; LIN, 2011) and synthetic data, depicts an example of non-linear Radial Basis Function (RBF) SVM classifier (with two input dimensions and three classes). Along with the data points, the final classification regions show how the SVM optimization process tries to maximize the separation between the classification frontiers and the data regions.

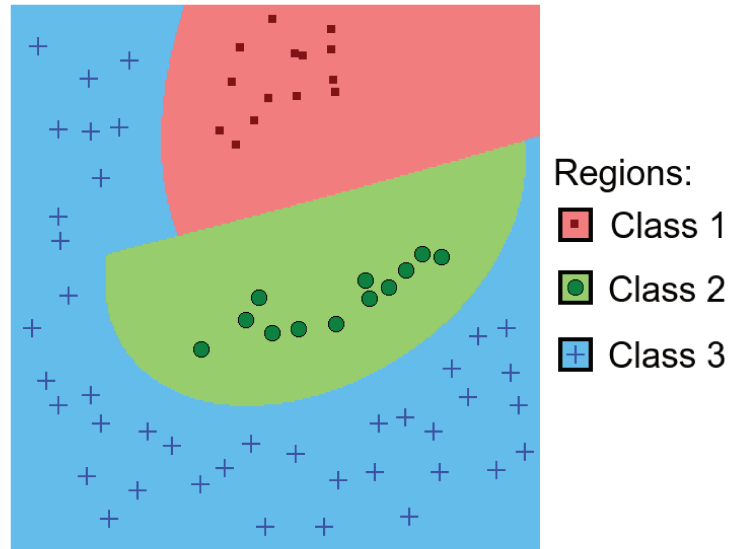


Figure 3 – Three-class SVM/RBF classifier with large margins

Two papers (ANGUITA; BONI; RIDELLA, 2003b; ANGUITA; BONI; RIDELLA, 2003a) present a research group proposal for a SVM learning digital architecture and discusses its implementation on an FPGA. The authors provide a detailed analysis of the quantization effects of the SVM parameters. A considerable amount of comparison tests is performed with varying register lengths for fixed-point implementations and comparing them to a floating point reference algorithm. In general, the performance of the proposed method matches or even improves the reference results. One of the main conclusions in these papers is that the quantization effect in specific areas benefits the generalization capability.

The SVM feed-forward phase is implemented in (ANGUITA et al., 2006) with an iterative algorithm in fixed-point arithmetic using only shift and add operations, avoiding multiplications. The authors describe a “hardware-friendly” kernel similar to the Gaussian

with a different base (2 instead of e) and L_1 norm. The proposal is based on a 1965 iterative algorithm and a thorough comparison is performed between different precision implementations regarding the error rate. When comparing the optimized kernel with the Gaussian, the authors find that the error rate in the approximated version is statistically equivalent in 11 out of 13 datasets. Final results show that the approach is similar in terms of error rate to the Gaussian kernel (implemented with 64 *bits* floating point operations). Regarding the fixed-point *bit* width, the accuracy starts to deteriorate between 12 and 8 *bits*, depending on the dataset.

The novel SVM architecture proposed in (AYINALA; PARHI, 2013) uses two approximation techniques to achieve a more efficient implementation: multiply-add operations with fixed width and exponential function based on a size-optimized Look-Up Table (LUT). Considering linear, polynomial and RBF (Radial Basis Function) kernels, the authors note that the core computation modules are dot-product, L2-norm and exponent. The first two benefit from approximations in a multiply-accumulate operation (proposed in (CHO et al., 2004)) and the trade-off between *bit* width (12 and 16) and accuracy is shown. The RBF implementation is optimized by a proposed LUT-based exponent function with different quantization step sizes. Final results are presented by comparing not only the *bit* lengths but also the use of both techniques combined or separated. With the optimizations combined and choosing 16 *bit* words, the authors claim 31% energy savings without affecting SVM accuracy.

It had been shown in (ARNOLD et al., 1997) that Logarithmic Number Systems (LNS) provide an efficient way of implementing back-propagation training in neural networks. The same system is applied to SVMs in (KHAN; ARNOLD; POTTENGER, 2005), in which the authors propose a hardware implementation using a linear kernel. A software simulation of the LNS is also provided for comparison and no statistically significant difference in the accuracy is found between this version and the hardware one. The difficulty due to the lack of representation of the “zero” value in LNS is mentioned but, when compared to a reference implementation, the proposed system achieves the following classification performance: in one dataset the accuracy is improved from 74% to 83.6% and in the three others it dropped between 1% and 4%. The authors compare the complexity of the FPGA implementations (in some cases reduced by half) as an indication of the energy efficiency improvement.

The use of LNS is also analyzed in (KHAN; ARNOLD; POTTENGER, 2004). The authors use some benchmark datasets to compare a reference SVM double floating point implementation with several LNS implementations with different *bit* precisions. The results show that, for the datasets considered, a general purpose SVM needs only 7 or 8 *bits* with LNS to achieve results within 1% of the reference implementation.

In (ESMAEELI; GHOLAMPOUR, 2012) the authors propose a more memory

efficient use of LNS for SVM classifiers. The improvement consists in using LNS for multiplication but fixed point for addition and subtraction. These two operations are more complex if implemented logarithmically, usually demanding look-up tables. A comparison is performed to show the high memory demands by these LUTs to achieve good accuracy. An optimized system is proposed to convert LNS to fixed point, using a reduced LUT and a shifter. The authors show that to achieve the same good accuracy in SVM, the memory demand is reduced to a small fraction ($\approx 1/200$) of the traditional LNS implementation.

Recent reports in the literature that achieve comparable results between heavy quantizations in both data and hyperparameters and reference floating point implementations of learning systems are analyzed in (SAKR et al., 2016), and later summarized in (SAKR et al., 2017). One of the criticisms about these works is that most of them are empirical studies, which leads to the main contribution of the paper: the derivation of analytical lower bounds on the precision requirements for stochastic gradient descent on SVMs. Most of the previous methods compare the different precisions with an “acceptable precision” reference but the proposed method bases its estimation on the converged weight vector of the reference algorithm. Results support the success of the approach and indicate considerable energy savings when assigning the numerical precision at the lower theoretical bounds. This is a clear advantage over previous methods which just show different compromise points among selected precision values.

The application presented in (WU et al., 2016) shows that an approximated SVM implementation for remote sensing and hyperspectral image classification, achieves 70% of power savings in the kernel operation in one dataset and 75% in other, both with comparable classification accuracy (approx. 1% drop on average for all classes). These final results are based on real data and compared to a precise ripple carry adder implementation on both classification accuracy and power consumption. In order to analyze resilience to input error, noise is injected in the original hyperspectral image. This evaluation is performed as follows: up to 45 LSBs (Least Significant Bits) random error can be inserted in 100% of data represented with 64 *bits* floating point per pixel, with no impact on classification accuracy. Algorithm resilience is also analyzed with a similar procedure and results show that the kernel accumulation computation module is a promising target for optimization. A detailed description of a hardware implementation of an approximate accumulator used in the kernel computation is presented.

A training dataset approximation for SVMs is proposed in (NANDAN; KHARGONEKAR; TALATHI, 2014). The authors present a method to select a representative subset of the training examples to alleviate the problem of quadratic time complexity growth with the training dataset size. The authors provide an extensive theoretical basis to support the solution. Such analysis is not frequently found in similar research, as seen in this survey. For the experiments, nine datasets were chosen to explore variations in the

amount of samples, feature complexity and density. In five of the datasets the algorithm provided considerable reduction of the training set, consequently also the training time, in some cases as high as 99.5%. For the other four datasets the reduction varied from 27.8% to no reduction in one case. In the first dataset group the proposed method obtained equivalent accuracy performance in three cases and considerably better in two, when compared to standard SVM implementations. Accuracy performance was found to be equivalent in the second dataset group, with high-dimensional problems, and training time was not much faster, as expected from the smaller dataset simplifications.

3.3 Generic Approximate Computing

This review does not provide a full coverage of AC methods and only focus on their ML applications. Many of the papers presented in this section either study a very specific operation that is commonly used in learning algorithms (without analyzing the full application) or apply a generic AC technique to more than one ML method.

The approach detailed in (VENKATARAMANI et al., 2015) explores the idea that most of the input data from real systems can be correctly classified with minimal computational effort. This means that only “hard to classify” examples should use the full computational effort of the classifier in systems aiming for energy efficiency. Although frequently resorting to SVM in their explanations, the authors claim that their methodology can be successfully applied to any classification algorithm. The cascaded classifiers output not only the classification result but also a confidence level. Based on this value, more complex classifiers are invoked if needed. The method is tested in three ML algorithms (SVM, ANN and Decision Trees) using several benchmark datasets. The scalable effort classifiers were designed to achieve the same final classification accuracy as the single-stage baseline. The average improvement in *operations/input* varied from 1.2 times to 9.8 times. The results are presented as average since they depend on the test set. In one extreme case, for example, 90% of the inputs are evaluated at a cost of just 0.2% of the baseline.

Although focused on the analysis of an imprecise multipliers based on imprecise adders, which are not focused on this review, the static error estimation technique presented in (HUANG; LACH; ROBINS, 2012) may be used as inspiration for other types of approximation approaches. Instead of expensive Monte Carlo simulations, which require the actual computation to be performed, the authors propose a static analysis which is based on probability mass functions representing the statistical distributions of errors. The method is an improvement of the classical interval arithmetic, which is limited to uniform distributions of errors for each variable and does not consider imprecise operations. The analysis shows very similar results to the ones obtained from simulations.

A Recursive Least Squares (RLS) algorithm implemented in fixed-point is presented

in (BOSMAN et al., 2013) as an efficient replacement for Linear Least Squares Estimation (LLSE), which is used to estimate model parameters in a closed form (without iteration). The main advantage of the recursive method is that it does not have to keep all the input data for calculation. The performance is analyzed regarding standard data anomalies: spikes, constant values, noise, drift (slow change in offset) and shift (sudden change in offset). The proposed method is compared to a reference RLS in floating point, a traditional LLSE and a windowed version of this method. In the best approach, the anomaly detection itself is performed by comparing the RLS absolute estimation error with an adaptive threshold. The windowed LLSE also has fixed memory requirements, but the RLS is less prone to over-fitting and theoretically requires less processing. There are too many aspects in the final results (anomaly types versus algorithms) to be summarized here, but RLS is found to be a promising solution for embedded and resource constrained devices.

The paper (CHIPPA et al., 2013a) is not restricted to ML and intends to show the high degree of resilience intrinsically present in many applications. The authors argue that the analysis and some insights into the nature of application resilience are good guidelines for future work in this area. The five sources of application resilience (present in most ML methods) are presented to explain why standard error injection, used to simulate non-deterministic hardware behavior, cannot be used in studies related to intentional computation approximation. A resilience characterization framework is presented to achieve the identification of potentially resilient computation kernels (avoiding the sensitive ones and parts which do not represent a considerable part of the execution time) and the characterization of these computations through approximation models. The resulting characterization is divided into three approximation models: data representation, arithmetic operations and algorithmic level. The effectiveness of approximate computing in applications that spend most of their computing time in resilient kernels is analyzed extensively. Results and conclusions are grouped into several parts, including: the relevance of granularity of approximation adjustments; similarity between frequent small errors and rare large ones; the importance of relative scale of input data; the advantage of application-aware approximation over an application-agnostic one.

Based on their previous work (CHIPPA et al., 2013a) to characterize application resilience, the authors propose in (CHIPPA et al., 2013b) a method for dynamically adjusting approximations by analyzing the resilience and acting in a manner similar to a feedback control mechanism. The authors demonstrate that static settings for approximations do not handle well situations in which the resilience varies with different problems or even within the same problem and datasets. This leads to either missed opportunities for energy savings or degraded output quality. A scalable effort processor is presented, providing “virtual control knobs” which adjust: the operating voltage (which cause timing errors), variable bit widths (which leads to quantization and truncation errors) and algorithm

specific (SVM and k-Means) parameter simplifications. In order to estimate quality, two approaches are implemented: infrequently comparing the system outputs with and without scaling and low overhead sensors at all levels (circuit, architecture and algorithm). The proposed system is compared to reference implementations and two different approaches based on static scaling. Consolidated results show that the dynamic method combines the best performance from both static methods: energy reduction similar to aggressive adjustments (54% to 69%) with error overshoot similar to the conservative ones (less than 1%).

A framework is proposed in (ZHANG et al., 2014) to dynamically balance the trade-off between output quality, with guarantees, and computational effort for iterative methods. The solution is divided into two parts: an offline characterization and an online reconfiguration. The offline stage is executed once for each application and identifies the error-resilient parts which are candidates for approximations. With a lightweight quality estimator used at each iteration, the online stage reconfigures the approximation modes during execution. Two strategies are proposed: the incremental one starts with the lowest accuracy level and increases the quality requirements with time; the adaptive one performs its adjustments based on the contribution of each approximation component to the convergence of the solution. Results show that energy savings, compared to a reference without dynamic adjustments, range from 25% to 52% using the incremental approach and from 28% to 63% with the adaptive one.

A linear discriminant analysis (LDA) algorithm is proposed in (ALBALAWI; LI; LI, 2014) for efficient implementations of binary classifiers using fixed-point arithmetic and small word lengths. The standard LDA algorithm, normally implemented with double-precision floating-point operations, was redesigned to add robustness to rounding errors and overflows. The feature vectors can be safely scaled and rounded to their fixed point representation but the weight vectors must be handled more carefully. Feature vectors are modeled as multivariate Gaussian distributions so that their multiplication by a weight vector and projections also yield Gaussian distributions. Using this model, it is possible to adjust the fixed-point conversion within a certain confidence interval without causing overflow in these operations. With real datasets, results show that the proposed method achieves $1.8\times$ power reduction without sacrificing classification accuracy when compared to a regular LDA simply adapted for fixed-point implementation.

A detailed analysis of implementation techniques for machine learning algorithms in a low-power and high-performance 16 *bits* DSP is presented in (BHARATI; JHUN-JHUNWALA, 2015). The analysis is performed separately for each algorithm, listing all proposed optimizations, but the authors claim that the solutions apply to most commercial DSPs. Approaches include: an approximate expression for the sigmoid activation function, efficient dot product implementation using MAC operations and lookup tables, optimized

fixed count loops and an approximated formula for the magnitude of a complex value. The results show drastic reduction in cycle counts (70% to 91%) when compared to reference floating-point implementations, without considerable compromise of accuracy.

The recent work (MITTAL, 2016) is not focused on ML applications of AC, but the organization provided by the authors can be applied to facilitate the selection process for each technique applicable to a specific ML algorithm. Characterization of approximable program portions are grouped into: error injection, output quality monitoring, source code annotations and compiler optimizations. The groups of this classification apply seamlessly to ML. Conversely, not all approximation strategies apply to ML. The relevant ones in this context are: precision scaling, loop perforation, data fetching simplifications, function results reuse and multiple kernel precisions.

3.4 Comparisons and Limitations of Current Techniques

Tables 1 and 2 merge the Sections 3.1, 3.2 and 3.3 according to the classification proposed in this Chapter in order to summarize the research efforts that focus on the application of AC to ML. The references show which methods of each type were used, considering only the 38 presented solutions (disregarding the references cited only as previous and similar works or inspiration). Two papers (CHIPPA et al., 2013a; MITTAL, 2016) from section 3.3 were excluded from Table 1 because they were not focused on a complete implementation. Improved implementations from the same authors (that were analyzed together in this survey) are explicitly referenced in the tables.

It is worth mentioning that only 6 methods proposed a solution that was both *automatic* and *dynamic*. This confirms that most of the surveyed implementations are not generic regarding the ML method and also were not prepared to handle unknown datasets, adapting the approximations. Despite their potential to be automatically adapted to different problems, four of them used from two to four datasets to verify their performance.

Understandably, it is not trivial to test very complex models with too many datasets, due to the required processing time. On the context of AC, it should not be forgotten that these datasets are typically the most resilient ones (since they are frequently noisy and redundant). As an evidence that current research is not focused on generic approximated solutions, Figure 4 represents how extensively the literature tests the proposed methods. Papers which did not present a complete solution or analyzed only the hardware impact of the simplifications are plotted as “zero datasets”.

The majority of the methods (22) applied a *static* and *guided* approach and 31 implemented the **data representation** approximation (17 applied only this technique). This fact strengthens the claims that ML methods are usually resilient to noise but if the

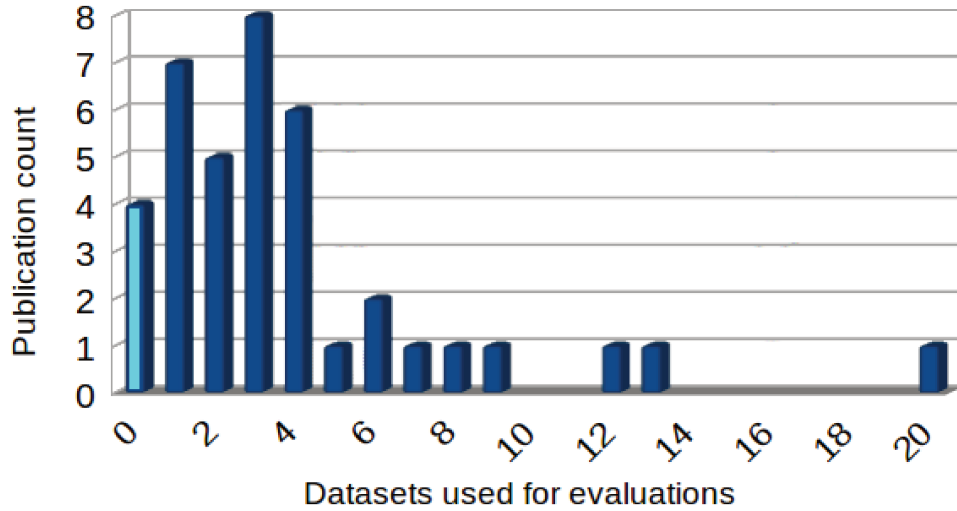


Figure 4 – Number of datasets used for concept validation.

method is not *dynamic*, the selected approximation levels may not be adequate to other datasets (or network topologies in the case of ANNs). Conversely, if the approximation solution is specifically designed for one method, the *automatic* approach may represent an unnecessary complexity, since frequently the resource usage is known a priori and very specifically located.

One example of the focus on specific methods, the sum of products operations seem to attract a lot of interest. Approximations on *complex functions* were rarely found on the surveyed research. The computational cost of activation functions were representative on smaller ANNs (SCHRAUDOLPH, 1999) but the increased size of current models and simpler functions like ReLUs - Rectified Linear Units (NAIR; HINTON, 2010) - probably influence this tendency. Specific techniques, adequate for FPGA implementations, are still being presented (DIAS; SALES; OSORIO, 2016). These may be relevant also for multi-class problems that rely on *Softmax* (DUNNE; CAMPBELL, 1997) outputs, due to their use of the exponential function.

It is not a surprise that deep learning and computer vision or big data applications dominate the reviewed AC research when ANNs are considered, due to their fast increase in popularity in this decade and high performance demand. Its is important to note that these problems rely on very large, redundant an usually noisy datasets. This is a crucial set of characteristics to explain why the extreme implementation simplifications found in this work are possible. A similar level of interest was not found when AC is applied to another promising field: online and autonomous ML used in resource constrained smart devices. When mixed precision approaches are not used for training, even standard “half-precision” floating points (with 16 *bits*) result in training problems with some simple datasets, as shown in (TORRES; TORRES, 2017).

Searching for the highest acceptable approximation level on specific problems is a

completely different goal than defining an efficient and generic method that works equivalently to precise implementations, even if not reaching the highest possible optimization in each one. Additionally, many papers that present expressive approximations restrict their validation to a single class of problems (e.g. image classification). It is important to note that the network configurations (number of layers, nodes and connections) used in this thesis are the same minimal ones employed in other references (as a form of regularization). It is not uncommon to see larger networks in articles analyzing low precision training, which partially defeats the purpose of the simplifications.

Achieving the highest possible optimization usually means that the performance degradation reaches the limit of acceptability for a specific application. The generic approach is still rarer in the published research, but commonly found in commercial hardware offers (with more conservative solutions). Both the recently released NVIDIA Volta architecture (MICIKEVICIUS et al., 2018) and Google TPUv2 (the second version of the Tensor Processing Unit) rely on mixed precision 32 *bits* FP operations to mark their position as training platforms and not only as inference accelerators. The trend continues as the third TPU generation, announced in May 2018, still relies on a mixed precision approach.

This analysis reinforces the relevance of an architecture and method which could be applied to ML problems, both in inference and training, more efficiently than a precise implementation and equivalently reliable. Even in the cases where an eventual small performance (e.g., accuracy) penalty could not be accepted in the final training runs, approximated solutions could be used for the costly hyper-parameter tuning phases. Using the categorization proposed in the beginning of this Chapter, the group of methods implemented in this thesis, which are presented in the next chapters, can be classified as: ***dynamic***, ***guided***, based on ***data representation***, ***mathematical operations*** and ***complex functions***. Additionally to being grouped with a small number of solutions proposed in the literature, this thesis presents a method which uses the same representation in the entire ANN, allowing the implementation of a single type of FPU. It should also be clear that when “equivalency” to precise methods is stated, an evaluation that the behavior is “identical” to precise method is obviously not being implied. Even full 32 *bits* FP implementations may face numerical problems or suffer from “vanishing gradients” in deep networks. Approximated operations will certainly reach such limits sooner than precise references, which could result in, as an example, different optimal hyper-parameters (like learning rates or batch sizes).

Table 1 – Approximation Type

Method	Uses
Static	(ALBALAWI; LI; LI, 2014; ANGUITA; BONI; RIDELLA, 2003b) (ANGUITA; BONI; RIDELLA, 2003a; ANGUITA et al., 2006) (ARNOLD et al., 1997; AYINALA; PARHI, 2013) (BHARATI; JHUNJHUNWALA, 2015; BOSMAN et al., 2013) (COURBARIAUX; BENGIO; DAVID, 2014) (COURBARIAUX; BENGIO; DAVID, 2015; DAS et al., 2018) (DRUMOND et al., 2018; ESMAEELI; GHOLAMPOUR, 2012) (FERNANDEZ-DELGADO et al., 2011) (GOODFELLOW et al., 2013; HASHEMI et al., 2016) (HINTON et al., 2012; HUANG; LACH; ROBINS, 2012) (KHAN; ARNOLD; POTTENGER, 2005) (KHAN; ARNOLD; POTTENGER, 2004) (LI et al., 2017; MICIKEVICIUS et al., 2018) (SAKR et al., 2016; SRIVASTAVA et al., 2014) (WU et al., 2016; WU et al., 2018)
Dynamic	(COURBARIAUX; BENGIO; DAVID, 2014) (CHIPPA et al., 2013b; GUPTA et al., 2015) (HAN; MAO; DALLY, 2015; KIM; SMARAGDIS, 2016) (KÖSTER et al., 2017; LIN; TALATHI, 2016) (NANDAN; KHARGONEKAR; TALATHI, 2014) (VENKATARAMANI et al., 2015; WAN et al., 2013) (ZHANG et al., 2015b; ZHANG et al., 2014)
Guided	(AGRAWAL et al., 2016; ARNOLD et al., 1997) (ANGUITA; BONI; RIDELLA, 2003b) (ANGUITA; BONI; RIDELLA, 2003a) (ANGUITA et al., 2006; AYINALA; PARHI, 2013) (BHARATI; JHUNJHUNWALA, 2015; DAS et al., 2018) (BOSMAN et al., 2013; DRUMOND et al., 2018) (COURBARIAUX; BENGIO; DAVID, 2015) (ESMAEELI; GHOLAMPOUR, 2012) (FERNANDEZ-DELGADO et al., 2011) (GOODFELLOW et al., 2013) (GUPTA et al., 2015; HASHEMI et al., 2016) (HINTON et al., 2012; HUANG; LACH; ROBINS, 2012) (KHAN; ARNOLD; POTTENGER, 2005) (KHAN; ARNOLD; POTTENGER, 2004) (KIM; SMARAGDIS, 2016) (LI et al., 2017; LIN; TALATHI, 2016) (MICIKEVICIUS et al., 2018) (SRIVASTAVA et al., 2014) (VENKATARAMANI et al., 2015) (WAN et al., 2013; WU et al., 2018; WU et al., 2016)
Automatic	(ALBALAWI; LI; LI, 2014; CHIPPA et al., 2013b) (HAN; MAO; DALLY, 2015; KÖSTER et al., 2017) (NANDAN; KHARGONEKAR; TALATHI, 2014) (SAKR et al., 2016) (ZHANG et al., 2015b; ZHANG et al., 2014)

Table 2 – Approximation Method

Technique	Uses
Data Represent.	(ALBALAWI; LI; LI, 2014; ANGUITA; BONI; RIDELLA, 2003b) (ANGUITA; BONI; RIDELLA, 2003a; ANGUITA et al., 2006) (ARNOLD et al., 1997; AYINALA; PARHI, 2013) (BHARATI; JHUNJHUNWALA, 2015; BOSMAN et al., 2013) (CHIPPA et al., 2013a; CHIPPA et al., 2013b) (COURBARIAUX; BENGIO; DAVID, 2014) (COURBARIAUX; BENGIO; DAVID, 2015; DAS et al., 2018) (DRUMOND et al., 2018) (ESMAEELI; GHOLAMPOUR, 2012; GUPTA et al., 2015) (HAN; MAO; DALLY, 2015; HASHEMI et al., 2016) (KHAN; ARNOLD; POTTENGER, 2005) (KHAN; ARNOLD; POTTENGER, 2004) (KIM; SMARAGDIS, 2016; KÖSTER et al., 2017) (LI et al., 2017; LIN; TALATHI, 2016) (MICIKEVICIUS et al., 2018; MITTAL, 2016) (SAKR et al., 2016; WAN et al., 2013) (WU et al., 2018; WU et al., 2016) (ZHANG et al., 2015b; ZHANG et al., 2014)
Math. Operations	(BHARATI; JHUNJHUNWALA, 2015) (BOSMAN et al., 2013; CHIPPA et al., 2013a) (FERNANDEZ-DELGADO et al., 2011) (HUANG; LACH; ROBINS, 2012; WU et al., 2016) (KIM; SMARAGDIS, 2016; MITTAL, 2016)
Complex Functions	(ANGUITA et al., 2006) (AYINALA; PARHI, 2013)
Simplified Algorithms	(BHARATI; JHUNJHUNWALA, 2015) (CHIPPA et al., 2013b) (COURBARIAUX; BENGIO; DAVID, 2015) (FERNANDEZ-DELGADO et al., 2011) (GOODFELLOW et al., 2013) (HAN; MAO; DALLY, 2015) (HINTON et al., 2012; MITTAL, 2016) (SRIVASTAVA et al., 2014) (VENKATARAMANI et al., 2015) (WAN et al., 2013; ZHANG et al., 2015b)
Memory (Reduc./Skip.)	(BOSMAN et al., 2013; CHIPPA et al., 2013a) (ESMAEELI; GHOLAMPOUR, 2012) (HAN; MAO; DALLY, 2015) (NANDAN; KHARGONEKAR; TALATHI, 2014) (ZHANG et al., 2015b)

4 ANN Implementations

This chapter details how the ideal operations in the *feed-forward* and *back-propagation* phases of ANN inference and training, as introduced in Section 2.1, translate to real implementations. Problems such as the “vanishing gradient”, which arise even in precise representations, may considerably harm the training process if not dealt with. Section 4.1 presents the IEEE format for limited precision FP numbers, which has been an industry standard for more than three decades. Section 4.2 lists some important caveats when limited precision representation is used in ANNs and Section 4.3 details how an open source ANN library was modified to become an easily extensible framework for experiments with different number representations, using emulated code to abstract all operations. The first numerical type integrated (IEEE FP with 16 bits), used for the tests in this Chapter, is also provided by code extracted from another open source library. This Section also shows comparisons of the training process evolution with a precise baseline when standard FPs are used.

4.1 Floating Point Review

In order to better understand the approximations and adjustments that are detailed in Chapter 5, some important and fundamental details of the FP representation are presented in the following sections. The reader is referred to the official standard (IEEE, 2008) for a complete explanation.

4.1.1 Standard Floating Representation

For a fixed space number representation, FPs have become an ubiquitous way to achieve an excellent compromise between digital implementation complexity and useful range. By using a binary version of the scientific notation, the same format can represent very small and very large numbers, with different resolutions. The 16 *bits* binary format (FP16 or “half precision”) is defined as follows:

$$B \ S^0 \cdot S^{-1}S^{-2}S^{-3}S^{-4}S^{-5}S^{-6}S^{-7}S^{-8}S^{-9}S^{-10} \times 2^E \text{ where } E = E^4E^3E^2E^1E^0$$

A single *bit* B is used to hold the signal information (‘1’ meaning a negative number) which is common to higher precision formats (32 and 64 *bits* are the most commonly used and respectively named “single” and “double” precision). Normalized significands in base 10 are represented in the interval $[1.0, 10.0)$, which translate to $[1.0, 2.0)$ when a base two exponent is used. This means that S_0 will always be 1 in binary normalized numbers, so

its representation is omitted. Significand lengths of 10, 23 and 52 correspond to the 16, 32 and 64 *bits* binary formats, respectively.

The remaining *bits* are used to represent the exponent (E) as a non negative integer number, which is shifted by a standardized bias (equal to +15 in the “half precision” format). For example: the minimum exponent for a normal number in 16 *bits* binary format is -14 , which is encoded as 1. When both significant and exponent are equal to zero, ± 0 is represented, depending on the sign *bit*. The complete bit representation is depicted in Figure 5.

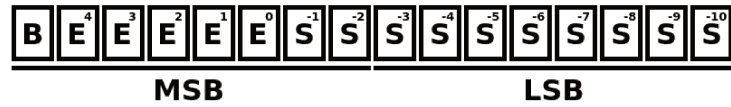


Figure 5 – Bit representation of the FP16 format.

The exponent is also used to encode a few exceptional conditions in the binary representation itself. The smallest exponent ($E = 0$) is used to encode denormalized numbers, meaning that S_0 is assumed to be 0. These numbers may be considered a “soft representation” of an underflow (COONEN, 1981) (an operation that results in a number with an absolute value smaller than the smallest representable absolute number). A graphical comparison between subnormal numbers and normal ones is presented in Figure 6: the 2000 smallest positive numbers in FP16 have their values represented if subnormal numbers are assumed (Red) or without this exception (Green). The plot clearly shows how significantly smaller values are reached by denormalized numbers.

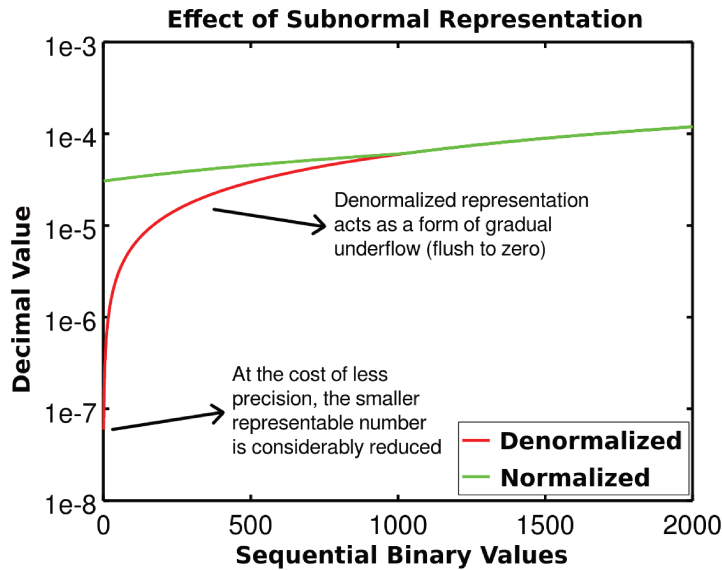


Figure 6 – Effect of subnormal numbers on the lowest exponent

The first impression is that such feature is welcomed to ease the effort of handling underflow conditions. It can be argued, instead, when such feature is useful, especially

considering that floating point programming is not always a trivial task (GOLDBERG, 1991). Such subnormal numbers add considerable complexity to FPU implementations, which may even disregard such support, leaving it to be handled as an exception at software level. Even in modern CPUs the effects of operations involving these numbers may be noticeable (LAWLOR et al., 2005; ANDRYSCO et al., 2015), furthermore the relative error may be much higher, considering the reduced number of effective *bits* in the significand. The following excerpt from IEEE 754R minutes of September 19, 2002 is useful to understand the rationale behind the committee’s decisions involving subnormals:

“[...] Whatever causes one underflow will usually cause a lot more. So occasionally a program will encounter a large batch of underflows, which makes it slow. The loss of speed will upset someone. [...]

If we want reliable codes, we must make codes easy to prove, and we must provide default behaviors that give numerically naive programmers the best chance of avoiding problems.”

The highest representable exponent (with all *bits* set to 1) is also used to encode two other exceptions. When all significand *bits* are set to zero the numbers represent infinity ($\pm\infty$, depending on the sign *bit*). This exception is returned whenever the operation results in a number with an absolute value higher than the largest representable one. Significand *bits* different from zero signal the other exception, NaN (Not-a-Number), triggered by invalid operations which are not covered by simple underflows or overflows (e.g. 0/0).

4.1.2 Standard Floating Point Operations

The basic FP arithmetic operations will be briefly analyzed, but a common aspect will be omitted in each explanation: the handling of subnormal numbers. Such exception to the common flow demands that this type of numbers is normalized before the actual operations are performed. Moreover the underflowed results are verified to determine if they are representable as denormalized values. Not only these operations add complexity to FPU implementations, but they also have indirect effects on the resources required to handle normalized numbers (e.g. internal significand representation).

Addition and subtraction share the common step of exponent adjustment and significand alignment. In order to add or subtract the significand, which can be handled as a binary integer, each bit position must be aligned to another with the same magnitude. For this reason, if the operands exponents do not match, they have to be adjusted firstly in one of the numbers, which requires that its significand is shifted accordingly. The resulting significand may be outside the range $[1.0, 2.0)$, in which case its contents must be shifted and the exponent adjusted, if possible. It should be noted that the exponent bias is not

used in these operations, since it is only required that the encoded exponents are aligned for the addition or subtraction to be performed.

The resulting exponent for the multiplication is just the addition of the operands exponents subtracted by the bias value. The significands are directly multiplied as integers, what may result in much more digits than the representation limit, requiring extra *bits* to be dropped after the rounding operation. Resulting significands ≥ 2 are also adjusted together with the exponent, if possible.

Due to the complexity of the integer division hardware circuits, the significand division is usually implemented in more efficient ways such as using multiplication operations and LUTs. FP division can be quite hard to implement correctly in an optimized way, which probably plays a role in the occurrence of defects like the “Pentium FDIV bug”. Resulting significands may also require adjustments and the final exponent depends on this process, after being determined by the difference of the operands exponents and bias addition.

Every operation generating more significand *bits* than the maximum representable value is subjected to rounding. From the several available options, the most commonly used is rounding to the nearest value. This method results in a maximum error of 1/2 LSB in the significand ($\approx \pm 0.05\%$ in FP16). The non-standard stochastic rounding may also be applied and is especially useful in low precision arithmetic applied to ANN training. The main reason behind this usage is that the explicit “numeric noise” acts as a form of implicit annealing and also acts as a regularization mechanism. Regardless of which from these two methods is used, correct rounding is an expensive operation to be implemented and it must also include a normalization step, since it may cause the resulting significand to be ≥ 2 . Simpler solutions like truncating the significand are normally avoided due to the fact that they add a bias to the rounding error.

4.2 Mathematical Operations

Several characteristics of ANNs may present both opportunities and challenges to low precision representations and operations. The following sections analyze some of these issues and techniques related to them.

4.2.1 Small Values Accumulation

It comes directly as a consequence of a fixed length significand/exponent representation that adding or subtracting numbers with exponents that are too far apart may result in cancellation (the result equals one of the numbers). A graphical representation of how this problem easily arises on less precise formats is shown in Figure 7: the x axis depicts how many times the value 0.001 is added to a FP number starting from 0 and the

y axis indicates the accumulated value for two FP representations. The actual shape of the curve varies with the rounding method, but the saturation value for this example in 32 *bits* FPs is almost 4 orders of magnitude larger than the one for FP16.

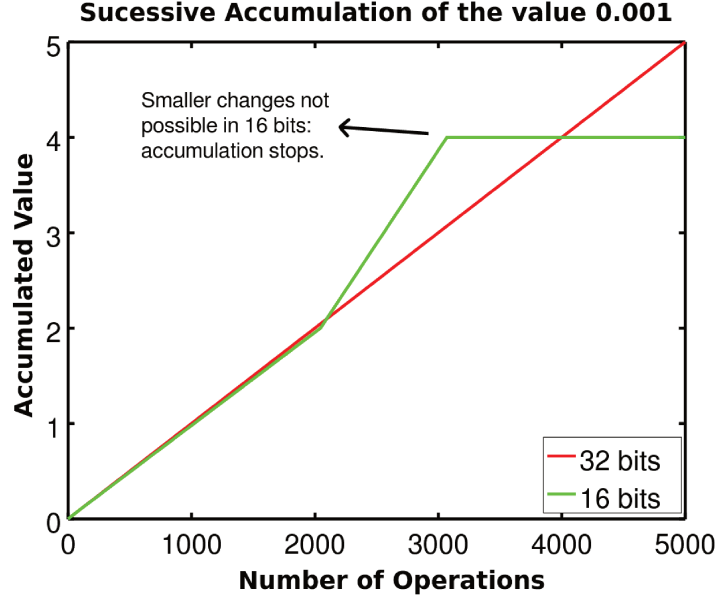


Figure 7 – Accumulation error due to limited precision

The IEEE FP standard specifies an operation which can be applied to handle the frequent sum-of-products calculations in ANNs, and is commonly supported by commercial hardware: FMA (Fused-Multiply-Add). The main idea is not focused on minimizing the error due to small values accumulation, although it may provide some improvements. Equation 4.1 represents a single step of a sum-of-products operation, in which A_n and B_n hold the values to be multiplied, C_{n-1} the previous accumulated value and C_n the next one. If the full step is broken in two completely separate arithmetical operations, the product result must be rounded and normalized, discarding the extra significand *bits*, before an addition is performed. These intermediate rounding and normalization procedures can be skipped, saving resources in the whole process, improving the performance and may also present slightly more precise results (due to a better final rounding).

$$C_n = (A_n \times B_n) + C_{n-1} \quad (4.1)$$

The same idea could be extended if the main focus is on an efficient and long accumulation without losing too many small contributions. Both the argument C_{n-1} and the result C_n could be represented in a more precise representation (e.g. 32 *bits*), while the product arguments are passed in a vector with smaller precision (e.g. 16 *bits*). Known as Fused Multiply-Accumulate (FMAC) or simply MAC, this technique has also become frequently available in several hardware platforms, although it is not always implemented with mixed precisions.

In most ANN structures commonly used nowadays, including Deep Learning applications, the number of connections per node (normally between dozens and thousands) does not represent a critical reliability issue for representations as low as FP16. Additionally to this structural characteristic, random small and adaptive weight initialization (GLOROT; BENGIO, 2010), associated with regularization techniques that avoid large weight values, contribute to the resilience of the sum-of-products realization in the *feed-forward* phase.

For problems with large datasets, it is not efficient to apply weight changes during training for each example, which is the classic Stochastic Gradient Descent (SGD) method. Full-batch optimization methods, which accumulate slope calculations for each weight for the entire dataset, will also suffer from slow convergence, in spite of being very adequate for massive parallelization due to training set partitioning. Mini-batch training methods, which represent the compromise between these two strategies, will also have the advantage of being a solution to avoid large errors due to small values accumulation. This benefit, a direct effect of smaller gradient accumulation sequences, may be possible even with very small initial learning rates. By performing slope accumulations for small subsets of training examples, the weight adjustments will not reach values that are too large when compared to each contribution, avoiding cancellation errors.

4.2.2 Deep Learning

Even before the work presented in this thesis is implemented in hardware and reaches the required performance to make it feasible to analyze Deep Neural Networks (DNNs), it is important to note issues inherent to these topologies. These complex structures are associated with difficult problems, usually represented by very large datasets. Although the mini-batch training approach attenuates the accumulation errors, the “Vanishing Gradient” problem may be an issue even for precise implementations. When methods based on *back-propagation* are used to train ANNs with many layers, the ones closer to the inputs may suffer from very slow training speeds due to small gradient values. Activation functions that saturate their outputs (like the sigmoidal ones) tend to diminish derivatives for input values far from the outputs. Other activation functions like Rectified Linear Units (ReLUs), their variations and the slightly more complex Exponential Linear Units (ELU) have recently become popular and counteract this problem that contributes to smaller gradients for the first layers during *back-propagation*.

A detailed study of the “Vanishing Gradient” problem, which also affects RNNs (Recurrent Neural Networks) with specific datasets, is presented in (SUTSKEVER et al., 2013). The authors argue that both the weight initialization and a well tuned momentum factor are crucial for an SGD based training to perform well. The datasets used so far in this work do not demand networks that are deep enough to cause the problem mentioned here, but adaptive methods have been considered and will be presented, as well as different

activation functions.

4.2.3 Activation Functions

Activation function implementations may be as simple as testing if the input value is negative (e.g. ReLU) or involve more expensive mathematical operations such as the exponential function. Since the derivatives are also used in the learning phase, it is an interesting feature to be able to calculate them based on the output value, instead of repeating the execution of the resource intensive mathematical method. Equations 4.2 to 4.4 present three common activation functions with their derivatives and Figure 8 compares their output.

$$f(x) = \text{ReLU}(x) = \max(0, x) \quad , \quad f'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \quad (4.2)$$

$$f(x) = \text{ELU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \exp(x) - 1 & \text{if } x < 0 \end{cases} \quad , \quad f'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ f(x) + 1 & \text{if } x < 0 \end{cases} \quad (4.3)$$

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \quad , \quad f'(x) = 1 - f(x)^2 \quad (4.4)$$

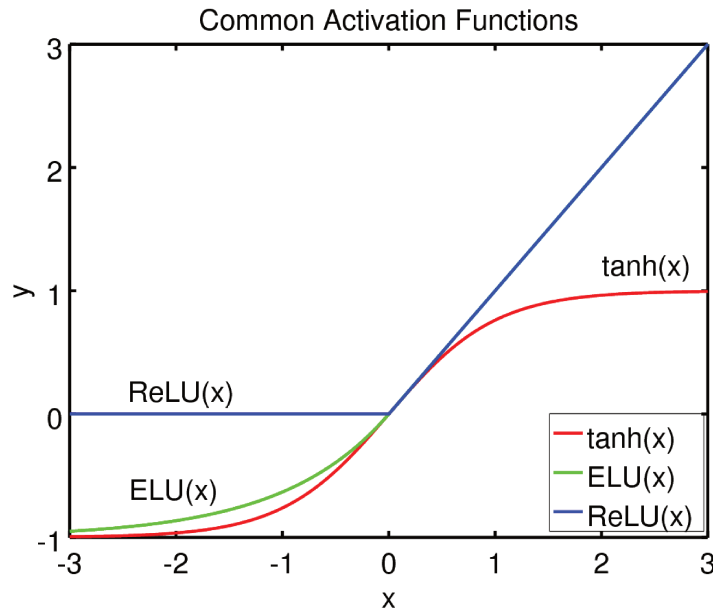


Figure 8 – Activation Functions Comparison

Optimized activation function implementations have been frequently found in the literature for decades and new specific proposals are still being made (DIAS; SALES; OSORIO, 2016). Unless supported by hardware (which is the case for the most used

general purpose CPUs), precise implementations of exponentials are usually considered an overkill which may compromise the ANN performance. LUT-based implementations (even without numerical refinement) may be sufficient for most applications, including the training phase. Figure 9 depicts a LUT based implementation with interpolation and its relative error. The continuous line shows a precise reference calculated with the standard library exponential function. The crosses show several line segments with interpolated values between the LUT values. The relative errors are presented with dotted lines. Even less precise solutions are found in the literature, as well as implementations based on approximations of the mathematical functions, like exponentiation.

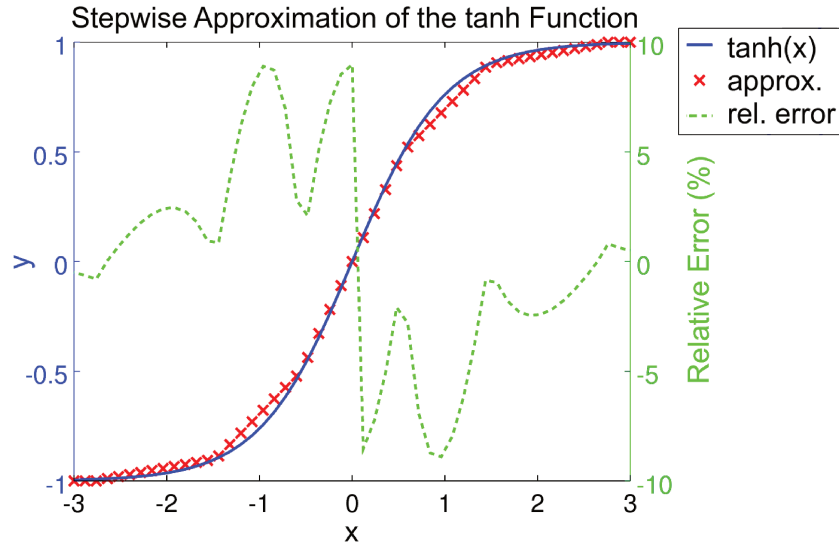


Figure 9 – Stepwise approximation of $\tanh(x)$, with interpolation.

Contrary to these simple and efficient implementations, functions like Softmax (Equation 4.5), which is commonly found in the output of mutually exclusive multi-class classifiers, are interesting targets for careful optimizations. Due to the fact that Softmax is multi-variate (each x represents the aggregate input of a neuron in a certain layer with K neurons), LUT based implementations may not be practical for a large number of dimensions. Not only the table size but also the interpolations would require a considerable amount of resources. For this reason, the explicit exponentiation operations are the preferred base for implementation. Performance issues due to the cost of exponentiation when Softmax is used with a large number of classes motivates research on optimizations, like described in (JOULIN et al., 2017). Unlike simpler activation functions, the divisions and the exponentiations may result in instabilities (e.g. NaNs or infinities). Subtracting the maximum x_k in the layer from each x_i before taking the exponentials is a simple countermeasure against infinities, for example.

$$\text{Softmax}(\vec{x})_i = \frac{e^{x_i}}{\sum_{k=1}^K e^{x_k}} \quad (4.5)$$

4.3 Baseline Tests with standard IEEE FP (64 vs 16 *bits*)

With the details of ANN implementations based on standard FPs already presented, it is interesting at this point to compare a double-precision baseline with the standard 16 *bits* FP. Sometimes this “half precision” representation is even considered a baseline for comparison with approximate implementations, but since some values during training reach magnitudes lower than what FP16s are capable of representing, a thorough analysis is required. Using a standardized floating point format has the advantage of making this first investigation relevant for the growing amount of available systems (compilers and hardware) supporting this representation.

The “Fast Artificial Neural Network Library”¹ (FANN) (NISSEN, 2012) was used as a basis for the implementation, but was heavily modified. One of the main changes was the integration of part of the Berkeley SoftFloat Library (HAUSER, 2017) to perform half-precision standard FP operations². Other important modifications include:

- A complete abstraction of all arithmetic operations and conversions
- Restructuring of the ANN internal representation
- Added support for POSIX Multi-threading
- Removal of unwanted features
- Implementation of detailed operation statistics
- Added compilation option optimized for embedded targets (without statistics and terminal IO)
- Creation of a flexible binary with all ANN definitions selectable at runtime
- Adaptation of RProp to conform to the original iRProp-
- Added support for RMSProp and normalized initialization
- Added support for ReLU activation and Softmax outputs

All coding was implemented in ANSI C, without architecture-specific assembly optimizations, and compiled with GCC 5.4 for a Linux based 64 *bits* system with an Intel Core-i5-2537M CPU. Compiler optimizations which improved the training time when using the approximated implementations (like aggressive “inlining”) were enabled but no accelerations to the native floating point operations were activated.

¹ Source code also available at: <<https://github.com/libfann/fann.git>>

² Source code also available at: <<https://github.com/ucb-bar/berkeley-softfloat-3.git>>

This setup with software emulation and abstraction of arithmetic operations provided great flexibility to explore approximations, but had a relevant drawback: the performance of the implementation made it infeasible to test complex datasets. Using FP16 only for the representation and performing all operations with native FP instructions would be an option for better performance, but would not allow these trials to be considered as equivalent to standard half precision hardware.

In order to prepare the library for the changes and experiments presented in the next chapters, variables used in the entire implementation were split in the following groups, whose types are selectable at compile time:

Group 1 variables used for external interaction (dataset loading and saving, ANN persistence, real time statistics etc) fixed in hardware native floating points

Group 2 variables used directly by the neurons (weights, steepness, neuron inputs and outputs) and the operations involving only these variables (except activation functions and derivative functions, which belong to **Group 3**)

Group 3 all other variables involved in the forward and backward phases of the ANN execution (including all parameters and operations of the training algorithms)

Disregarding **Group 1** and analyzing the variables that are relevant for the actual ANN operation, the rationale for the division was as follows. Due to the known resilience of the inference phase, in the **Group 2** were included the variables participating in this process. These variables, especially the connection weights, are the most important for memory usage (space and traffic) in the forward phase for large fully-connected networks. Additionally, regarding **Group 2**, it should be noted that representing the training data in this format may lead to a significant contribution of these variables to the overall gain with the approximation, due to lower memory usage. Some variables used in the inference phase do not require a considerable amount of space, but they remained in this group in order to reduce the need for conversions, since they are frequently used in *feed-forward* operations. As an exception, activation functions were not included in **Group 2**, because their relevance in resource usage decreases with the network size and complexity: since the activations are executed only once for every full sum-of-products input accumulation and their resources are fixed regardless of input fan-in, the more connections each neuron has, the less relevant the activation cost will be when compared to the of sum-of-products in the inputs. Additionally, for the preliminary studies, specific approximations for activation functions could be explored without mixing the effects of LUT-based implementation, for example, with the errors related to representation accuracy. With this reasoning and the irrelevant variables in **Group 1**, the **Group 3** was already

defined: it held the variables more directly related to the training process, which could require more precise representations.

Internally in the library code, the variable groups were associated to specific types defined at compile time. With the downside of resulting in more complex internal source code, this design decision allowed the use of the same library for native and simulated compilations without compromising the user visible code. A short segment extracted from the incremental weight adjustments code is presented in Listing 4.1, where **fann_type_ff** and **fann_type_bp** refer respectively to variables in the **Group 2** and **Group 3**, as defined previously. Two arithmetic operations (*add* and *mac*) are shown with the necessary type conversions. All calls prefixed with **fann_bp** or **fann_ff** are actually macros which may refer to native C code (e.g. operations for FP types) or simulated functions for various approximated implementations, defined at compilation time. Depending on the actual format being simulated, these functions can be implemented with the help of native FP operations or entirely in emulations based on integer types.

Listing 4.1 – Internal example of FANN variable types and operations

```
fann_type_bp tmp_error , delta_w , *weight_slopes ;
fann_type_ff *weights , learning_momentum ;

/* ... */
delta_w = fann_bp_mac( fann_ff_to_bp( learning_momentum ) ,
                      weight_slopes[w] , tmp_error ) ;
weights[w] = fann_bp_to_ff( fann_bp_add( delta_w ,
                                         fann_ff_to_bp( weights[w] ) ) ) ;
```

The code snippet presented in Listing 4.2 shows how the application programming interface provided by the library allows high level definitions of ANN implementations, and is not affected by the internally defined types and operations. The internal complexity of the type conversions and simulation is transparent to application code. Besides the actual numerical representations and respective arithmetic operations, which are fixed at compilation, all other ANN aspects are defined at runtime. For this reason, each simulated format is compiled to a different binary, using the same code base.

Listing 4.2 – Simple example of FANN application code

```
struct fann_data *train_data , *test_data ;
struct fann *ann ;

/* read dataset folds for training and test purposes */
train_data = fann_read_data_from_file( "dataset.train" ) ;
```

```

test_data = fann_read_data_from_file("dataset.test");

/* creates the main ANN structure */
ann = fann_create_standard_args(threads, num_layers,
                                train_data->num_input,
                                num_neurons_hidden,
                                train_data->num_output);

/* normalized initialization of weights */
fann_init_weights(ann);

/* optional callback function to manage the training process */
fann_set_callback(ann, train_callback);

/* ANN hyper-parameters */
fann_set_activation_function_hidden(ann, FANN_RELU);
fann_set_activation_function_output(ann, FANN_SIGMOID);

/* Depending on the training algorithm, other
   hyper-parameters are available */
fann_set_training_algorithm(ann, FANN_TRAIN_RPROP);

/* This function returns when training stop criteria is reached
   or when the callback function explicitly interrupts it. */
fann_train_on_data(ann, train_data, max_epochs,
epochs_between_reports, desired_error);

/* Fills the ann structure with test error statistics.
   May be called during training from the callback. */
fann_test_data(ann, test_data);

```

Weight initialization was a factor kept constant among all tests and datasets. Each random seed was repeated once in the FP64 and FP16 runs. This assured that, for every execution, both representations started from the same random weights (apart from the reduced numerical representations). Since all trials were trained with the same order of examples using the same algorithms and hyper-parameters, this procedure allowed a more direct comparison of the approximations effects in the training epochs.

No regularization component or stop criteria was defined for all the comparisons in this study. Hyper-parameters and network topologies were defined based only on the FP64

runs and repeated in the FP16 tests. These experimental design decisions associated with paired weight initializations provided a more direct way to graphically compare training evolution and generalization capacity of the implementations. Throughout the following chapters, paired plots of the train and test average accuracies of two ANN implementations will be compared to analyze several differences in the training progress. An important fact is that maximum test accuracies should not always be compared between different graphs because they may happen at different epochs, causing the averaged value to be lower if the network tends to overfitting. For this reason, confidence intervals differences may also provide important informations.

To avoid performance differences in datasets due to class imbalance and also to show more clearly possible effects on the minority classes, the reported accuracy is the average of individual classes, and not the global one. Equivalently, for binary classification problems, the accuracy is the average between True Positive Ratio (TPR) and True Negative Ratio (TNR). In the final tests, presented in Chapter 6, a unified approach was used, by calculating the geometric mean of individual class accuracies, which captures more aggressively the “catastrophic forgetting” of minority classes. Graphical comparisons will always be presented with 95% confidence intervals for the average on each epoch.

Table 3 summarizes the main characteristics of the classification datasets selected for this preliminary study. They were chosen from common benchmarks which have been used for a long time (PRECHELT et al., 1994) to explore a mixture of following parameters: set size, input complexity and number of classes (mutually exclusive). Class imbalance also varies and will be analyzed later, when the approximated version is evaluated. All datasets are freely available from open repositories like (LICHMAN, 2013) and usually accompany ML software packages, like FANN. MNIST was obtained directly from the original Yann LeCun’s website: <<http://yann.lecun.com/exdb/mnist/>>.

Table 3 – Characteristics of the benchmark datasets

Dataset	Inputs	Outputs	Training Set Size	Testing Set Size
MNIST	784	10	60000	10000
Breast Cancer	30	1	455	114
Thyroid	21	3	3600	3600
Soybean	82	19	342	341

Regarding the training methods, the following comparison was based on simple standard back-propagation in batch modes (mini-batch for the larger datasets and full-batch for the two others). Not even incremental SGD with a momentum factor was included in this phase. Adaptive algorithms, more robust to hyper-parameter changes and with faster convergence, like iRProp (IGEL; HÜSKEN, 2000), RMSProp (TIELEMAN; HINTON, 2012) or Adam (KINGMA; BA, 2014) will be considered in the next chapters.

The following comparisons are based on two standard FP representations, already defined at the beginning of this section, allocated to the variable **Groups 2** and **3**. The purpose was to analyze the effects of lower resolution arithmetic in simple training methods, even before further approximations were introduced.

Figure 10 depicts the average training behavior for the two datasets (MNIST and Soybean) that did not present significant difference between the two representations. Both the generalization pattern and the accuracy spread were very similar. No relevant difference in convergence speed was observed and also no trial failed to converge.

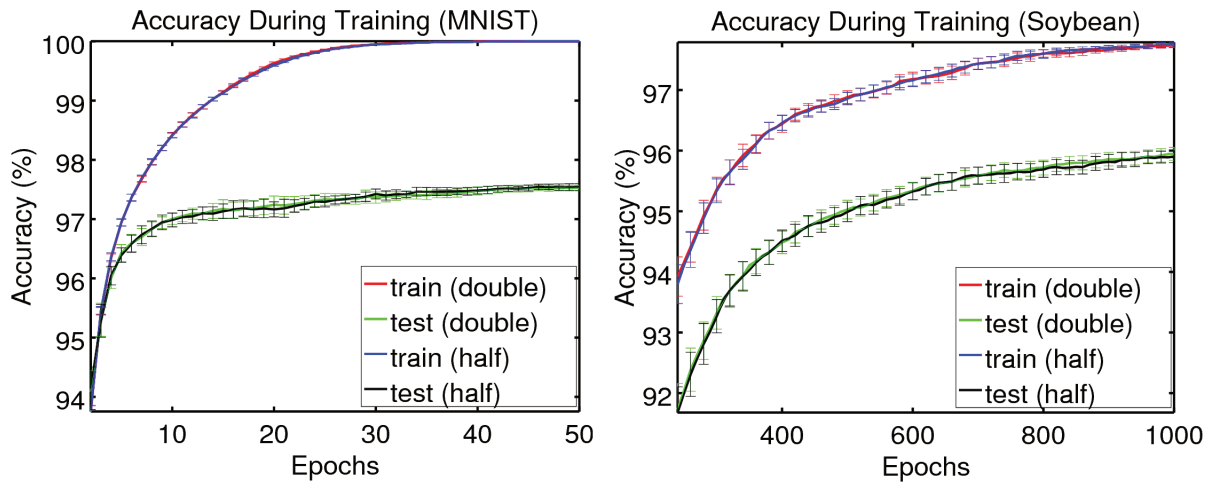


Figure 10 – Training Process Comparison between half and double precision FP representations for the MNIST and Soybean datasets

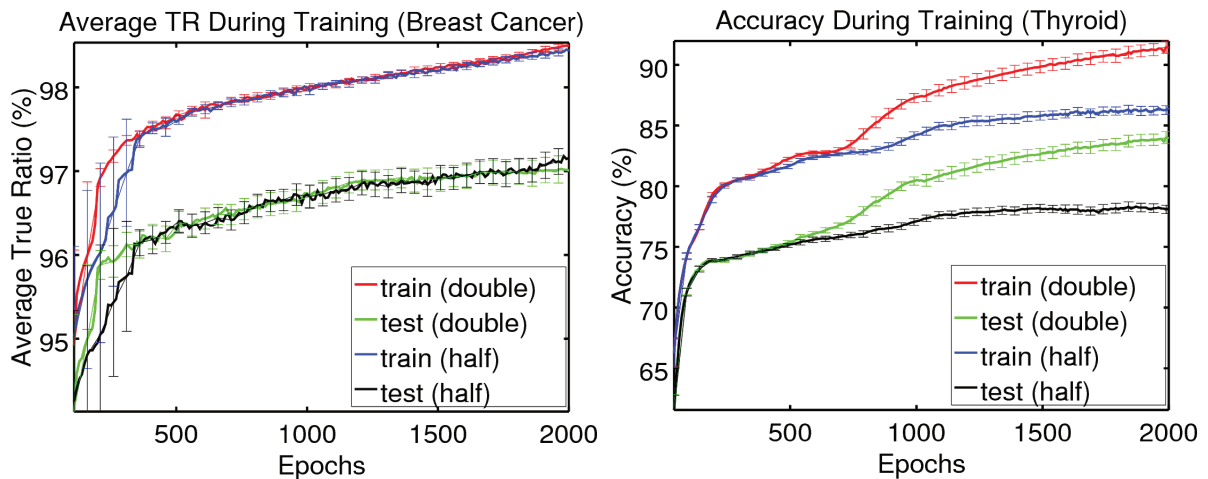


Figure 11 – Training Process Comparison between half and double precision FP representations for the Breast Cancer and Thyroid datasets

Figure 11 exhibits the training process for two datasets which behaved differently with lower precision. The larger spread in the first epochs for the Breast Cancer dataset is due to different convergence speeds between trials, but the average true ratio converges to similar values at the end. A single initialization failed to converge and this happened

to both precisions. The Thyroid dataset presented a different behavior: the learning process stagnated in the less precise (FP16) version and kept progressing in the reference implementation. There is a possibility that setting a different a learning-rate could provide better results for FP16 in these datasets, but this would violate the premise of performing comparisons under the same conditions. Additionally, the evidence that the lower precision ANN was not equivalent in these cases can not be disregarded.

As it can be seen in these comparisons, FP16, which is sometimes used as a baseline for approximation studies, is not a direct “drop-in” replacement for more precise FPs in ANN training. Even considering that hardware manufacturers like NVIDIA show impressive results using their powerful GPUs, implementation details should be analyzed with care. These implementations may include a mixture of FP32 operations, sometimes used only internally, or specific ANN architectures and training algorithms adjusted to perform well with low precision arithmetic. Since this work aims at optimized hardware ANN implementations, it is an important factor to observe that even before further simplifications are added to the system with reduced precision, attention is required to provide reliable operation.

5 Simplifications and Approximations

In Chapter 4, a reduced standard FP representation was presented as a first step to optimize the ANN training process (by reducing its memory footprint) and simplify the FPU (by using fewer circuits or iteration steps in basic operations). This Chapter introduces further optimizations that simplify the hardware design by eliminating exceptions to this reduced FP format (Section 5.1) and approximating higher level mathematical operations frequently used in ANNs (Section 5.2). Section 5.3 repeats the same test procedures already detailed, replacing the double FP baseline by the IEEE FP16, to verify if the approximated implementation with extra simplifications introduced further performance penalties. All approximations proposed in this chapter are based on the FP16 format, using the environment described in Section 4.3.

5.1 Floating Point Simplifications

As previously mentioned, the lowest and highest exponents in the IEEE FP representation are used to represent exceptional conditions. Removing these exceptions has a few direct beneficial effects and some changed behaviors regarding the lack of indication of such occurrences. This section analyzes the rationale and effects of these modifications.

5.1.1 Removal of Infinities and NaNs

By replacing the highest exponent with an extra set of normal numbers, 1024 new values become representable in the FP16 format. This means that the largest magnitude is doubled (from 65504 to 131008, considering the default bias). This improvement represents a mere $\approx 1.6\%$ increase in the representable numbers set which, with the default bias, is outside the most useful range for ANNs since models are typically initialized with very small weights (positive and negative). No relevant FPU simplifications result from this change since the overflow condition must still be detected and the maximum positive (or minimum negative) number be returned as a result. Conversely, infinite arguments do not have to be checked at the operation inputs.

What is more relevant for reliability is that in the same situations when infinities would occur, the highest possible number is returned as the operation result. This can be viewed as a form of “graceful degradation”, since invalid numbers that would interrupt the learning process are replaced by valid values. If this operation was, for example, in the gradient calculation, the result could be just a smaller learning adjustment step in one weight, instead of a failed learning sequence. The fact that the migration from FP32

to FP16 drastically reduces the largest representable number, potentially increasing the likelihood of overflows, is partially mitigated by this conservative behavior. Some adaptive training algorithms even limit the largest weight adjustment step as a hyper-parameter.

The IEEE standard is very specific about the invalid operations that should result in NaNs, also making distinctions between the quiet and signaling types. The simple removal of infinities eliminates many of these situations. The approximate implementations of the reciprocal square root functions (detailed in Section 5.2) associated with their specific use in this research, eliminates another possibility. The simplicity of ANNs, which requires only a subset of the standard operations, narrows down the remaining possibilities to a single one: the division of zero by zero. This operation occurs in only two situations in all analyzed methods: the Softmax output function and some adaptive methods which dynamically adjust the learning speeds of each weight. The first situation can be reasonably handled by the mathematical absurdity $0/0 = 0$, which makes sense in this case, since no output was activated, but for training purposes the error will be high for the one which was not activated. The second situation can be easily circumvented by testing the denominator, which also avoids the division by zero.

Besides eliminating NaNs and infinity representations, some situations in which exceptions occur may also set hardware flags: inexact result, underflow, overflow, infinite and invalid operation. All the tests required to generate these flags were also removed. The inexact result is the only condition not previously analyzed. It means that the returned value could have been more precise if more *bits* were available in the format (in other words, if the value was rounded), which is useless for this type of application. When grouped, the simplifications detailed so far lead to considerable resource savings, as shown later in this section.

5.1.2 Restrict Format to Normalized Numbers

The removal of the subnormal number representation exception, and the use of its exponent to represent normalized numbers, has a direct and minor effect similar to the previous one: the smallest normal magnitude in the FP16 format is reduced by half (from 6.1095×10^{-5} to 3.0547×10^{-5} , considering the default bias). Taking into account that these new normal values replace the subnormal ones, the limit is actually worsened since the smallest subnormal magnitude is 5.9605×10^{-8} . It should not be forgotten that this mode of operation is a form of “graceful degradation” applied to the underflow situations. Not only there is a performance penalty in their use, but the FPU must have extra complexity to handle this exceptional situation. Other undesirable characteristic is that the smaller the number of effective *bits* in the subnormal significand (which is smaller than 1.0, thus has leading zeros) the larger the relative error of the operations. This leads to the conclusion that, even if supported, subnormal numbers should be avoided.

The problem with the reduced range of small magnitudes in FP16, aggravated by the removal of subnormals, is that such low values are common in ANNs, especially considering the weight initialization values. Even simple numerical techniques, like adding a small value (e.g. 10^{-7}) to a positive denominator before making a division in order to avoid a division by zero, become problematic. Additionally, mixed precision approaches implemented in standard hardware may rely on this exception as an extended representation range, opening the possibility for a performance cost. Conversely, in FPGA implementations the subnormals are normally not included due to resource usage, as can be observed in two commercial and two academic examples: Altera (ALTERA, 2016), Xilinx (XILINX, 2017), FloPoCo (DINECHIN; PASCA, 2011) and VFloat (WANG; LEESER, 2010).

In order to exemplify the order of magnitudes seen in typical weight adjustments during training, Figure 12 depicts trials of the MNIST dataset using the same initialization, with the conditions detailed in Chapter 4, comparing FP16 without subnormals and FP32 executions. The plots compare the maximum (green and black), which are very similar, and the average (red and blue) absolute values for the weight changes in each layer during training. Layer 3 (Output) is omitted due to its similarity with Layer 2. The minimum non-zero absolute values are not represented since they are very small in the FP32 execution (lower than 10^{-10} and frequently reaching values that are subnormals even in this format).

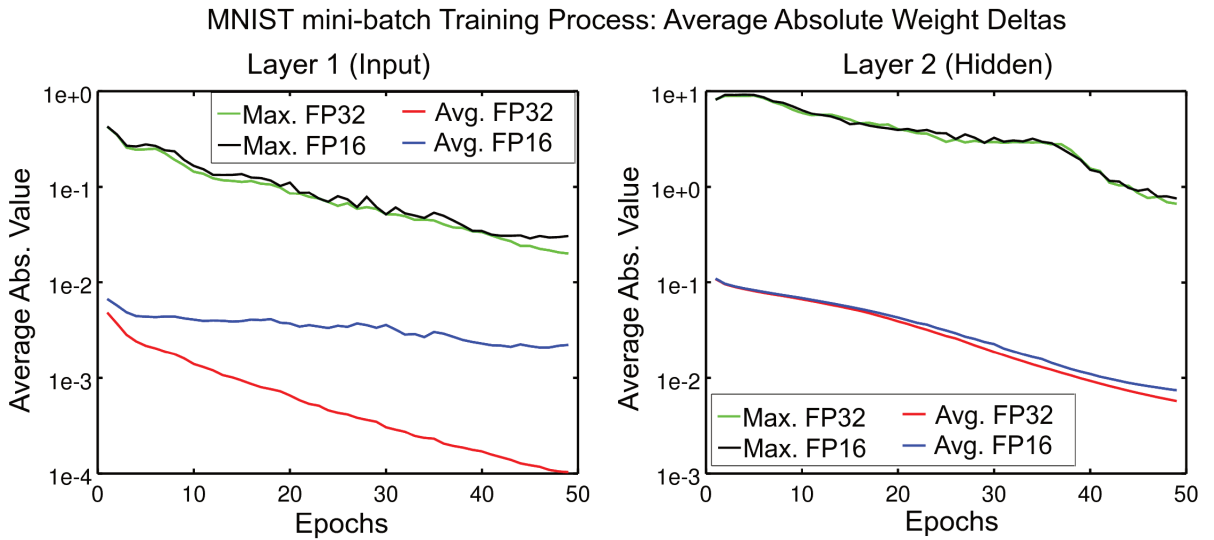


Figure 12 – Average and maximum absolute values for weight adjustments

It is worth emphasizing that even with a dataset (MNIST) in which no relevant difference was found in the average performance (progress in training accuracy and generalization), noticeable differences are verified in the weight adjusts. When just the average absolute values are compared, the differences are only significant in the first layer, which has the smaller adjustments, where they differ by more than an order of magnitude by the end of the training. This is an indication that, for this problem and in these training

conditions, the small weight changes missed in the FP16 format were overwhelmed by other dataset characteristics, like redundancy. It should also be noted that the low magnitudes (when compared to the FP16 representation limits) imply that the subnormal range of FP16 would have been frequently used in this process, especially at the end of the training in the first layer, when even the average value got close to the smallest magnitude limit.

Restricting the representation and arithmetic operations to normalized numbers has many direct beneficial effects, as already mentioned, but also an indirect one. When the operations are performed, the possibility of a subnormal value as a result has to be considered, which requires the internal number representation to have enough precision to generate such results. Removing the circuits (or code) to handle subnormal numbers also gives room to reduce the internal bit representation in some operations. This fact was explored in the optimized implementation and the effects are presented in the following section.

5.1.3 Preliminary Implementation of FP arithmetic

Even before the final hardware implementation, a preliminary evaluation of the gains obtained with the described simplifications was performed. Though the results of this analysis can not be directly extrapolated to hardware FPUs, they suggest that the effects of the simplifications were significant in this proof of concept. A similar gain could be observed in very low end CPUs, which base their FP implementations on libraries that normally use native integer arithmetic internally, like the one used in these tests. Direct FPGA implementations may also yield similar benefits, since these components provide internal DSP units capable of performing integer FMA operations.

The reference “SoftFloat” IEEE compliant implementation was firstly trimmed to include only the functions used by the modified FANN library (the contents of the methods were not modified). The resulting code is the library used to run the tests with Standard IEEE FP16 and resulted in a compiled binary ¹ with 125,264 bytes. The first step for the approximation operations was the removal of exceptional conditions and

¹ For compilation off all binaries, all optimizations from level 1 were selected (with -O1) but the flags from levels 2 and 3 were hand-picked (instead of using -O2 and -O3):

```
-finline-functions -funswitch-loops -fpredictive-commoning -fgcse-after-reload
-fipa-cp-clone -ftree-loop-distribute-patterns -ftree-slp-vectorize
-fvect-cost-model -ftree-partial-pre -ftree-pre -fstrict-aliasing
-fstrict-overflow -fgcse -finline-small-functions -fthread-jumps -falign-functions
-falign-jumps -falign-loops -falign-labels -fcaller-saves -fcrossjumping
-fcse-follow-jumps -fcse-skip-blocks -fdelete-null-pointer-checks -fdevirtualize
-fexpensive-optimizations -ftree-vrp -fgcse-lm -fhoist-adjacent-loads
-findirect-inlining -fipa-cp -fipa-sra -foptimize-sibling-calls -foptimize-strlen
-fpartial-inlining -fpeehole2 -freorder-blocks -freorder-blocks-and-partition
-freorder-functions -frerun-cse-after-loop -fsched-interblock -fsched-spec
-fschedule-insns -fschedule-insns2 -ftree-builtin-call-dce -ftree-switch-conversion
-ftree-tail-merge
```

flushing the subnormal numbers to zero. At this point a reduction of almost 40% in size was already observed. Further optimization possibilities resulting from the removal of subnormal numbers were identified, which simplified the implementation of some arithmetic operations. The final version of the approximated library, with the exact same API and supported methods, reached a size of 59,088 bytes, representing $\approx 53\%$ of size reduction. Stress tests were performed with all arithmetic operations in both libraries in order to identify any abnormal behavior. The arguments for each operation (two or three) were generated randomly, each operation was performed and the results were compared. Different results would interrupt the test sequence and the ones that increased the maximum relative error found so far were displayed for inspection. The maximum observed error was $\pm 0.05\%$ in both libraries, as expected from the significant representation length and rounding method. These figures exclude the operations involving subnormals which are performed with worse resolution. For each operation the maximum error was reached after a few thousand operations but the stress tests were executed hundreds of millions of times for each one, increasing the confidence in the correctness of both implementations.

Optimized FPGA implementations should be performed directly with an appropriate hardware implementation tailored language, but a preliminary evaluation could also be obtained using the Vivado High-Level Synthesis tool, offered by Xilinx. With this method, the libraries source codes are used as inputs in the design process. A full FPU was not implemented using this method, but the most important operations were compared: FMA, multiplication and addition (which also includes the subtraction). The comparison between the results provided by the tool are presented in Table 4, where the rows identified as “Total” summarize the resource usage. Relative reductions from 26% to 45% were observed for the LUTs and Flip-Flop (FF) resources. Due to inherent code obfuscation techniques of the Synthesis tool it was not possible to determine the reasons for the considerable reduction in DSP usage (from 16 to 1, in the DSP48E columns) and the higher RAM usage (from one to two blocks, in the BRAM_18K columns). The analysis shows that relevant benefits were still present when the software designs were migrated to the FPGA. It is important to note that an efficient FPGA implementation approach would take advantage of common blocks sharing (such as the final packing and rounding module) and also from the component peculiarities, like the *bit* width of the internal DSPs. Another important observation is that when a full ANN implementation is considered, the FPU is only part of the whole architecture and these benefits are expected to be diluted.

This research has not yet evaluated alternative rounding solutions, fixing its implementation to the most conservative (and expensive) choice: round to the nearest value. There are less complex options, like the simple elimination of extra *bits*, which doubles the maximum error due to rounding and, more importantly, introduces a bias into that error. Stochastic rounding has also been analyzed in the literature as an option to accumulating long series in values with higher precision.

Table 4 – FPGA resource comparison after synthesis using the Vivado HLS

Standard IEEE FP					Approximate FP				
Name	BRAM_18K	DSP48E	FF	LUT	Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-	DSP	-	1	-	-
Expression	-	-	-	-	Expression	-	-	2101	2274
FIFO	-	-	-	-	FIFO	-	-	-	-
Instance	1	16	5455	5125	Instance	1	-	761	658
Memory	-	-	-	-	Memory	1	-	0	0
Multiplexer	-	-	-	13	Multiplexer	-	-	-	170
Register	-	-	3	-	Register	-	-	442	-
Total	1	16	5458	5138	Total	2	1	3304	3102

FMA

Name	BRAM_18K	DSP48E	FF	LUT	Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-	DSP	-	1	-	-
Expression	-	16	495	1034	Expression	-	-	336	419
FIFO	-	-	-	-	FIFO	-	-	-	-
Instance	-	-	833	845	Instance	1	-	761	658
Memory	1	-	0	0	Memory	1	-	0	0
Multiplexer	-	-	-	185	Multiplexer	-	-	-	64
Register	-	-	369	-	Register	-	-	101	-
Total	1	16	1697	2064	Total	2	1	1198	1141

Mul.

Name	BRAM_18K	DSP48E	FF	LUT	Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-	DSP	-	-	-	-
Expression	-	-	0	18	Expression	-	-	0	18
FIFO	-	-	-	-	FIFO	-	-	-	-
Instance	1	-	4123	4179	Instance	3	-	2968	2770
Memory	-	-	-	-	Memory	-	-	-	-
Multiplexer	-	-	-	35	Multiplexer	-	-	-	53
Register	-	-	13	-	Register	-	-	101	-
Total	1	0	4136	4232	Total	3	0	3069	2841

Add.

An improvement to achieve resilient training of ANNs is to dynamically adapt the FP range for each problem and network structure by adjusting the exponent bias. Since even standard FP16 numbers showed mixed results in the first tests, further reducing the representation ranges, by restricting the format to normalized values, may compromise even more the performance with some problems. The implementation of such feature is relatively straightforward, but the adjustment mechanism may be implemented at several levels and granularities, with different complexity compromises. The evaluations could be performed at each batch, mini-batch or even for each training example. The adjustments could be applied to each neuron, layer or to the whole network, avoiding the conversion operations. Non-automatic adjustments are not desirable, since they add hyper-parameters to the training.

5.2 Math Operations Approximations

As mentioned in Chapter 4, LUT based implementations for the most complex activation functions used in ANNs are a common way to optimize their hardware implementations. Optimizing a basic mathematical operation used in the activation function

definition is another option. This approach has the benefit that the same operation, for example the exponentiation, can be used in other functions. In this case, besides the Softmax output activation, the exponential could also be used to decay hyper-parameters in some adaptive training methods. The other operation approximation will merge a division and a square root into a single method, which is also useful in some algorithms.

The first two methods presented in this section are adaptations to the simplified FP16 format of approximation techniques proposed several years ago. The first technique (SCHRAUDOLPH, 1999) was associated with ANNs from the beginning, but the second one appeared in the context of the early 3D gaming platforms (KUSHNER, 2002) without being directly related to a scientific publication. These techniques share a common background of computing platforms that performed much better with integer operations than with floating points. For specific resilient applications (like ANNs and fast 3D rendering) they benefited directly from the FP representation format (BLINN, 1997) to obtain better performance. It is interesting to see how these two fields have been sharing performance techniques for a long time and this trend continues as so many important ML results nowadays rely on powerful GPUs. Finally, a third approximation is proposed with similar techniques. It replaces the reciprocal operation in the second method by a full division (taking the two arguments). Interestingly, the implementation is even simpler, at the cost of a larger relative error.

5.2.1 Exponentiation and Activation Functions

The original idea for this approximation was defined for double precision representations. According to the author, it is equivalent to a LUT implementation with 2048 entries with interpolation. The half-precision modification proposed here, despite having a larger error, will behave similarly. This adaptation is based on the same principle but it uses all the 16 *bits* of the resulting FP16 number while the double precision version performs its main operation on the upper 32 *bits* of the FP number.

Figure 13 details how the FP16 format defined in Section 4.1.1 is represented. The FP format is intuitively an exponentiation operation with base 2 for small numbers. If a small integer number x , within the valid range $[-15, 14]$, is added to the exponent bias (15) and shifted 10 positions to the left, the resulting integer (i), interpreted as a FP number, equals 2^x . The shifting operation may be replaced by a multiplication by a power of two integer (in this case $2^{10} = 1024$), which results in the Equation 5.1.

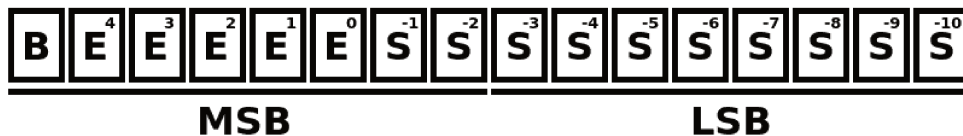


Figure 13 – Bit layout of the FP16 format

$$i = 2^{10}(x + 15) \Rightarrow i = 1024x + 15360 \Rightarrow i = (\text{int})(1024x_f + 15360) \quad (5.1)$$

If x_f is a fractional number (regardless of the representation) i becomes the conversion of the resulting number to an integer, discarding the fractional part. But in this case, the *bits* that are assigned to the significand part of the FP number may not be zero. This difference is what makes the linear interpolation between the integer values possible, and automatic. Remembering that the exponent part refers to the power of the first *bit* 1 in the significand, which is omitted, the next *bit* (S_{-1}) is worth 0.5, the next one 0.25, and so the sequence follows. This means that the lower order *bits* of the multiplied number will fill the gap between two integer exponents, proportionally to the fractional part of the input.

Two more steps are needed to conclude the approximation. The first is that the 2^x operation must be transformed in e^x . This is achieved by simply dividing $x/\ln(2)$, according to Equation 5.2, so that the resulting multiplication becomes $1024/\ln(2) \approx 1477.32$. Finally, a small adjustment should be performed in the sum with 15360, which results in a minor reduction of the maximum approximation error. Since the search space for FP16 is very small, this value was obtained with exhaustive search. The adjusted value found was 15320, resulting in the final expression for the simplification, given in Equation 5.3.

$$2 = e^{\ln(2)} \Rightarrow 2^{x/\ln(2)} = \left(e^{\ln(2)}\right)^{x/\ln(2)} = e^x \quad (5.2)$$

$$i = (\text{int})(1477x_f + 15320) \quad (5.3)$$

Figure 14 presents the final approximation results in a narrow range, compared to a precise double precision reference. The relative error curve repeats the same aspect in all the usable input range, which is: $-10.367 < x < 11.805$ (arguments outside this interval result in non-representable answers). For values outside this range the function returns 3.0756×10^{-5} and 1.3056×10^5 , respectively.

5.2.2 Reciprocal Square Root

This approximation method also takes advantage of the intrinsic exponentiation operation of the FP format, but for a different purpose. Like many other implementations of complex functions, it relies on an initial estimate for the result (usually obtained from a LUT) followed by some iterative refinement operations. The better the initial guess for the answer, the fewer iterative adjustments are required until the desired accuracy is reached. The original method relies on single precision FP numbers and this adaptation modifies it

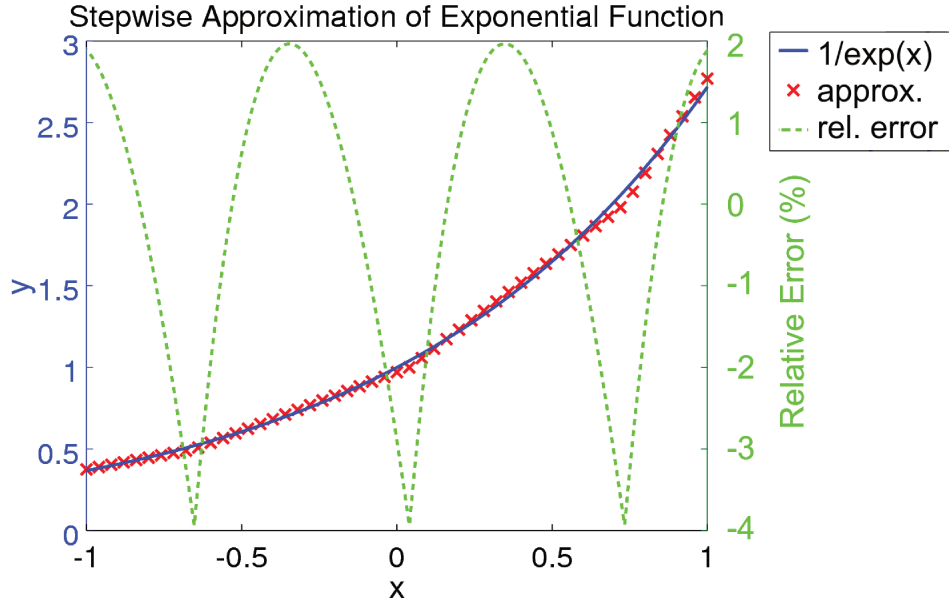


Figure 14 – FP16 exponentiation approximation

for the FP16 format. A very detailed analysis of the method was provided by the Technical Report (LOMONT, 2003), on which this adaptation was based.

The first part is the most interesting one in the proposed implementation. It uses the exponent representation to obtain an initial guess for the operation, without a LUT. Considering a number with the significand = 1.0, the reciprocal square root could be obtained by multiplying the exponent by -0.5 . Integer division by 2 is quite simple since it can be implemented with a single right shift, but this operation has to cope with two issues: the FP exponent is biased and if the value is odd the LSB from the exponent will be assigned to the MSB of the significand. The sign *bit* is not a problem since the reciprocal square root is only used for positive numbers in the targeted ANN methods². Equation 5.4 details the operation and the final exponent encoding in which E is the initial exponent and E_R the resulting one.

Reciprocal Square root of numbers with significand = 1.0:

$$\left(2^{E-15}\right)^{-1/2} = 2^{(15-E)/2} \quad (5.4a)$$

$$E_R = \frac{15 - E}{2} + 15 = 22.5 - \frac{E}{2} \quad (5.4b)$$

If exponents were represented in fixed point, with a single fractional *bit*, this simple operation, composed by a shift and a integer subtraction would obtain an exact answer for pure exponential arguments. Instead, lets consider a subtraction from 22 (which, encoded in the appropriate position would be 0x5800), slightly underestimating the answer. Two cases must be analyzed: even argument exponents and the odd ones, on which the shift causes the

² In both RMSProp and Adagrad the argument for the reciprocal square root is a sum-of-squares. The condition with the argument = 0 must still be handled.

exponent LSB to become MSB of the significand. Figure 15 shows how estimates for even exponents (starting from 0) are underestimated and the other ones become overestimated due to the significand changing to 1.5 (due to *bit* shifting). These estimates are represented as crosses, above and below the reference reciprocal square root value, considering only integer exponents and the operations just described, affecting the mantissa value.

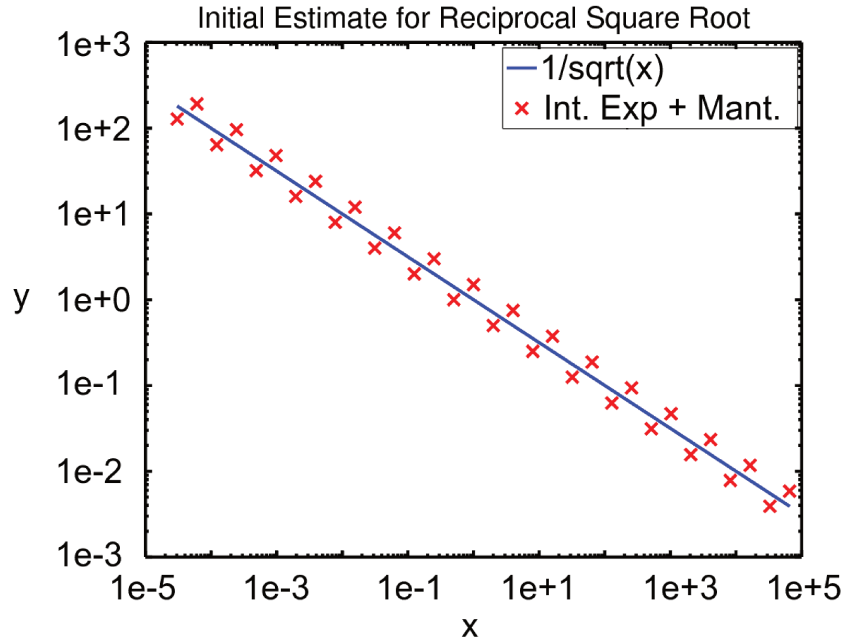


Figure 15 – Approximations for powers of 2

Similarly to the exponential approximation, the argument significand will provide an interpolation between the whole exponents, but it will not be a linear one and will also improve the approximation in these points. The best value for the initial subtraction was adjusted by exhaustive search ³, minimizing the absolute error on the initial guess: 0x59BB.

The second part of the approximation is a single iteration of the Newton-Raphson ⁴ method. If x_0 is a first guess for x that makes $f(x) = 0$, Equation 5.5 indicates how a better value for x (x_1 , for iteration 1) could be estimated by using the first terms of a Taylor approximation for $f(x)$.

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \quad (5.5)$$

³ All subtraction values from 0x5800 to 0x60FF were tested and, for each one of them, all possible arguments (positive numbers) were evaluated. For each subtraction value, the maximum relative error resulting from all arguments was noted. The subtraction value which provided the better absolute maximum error in the initial guess was used.

⁴ <http://mathworld.wolfram.com/NewtonsMethod.html>

In order to use the Newton-Raphson, the reciprocal square root calculation must be transformed into a root-finding problem, based on the efficient initial estimate detailed previously. For an argument $\mathbf{a} > 0$, we have to find a value x that gives us $x = 1/\sqrt{\mathbf{a}}$, starting from a first approximation $x_0 \approx 1/\sqrt{\mathbf{a}}$. We define $f(x) = x^{-2} - \mathbf{a}$ which will reach $f(x) = 0$ for the correct answer. Using the first derivative for $f(x)$ and applying it to the Equation 5.5, we define an analytical expression for the first iteration of the Newton-Raphson method. This is the most computationally intensive part of the reciprocal square root approximations since it is regularly performed as floating point operations: 4 multiplications and one subtraction, as indicated by Equation 5.6.

$$f(x_0) = (x_0)^{-2} - \mathbf{a} \Rightarrow f'(x_0) = -2x_0^{-3} \quad (5.6a)$$

$$x_1 = x_0 - \left(\frac{x_0^{-2} - \mathbf{a}}{-2x_0^{-3}} \right) \quad (5.6b)$$

$$x_1 = x_0 - \left(-0.5x_0 + 0.5\mathbf{a}x_0^3 \right) \quad (5.6c)$$

$$x_1 = x_0 \left(1.5 - 0.5\mathbf{a}x_0^2 \right) \quad (5.6d)$$

In the previous explanation the variable x was used in a more adequate way to relate the equation definition to the Newton-Raphson method, which is a root-finding algorithm. If the first part of the explanation (the *bit* shift and integer subtraction operations to obtain the first estimate) is merged with the numerical method iteration just defined, a final expression can be obtained (Equation 5.7) for the complete reciprocal square root approximation. The approximation $Y = 1/\sqrt{X}$ is then calculated in two steps, considering that in the first part the argument X is handled as a 16 *bits* integer and in the second part as a regular FP16 number, as well as G , the first estimate for the numerical method.

$$G = 0\mathbf{x}59\mathbf{BB} - (X \gg 1) \quad (5.7a)$$

$$Y = G \left(1.5 - 0.5XG^2 \right) \quad (5.7b)$$

Figure 16 depicts a precise reference for $y = 1/\sqrt{x}$ in blue. On the left, this reference is compared to the first approximation G (in red), obtained by the first step and interpreting G as a FP number. On the right, the same reference is compared to the full operation, which includes one step of the numerical approximation. The relative error patterns repeat themselves in the whole usable input range ($6.1095 \times 10^{-5} < x < 3.3952 \times 10^{+4}$).

This approximation will not be tested on the comparisons that close this Chapter, since it is not used in the simple training methods presented here. Due to the low precision requirements of some resilient applications, it is worth noticing that even the first approximation step already results in similar relative errors to the ones obtained at the exponentiation approximation.

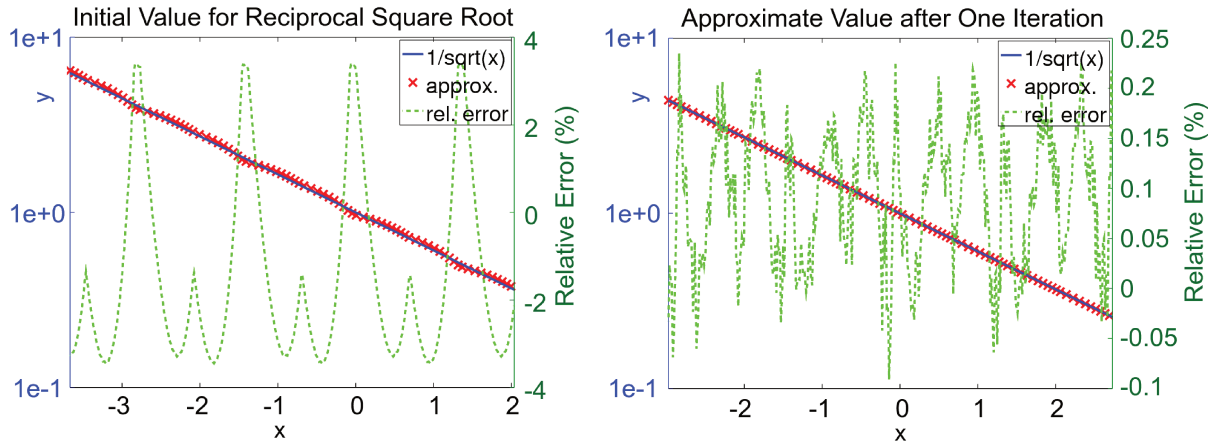


Figure 16 – Reciprocal Square Root Approximations

5.2.3 Division by Square Root

Extending the same ideas used in the previous approximations, another technique that includes a division in the operation is proposed, giving an approximate answer for the expression $y = b/\sqrt{a}$. Additionally, this is implemented in a way that accepts dynamic FP bias values. The rationale for the new method is similar to the other two adapted approximations, by initially considering only power of two arguments (both significands have only zeroed *bits*). The resulting encoded exponent (E_y) can be obtained by direct operations from the argument's encoded exponents (E_a and E_b). The equation (5.8) details the operations, where FP bias is also considered variable and is represented as **BIAS**.

$$E_y - \text{BIAS} = E_b - \text{BIAS} - \left(\frac{E_a - \text{BIAS}}{2} \right) \quad (5.8a)$$

$$E_y = E_b - \frac{E_a}{2} + \frac{\text{BIAS}}{2} \quad (5.8b)$$

Divisions by two are efficiently implemented as 1 *bit* shifts. Similarly to the other methods, the exponent operations are not performed separately. Instead, the arithmetic and shift operations are performed with the whole represented FP numbers, including the significands. The same error compensation explained in Section 5.2.2 happens when the fractional LSB of the exponent's operation “leaks” into the significand. Both significands also provide a interpolations without extra cost and a negative argument b preserves its sign *bit* due to the shift operation in the a argument (which is always positive, as mentioned previously).

Fig. 17 depicts, in a limited range, the relative error for the approximated $y = b/\sqrt{a}$ operation, using logarithmic scales for the two arguments, and a double precision FP standard implementation as reference. Invalid arguments are not considered since they are handled before the approximation, returning either zero or the maximum representable number (with the same sign as b). Not considering these underflow and overflow conditions,

the relative error limits in the entire useful output range (including exponent biases varying from 15 to 31) are $\approx \pm 8.866\%$.

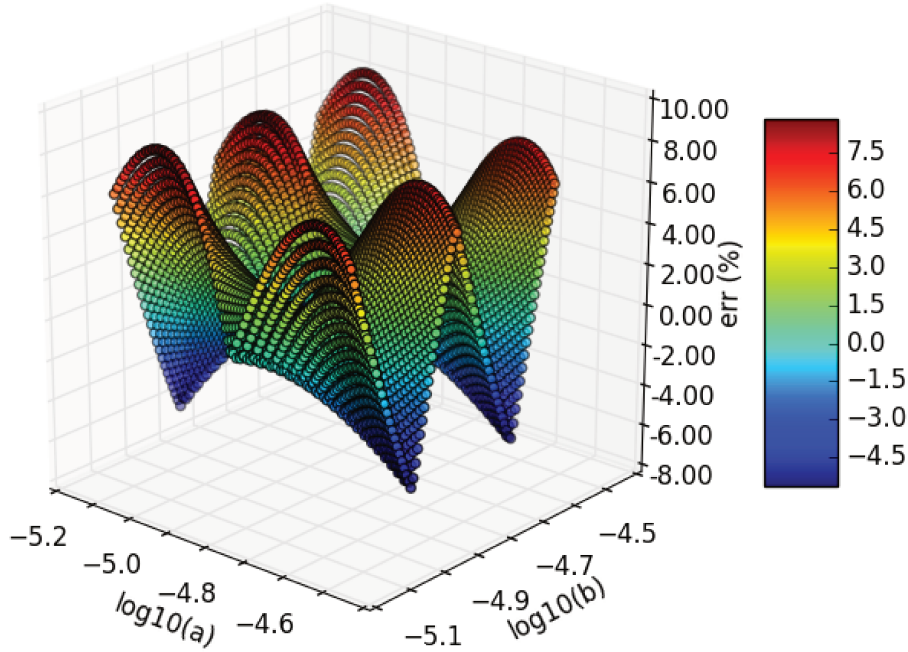


Figure 17 – Division by Square Root Approximation

The Newton-Raphson method is very effective in reducing the initial approximation error, but it is computationally expensive: each step performs 5 multiplications, one division and one subtraction. Fig. 18 depicts how two numerical steps reduce the maximum approximation error. For each value of a , calculated with FP16 BIAS = 24, the largest errors for all possible values of the argument b which lead to valid results are tested. Both steps are performed with single precision operations. As in the previous example, this numerical method was not included in the final approximate implementation.

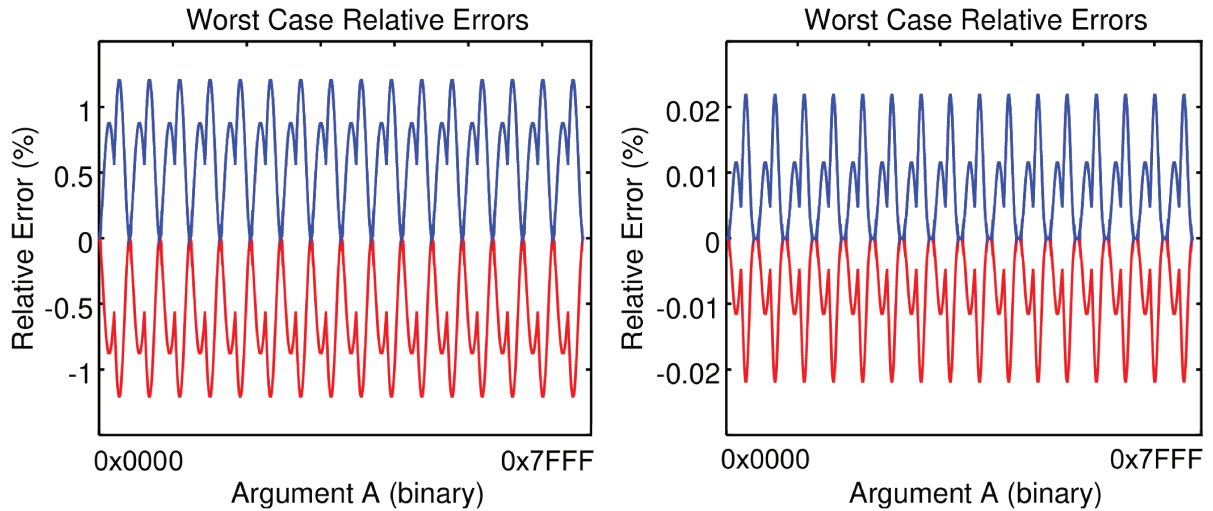


Figure 18 – Division by Square Root Approximation: 2 Newton-Raphson steps

5.2.4 Preliminary Implementation of Complex Functions

Similarly to what was reported in the Section 5.1.3, a preliminary viability analysis was performed to verify if the proposed complex function approximations resulted in significant simplification. Since the SoftFloat library does not provide a precise exponentiation function for comparison, the synthesis of the FMA operations were repeated as references. This new synthesis was necessary for a fair comparison because, at this stage, a newer Xilinx Vivado version (2018.3) was used, which produced better HLS results in both the baseline and approximate version. Additionally, the approximated multiplication code with variable FP bias, mentioned in Section 5.2.3, modifies these approximated source code implementation basis.

The synthesis results are presented at Table 5. When compared to Table 4, the same columns with non-zero values are shown but only the rows previously identified as “Total” are presented for each entry. Like in the previous analysis, the Softfloat original code is used as the HLS input for the “Standard IEEE FP” synthesis rows while the other ones are generated from the approximated code. The direct comparison shows that the exponentiation approximation is even simpler than the approximated FMA operation. It is also clear that the division by square root, including the variable FP bias, is so simple it could be implemented only with combinational logic.

Total Resource:	BRAM_18K	DSP48E	FF	LUT
FMA: Standard IEEE FP	1	16	2079	6062
FMA: Approximate FP	2	1	892	3426
Exp. Approximate FP	2	1	598	2177
sqrt(): Standard IEEE FP	1	34	1751	2763
b/sqrt(a) Approximate FP	0	0	0	157

Table 5 – High Level Synthesis of Complex Functions

5.3 Comparative Tests: Standard IEEE FP16 vs Approximated

The purpose of the following comparison is to verify if the approximations presented in this Chapter had any relevant effects on the ANNs performance. For this reason, the baseline for comparison will be the same standard IEEE FP16 (half precision) implementation tested in Section 4.3, with the same initializations. The comparison method is also the same, graphically analyzing the average accuracy during training, with 95% confidence intervals. In all these comparisons, the approximations presented in this chapter are implemented in the trials identified as “**approx. 16**”.

In Figure 19 the same two datasets which previously showed no significant difference to the double precision implementation, also behaved similarly when implemented with

the approximate operations. Exactly as before, no approximated trials failed to converge. Due to the lack of difference in average behavior and the narrow confidence intervals, the approximate implementation can be considered equivalent to the standard FP16 for these datasets and training method.

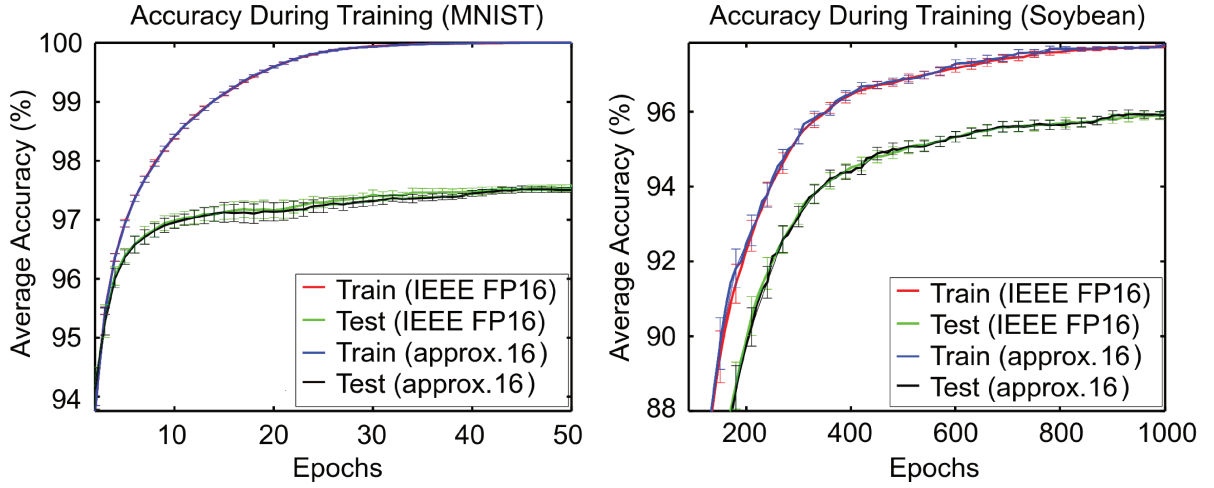


Figure 19 – Training Process Comparison between standard IEEE and approximate FP16 representations for the MNIST and Soybean datasets

Two of the datasets compared to a precise implementation in Section 4.3 presented significant differences to the baseline. Conversely, when the standard FP16 trials are compared to the approximated version, as depicted in Figure 20, no evidence was found that indicates worse or better training performance of the approximated FP16 implementation and the simple exponential method. As it happened previously, the larger spread in the first epochs for the Breast Cancer dataset is due to slower convergence speeds of some trials. The same initialization which failed to converge previously, also diverged in the approximated version. In the Thyroid dataset, the same slower learning progress was observed, with similar averages and narrow confidence intervals. For this reasons, the approximate implementation can also be considered equivalent to the standard FP16 in these conditions.

To summarize the analysis: no evidence was found to indicate that the approximations and simplifications proposed in this Chapter worsened the performance of the ANN training procedures when compared to the low precision IEEE FP16. Unfortunately, as shown in Chapter 4, even this standard format was proven to not always be equivalent to the more precise ones. The two relevant differences found when the comparison was based on a double precision implementation will be handled in the Chapter 6, when adaptive

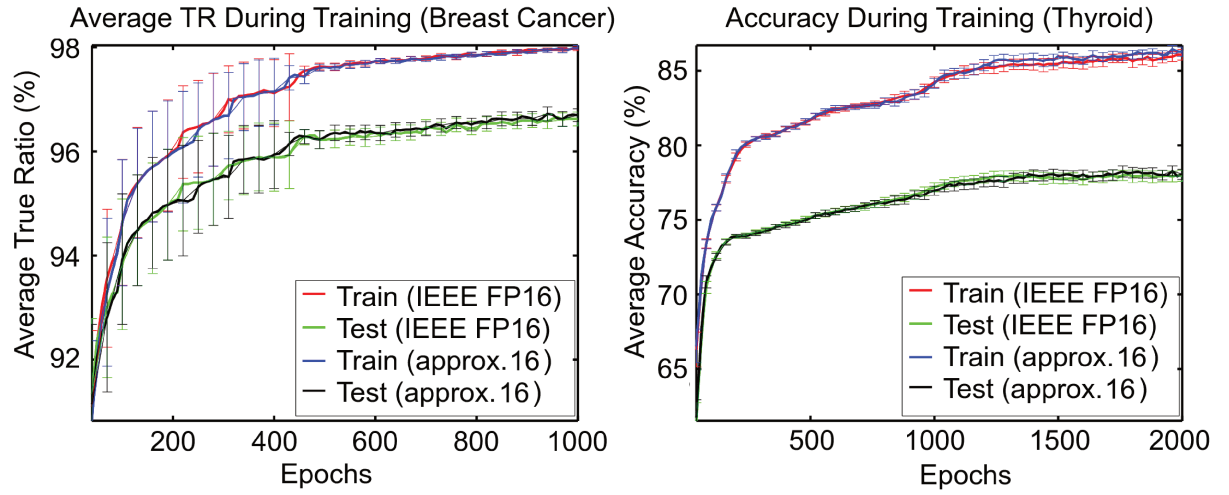


Figure 20 – Training Process Comparison between standard IEEE and approximate FP16 representations for the Breast Cancer and Thyroid datasets

methods are introduced and used with an automatic FP bias adjustment mechanism.

6 Resilient Training of ANNs

Chapter 4 presented the details of ANN implementations based on standard FP operations, showing evidence that FP16 is not always a direct replacement for more precise formats. In Chapter 5, approximations that simplify the FP16 implementations were analyzed and no evidence was found to consider them less reliable than standard FP16 for basic ANN training. This Chapter will analyze other learning techniques that could be more efficient than conventional Gradient Descent by adapting the weight change mechanism to each connection. This is evaluated as a better approach than, for example, increasing the global learning rate to solve problems like the ones found in the FP16 implementations. For this reason, these methods rely on storing extra information for each weight, adjusting it as the training progresses. Several robust adaptive methods have been proposed in the literature as a mechanism to dynamically and individually adjust weights at different speeds. These methods differ regarding how much extra memory is required for the adaptive behavior.

Efficiency, as mentioned, is a broad concept when related to this analysis. A network that converges faster (with less evaluations of the training examples) requiring much more memory or computing, may require even more energy than a simpler and slower one. The memory use alone may result in the infeasibility of an FPGA implementation, by not allowing a good localization of the parameters ¹. As the final target for this work is an efficient hardware implementation, only methods that store a single extra variable for each weight will be analyzed. Not even the use of temporary variables with higher precision (like accumulations in FP32) will be considered, for the same reason.

Finally, an automatic method to dynamically adapt the approximated FP16 representation range during training, for each neuron and without extra hyper-parameters, is proposed. Adaptive training methods for ANNs improve the equivalence of low-precision arithmetic to the reference ones, but were not enough to reach a desirable performance on all tested datasets. The final system is also tested against the **bfloat16** format, used in the accelerated TPU platforms, designed by Google (<<https://cloud.google.com/tpu/docs/bfloat16>>), and also being adopted by Intel (HENRY; TANG; HEINECKE, 2019) and ARM (BURGESS et al., 2019). A novel FP representation with dynamic precision is also used for comparison, in its fixed size form: the **posit16**.

¹ If the parameters used for a node are not stored in a distributed way, close to the processing elements, the data traffic to external memory may dominate the training costs, as stated, e.g. in (HAN et al., 2015) and (MISRA; SAHA, 2010).

6.1 Adaptive Training Mechanisms

This Section will use the same notation for the weight updates in all its Parts, adding terms and operations according to the analyzed method. Equation 6.1 details the update rule for GD where the parameters θ_n (the weights and biases) are updated by applying at iteration n a change $\Delta\theta_n$. This change is a small fraction (given by η , the learning rate) of the gradient ∇_θ of the loss function $L(\theta_n; x; y)$ with respect to the parameters, given the inputs x and outputs y . The negative sign comes from the fact that the loss function is the objective to be minimized. An incremental approach applies this change for each training example, i.e. a $(x; y)$ pair, while batch methods accumulate the gradient contributions of several examples before adjusting the parameters.

$$\Delta\theta_n = -\eta\nabla_\theta L(\theta_n; x; y) \quad (6.1a)$$

$$\theta_{n+1} = \theta_n + \Delta\theta_n \quad (6.1b)$$

The following parts present different methods to determine the change $\Delta\theta_n$ and the analysis is focused on specific datasets, according to the problems found.

6.1.1 Gradient Descent with Momentum

The basic idea for a momentum term is to change the “velocity” of weight updates instead of their “position”. This analogy considers each connection value as a particle that should be moved, as fast as possible, without instabilities, to improve the network loss. If successive updates happen in the same direction, the weight change will increase its rate, or momentum. Conversely, gradients that frequently change sign will not be heavily adjusted. Equation 6.2 summarizes this update scheme where γ , usually < 1 , determines how much of the previous “velocity” $\Delta\theta_{n-1}$ is kept in the current step. This means that for each weight an extra parameter should be stored and updated during training.

$$\Delta\theta_n = \gamma\Delta\theta_{n-1} - \eta\nabla_\theta L(\theta_n; x; y) \quad (6.2)$$

The momentum term did not improve the slightly lower convergence speed in some specific initializations of the Breast Cancer dataset. By removing 3 (out of 150) initializations that could be considered outliers due to delayed convergence, all of them common to both representations, no significant difference was found between the two precisions. This similarity also happens in the previous test, if outliers are excluded. Figure 21 shows how the approximated FP16 (as defined in Chapter 5 and identified as “**approx.**”) behaved similarly to the reference when momentum was used (plot on the right), but with a slightly faster accuracy growth when compared with regular GD (on

the left). The reference is the same double precision floating point (IEEE FP64) used in Chapter 4. The same reason defined in Section 4.3 justifies the use of Average True Ratio.

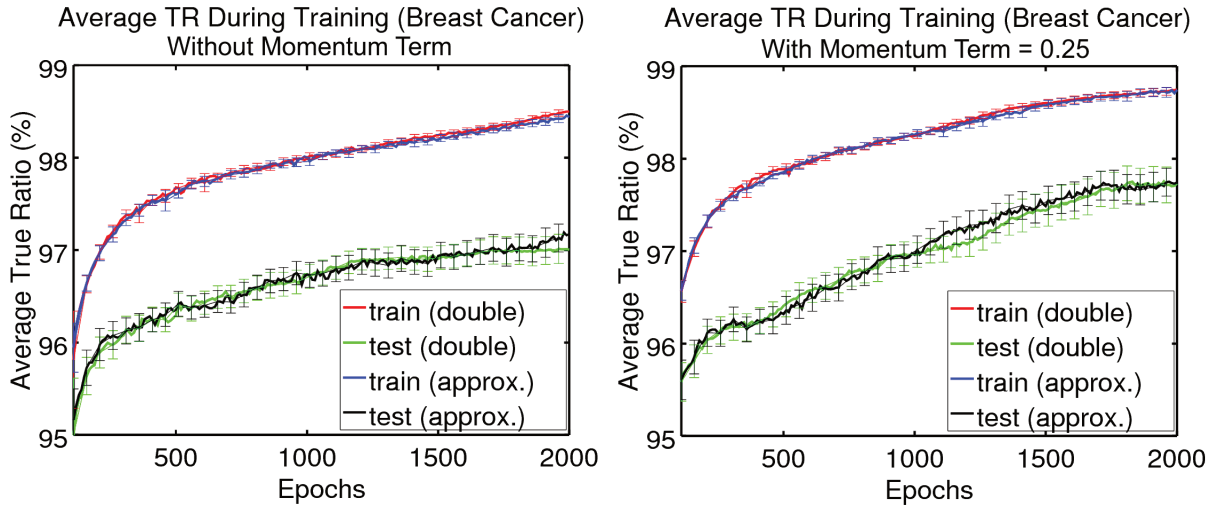


Figure 21 – Breast Cancer without outliers: comparison of the effect of the momentum term using a double precision FP64 as baseline

The first experiments with momentum applied to the Thyroid dataset showed that the addition of this term did not minimize the problems found when standard FP64 and FP16 were compared with mini-batch training. To extend the comparison scope, the first training setup was repeated using a full-batch approach, which is not the ideal one for large datasets since it only applies the weight changes once for each complete evaluation of the dataset. Figure 22 compares the training progress with this method without (to the left) and with the momentum term (to the right). The higher precision training (double precision FP64) presented instabilities late in the process which were not observed in the approximate version (as defined in Chapter 5 and identified as “**approx.**”). It is worth mentioning that this is the most unbalanced dataset in these tests. The two minority classes represent only 2.3% and 5.1% of the training examples and the rest belongs to a single class. The use of a cost based approach to handle the class imbalance caused a sequential progress in the learning process, and the instabilities coincide with the epochs when the accuracy of the third class is increased. This problem would go unnoticed if global accuracies had been used. The momentum term improved the stability and the accuracy progress. It reduced the difference between the precisions, but a small advantage is still present for the FP64 implementation.

6.1.2 iRProp-

The original RProp (“Resilient back-Propagation”) algorithm (RIEDMILLER; BRAUN, 1993) offered a very simple way of learning the weight adjustment rates adaptively. The basic idea is that just the signs of the gradients in consecutive epochs are used to decide if the absolute value of $\Delta\theta_n$ should be increased or reduced and in which direction

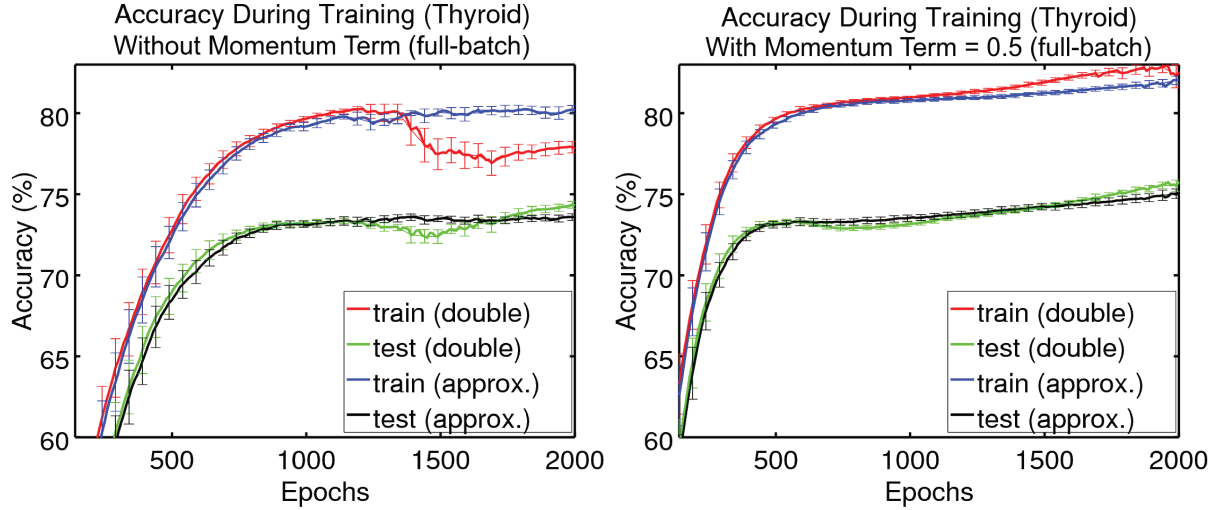


Figure 22 – Thyroid in full-batch mode: comparison of the effect of the momentum term using a double precision FP64 as baseline

it should be applied. When a partial derivative changes its sign, it is assumed that the last update was too large and the network has jumped over a local minimum. In this case the absolute value of $\Delta\theta_n$ is multiplied by a positive factor $\mu^- < 1$. If the gradient sign does not change, the absolute value of $\Delta\theta_n$ is multiplied by a factor $\mu^+ > 1$ to increase the convergence speed. The choice of the correction factors $0 < \mu^- < 1 < \mu^+$ is considered an optional hyper-parameter adjustment, but eventual changes must be performed carefully as they are crucial for convergence.

A few modifications were proposed in (IGEL; HÜSKEN, 2000), from which the iRProp- variation was chosen for the tests performed here due to its performance and simplicity. Not only these proposals but also more recent ones, like (BAILEY, 2015), keep the premise of using this algorithm in full-batch mode. This is probably the main reason that explains the lack of popularity of this robust method in a context of great achievements related to Deep Learning with large datasets.

In Algorithm 6.1, as defined in (IGEL; HÜSKEN, 2000), $\Delta\theta_n^+$ is the absolute value of the weight adjustment, stored between epochs. The function `sign(x)` returns -1 if $x < 0$, 1 if $x > 0$ or 0 otherwise. After the last step, only the sign of the gradient must be stored and a flag indicating if, in the previous step, a weight adjustment was not performed (caused by the “**else if**” branch in line 6). The actual steps are limited by fixed and global values (Δ_{min} and Δ_{max}).

Algorithm 6.1 – iRprop- Method

```

1: for all parameters  $\in \theta_n$  do
2:   if  $(\nabla_{\theta}^n \cdot \nabla_{\theta}^{n-1} > 0)$  then
3:      $\Delta\theta_n^+ \leftarrow \min(\mu^+ \cdot \Delta\theta_{n-1}^+, \Delta_{max})$ 
4:   else if  $(\nabla_{\theta}^n \cdot \nabla_{\theta}^{n-1} < 0)$  then
5:      $\Delta\theta_n^+ \leftarrow \max(\mu^- \cdot \Delta\theta_{n-1}^+, \Delta_{min})$ 
6:      $\nabla_{\theta}^n \leftarrow 0$ 
7:   else
8:      $\Delta\theta_n^+ \leftarrow \Delta\theta_{n-1}^+$ 
9:   end if
10:   $\Delta\theta_n \leftarrow -\text{sign}(\nabla_{\theta}^n) \cdot \Delta\theta_n^+$ 
11: end for

```

Previously this research attempted to implement some minor modifications to the iRProp- method to improve its behavior with reduced precision and the normalized weights initialization. The main changes were: adjusting the initial step as a relative value of the uniform random distribution limits in each layer, causing a hyper-parameter to be used as a fixed value; creating a mechanism to restart the weight change process when this was interrupted by a flush to zero (more common with reduced precision). An option was also evaluated to restart the learning process only for significant weights if the network shifts to a different position in the optimization space where a specific change becomes relevant again.

The proposed modifications improved the training behavior in most datasets, but caused a slightly worse performance in one of them. Specially when generalization is analyzed, the results are mixed, with slight advantages for each FP representation on different datasets. iRProp- is a robust algorithm, with noticeably better convergence speeds for many datasets. The fact that this method does not rely on the gradient value was considered an advantage for low precision training but, even with guided modifications, equivalent behavior was not generally achieved.

6.1.3 RMSProp

The previous sections analyzed two adaptive techniques as attempts to improve, at the training method level, the performance difference found between some FP64 and FP16 implementations. The well known beneficial effects of the momentum term improved the results for the two datasets which presented differences earlier. This improvement was not enough to consider the approximate implementation as equivalent to the precise one. The impressive convergence speed and reliability of iRProp-, even with modifications, were not enough to provide equivalent results and do not cover the problems with large datasets

as it is a method known to be adequate only for full-batch training. This leads to the current investigation of RMSProp, still keeping the constraint of a single extra adjustable parameter for each weight.

An interesting fact about RMSProp is that it achieved “huge empirical success”, as stated in (MUKKAMALA; HEIN, 2017), before a rigorous theoretical analysis was published. For this reason, most of the papers analyzing RMSProp cite a slide (TIELEMAN; HINTON, 2012) from the Geoffrey Hinton’s on-line ANN course, where he quickly mentions the method as an unpublished idea and also some possible improvements worth investigating. The algorithm implemented for the tests in this section is based on the details provided in (MUKKAMALA; HEIN, 2017), due to the informality and brevity of Hinton’s presentation, but it does not include the improvements proposed in the paper. The datasets used for the following tests did not require the dynamic hyper-parameters adjustments proposed by the author as a way to improve reliability.

Equation 6.3 details the original RMSProp algorithm. A small term added to the denominator to avoid division by zero is omitted as it is not used in the approximated implementation, since division by zero is handled differently (without an exception). A running average of the squared gradient is stored in ∇_{θ}^{avg} , and adjusted by the factor $\beta < 1$. Originally this parameter was proposed as static but the in (MUKKAMALA; HEIN, 2017) it is argued that this may lead to divergence and should be adapted during training. The method is sometimes compared to RProp but with the advantage that gradient fluctuations between mini-batches do not cause instabilities because the running average provides a normalization. It should also be noted that the learning rate is attenuated with the training progress (since it is divided by \sqrt{n}).

$$\nabla_{\theta}^{avg} = \beta \nabla_{\theta}^{n-1} + (1 - \beta)(\nabla_{\theta}^n)^2 \quad (6.3a)$$

$$\Delta\theta_n = \frac{\eta}{\sqrt{n}} \times \frac{\nabla_{\theta}^n}{\sqrt{\nabla_{\theta}^{avg}}} \quad (6.3b)$$

When ∇_{θ}^{avg} reaches very low values and is flushed to zero, the adjustments for these specific weights may be interrupted if the gradient does not reach higher values as the training progresses. To cope with these problems, a simple modification was implemented in this thesis. The running average is initialized to a small value (e.g. $\nabla_{\theta}^{avg} = 1 \times 10^{-4}$) and as soon as it reaches zero the update rule for that respective weight falls back to regular GD with momentum, using the same learning rate. The running average stops being updated for that weight so the new scheme is maintained until training is interrupted. Since the variable holding the average gradient becomes available, it can be used to hold the dynamic momentum term without increasing the memory usage. Alternative methods for the learning rate decay have not been evaluated, as they would imply another

hyper-parameter to be tuned. It should also be noted that this modification was enabled in the trials for both precisions, but rarely activated in the precise ones (since the average value drops to very low value without being flushed to zero). It is worth noting that automatic FP bias adjustments, detailed in Section 6.3, make this handling non-significant to increase training reliability.

Figure 23 presents the results which compare the average training accuracy progress in the same two precision implementations (identified as previously) for the MNIST dataset. The first noticeable difference on the left plot is the faster convergence speed when compared to normal gradient descent, which reached its maximum generalization after 50 epochs, against 28 for RMSProp. The graph on the right provides a more clear view of the accuracy differences, showing a very small but statistically significant advantage regarding overfitting resistance in the FP16 approximated version.

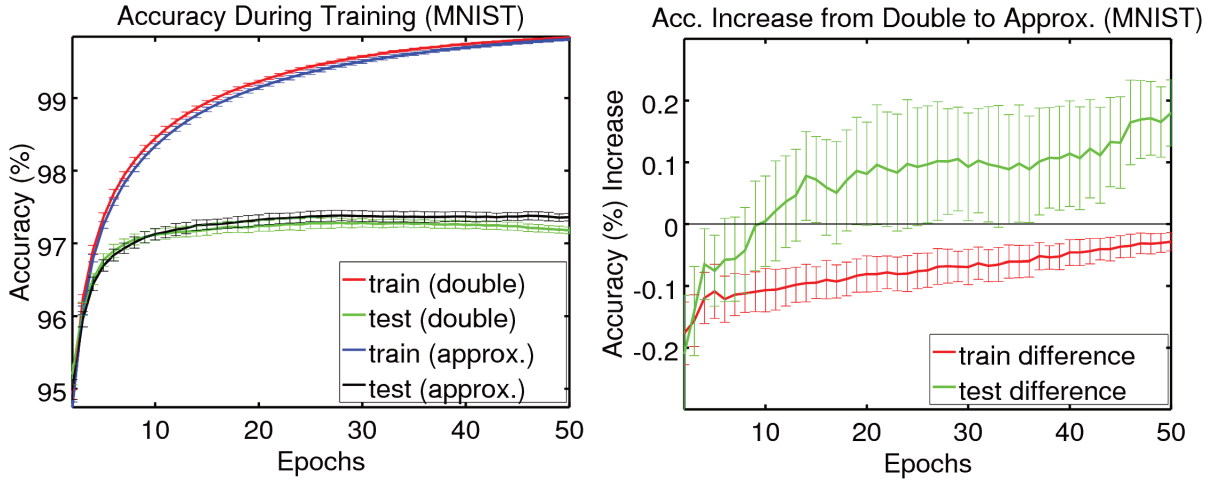


Figure 23 – MNIST dataset: comparing the RMSProp adaptation implemented with FP16 approximations and a double precision FP64 as baseline (positive values on the right mean better “**approx.**” performance)

6.2 Exploring Different FP16 Ranges

In Sections 6.1.1, 6.1.2 and 6.1.3, simple adaptive methods provided more reliable results and faster convergence with approximated FP implementations, when compared with regular GD training. Despite this improvement, a general equivalent behavior to precise arithmetic was not observed for all datasets. With these robust training algorithms, a method could be explored to provide a flexible representation for variables in **Groups 2** and **3**, adapting the precisions to each problem. Representation ranges could be shifted towards smaller numbers, for example, to provide better precision for variables which never operate with the larger representable values. To avoid considerable overhead, this feature should be able to operate without requiring complex conversion procedures between different precisions. This could be considered a more reliable approach than to modify the

training methods themselves to cope with problems caused by the approximations, after their occurrence.

Before a different representation is evaluated for the variable group which may require more precision, it should be considered if, in some cases and to some extent, less precision is in fact better. Not only some tests reported in this thesis, but also the literature presented in Chapter 3, provide examples where, by a small margin, the approximated implementation behaved better than the reference one regarding average test accuracy. Even early experiments like the ones presented in (SIETSMA; DOW, 1991) and seminal papers like (BISHOP, 1995) reinforce the confidence in systematic use of noise as a regularization mechanism. For the datasets and training methods, where more precision brings also more sensitivity to let noise become detrimental in the learning process, there is a rich set of explicit regularization methods available in the literature.

If smaller numbers are indeed necessary in some cases during the *back-propagation* phase and consequently in the gradient calculation processes, a simple way to avoid their underflow would be to increase the value of the loss function, by multiplying it by a growing factor as the training progresses. The actual loss value must not be calculated for the training methods evaluated here, but the multiplication would increase its derivative, and that would result in larger weight gradients. It must be taken into account that this global method may lead to large weight updates in some cases since it can not selectively adjust the gradients for different layers or nodes. This limitation becomes even more important for Deep ANNs, due to the “vanishing gradient” problem.

As detailed in Section 4.1.1, the FP exponent is encoded in positive integer numbers, shifted by a constant value. Choosing different bias values also changes the representation limits, with only minor changes to the FP operations. To reach smaller values, the bias should be increased, sacrificing the largest representable number. Table 6 details some different absolute ranges for normal numbers provided by biases larger than the standard one. As a general rule, if the smallest normal significant equals 1.0 (with all fractional *bits* set to zero) and the largest one equals 1.9990234375 (with all fractional *bits* set to one), the ratio between the maximum and minimum representable absolute numbers does not change with the bias (Equation 6.4).

Table 6 – Different ranges provided by larger exponent biases

Bias	Smallest Positive	Largest Positive
19	1.9092×10^{-6}	8.1889×10^3
23	1.1933×10^{-7}	5.1175×10^2
27	7.4579×10^{-9}	3.1984×10^1
31	4.6611×10^{-10}	1.9990×10^0

$$\min = 1.0 \times 2^{0-\text{bias}}; \max = 1.9990234375 \times 2^{31-\text{bias}} \quad (6.4a)$$

$$\frac{\max}{\min} = 1.9990234375 \times 2^{31-\text{bias}-0+\text{bias}} \approx 2^{32} \quad (6.4b)$$

$$\max \approx 4.29 \times 10^9 \min \quad (6.4c)$$

To evaluate the effect of a more precise representation in the variable **Group 3**, test trials were performed using different bias values restricted to these variables and their operations. The reciprocal square root approximation was replaced by a precise implementation and the only difference between the trials was the bias value used for the more precise variables. The format used in **Group 2** was kept constant as the approximated FP16 (as defined in Chapter 5) with default bias (= 15) and the training algorithm was the one detailed in Section 6.1.3, with the same hyper-parameters (which were adjusted for the baseline in each dataset). The bias value for the precise trials was chosen to match the same minimum absolute magnitude provided by the standard FP16 format: 5.9663×10^{-8} (in subnormal representation).

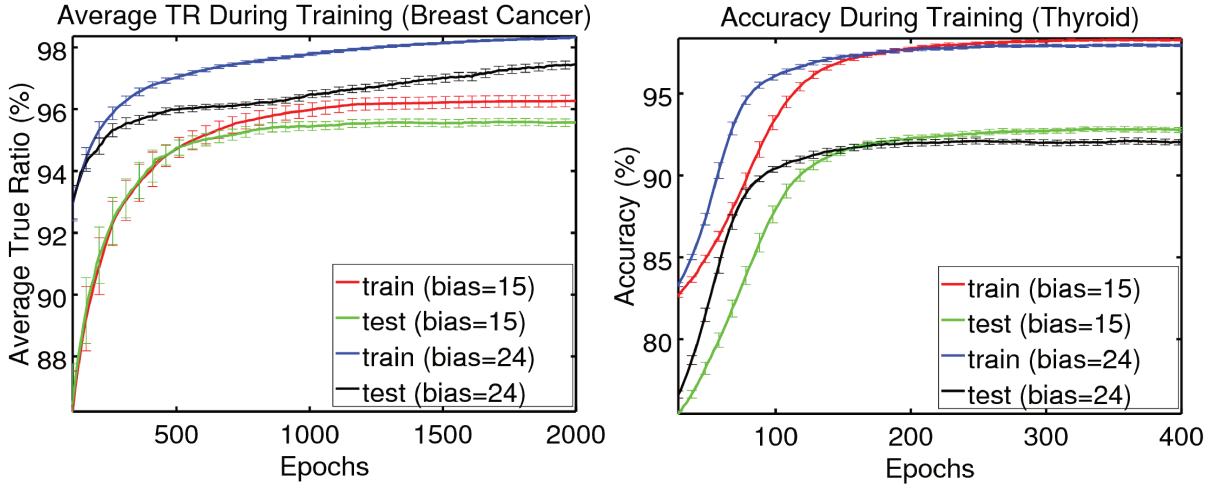


Figure 24 – Modified RMSProp training with the approximate FP16 using different exponent bias values for variable in **Group 3**

Noticeable differences are present in the results for the two datasets depicted in Figure 24. The same reason defined in Section 4.3 justifies the use of Average True Ratio for the Breast Cancer dataset. A more precise representation in this dataset resulted in considerably faster convergence and higher average accuracy. Outliers (6 non-convergent out of 150 trials) happened only in the reference implementation (bias=15). The Thyroid dataset, trained with mini-batches, presented very stable behavior and much faster convergence. The more precise trials were even noticeably faster than the default ones but at the expense of a slightly worse generalization. The same procedure was applied to the Soybean and Gene datasets, with less relevant differences. In the second case only a slightly worse

variability (accuracy standard deviation at each epoch) was observed in the trials with $bias = 24$.

It is evident from these tests that the bias selection affects the training results even with a more robust algorithm. Adjusting the precision on the **Group 3** considering the bias value as an extra independent hyper-parameter is not reasonable since the purpose of the simplifications is to make the learning process more efficient (and hyper-parameter search must be included in the training cost for a fair evaluation). Evaluating metrics, like the frequency of underflows and overflows, may also have a considerable cost if applied to all operations. Conversely, verifying only final values like weight changes at the end of each training data batch may lose important information regarding cancellations, for example. A light-weight heuristic to adjust the bias for separately for each neuron during back-propagation is presented in Section 6.3. The method is completely automatic, not requiring any a-priori search for hyper-parameter values.

6.3 Dynamic FP16 Bias Adjustments

As mentioned previously, the exponent representation in FP is shifted by a bias that allows simple operations using the unsigned format and simple FP magnitude comparisons. By choosing a central value (15) for the bias, the IEEE 754 standard creates an almost symmetrical exponent range: $[-15, 16]$. An interesting effect of this decision, specially useful for ANN inference, is that practically half the representable values are within -1.0 and 1.0 . For ANN training, and other GD optimizations, as the process advances smaller adjustments are performed until they stop improving the model. Higher exponents, in this context, waste representation space for large magnitudes that never occur at the expense of a reduced capacity to represent small numbers. For this reason, this thesis proposes a simple mechanism to automatically shift the representation range for variables involved in training, separately for each neuron, while keeping the inference variables in the standard range.

The decision to keep the FP bias constant in the variables used for the inference phase is mainly based on two aspects: it allows a direct deployment of trained models to platforms that support the standard FP16 format; it reduces the computational cost of the inference operations during training, by not requiring conversions. Additionally, it should also be noted that standard FP16 is becoming a safe choice for the inference phase in most datasets, and is even considered “too precise” in some cases (as seen in Section 3.1). The **bfloat16** format (detailed in Section 6.4), which is even simpler for mixed-precision approaches, is also gaining momentum, reaching commercial solutions beyond the original Google’s proposal: ARM and Intel are also adding support for this format.

For the dynamic range adjustment in the variables used only in training (**Group**

3, as defined in Section 4.3), the FP exponent bias is set separately for each neuron, and updated as the GD optimization progresses. The purpose of a completely automatic method is mainly to avoid the creation of extra hyper-parameters, like error thresholds that guide adjustments to the biases, and also adapt the network to changing precision requirements. Basically three conditions could guide the adjustments: overflows (when the result magnitude is larger than the representable limit), underflows (when the result magnitude is smaller than the lower limit) and cancellation (e. g. when a small number is added to a larger one, or subtracted from it, without changing its value). It should be clear that by shifting the representation range to avoid overflows, underflows could become more common. Also important is that cancellations among the more precise variables are not directly affected by the bias shifts, as these errors are related to the significand *bit* length, which is not changed, and the exponent difference, which is not affected.

Since acting on both overflows and underflows would lead to antagonistic decisions, the proposed method is directed at quickly avoiding the occurrence of the first error type. For each training mini-batch, a single overflow shifts the current neuron's bias to a smaller value (limited to 15), which increase the largest representable magnitude. In contrast, a shift to a larger bias (limited to 31) only happens if no overflow is observed for operations with that neuron in a whole epoch (the period that evaluates every training example once). Acting on a mini-batch granularity is important as it avoids conversions of temporary values (zero has the same representation, regardless of the bias), without taking too long to act on overflows. If a finer granularity than a mini-batch was chosen, temporary gradient accumulation and training parameters would have to be adapted before the weight adjustment.

In the experiments, detailed in Section 6.4, this research found different behaviors for these adjustments. With the MNIST and Thyroid datasets, the biases stabilize in distributions (in the first case similar to a Gaussian), with centers not close to the limits. Figure 25 depicts the FP bias distribution at the end of the training process. In the other problems, all the neuron biases shift to the maximum value (the most precise one) at the end of the training, and in some of them this process happens very early. This difference reinforces the importance of an adaptive method as opposed to one that creates a new (global or distributed) fixed hyper-parameter.

It should be noted that this proposed method, although experimented earlier, has some similarities to the shared tensor exponent, presented in (KÖSTER et al., 2017), where the authors also criticize methods that react to overflow thresholds, instead anticipating them. The granularity of the adjustments (which in this thesis is per neuron, as opposed to the data representation itself in (KÖSTER et al., 2017)), the fact that the proposed method requires a single FPU type for all numbers and the simple adjustment mechanism (quickly avoiding overflows, instead of trying to predict them as in (KÖSTER et al., 2017)),

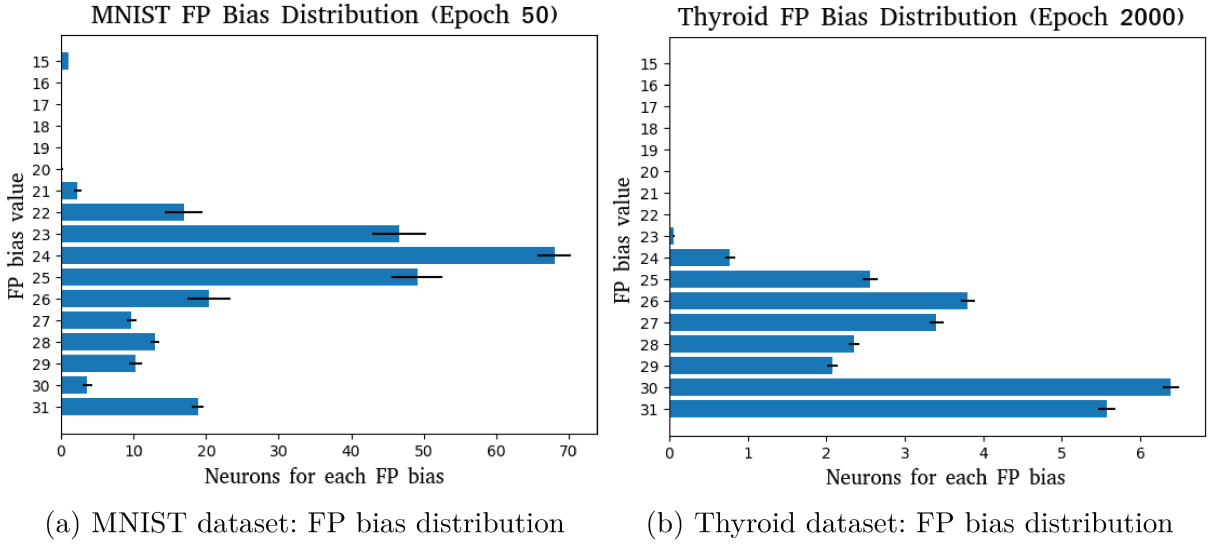


Figure 25 – FP bias distribution among neurons (end of training)

may be considered important differences. It is also very relevant that the bias value is stored for each neuron, regardless of how many input connection weights are present. This is more space efficient than sharing an exponent representation for small vectors of FPs.

Figure 26 illustrates how the FP biases are grouped for each neuron. In the inference phase all the operations are performed without range conversions, since all related variables use the default FP16 bias. During gradient accumulations and weight adjustments every operation includes the specific biases for each argument that comes from different neurons or from the variables in the previous group. Since the bias is not stored for each variable, its cost in memory usage is amortized: the wider the layers, more closer this solution is to the ideal 50% memory savings in the variables used for training (when compared to the standard mixed precision implementations). If the bias value were stored for each variable, increasing the *bit* length of each number, it would be much simpler to just add extra *bits* to the exponent representation.

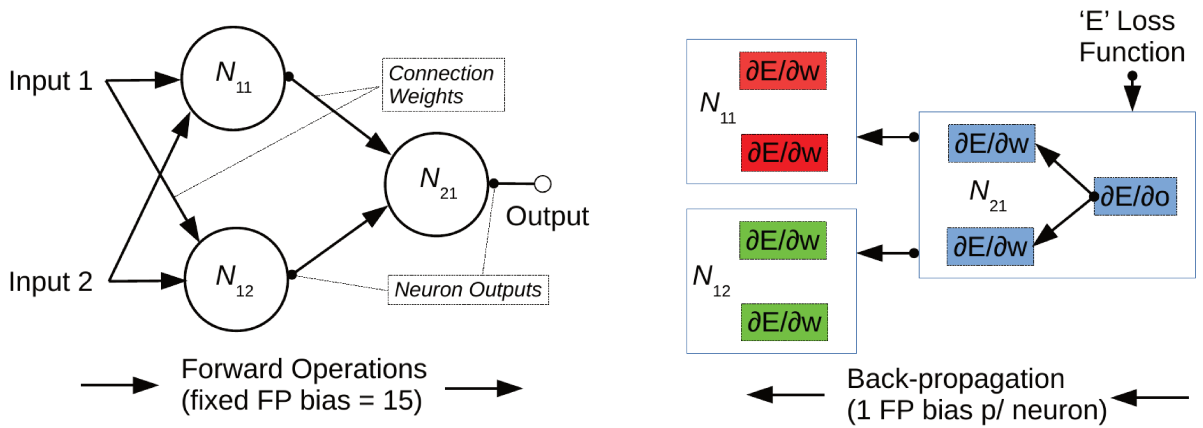


Figure 26 – FP bias: feed-forward and back-propagation

The addition and subtraction operations do not take into consideration the bias values, but only normalize the significands to equal exponents, what is also required for constant biases. For this reason, the dynamic bias has no effect in the hardware implementation of these operations. For multiplication and division, the actual bias value is required to determine the resulting exponent, but the only difference is that the operation involves a dynamic value instead of a constant one.

6.4 Experiments with Dynamic FP16 Bias

In the experiments with the complete method, additionally to the comparison with a precise reference, the approximated solution is also compared to the recently proposed **bfloat16** format. It should be clearly stated that this reduced format is intended to be used in a mixed-precision approach, which implies twice the memory used by training values. Its clever design was specifically optimized for this approach by aiming at a very fast and simple conversion from FP32 to **bfloat16**. Figure 27 compares ² two IEEE floating point formats (**FP32** and **FP16**) to Google’s **bfloat16**. Using the same formats for exponents explains the simple conversion and the similar range (subnormals in **FP32** are disregarded as they are flushed to zero). Arithmetic operations between **bfloat16** values already result in extra significant *bits*, which partially fill the available space in the **FP32** results.

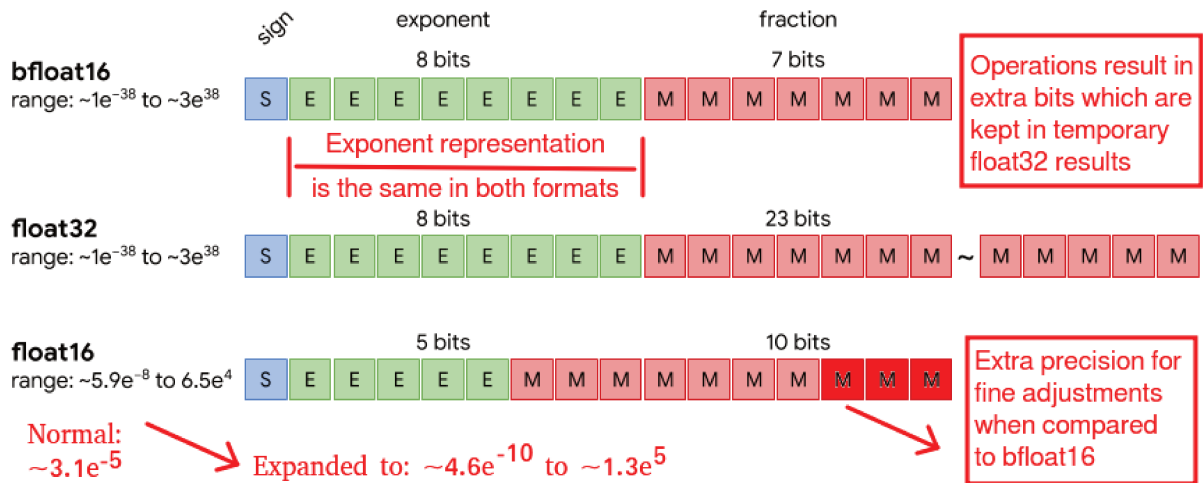


Figure 27 – Comparisons between different FP formats

It should be noted that when the approximated **FP16** with dynamic bias is compared with IEEE **FP16**, the minimum representable values are reduced by two orders of magnitude (much more if only normalized numbers are considered). Conversely, both these formats allow more precise adjustments when compared to **bfloat16** due to three extra *bits* in the significand.

² Adapted from <<https://cloud.google.com/tpu/docs/bfloat16>> under Attribution 4.0 International (CC BY 4.0) license, available at <<https://creativecommons.org/licenses/by/4.0/>>

The complete approximated solution, identified as *soft-ap*, is compared to a precise double precision reference and also to the **bfloat16** format. The results are depicted in the Figures: 28, 29, 30, 31 and 32. The plots to the left (a) present the comparison with the reference while the ones to the right (b) show the approximated performance compared to the **bfloat16** implementation in all the variables, hence with the same memory cost as the approximated solution.

No relevant difference in the test performance is found when the approximated method is compared to the precise reference. A minor advantage in generalization is observed in three problems (Thyroid, Breast and Gene) and a slightly inferior resistance to overfitting is present in the Soybean dataset. When the basis for comparison is the **bfloat16** format, similar generalization is observed in the MNIST and Breast datasets, but superior performance is observed in the other three problems, by large margins (one in accuracy and the other in convergence speed) in the Thyroid and Soybean datasets.

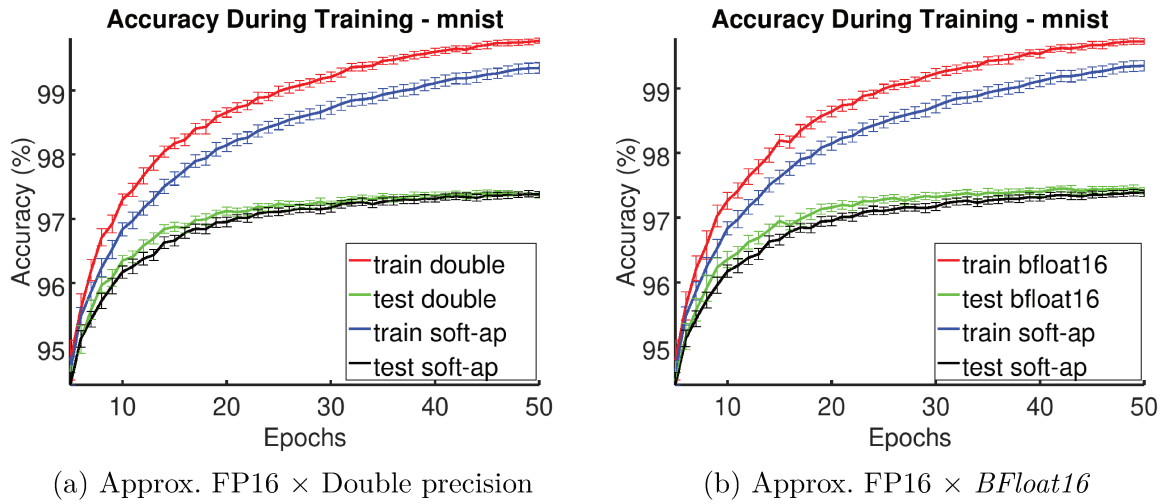


Figure 28 – Final performance comparisons with the MNIST Dataset

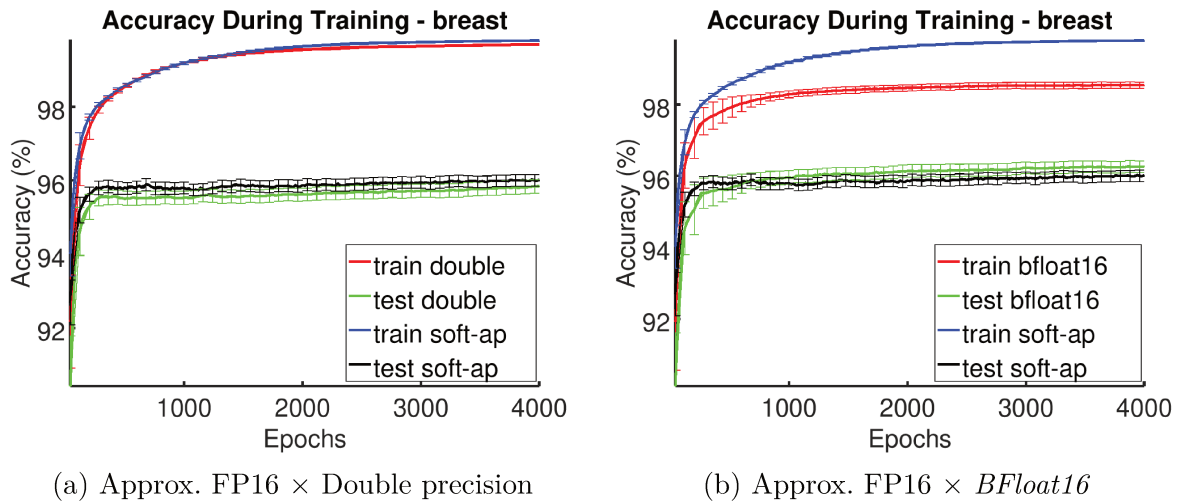


Figure 29 – Final performance comparisons with the Breast Cancer Dataset

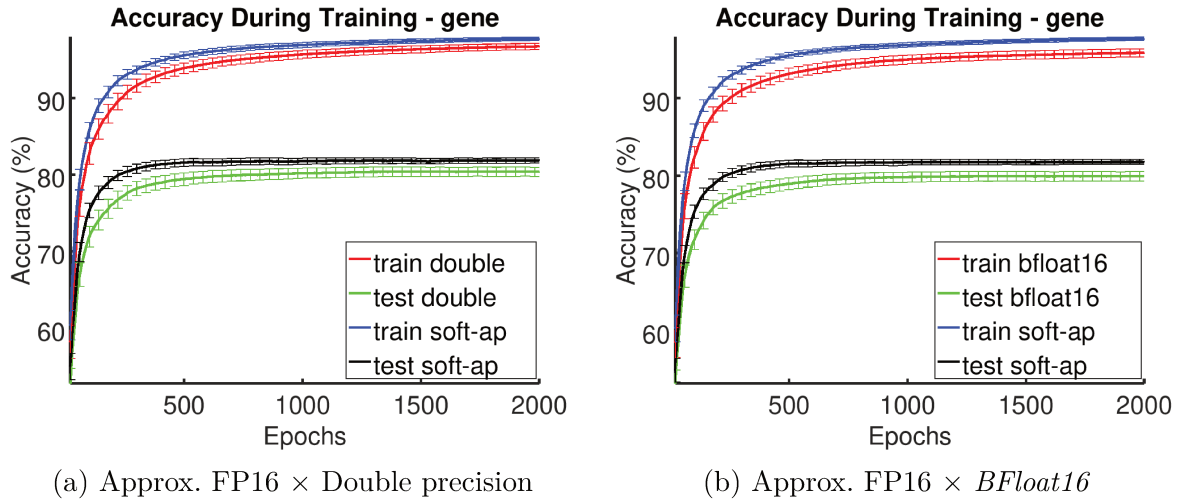


Figure 30 – Final performance comparisons with the Gene Dataset

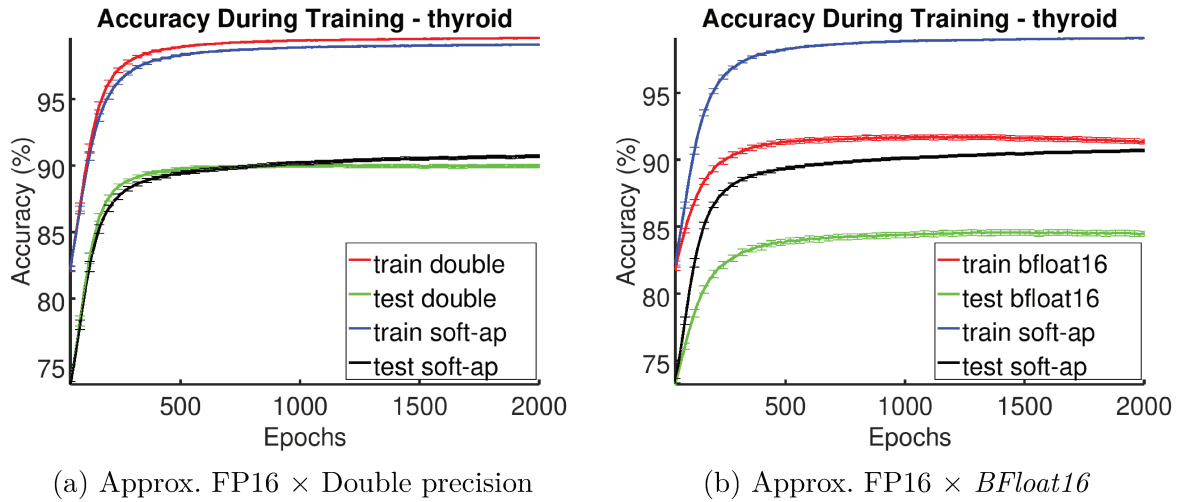


Figure 31 – Final performance comparisons with the Thyroid Dataset

An important validation is the addition of extra datasets to test the proposed methods after the emulated implementation is complete and mature (without extra changes after the execution of these tests). The chosen problems were the same ones used in (CASTRO, 2011), which are specially hard for ANNs without specific training techniques to handle unbalanced datasets. To keep the same training and test procedures previously defined, such approaches for unbalanced datasets are not included, which probably resulted in the exclusion of one variation from one dataset (glass9), due to failure in reaching a stable test behavior in the double precision baseline. The results are presented in using the same graphical format, but due to space and formatting constrains, are detailed in Appendix A.

The full approximated method performance can be summarized as follows:

- when compared to the double precision baseline: 11 datasets present equivalent test behavior, 4 show slightly better generalization and one shows worst overfitting

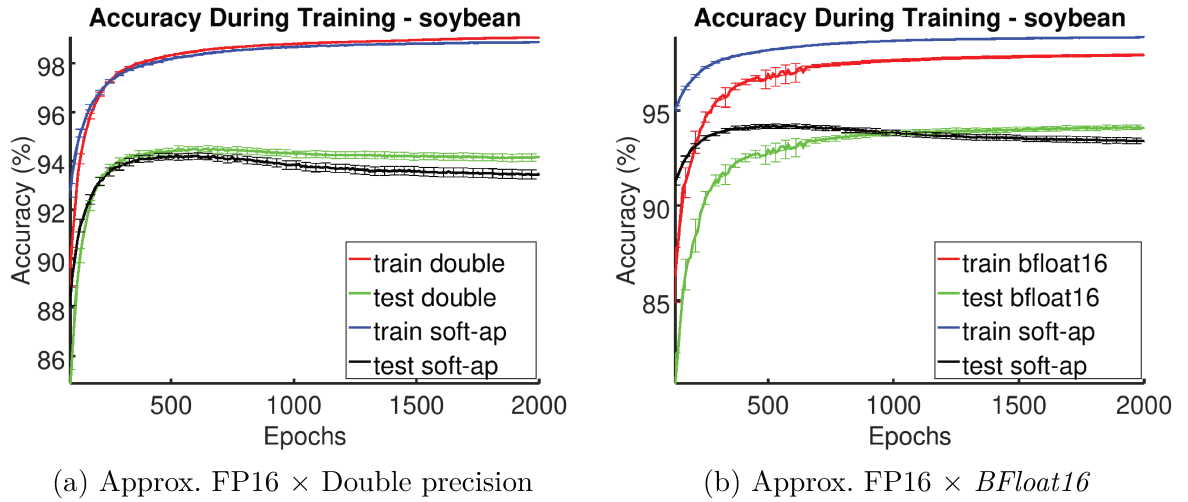


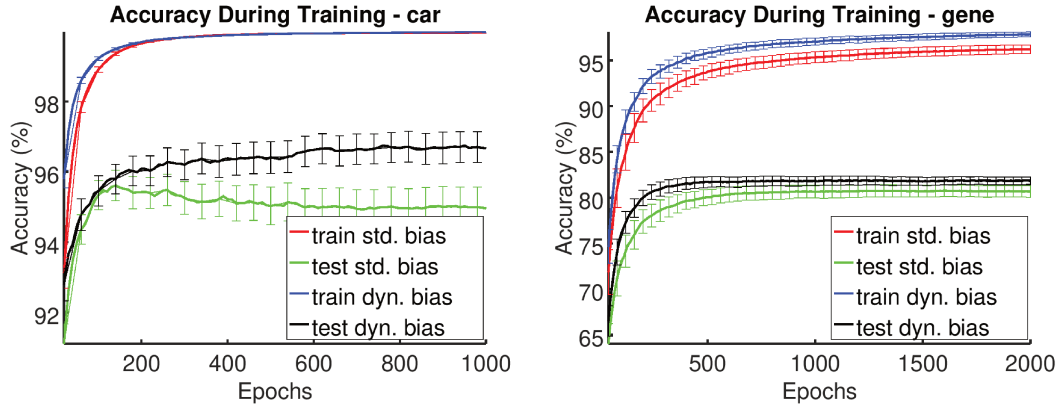
Figure 32 – Final performance comparisons with the Soybean Dataset

resistance, but with similar maximum values and only a $\approx 0.1\%$ accuracy difference by the end of training.

- when the **bfloat16** format is considered, 8 datasets present equivalent test performance, 7 show better performance (three $> 5\%$ and one $> 2\%$) and the same one shows worse overfitting resistance, with the same characteristics detailed in the previous comparison.

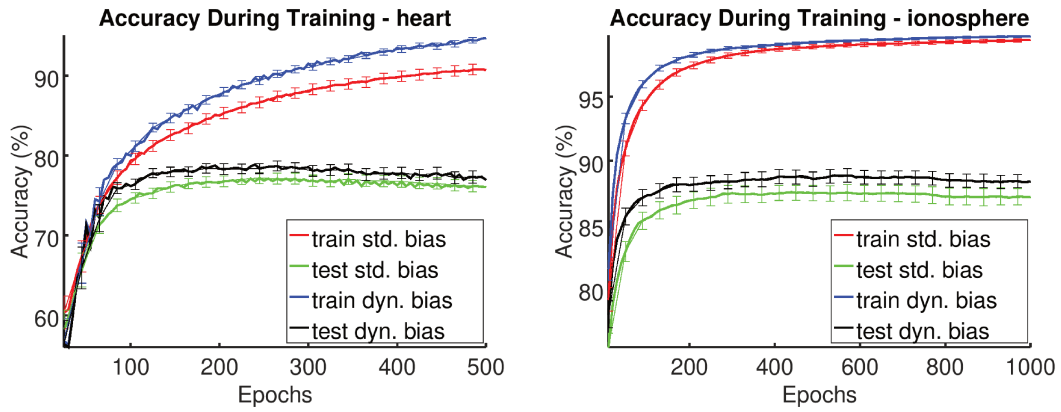
6.5 Experiments with Fixed FP16 Bias

As a final step to improve the confidence in effect of the dynamic FP bias method on the training resilience, an extra set of experiments was performed. Additionally to the broad confidence in the mixed precision approach for generic hardware implementations (e.g. Google, Intel, ARM and RISC-V), as opposed to pure FP16 solutions, the experiments performed since the beginning of this research confirmed that full approximated solutions needed more features to achieve equivalent results. For this reason, the experiments with all datasets were repeated with the approximated FP16 and single difference: the dynamic bias adjustments were disabled and set to the default value (15) defined by IEEE. For brevity, only the results with relevant differences in generalization are depicted in Figures 33, 34, 35 and 36, all with advantage for the dynamic approach. The relevant differences in generalization, all in favor of the dynamic method, add evidence for the importance of localized and dynamic range adjustments.



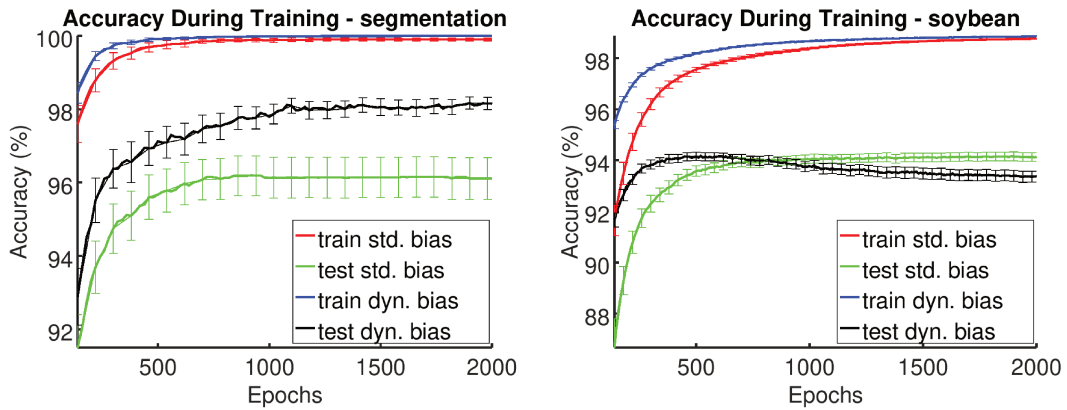
(a) Default FP16 bias \times Dynamic FP16 bias (b) Default FP16 bias \times Dynamic FP16 bias

Figure 33 – Performance influence of the Dynamic bias method



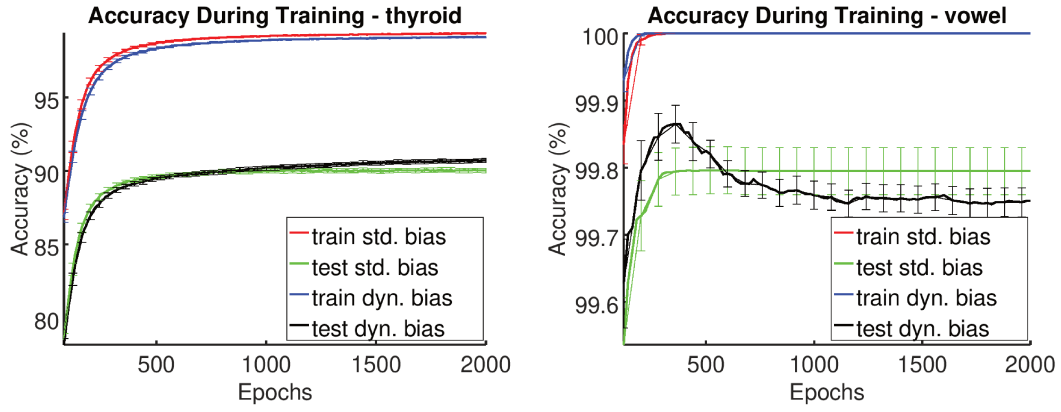
(a) Default FP16 bias \times Dynamic FP16 bias (b) Default FP16 bias \times Dynamic FP16 bias

Figure 34 – Performance influence of the Dynamic bias method



(a) Default FP16 bias \times Dynamic FP16 bias (b) Default FP16 bias \times Dynamic FP16 bias

Figure 35 – Performance influence of the Dynamic bias method



(a) Default FP16 bias \times Dynamic FP16 bias (b) Default FP16 bias \times Dynamic FP16 bias

Figure 36 – Performance influence of the Dynamic bias method

6.6 Comparison with Posit (16 bits)

Despite not having reached, at the time of this writing, the same level of adoption seen for **bfloat16** in practical implementations, the **posit16** format (GUSTAFSON; YONEMOTO, 2017) deserves attention due to its interesting design goals and decisions. Figure 37 depicts all the positive values representable in this format, as an illustration for the following discussion.

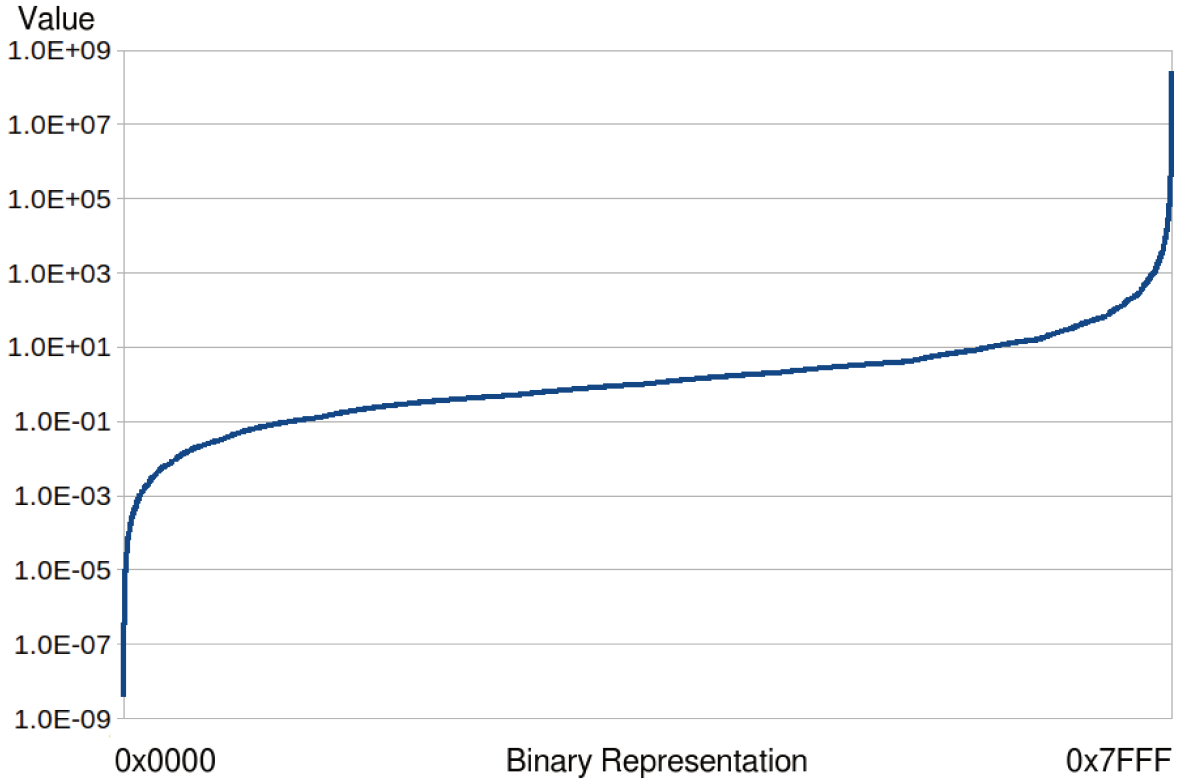


Figure 37 – All positive values representable in the **posit16** format

The ideas published in (GUSTAFSON, 2015b) and presented in (GUSTAFSON, 2015a) (named “Unum” format) received several criticisms which initially resulted in the improvements published in (GUSTAFSON, 2016) and later in the fixed size format presented in this Section. These iterations were crucial to allow more “hardware-friendly” implementations (without large LUTs and with fixed total length), like the one proposed in (PODOBAS; MATSUOKA, 2018).

The format does not provide a gradual underflow by the means of an exception, as in the IEEE FP subnormal numbers. Conversely, this graceful degradation is achieved with “tapered precision”, following the author’s denomination, which not only provides this behavior for underflow conditions, but also achieves the same effect in the numbers with large magnitudes. Numbers around the unity (e.g. between 0.1 and 10) have the highest precision, while larger and smaller magnitudes gradually exchange precision for range. To obtain this behavior, the format, which has a fixed total size, encodes an extra value in the “regime bits” with variable length, as depicted in Figure 38.

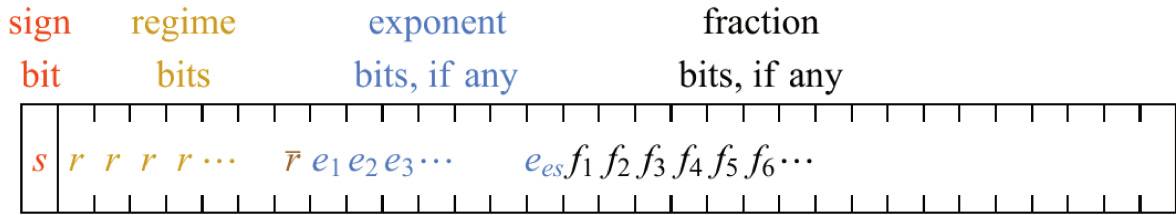


Figure 38 – Internal fields in the generic Posit format with fixed total *bits*. Source: (GUSTAFSON; YONEMOTO, 2017), unchanged

The effect of the new “regime bits” field is observable by comparing the Equation 6.5 (which represents a normalized IEEE FP number) to Equation 6.6 (representing the generic Posit encoding). The exponent value (*exp*) is present in both formats, but shifted by the *bias* in the IEEE encoding. The *mantissa* is the same, with only the fraction *bits* represented. The encoded value *k* in the regime *bits*, together with the exponent field length (*e_{bits}*), creates an extra power of two multiplier. The variable length fields are the main responsible by the extended range without losing precision in a large portion of the representable values. Conversely, this characteristic is also the main responsible for slightly more complex hardware implementations, since decoding of exponent and mantissa are only possible after the length of the regime *bits* is determined.

$$val = (-1)^{sign} * 2^{exp-bias} * 1.mantissa \quad (6.5)$$

$$val = (-1)^{sign} * (2^{2^{e_{bits}}})^k * 2^{exp} * 1.mantissa \quad (6.6)$$

Some characteristics of **posit16** are also present in the approximated FP16 format used in this thesis: the same fixed *bit* length, absence of subnormals, no encoding for infinity, saturation to the largest representable number and extended range (when compared to IEEE FP16). Despite the similarities, there is a difference that may be crucial when gradient descent optimization is considered: the representation error in the range responsible for the smaller numbers is quite large when compared to normalized FP16 representation (with dynamic bias). As an example, in **posit16** there are only 16 representable values between $1.0 * 10^{-6}$ and $3.7 * 10^{-9}$, which is the smallest non-zero magnitude.

The SoftPosit library³, which was based on the same library (SoftFloat) used in this thesis for the IEEE FP16 baseline, was integrated to the simulated environment described in Chapter 4 for comparisons using the same graphical method. Several Posit formats and operations are supported by the SoftPosit library. For consistency with other tests, the fixed format with 16 *bits* was used for feed-forward and back-propagation operations, resulting in the same memory usage. The detailed performance differences between **posit16** and the format proposed in this thesis (identified as **soft-ap**) are presented in Appendix B. The main aspects can be summarized as follows:

- No inferior generalization performance was observed for **soft-ap**.
- Statistically significant worse results in generalization for **soft-posit16** are present in the experiments with 7 datasets (from $\approx 1\%$ to $\approx 2\%$).
- In two datasets **soft-ap** presented better overfitting resistance, while in other two **soft-posit16** was superior in this aspect.

³ Source code available at: <<https://gitlab.com/cerlane/SoftPosit>>

7 Preliminary optimized implementations

Despite the obvious advantage in memory footprint when compared to mixed precision solutions, and the resource reduction estimated with HLS in Sections 5.1.3 and 5.2.4, it is important to verify the practicality of real implementations. The approximations and methods proposed in Chapters 5 and 6 are implementation agnostic, since the concepts could be realized in FPGAs, ASICs or in fully customized ICs.

The impact of higher level approximations was evaluated on a widely used ARM processor for embedded projects and the results are presented in Section 7.1. Due to the low cost and faster development cycle, an ARM-FPGA hybrid platform was used to explore practical implementation concepts for an ANN accelerator. This viability analysis is presented in Section 7.2.

7.1 Exponential Approximation with Hardware IEEE FP16

The Broadcom SoC (System on Chip) BCM2837 includes four ARM Cortex A53 cores. These CPUs support IEEE FP16 as a storage format and provide conversion instructions to allow the 32 *bits* FPU to be used efficiently. Throughout this Section, all references of FP16 and FP32 refer to the native support provided by this CPU, but even with hardware conversion instructions there is a considerable processing overhead in using the reduced format. This impact is clearly noticeable in Table 7, where the average training times (in seconds) for the reference FP32 implementation are presented in the **fp32fp32** column. The **fp16fp16** presents the training times for the implementation that use the FP16 storage format for all variables, thus demanding conversions every time a floating point is read from or written to RAM to perform operations. The implementations in both columns use precise *exp()* and *sqrt()* operations (provided by the standard math.h library), and the second one uses the VSQRT hardware instruction while the first one is implemented in software. The use of FP16 for all variables implies the largest memory usage reduction (50% less space required for variables converted from FP32 to FP16) but, as detailed next, the mixed precision savings may approach this figure for large datasets. Regarding CPU usage, code profiling showed that computational cost was more relevant in the inference phase than in back-propagation, and the *exp()* function was one of the main causes.

The **fp32fp16** column presents the training times for implementations which use a mixed precision approach: back-propagation, gradient accumulation and weight updates are performed in FP32 while inference data and operations use FP16. Both approximations for the *exp()* and *sqrt()* (presented in Sections 5.2.1 and 5.2.3, respectively) operations

were used but a profiling showed that the first one was the main responsible for the efficiency gain. This is consistent with the fact that this architecture provides a VSQRT instruction to calculate the square root, but there is no support for exponential calculation in the FPU.

Table 7 shows how in two cases (**Breast Cancer** and **Gene**) the mixed precision approach (**fp32fp16**) with approximations completely compensates the processing overhead caused by FP format conversions (the training times in these cases are not worse than the **fp32fp32** reference). For the other three cases, relevant processing time reduction is observed and the training times are between the reference and the full FP16 implementation. Figure 39 presents the same results in relative terms, considering the **fp32fp32** implementation as a comparison baseline.

Dataset	fp32fp32	fp16fp16	fp32fp16
Gene	33.0(± 0.36)	39.9(± 0.72)	33.0(± 0.56)
Soybean	47.0(± 0.21)	61.3(± 0.71)	55.1(± 0.57)
Thyroid	116(± 0.14)	166(± 0.73)	136(± 0.43)
Breast Cancer	8.05(± 0.07)	9.91(± 0.19)	7.94(± 0.18)
MNIST	3002(± 88)	4002(± 78)	3405(± 54)

Table 7 – Training times (in seconds, with 99% confidence intervals) in the Broadcom BCM2837 SoC. The mixed precision approach is presented in **fp32fp16**, while in the others all variables use the same representations (FP16 or FP32)

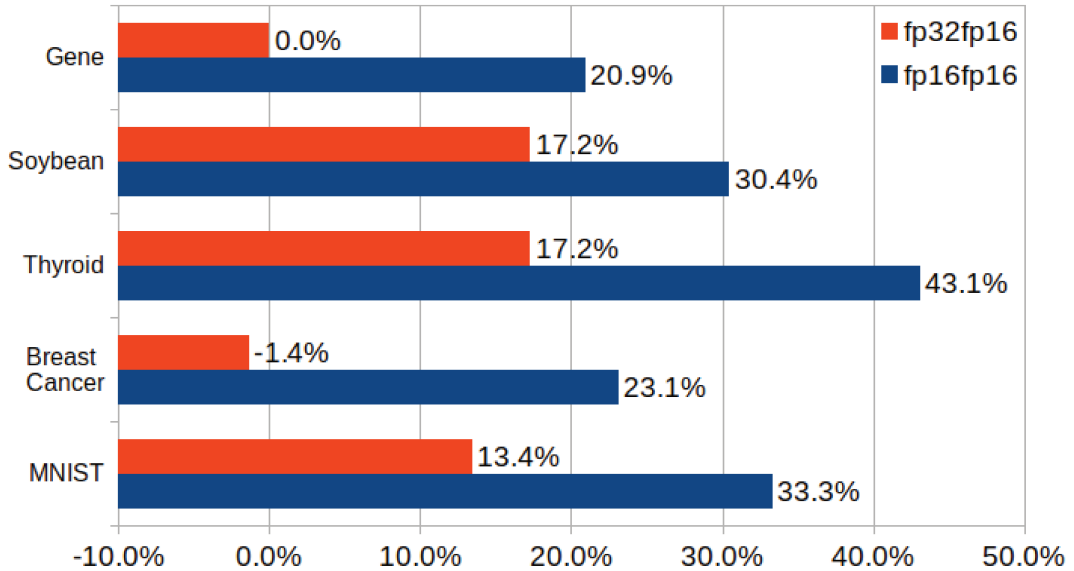


Figure 39 – Relative increase in training times (compared to the **fp32fp32** reference).

For large datasets, most of the memory gain is already present in the mixed precision approach (due to the fact that the dataset is loaded in RAM with the same format used in inference). Consequently, the overhead of precision scaling for the full FP16 implementation was not considered a viable option in this platform. Table 8 presents

the memory usage reduction for each dataset when the mixed precision implementation (**fp32fp16**) is compared to the reference (**fp32fp32**). Since the memory usage reduction in FP variables would be 50% if the comparison was with the **fp16fp16** implementation, and non-FP variables are not affected by the use of FP16, it is clear that, with respect to total memory usage, the mixed precision approach is very efficient if large datasets fit in RAM. For a realistic comparison, the figures in Table 8 consider all dynamically allocated memory, including non-FP variables.

Dataset	Memory Saved (%)	Dataset Size (variables)
Soybean	39.2	68,983
Breast Cancer	44.6	18,208
Thyroid	45.5	172,800
Gene	48.8	390,525
MNIST	49.6	55,580,000

Table 8 – Memory usage reduction (from **fp32fp32** to **fp32fp16**) and dataset sizes

Regarding accuracy behavior during training with the mixed precision approach, only one dataset showed statistically significant differences, although it is arguable if they are relevant. The results for the Soybean dataset are compared in Fig. 40. It is clear that, later in the training process, quite after the point of maximum generalization, the approximate implementation showed a slightly inferior resilience to overfitting.

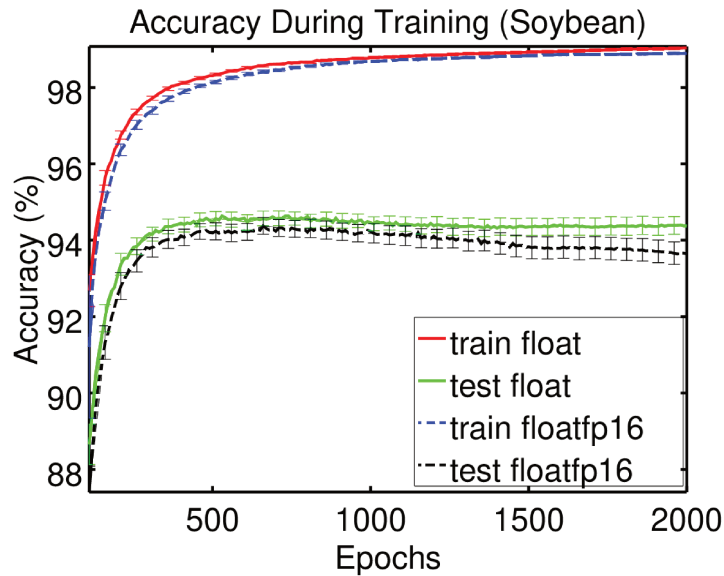


Figure 40 – Soybean accuracy in ARM Cortex A53.

Newer cores like Cortex-A55 and Cortex-A75 support native operations in FP16. When combined with vector based FP instructions (SIMD - Single Instruction Multiple Data) it is expected that FP16 computation achieves a higher throughput. In this context the *exp()* and *sqrt()* approximations may have a higher relative impact on the performance and the complete implementation will most likely outperform the FP32 reference (without

SIMD), without impact on accuracy (since mixed precision is becoming a de-facto standard for such architectures). The experiments presented in this section show evidences which indicate that DL/ML problems based on larger datasets may benefit from the proposed higher level function approximations in the conditions analyzed. This improvement was achieved using native IEEE FP16 representation, without the other proposed FP16 approximations.

7.2 Implementation of Approximated FP16 Operations in FPGA

Preliminary analysis of the complexity reduction obtained from FP16 hardware implementation, presented in Sections 5.1.3 and 5.2.4, provided evidence to support the claim that the approximate behavior requires considerable less resources for FP operations and complex functions. Synthesizing hardware based on source code for generic processors, or any form of High Level Design (HLD), has the obvious advantages of reducing design effort and increasing confidence in the generated modules. Conversely, it is not unexpected that this approach results in less efficient resource usage in the final hardware device, even if it is based on a low level language like C.

Project flows for more efficient hardware designs can be very different depending on the final target (e. g. custom ICs, ASICs or FPGAs) but, it is important to test the implementation in at least one of these options. Mainly for its relatively low cost and short development/test/improve cycles, FPGAs are very popular for such prototyping. This option received one more incentive, as the main manufacturers started to offer programmable hardware integrated with generic processors, internally connected by high-speed buses and direct access to external memory.

The Xilinx Zynq[®]-7000 family integrates ARM[®] processor cores with an FPGA fabric. This architecture is particularly appropriate for this problem, since the CPU can be dedicated to higher level tasks, while exporting specific (highly parallelizable) procedures to be accelerated in the FPGA. Testing is also facilitated, since code running in the processor can use the IEEE compliant FP instructions to verify the FPGA implementation.

The chosen model is the xc7z020clg484, which contains the resources summarized next. It is clear from these specifications that the device has a CPU powerful enough to run a full operating system but the programmable hardware, although powerful, is not among the high-end devices from the same manufacturer.

- CPU dual-core ARM[®] Cortex[®]-A9 (ARMv7-A architecture)
- FPGA Artix[®]-7 with:
 - 53.200 Look-Up Tables (LUTs).

- 106.400 Flip-Flops (FFs).
- 140 RAM units with 36 KiB each (4.9 MiB in total).
- 220 arithmetic processors (DSPs).

7.2.1 Preliminary Floating Point Unit

Choosing a good comparison benchmark is important to evaluate the impact of the variable bias approach. The IEEE FP standard does not detail how the FPUs should be designed, and the optimized implementations are active research and development topics. For this reason, there are no standard or reference implementations of either FP32 or FP16 units. The main FPGA manufacturers provide configurable FP IPs optimized for their products, but their closed form would not allow customizations.

As mentioned in Section 5.1.2, ignoring sub-normal exceptions is the rule in FPGA FP implementations, which excludes the most relevant simplification from the IEEE standard comparison baseline. From the two mentioned open source options, VFloat (WANG; LEESER, 2010) is chosen instead of FloPoCo (DINECHIN; PASCA, 2011), mainly due to its clear, modular and standard design. Additionally, the second one is a code generation tool for specific FPGA models, and uses a different representation format (exceptions are encoded as separate signals).

Two common targets for ANN accelerators, due to their parallelization capabilities and relevant performance impact, are convolutions (which are not covered in this thesis) and FMA/FMAC operations. The two base operations to implement these fused-multiply-add modules were chosen as the base units for comparison, merged in a simple FPU. The exponential approximation could later use the separate multiplication for an implementation focused on space saving. The square root approximation would not provide a fair comparison to VFloat since even its HLS implementation is very small and the counterpart would require merging the division and square root modules. Under these conditions, the implementation and analysis that follows are based on this simple two-operations FPU.

The first attempted approach was to use VFloat as a base VHDL implementation and simplify it to obtain some of the approximation gains. Despite initial satisfactory results regarding size reduction, this modification was abandoned for two reasons: it was not possible to reduce the architecture depth (the amount of sequentially connected modules), without practically re-writing it, to optimize Flip-Flop usage; stress tests showed some *1-bit* round errors. The final implementation for the two main operations (addition and multiplication) is an asynchronous VHDL code, without sequential stages, written from scratch and fine-tuned to catch rounding errors larger than half LSB, including flushes to zero and overflows (which are saturated in the approximated implementation, as described in Chapter 5). The two operations were merged in a simple FPU, without

sharing the common modules (e. g. normalization and rounding), for a fair comparison with the VFLoat counterparts.

The sequential logic which is saved from the FPUs is available to be dedicated to the processing of different neuron layers with the same floating point hardware or implementing the control logic that will manage the inference and back-propagation processes. Since layers are not simultaneously activated, pipelining different training samples in the same batch would be an option, but it may be expensive in terms of memory usage in this application. For the back-propagation phase, intermediate values must be stored for the gradient calculations during each batch. Using FFs for a pipelined FPU would obviously increase the throughput but would require even more complex control circuits.

Using the Xilinx Vivado to perform a post-implementation timing simulations for the approximated multiplication circuit with dynamic bias resulted in latencies from 14 to 18 ns with different operands. For the addition operation, delays from 20 to 24 ns were found. These figures would lead to a worst case throughput in the order of 40 million FP operations per second in each FPU. Enabling the usage of DSPs in the synthesis of the multiplication circuit resulted in slightly reduced delays: from 13 to 16 ns. Even if enabled, the addition synthesis was performed without DSPs.

Table 9 presents the comparison of the implemented modules with the baseline. RAM usage, equal to zero in all designs, is omitted. DSP usage was explicitly disabled in the synthesis tool to allow the results to be extrapolated to other platforms. Since both approaches are fully parameterizable, FP32 representations, with *8-bits* exponents and *23-bits* significands are also included as references to evaluate the gain from a mixed precision solution. The lines marked with **VFLoat** refer to the simple FPU previously described, using the modules from this open source reference. The lines marked with **FP Approx.** refer to the implementation proposed in this thesis, but the adjustable FP bias (designed as an extra operand input) is only enabled in the implementation presented in last line, marked with “+ **Bias**”.

Format (exp. <i>bits</i> , sig. <i>bits</i>)	FPU	LUT	FF
FP32 (8,23)	VFLoat	1195	584
	FP Approx.	1164	2
FP16 (5,10)	VFLoat	386	305
	FP Approx.	421	2
	FP Approx. + Bias	422	2

Table 9 – Comparison of FPGA resources for the VFLoat based FPU with two-operations (multiplication and addition) the respective approximated implementations.

Analyzing the resource usage of a single FPU in the context of the chosen Zynq device, it can be seen that it uses only $\approx 0.79\%$ of the available LUTs. Even without considering that a specific synthesis would reduce this figure in favor of the use of DSP

modules, it is clear that there is room for a considerable amount of parallel operations. The synthesis for the FP32 format indicated a $\approx 2.6\%$ reduction in LUTs with the approximated implementation, but the reduced FP16 FPU was $\approx 9.3\%$ larger than its baseline. This difference will be further analyzed in Section 7.2.3, where a device specific synthesis is performed.

In Chapter 5, Tables 5 and 4 provided comparisons resource usage using HLS and similar baselines to show the effect of the proposed approximations in implementation agnostic modules. Evaluating the feasibility of a simple FPU implemented directly in VHDL, targeting an FPGA with enough room for many parallel operations, is certainly an assuring first step for a full implementation. This also adds evidence to the lack of efficiency of using HLS for complex modules. Despite these results, for a full realization of this acceleration, many aspects related to the chosen platform must be analyzed.

All parameters should fit in the internal RAM to reduce the communication between the processors and the FPGA. It is worth noting that LUTs can also be used as RAM, but the amount of available memory is only appropriate to simpler problems, which are commonly related to the embedded systems. Despite this limitation, the amount of FPGA memory blocks provided by the Zynq device should not be considered too small for practical applications. The recently released Sipeed M1/M1W “AI Processor”¹, includes 2 MiB of RAM for its CNN (Convolutional Neural Network) accelerator while the other 6 MiB are used by two 64 *bits* RISC-V cores.

There are considerable differences between the operations in the input layer (which receives the data examples sequentially from the interconnection bus), the hidden layers (which receive all the inputs simultaneously) and the output layer (which must calculate the output error and start the back-propagation process). As a next step in this viability analysis, Section 7.2.2 proposes a mechanism to create memory mapped blocks to be used as separate and distributed storage for different ANN layers in the specific ARM-FPGA device being considered.

7.2.2 Communication between CPU and FPGA

Very simple ANN architectures for FPGA have been proposed, like the one in (SAHIN; BECERIKLI; YAZICI, 2006), with a single hidden layer with three neurons, using FP32 FPUs. The paper (HIMAVATHI; ANITHA; MUTHURAMALINGAM, 2007) presented a slightly larger one, with up to three hidden layers with 5 neurons each, using more restricted fixed point arithmetic (mixing 9, 17 and 25 *bits* for different parameters). Recently, accelerating convolution operation is becoming more common, like seen in (ZHANG et al., 2015a) and (QIU et al., 2016), since practical results with popular benchmark datasets are achievable. These accelerators allow the offloading of specific tasks

¹ Sipeed M1/M1W with Kendryte’s K210 AI chip: <<https://wiki.sipeed.com/en/maix/module/m1.html>>

to the FPGA while the higher level (and more complex) training process is implemented in a general purpose processor. Vendors like Intel and Xilinx have been offering integrated CPU-FPGA devices, connected by specific buses, which must be used efficiently in a full implementation. This Section evaluates one of these interconnection mechanisms, allowing a full implementation which could be based on the modified FANN library, described in Chapter 4, offloading the FP16 operations to the FPGA.

The Xilinx Zynq[®]-7000 integrated IC provides a high performance standard communication bus between the CPU and the FPGA: the Advanced eXtensible Interface (AXI), which is part of the ARM Advanced Microcontroller Bus Architecture 4. Xilinx has chosen ² AXI as a standard interface for its IP (Intellectual Property) blocks, and provides modules to implement the following communication methods:

AXI4-Lite the simplest memory mapped interface, using fewer signals and less logic, requiring each data transfer to be associated with its respective address

AXI4-Full achieves higher throughput using more signals and logic for state control, allowing burst transfers (sequenced data associated with a single base address)

AXI4-Stream provides a data-flow mechanism, without explicit memory mapping (with extra modules is used to implement asynchronous DMA transfers)

Due to its simplicity, AXI4-Lite was used for the tests during the implementations presented in Section 7.2.1. Conversely, the ANN accelerator should use a more efficient method for data transfer since the data presented in the inputs is organized as a vector, always with the same size and order. That favors the “Full” interface when compared with the “Lite” one, but does not justify the extra cost (both in FPGA resources and CPU code) to handle asynchronous DMA transfers. Without mini-batch parallelization or pipelining, ANN training is synchronous, since the error obtained from one input must be back-propagated before the next one is processed.

Due to the fact that AXI is a master-slave point-to-point interface, Xilinx provides an “AXI SmartConnect” module, which transparently maps several independent slave modules to a single master. Each slave module has a relatively small mapped memory limit (1 KiB, organized as 256 32 *bits* words), viewed as a different base address vector from the application code running in the CPU. If memory usage can be divided between read-only and write-only blocks, this limit can be doubled. Figure 41 presents a block diagram with two slave modules: each one is an instance of the same user defined IP (fully customizable), with the maximum supported amount of internal read/write memory (the FPU was not integrated in this experiment).

² Xilinx AXI Reference Guide:<https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf>

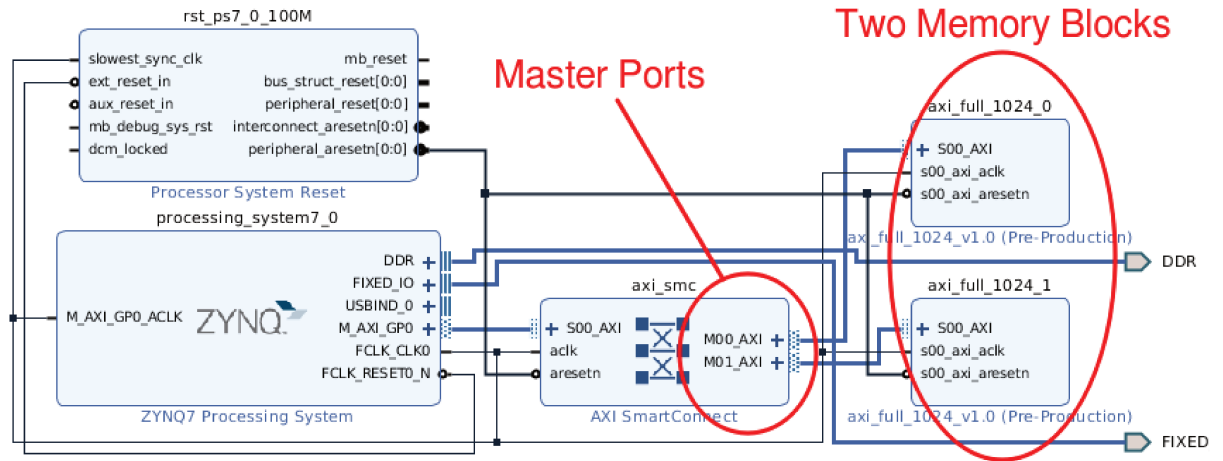


Figure 41 – Zynq subsystem block diagram with two memory blocks.

To evaluate how the AXI4-Full memory modules would scale, the same slave IPs were repeatedly inserted (with different base memory addresses), and each new configuration had its implementation resources measured. This verification was important since it was previously verified that the AXI4-Lite modules did not scale well when increased to their maximum memory capacity: the Vivado synthesis reported that the module was “not ideal for floorplanning”, recommending a hierarchical approach. Except when increasing the design from one to two slave modules, which resulted in an increase of 2637 LUTs and 4021 FFs, further additions resulted in a very linear resource usage increase, as depicted in Figure 42. Regarding BRAM usage, each new module resulted in 2 blocks allocated.

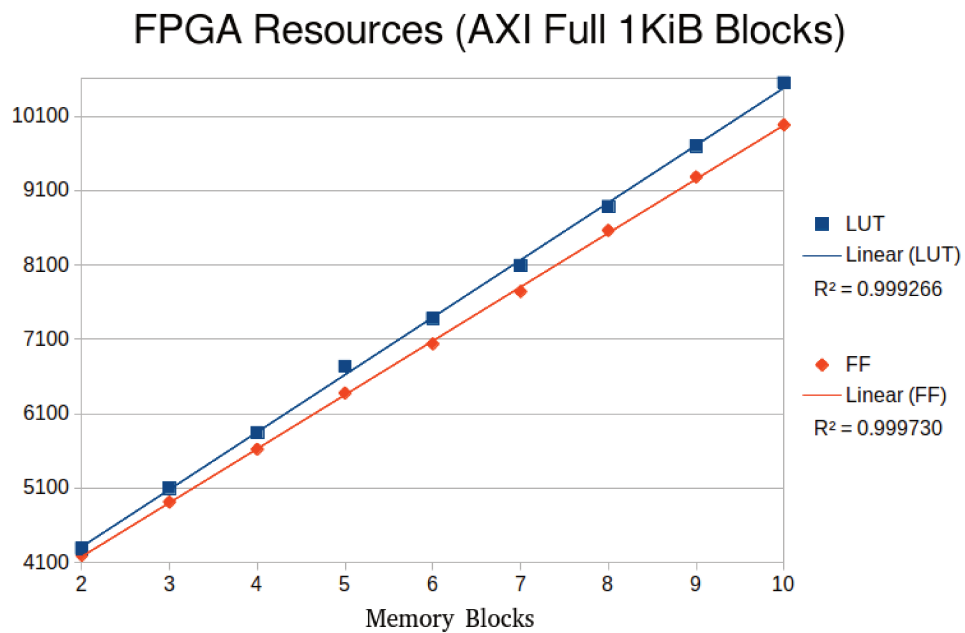


Figure 42 – LUTs and FFs allocated for each memory block added to the bus. Each separate AXI slave memory block could implement an ANN layer.

7.2.3 Structure for a multi-layer ANN in FPGA

Merging the blocks presented in Sections 7.2.1 and 7.2.2, this Section closes this Chapter by presenting a basic structure that could be used to implement fully connected ANNs using the techniques proposed in this thesis. As a proof-of-concept, to estimate the feasibility of an ANN with a considerable amount of parallelism, each layer included 8 FPU hardwired to specific memory mapped inputs, performing FMA operations independently. All blocks contain 2 KiB of memory mapped to the CPU using the AXI bus, additionally to separate buffers for direct connection between layers. The memory mapped block is divided in half, to map one part to write only memory and the other for read only operations. Regarding the direct connections between the blocks, three types of IPs were created, as follows:

input dedicated output connections only send data to the next layer

hidden dedicated connections are available for input and output, connecting this IP to two other layers

output the dedicated connections are used only for data input

Figures 43 and 44 detach only the four ANN layers from the block diagram of the full proof-of-concept to indicate how the layer interconnection is performed (the omitted modules are the same ones depicted in Figure 41). The signals prefixed with “s00_axi_” are used to connect each layer to its respective master port at the “AXI SmartConnect”. The other connections are simple master-slave indexed transfers of separate buffers, not mapped to the CPU.

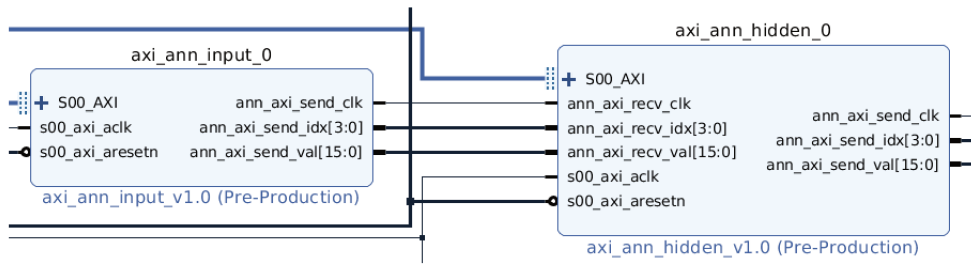


Figure 43 – Inputs and outputs of the input layer and the first hidden one.

The full design was synthesized for the target Xilinx Zynq®-7000 device, without any constraints regarding the implementation process. The total resource usage is indicated in Table 10 in both absolute numbers and relative usage of the target device. The large difference in FF usage is coherent with the design decision of a non-pipelined FPU architecture in the approximated FPU. The requirement of twice the amount of DPSs demand a further analysis, as follows.

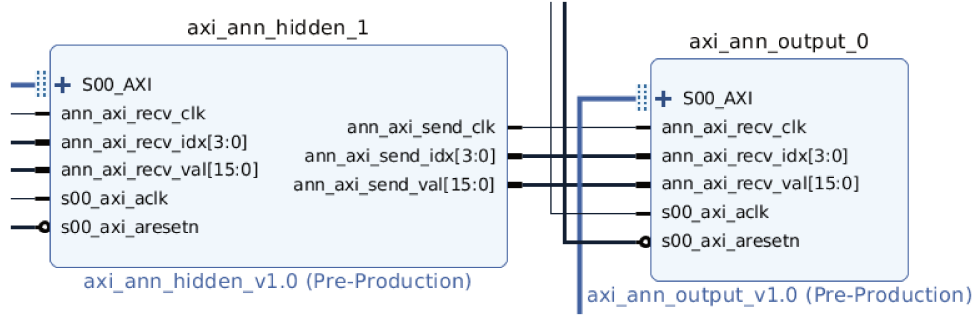


Figure 44 – Inputs and outputs of the second hidden layer and the output one.

Resources	FPU (FP16)	LUT	FF	DSP
Absolute	VFloat (5,10)	13,464	16,253	32
	FP Aprox. (5,10)	14,001	8,381	64
Relative	VFloat (5,10)	25.3%	15.3%	14.5%
	FP Aprox. (5,10)	26.3%	7.9%	29.1%

Table 10 – Comparison of FPGA resources after implementation

In Section 7.2.1, Table 9 presented results for a synthesis process with DSP usage explicitly disabled (to allow the results to be extrapolated to other platforms). If this restriction is removed, the synthesis tool allocates the internal DSPs available in the device for the arithmetic operations required for the FP calculations. The comparison of the FP32 FPU's show that the same amount of DSPs is required in both implementations. Conversely, with the shorter FP16 format an extra DSP is allocated by the synthesis tool, regardless of the dynamic FP bias feature. The difference in the results for the FP32 and FP16 implementations, which are realized by a simple change of two generic parameters (without extra changes in the VHDL code), indicate that there may be room for optimization in the approximated implementation, regarding the internal bit-widths in the FPU's.

Format (exp. bits, sig. bits)	FPU	LUT	FF	DSP
FP32 (8,23)	VFloat	580	555	2
	FP Aprox.	581	2	2
FP16 (5,10)	VFloat	282	285	1
	FP Aprox.	294	2	2
	FP Aprox. + Bias	312	2	2

Table 11 – Comparison of FPGA resources after implementation

The viability analysis presented in this Section was important to verify if the techniques proposed in this thesis were implementable in current of-the-shelf hardware. The resource usage results indicate that there is a considerable amount of space to implement the complex control logic which will guide the training process. The implementation of a framework which generates HDL code based on a high level description of the ANN would be an interesting research project from the current status of this work.

8 Conclusions and Future Directions

In this Chapter the main achievements and limitations of the research conducted for this thesis are summarized in Section 8.1. Most of the contributions of this research have been published in the paper (TORRES; TORRES, 2019). A few more investigations that are worth analyzing, possibly still in the simulation environment, are listed in Section 8.2. Section 8.3 discusses the plans and possible challenges regarding the implementation of the adaptive training methods in FPGAs. The final remarks and conclusions about the status of this work close this document in Section 8.4.

8.1 Analysis of the current status

Early in this research an investigation method was defined to focus the comparisons on the possible effects that different precisions could have on training performance. Simple comparisons of final test accuracies would hide different convergence trajectories and possible effects of increased generalization, commonly found in approximated solutions. To avoid losing relevant information in unbalanced datasets, the average class accuracy was initially used instead of the global one. This was further improved to the geometrical mean, which penalizes even more the minority class poor performance due to “catastrophic forgetting” and also in the beginning of the training process. In every comparison, the same structures (connections, nodes per layer, activation and loss functions), weight initializations, hyper-parameter adjusts were the same and even the pseudo-random training examples presentation order was preserved. This methodology allowed the verification of relevant effects that helped considerably in the investigations.

The first comparison reduced the floating point precision from FP64 to FP16 using a simple GD method for training. One of the datasets presented considerable performance decrease in the lower precision tests and further analysis indicated that the subnormal range of the standard FP16 was easily reached in these trials. Additional simplifications to the FP16 arithmetic were presented and approximations for the exponential and reciprocal square root functions were adapted to this reduced FP format. A new approximation for the division by the square root was also proposed. When compared to the standard FP16 representation, no evidence of differences in the performance was found between the two implementations. The approximated FP16 software implementation, based on native integer operations, resulted in a $\approx 53\%$ smaller compiled binary when compared to the standard one. High level synthesis was used to migrate the FP16 implementations to an FPGA, which allowed the verification that the space gains were still significant for the basic arithmetic operations and also for the complex functions.

Regarding the adaptive training algorithms, the first step was the addition of a standard momentum term. This modification provided better accuracy results than regular GD. Despite this, the new term was still not able to provide equivalent results between FP64 and the approximated FP16 in one dataset. For simpler datasets, which allow full-batch training, RProp was evaluated, but could not achieve equivalent performance for all tested problems without modifications and new hyper-parameters. As an option for large datasets, RMSProp was also tested, with minor modifications, and presented good results regarding convergence speed, accuracy and stability. Despite these results, a method to provide equivalent behavior with unmodified training algorithms was proposed. This feature consisted in an automatic and light-weight method to change the representation range of FP values in **Group 3**, related to back-propagation, for each neuron, by adapting the FP bias value. This mechanism provided comparable accuracy results without relying in the modifications in the training algorithms to handle “vanishing gradient” and underflow problems. The full method with the dynamic FP bias was later tested with a larger set of unbalanced problems, including comparisons with the recently proposed **bfloat16** and **posit16** formats, achieving excellent results.

Despite the good results it should be noted that in some cases the proposed method has shown less resistance to overfitting. This characteristic would not affect a full implementation with specific regularization techniques, since the training process would stop before the generalization is compromised. In fact, it could be considered an evidence for the resilience of the proposed method, which does not stagnate the learning process when gradient values become to low, and continues to improve the training error beyond the point of best generalization. A discussion about future works related to regularization can be found in Section 8.2.2, but it is an important note for this Section to observe that the current status of equivalence, without relying on specific training algorithms, may indicate that further approximations are still possible.

The main limitation of the current full implementation is the training time. Emulating all FP arithmetic operations with native integer instructions in a general purpose CPU is extremely flexible for development purposes but it increases training time by an order of magnitude when compared to native FP instructions. A partial speedup can be obtained by using the approximated FP16 only for storage and implementing the FP operations using native instructions. Unfortunately this is still 4 to 5 times slower than a fully native FP implementation. All the time measurements just mentioned were performed with the MNIST dataset, which can be considered simple when compared to recent Deep Learning applications, with large datasets. For smaller models, more frequently found in embedded platforms, the first step to an optimized accelerator in FPGAs was realized, with a VHDL implementation of a simple ALU. A comparison of the resource usage confirmed that the final design, with variable FP bias, is comparable to a reference in the literature in combinational logic, but practically reducing to zero the cost in sequential logic, due to its

architecture. With a specific CPU/FPGA IC as an example, a basic structure for a fully connected ANN was synthesized, with promising results regarding the amount of used resources and their scalability to larger network structures.

8.2 Future Improvements

The investigation method detailed in Section 8.1 will not be adequate for a statistically significant comparisons to other approaches in the literature. For this to be possible a full training method must be implemented to stop the learning process and provide a final test accuracy for each trial. Besides this required change, other possible improvements may be analyzed, and are detailed in this Section. It is important to note that these proposals may be attempted after the hardware implementation, which is discussed in Section 8.3.

8.2.1 Training Methods

The RProp algorithm may be considered to have a limited reach for complex problems, due to its intrinsic dependency on full-batch training. Conversely, its simplicity and resiliency to avoid an expensive hyper-parameter search deserve extra efforts which could allow its reliable use with approximated methods. The iRProp- variation, as an example, presents the desirable characteristics of not requiring the actual calculation of the loss function value and also performing its weight adjustments without depending on the precise value of the gradient, but relying only in its sign. The initial idea of restarting the learning process for specific weights may be revisited and improved for more aggressive approximated formats.

A modified version of RMSProp was proposed by adding a momentum term to the “fall-back” scenario described in Section 6.1.3, but equivalent convergence performance was later obtained without relying on this feature (due to the dynamic FP bias adjustments). This mechanism may have its effectiveness reevaluated with more aggressively approximated formats, as a method to recover from the situations where small gradient values are flushed to zero, harming the fine-tuning phase of the learning process.

If the opportunity to evaluate other training algorithms is presented, another adaptive proposal with similar resource usage to RMSProp (a single extra parameter for each weight) is Adagrad (MUKKAMALA; HEIN, 2017), which also shares the same division by square root approximation already implemented. More complex options in terms of resource usage (both memory and computing), like Adam (KINGMA; BA, 2014), may also be studied but that could be postponed as a possible improvement after the full FPGA implementation, if the available hardware capabilities allow this extra demand.

8.2.2 Regularization

Regularization was explicitly excluded in the experiments as a mechanism to improve the chance of better generalization capabilities (due to limited precision) being observable. This effect has been verified with relevant magnitudes in specific cases, which means that more precision in these conditions resulted in slightly worse results. A fully implemented training algorithm will include a form of regularization, which may have beneficial effects also in these cases.

It is a well known fact (DENIL et al., 2013) that complex ANN models are frequently over-parameterized, which means that not only they can be simplified after training but also have better resilience to approximations (as seen in Section 3.1). Regularization provides a way to avoid the inherent overfitting tendency in these structures. This feature may be implemented by, for example: stopping the learning process early (based on a separate validation dataset); reducing the model complexity (without affecting the maximum accuracy or convergence capabilities); adding a penalization to large weights in the loss function (method known as *Weight Decay*). A cost evaluation of such approaches should be performed in the FPGA implementation.

Implementing the bias adjustments associated to an efficient regularization method, like *DropOut* (SRIVASTAVA et al., 2014), may be able to avoid the occasional negative effects regarding the overfitting resistance. Batch Normalization (IOFFE; SZEGEDY, 2015) is also a promising strategy to avoid drastic changes in the value ranges during training that could alter the ideal bias selection. By inserting normalization layers before the non-linearities, the network calculates the mean and variance for each input dimension within a mini-batch. After this procedure, the inputs are normalized and may be shifted and scaled by learnable parameters, so that each layer may adapt to different means and standard deviations using a standard *back-propagation* process. The authors claim that the method allows much higher learning rates, is more robust to bad initializations and even provides regularization. Unfortunately, the cost of this method is considerable and this may affect the feasibility of a full hardware implementation.

Stochastic rounding has received a lot of attention recently, when approximate computing techniques are used for training ANNs to solve problems with large datasets, without resorting to mixed precision (DALLY, 2015). The noise added to the training process, more relevant at the fine tuning stage, has been related to better generalization. This technique has the advantage to be directly related to the arithmetic operations implementation, and not to the training method.

In the AC literature related to ML, it is quite common to see slightly better generalization results for the approximated method being related to “pseudo-random” quantization noise. Adding noise to training (to the datasets, to weight values or even

to their adjustments) has been known for decades as a way to improve generalization. A good theoretical and generic analysis of quantization errors as a way to automatically provide this noise is still required. Other ways to explore these results are also possible: cancellations may impede small adjustments to weights, limiting their growth, which is also a regularization technique; weights with small values may lose their effectiveness since multiplications by other small values may be flushed to zero (this could be compared to a form of Dropout or Pruning, also affecting generalization).

8.3 FPGA Implementation

In order to validate the results obtained in this research, the presented methods should be implemented in FPGA. The intrinsic characteristics of FPGAs have been explored in efficient ANN implementations for a long time, e.g. (BOTROS; ABDUL-AZIZ, 1994). As these components evolve and increase their capacity, their applicability to more complex problems becomes more evident, like recently shown in (LIU et al., 2017). When compared to the optimizations proposed in this thesis, it should be noted that this latest example is focused on the inference phase, which allows significant approximations on parameters, inputs and internal operations.

This research is focused on the learning phase, which includes the inference with higher precision and adds considerable complexity to the system for handling all the operations involved in *back-propagation* and parameter adjustments. This extra resource usage certainly limits the ANN structure and model size but the intrinsic parallelization of FPGAs can be fully explored. The implementation should ideally limit the communications between the FPGA and external memory for maximum efficiency. For this reason, not only the logic elements required to implement arithmetic operations but also the memory blocks used for parameters and internal variables should be carefully considered.

Key concepts for a specific IC which merges an FPGA to two ARM cores were presented in Section 7.2. This device is an example of a recent trend of encapsulating FPGAs and CPUs in the same package, which should provide efficient solutions for embedded platforms. In these cases, even if the FPGA does not have the capacity to implement larger models, the flexibility of a general purpose CPU closely integrated with a reconfigurable and highly efficient parallel hardware may represent excellent optimization possibilities for several applications. Since one of the targets for the approximations proposed here are the embedded applications, such devices are an attractive option for the hardware realization and the preliminary results presented in this thesis allow an optimistic expectation on the complexity of real problems which can be optimized.

Depending on the FPGA capacity, the ANN training process provides even more parallelization opportunities than the inference phase. If all the nodes and parameters fit

in the component and a mini-batch process is performed, not only the neurons in each layer may operate in parallel but also sequential layer operations may be pipelined. In this case, the inputs and the intermediate activations must be temporarily stored to allow the *back-propagation* calculations. Most of the datasets used in this work require simple ANNs with two small fully connected hidden layers. For this reason, and comparing with recent Deep ANN research on FPGAs, high-end devices are not expected as a requirement to reproduce in hardware most of the results presented in this thesis.

Even simple architectures like the one proposed in (ÇAVUŞLU et al., 2011) may provide relevant results when a specific baseline for comparison is improved, which in this case is (SAVICH; MOUSSA; AREIBI, 2007). Deep ANNs are certainly an interesting topic, but their full implementation is not required in FPGA accelerators. In (ZHANG et al., 2015a), for example, only the convolution operations are optimized and the authors used High-Level Synthesis and FP32 representations in their approach. There are many possibilities to explore if the implementation is limited by a hardware with a lower specification grade.

8.4 Conclusion and Final Remarks

As ML applications currently demand efficient implementations both in embedded low power platforms and in high-end massively parallel solutions, the research community has been offering several approaches to implement such optimized systems. Neuromorphic Computing, which regained momentum, and Quantum Computing may lead to new paradigms for the field in the medium and long terms. For the short term, the variety of recent solutions applicable to the current technology of digital circuits (FPGAs, GPUs and ASICs) confirms that the state-of-the-art in this area is still evolving quickly.

This research approached the problem of implementing efficient and reliable ANN training methods by applying AC techniques to reduce the complexity of hardware implementations. Besides the optimizations on arithmetic operations, modifications to adaptive learning methods were proposed and evaluated as mechanisms to cope with the small values found during the training process. Despite the good results, an approach that is more in line with the hardware approximations was developed: a fully automatic and light-weight mechanism to adapt the FP representation range of each neuron during training. In this Chapter, several possible venues for further improvements and the final FPGA implementation were discussed. Despite the variety of possibilities, the current status of the work, supported by the simulation results and preliminary FPGA synthesis, indicates a promising scenario for the hardware implementation.

8.5 Contributions

During the time dedicated to this research, the following contributions were published in peer-reviewed publications and conferences:

- **Ferreira Torres, Vitor Angelo**; Torres, Frank Sill. ***A Comparative Analysis of Neural Networks Implemented with Reduced Numerical Precision and Approximations***. In: ENIAC - 2017 XIV Encontro Nacional de Inteligência Artificial e Computacional, Uberlândia, October 2-3-4-5, 2017. p. 193-204.
- Eduardo Ribeiro ; **Ferreira Torres, Vitor Angelo** ; Brayan Jaimes ; Mateus Braga ; Elcio Shiguemori ; Haroldo Velho ; Luiz Torres ; Antônio Braga. ***Weightless neural systems for deforestation surveillance and image-based navigation of UAVs in the Amazon forest***. In: ESANN2019 27th European Symposium on Artificial Neural Networks, 2019, Bruges, Belgium, April 24-25-26, 2019. p. 153.
- **Vitor Torres**; Brayan Jaimes; Eduardo Ribeiro; Mateus Braga; Elcio Shiguemori; Haroldo Velho; Luiz Carlos B Torres; Antonio Braga. ***Combined Weightless Neural Network FPGA Architecture for Deforestation Surveillance and Visual Navigation of UAVs***, Engineering Applications of Artificial Intelligence, Elsevier, 2020, Qualis A1, DOI: 10.1016/j.engappai.2019.08.021
- **Ferreira Torres, Vitor**; Torres, Frank Sill. ***Resilient Training of Neural Network Classifiers with Approximate Computing Techniques for Hardware-optimized Implementations***, Computers & Digital Techniques, IET, 2019, Qualis B1, DOI: 10.1049/iet-cdt.2019.0036. Temporary URL: <<http://ietdl.org/t/UiWFdb>>

Bibliography

AGARWAL, V. et al. Clock rate versus ipc: The end of the road for conventional microarchitectures. In: ACM. *ACM SIGARCH Computer Architecture News*. [S.l.], 2000. v. 28, n. 2, p. 248–259. Cited on page 1.

AGRAWAL, A. et al. Approximate computing: Challenges and opportunities. In: IEEE. *Rebooting Computing (ICRC), IEEE International Conference on*. [S.l.], 2016. p. 1–8. Cited 2 times on pages 1 and 28.

ALBALAWI, H.; LI, Y.; LI, X. Computer-aided design of machine learning algorithm: Training fixed-point classifier for on-chip low-power implementation. In: IEEE. *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. [S.l.], 2014. p. 1–6. Cited 3 times on pages 24, 28, and 29.

ALEKSANDER, I. Self-adaptive universal logic circuits. *Electronics Letters, IET*, v. 2, n. 8, p. 321–322, August 1966. ISSN 0013-5194. Cited on page 9.

ALEKSANDER, I.; THOMAS, W.; BOWDEN, P. Wisard: a radical step forward in image recognition. *Sensor review*, MCB UP Ltd, v. 4, n. 3, p. 120–124, 1984. Cited 2 times on pages 9 and 18.

ALTERA. *Floating-Point IP Cores User Guide*. [S.l.], 2016. Altera Corporation (now part of Intel). Cited on page 47.

ANDRYSCO, M. et al. On subnormal floating point and abnormal timing. In: IEEE. *Security and Privacy (SP), 2015 IEEE Symposium on*. [S.l.], 2015. p. 623–639. Cited on page 32.

ANGUITA, D.; BONI, A.; RIDELLA, S. A digital architecture for support vector machines: theory, algorithm, and fpga implementation. *IEEE Transactions on neural networks*, IEEE, v. 14, n. 5, p. 993–1009, 2003. Cited 3 times on pages 19, 28, and 29.

ANGUITA, D.; BONI, A.; RIDELLA, S. Svm learning with fixed-point math. In: IEEE. *Neural Networks, 2003. Proceedings of the International Joint Conference on*. [S.l.], 2003. v. 3, p. 2072–2076. Cited 4 times on pages 15, 19, 28, and 29.

ANGUITA, D. et al. Feed-forward support vector machine without multipliers. *IEEE Transactions on Neural Networks*, IEEE, v. 17, n. 5, p. 1328–1331, 2006. Cited 3 times on pages 19, 28, and 29.

ARNOLD, M. et al. On the cost effectiveness of logarithmic arithmetic for backpropagation training on simd processors. In: IEEE. *Neural Networks, 1997., International Conference on*. [S.l.], 1997. v. 2, p. 933–936. Cited 3 times on pages 20, 28, and 29.

AYINALA, M.; PARHI, K. K. Low-energy architectures for support vector machine computation. In: IEEE. *Signals, Systems and Computers, 2013 Asilomar Conference on*. [S.l.], 2013. p. 2167–2171. Cited 3 times on pages 20, 28, and 29.

BAILEY, T. M. Convergence of rprop and variants. *Neurocomputing*, Elsevier, v. 159, p. 90–95, 2015. Cited on page 64.

- BHARATI, K. S.; JHUNJHUNWALA, A. Implementation of machine learning applications on a fixed-point dsp. In: IEEE. *Electrical and Computer Engineering (CCECE), 2015 IEEE 28th Canadian Conference on*. [S.l.], 2015. p. 1458–1463. Cited 3 times on pages 24, 28, and 29.
- BISHOP, C. M. Training with noise is equivalent to tikhonov regularization. *Neural computation*, MIT Press, v. 7, n. 1, p. 108–116, 1995. Cited on page 68.
- BLEDSE, W. W.; BROWNING, I. Pattern recognition and reading by machine. In: ACM. *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*. [S.l.], 1959. p. 225–232. Cited on page 9.
- BLINN, J. F. Floating-point tricks. *IEEE Computer Graphics and Applications*, IEEE, v. 17, n. 4, p. 80–84, 1997. Cited on page 51.
- BOSER, B. E.; GUYON, I. M.; VAPNIK, V. N. A training algorithm for optimal margin classifiers. In: ACM. *Proceedings of the fifth annual workshop on Computational learning theory*. [S.l.], 1992. p. 144–152. Cited on page 19.
- BOSMAN, H. H. et al. Anomaly detection in sensor systems using lightweight machine learning. In: IEEE. *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*. [S.l.], 2013. p. 7–13. Cited 3 times on pages 23, 28, and 29.
- BOTROS, N. M.; ABDUL-AZIZ, M. Hardware implementation of an artificial neural network using field programmable gate arrays (fpga's). *IEEE Transactions on Industrial Electronics*, IEEE, v. 41, n. 6, p. 665–667, 1994. Cited 2 times on pages 9 and 96.
- BURGESS, N. et al. Bfloat16 processing for neural networks. In: IEEE. *Proceedings of the 26th Symposium on Computer Arithmetic*. [S.l.], 2019. p. 88–91. Cited on page 61.
- CASTRO, C. L. de. *Novos critérios para seleção de modelos neurais em problemas de classificação com dados desbalanceados*. Tese (Doutorado) — Universidade Federal de Minas Gerais - Programa de Pós-Graduação em Engenharia Elétrica, 10 2011. Supervisor: Prof. Antônio de Pádua Braga. Cited on page 75.
- CASTRO, H. A.; SWEET, M. R. Radiation exposure effects on the performance of an electrically trainable artificial neural network (etann). *IEEE transactions on nuclear science*, IEEE, v. 40, n. 6, p. 1575–1583, 1993. Cited on page 11.
- CASTRO, H. A.; TAM, S. M.; HOLLER, M. A. Implementation and performance of an analog nonvolatile neural network. *Analog Integrated Circuits and Signal Processing*, Springer, v. 4, n. 2, p. 97–113, 1993. Cited on page 11.
- ÇAVUŞLU, M. A. et al. Neural network training based on fpga with floating point number format and it's performance. *Neural Computing and Applications*, Springer, v. 20, n. 2, p. 195–202, 2011. Cited 2 times on pages 9 and 97.
- CHANG, C.-C.; LIN, C.-J. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, Acm, v. 2, n. 3, p. 27, 2011. Cited on page 19.
- CHIPPA, V. K. et al. Analysis and characterization of inherent application resilience for approximate computing. In: ACM. *Proceedings of the 50th Annual Design Automation Conference*. [S.l.], 2013. p. 113. Cited 3 times on pages 23, 25, and 29.

- CHIPPA, V. K. et al. Managing the quality vs. efficiency trade-off using dynamic effort scaling. *ACM Transactions on Embedded Computing Systems (TECS)*, ACM, v. 12, n. 2s, p. 90, 2013. Cited 3 times on pages 23, 28, and 29.
- CHO, K.-J. et al. Design of low-error fixed-width modified booth multiplier. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, IEEE, v. 12, n. 5, p. 522–531, 2004. Cited on page 20.
- COATES, A. et al. Deep learning with cots hpc systems. In: *International Conference on Machine Learning*. [S.l.: s.n.], 2013. p. 1337–1345. Cited on page 2.
- COONEN, J. T. Underflow and the denormalized numbers. *Computer*, IEEE, n. 3, p. 75–87, 1981. Cited on page 31.
- CORTES, C.; VAPNIK, V. Support-vector networks. *Machine learning*, Springer, v. 20, n. 3, p. 273–297, 1995. Cited 2 times on pages 1 and 19.
- COURBARIAUX, M.; BENGIO, Y.; DAVID, J.-P. Training deep neural networks with low precision multiplications. *arXiv:1412.7024 (Workshop contribution at ICLR 2015)*, 2014. Cited 3 times on pages 14, 28, and 29.
- COURBARIAUX, M.; BENGIO, Y.; DAVID, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In: *Advances in Neural Information Processing Systems*. [S.l.: s.n.], 2015. p. 3123–3131. Cited 3 times on pages 15, 28, and 29.
- DALLY, W. High-performance hardware for machine learning. *NIPS Tutorial*, 2015. Cited on page 95.
- DAS, D. et al. Mixed precision training of convolutional neural networks using integer operations. *Accepted Paper at the Sixth International Conference on Learning Representations*, 2018. ArXiv:1802.00930. Cited 3 times on pages 17, 28, and 29.
- DENIL, M. et al. Predicting parameters in deep learning. In: *Advances in Neural Information Processing Systems*. [S.l.: s.n.], 2013. p. 2148–2156. Cited on page 95.
- DIAS, F. M.; ANTUNES, A.; MOTA, A. M. Artificial neural networks: a review of commercial hardware. *Engineering Applications of Artificial Intelligence*, Elsevier, v. 17, n. 8, p. 945–952, 2004. Cited on page 10.
- DIAS, M. A.; SALES, D. O.; OSORIO, F. S. Automatic generation of luts for hardware neural networks. *Neurocomputing*, Elsevier, v. 180, p. 108–120, 2016. Cited 2 times on pages 26 and 36.
- DINECHIN, F. D.; PASCA, B. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, IEEE, v. 28, n. 4, p. 18–27, 2011. Cited 2 times on pages 47 and 85.
- DRUMOND, M. et al. Training dnns with hybrid block floating point. In: *Advances in Neural Information Processing Systems*. [S.l.: s.n.], 2018. p. 453–463. Cited 3 times on pages 17, 28, and 29.
- DUNNE, R. A.; CAMPBELL, N. A. On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function. In: *Proc. 8th Aust. Conf. on the Neural Networks, Melbourne, 181*. [S.l.: s.n.], 1997. v. 185. Cited on page 26.

- ESMAEELI, S.; GHOLAMPOUR, I. Reduced memory requirement in hardware implementation of svm classifiers. In: IEEE. *Electrical Engineering (ICEE), 2012 20th Iranian Conference on*. [S.l.], 2012. p. 46–50. Cited 3 times on pages 20, 28, and 29.
- ESMAEILZADEH, H. et al. Dark silicon and the end of multicore scaling. In: ACM. *ACM SIGARCH Computer Architecture News*. [S.l.], 2011. v. 39, n. 3, p. 365–376. Cited on page 1.
- FERNANDEZ-DELGADO, M. et al. Direct parallel perceptrons (dpps): fast analytical calculation of the parallel perceptrons weights with margin control for classification tasks. *IEEE transactions on neural networks*, IEEE, v. 22, n. 11, p. 1837–1848, 2011. Cited 3 times on pages 13, 28, and 29.
- GEPPERTH, A.; HAMMER, B. Incremental learning algorithms and applications. In: *European Symposium on Artificial Neural Networks (ESANN)*. [S.l.: s.n.], 2016. Cited on page 2.
- GLOROT, X.; BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. [S.l.: s.n.], 2010. p. 249–256. Cited on page 35.
- GOLDBERG, D. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 23, n. 1, p. 5–48, mar. 1991. ISSN 0360-0300. Disponível em: <<http://doi.acm.org/10.1145/103162.103163>>. Cited on page 32.
- GOODFELLOW, I. J. et al. Maxout networks. *ICML (3)*, v. 28, p. 1319–1327, 2013. Cited 3 times on pages 15, 28, and 29.
- GUPTA, S. et al. Deep learning with limited numerical precision. In: *ICML*. [S.l.: s.n.], 2015. p. 1737–1746. Cited 3 times on pages 14, 28, and 29.
- GUSTAFSON, J. The end of numerical error. In: *ARITH 22*. [S.l.: s.n.], 2015. p. 74. Cited on page 79.
- GUSTAFSON, J. L. *The End of Error: Unum Computing*. [S.l.]: Chapman and Hall/CRC, 2015. Cited on page 79.
- GUSTAFSON, J. L. A radical approach to computation with real numbers. *Supercomputing frontiers and innovations*, v. 3, n. 2, p. 38–53, 2016. Cited on page 79.
- GUSTAFSON, J. L.; YONEMOTO, I. T. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, v. 4, n. 2, p. 71–86, 2017. Cited 3 times on pages viii, 78, and 79.
- HAN, J.; ORSHANSKY, M. Approximate computing: An emerging paradigm for energy-efficient design. In: IEEE. *Test Symposium (ETS), 2013 18th IEEE European*. [S.l.], 2013. p. 1–6. Cited on page 1.
- HAN, S.; MAO, H.; DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. Cited 3 times on pages 16, 28, and 29.

- HAN, S. et al. Learning both weights and connections for efficient neural network. In: *Advances in Neural Information Processing Systems*. [S.l.: s.n.], 2015. p. 1135–1143. Cited 2 times on pages 16 and 61.
- HASHEMI, S. et al. Understanding the impact of precision quantization on the accuracy and energy of neural networks. *arXiv:1612.03940 (Accepted for conference proceedings in DATE17)*, 2016. Cited 3 times on pages 16, 28, and 29.
- HAUSER, J. *Berkeley SoftFloat*. 2017. <<http://www.jhauser.us/arithmetric/SoftFloat.html>>. Accessed: 2017-06-26. Cited on page 38.
- HAYKIN, S. *Neural Networks: A comprehensive foundation*. 2. ed. [S.l.]: Prentice Hall, 1998. 842 p. ISBN 978-0132733502. Cited on page 9.
- HENRY, G.; TANG, P. T. P.; HEINECKE, A. Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations. In: IEEE. *Proceedings of the 26th Symposium on Computer Arithmetic*. [S.l.], 2019. p. 69–76. Cited on page 61.
- HIMAVATHI, S.; ANITHA, D.; MUTHURAMALINGAM, A. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, IEEE, v. 18, n. 3, p. 880–888, 2007. Cited on page 87.
- HINTON, G. E.; SALAKHUTDINOV, R. R. Reducing the dimensionality of data with neural networks. *science*, American Association for the Advancement of Science, v. 313, n. 5786, p. 504–507, 2006. Cited on page 12.
- HINTON, G. E. et al. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580 (Technical report)*, 2012. Cited 3 times on pages 15, 28, and 29.
- HO, T. K. Random decision forests. In: IEEE. *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*. [S.l.], 1995. v. 1, p. 278–282. Cited on page 1.
- HOLLER, M. et al. An electrically trainable artificial neural network (etann) with 10240 floating gate synapses. In: *International Joint Conference on Neural Networks*. [S.l.: s.n.], 1989. v. 2, p. 191–196. Cited on page 10.
- HUANG, J.; LACH, J.; ROBINS, G. A methodology for energy-quality tradeoff using imprecise hardware. In: ACM. *Proceedings of the 49th Annual Design Automation Conference*. [S.l.], 2012. p. 504–509. Cited 3 times on pages 22, 28, and 29.
- IEEE. *IEEE Standard for Floating-Point Arithmetic*. [S.l.], 2008. IEEE Std 754-2008. Cited on page 30.
- IGEL, C.; HÜSKEN, M. Improving the rprop learning algorithm. In: ICSC ACADEMIC PRESS. *Proceedings of the second international ICSC symposium on neural computation (NC 2000)*. [S.l.], 2000. v. 2000, p. 115–121. Cited 2 times on pages 42 and 64.
- IOFFE, S.; SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: BACH, F.; BLEI, D. (Ed.). *Proceedings of the 32nd International Conference on Machine Learning*. Lille, France: PMLR, 2015. (Proceedings of Machine Learning Research, v. 37), p. 448–456. Disponível em: <<http://proceedings.mlr.press/v37/ioffe15.html>>. Cited on page 95.

- JIA, Y. et al. Caffe: Convolutional architecture for fast feature embedding. In: ACM. *Proceedings of the 22nd ACM international conference on Multimedia*. [S.l.], 2014. p. 675–678. Cited on page 2.
- JOULIN, A. et al. Efficient softmax approximation for gpus. In: *International Conference on Machine Learning*. [S.l.: s.n.], 2017. p. 1302–1310. Cited on page 37.
- JOUPPI, N. P. et al. In-datacenter performance analysis of a tensor processing unit. *arXiv:1704.04760 (To appear at the 44th International Symposium on Computer Architecture (ISCA), Toronto, Canada, June 26, 2017.)*, 2017. Cited on page 2.
- KHAN, F. M.; ARNOLD, M. G.; POTTENGER, W. M. Finite precision analysis of support vector machine classification in logarithmic number systems. In: IEEE. *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*. [S.l.], 2004. p. 254–261. Cited 3 times on pages 20, 28, and 29.
- KHAN, F. M.; ARNOLD, M. G.; POTTENGER, W. M. Hardware-based support vector machine classification in logarithmic number systems. In: IEEE. *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*. [S.l.], 2005. p. 5154–5157. Cited 3 times on pages 20, 28, and 29.
- KIM, M.; SMARAGDIS, P. Bitwise neural networks. *arXiv:1601.06071 (Proceedings of the 31 st International Conference on Machine Learning, Lille, France, 2015. JMLR)*, 2016. Cited 3 times on pages 15, 28, and 29.
- KINGMA, D.; BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. Cited 2 times on pages 42 and 94.
- KÖSTER, U. et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In: *Advances in Neural Information Processing Systems*. [S.l.: s.n.], 2017. p. 1742–1752. Cited 4 times on pages 16, 28, 29, and 71.
- KUSHNER, D. The wizardry of id [video games]. *IEEE Spectrum*, IEEE, v. 39, n. 8, p. 42–47, 2002. Cited on page 51.
- LAWLOR, O. et al. Performance degradation in the presence of subnormal floating-point values. In: *Proc. Workshop on Operating System Interference in High Performance Applications*. [S.l.: s.n.], 2005. Cited on page 32.
- LE, Q. V. Building high-level features using large scale unsupervised learning. In: IEEE. *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. [S.l.], 2013. p. 8595–8598. Cited on page 2.
- LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. *Nature*, Nature Research, v. 521, n. 7553, p. 436–444, 2015. Cited on page 2.
- LI, H. et al. Training quantized nets: A deeper understanding. In: *Advances in Neural Information Processing Systems*. [S.l.: s.n.], 2017. p. 5813–5823. Cited 3 times on pages 16, 28, and 29.
- LICHMAN, M. *UCI Machine Learning Repository*. 2013. Disponível em: <<http://archive.ics.uci.edu/ml>>. Cited on page 42.

LIN, D. D.; TALATHI, S. S. Overcoming challenges in fixed point training of deep convolutional networks. *arXiv:1607.02241 (As “Fixed Point Quantization of Deep Convolutional Networks” in Proceedings of the 33 rd International Conference on Machine Learning, New York, NY, USA, 2016. JMLR)*, 2016. Cited 3 times on pages 14, 28, and 29.

LIU, Z. et al. Throughput-optimized fpga accelerator for deep convolutional neural networks. *ACM Trans. Reconfigurable Technol. Syst.*, ACM, New York, NY, USA, v. 10, n. 3, p. 17:1–17:23, jul. 2017. ISSN 1936-7406. Disponível em: <<http://doi.acm.org/10.1145/3079758>>. Cited on page 96.

LOMONT, C. *Fast inverse square root*. 2003. 12 p. Disponível em: <<http://lomont.org/Math/Papers/Papers.php>>. Cited on page 53.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, Springer, v. 5, n. 4, p. 115–133, 1943. Cited on page 6.

MEAD, C. Neuromorphic electronic systems. *Proceedings of the IEEE*, IEEE, v. 78, n. 10, p. 1629–1636, 1990. Cited on page 1.

MEROLLA, P. A. et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, American Association for the Advancement of Science, v. 345, n. 6197, p. 668–673, 2014. Cited on page 2.

MICIKEVICIUS, P. et al. Mixed precision training. In: . [S.l.: s.n.], 2018. ArXiv:1710.03740. Cited 4 times on pages 17, 27, 28, and 29.

MISRA, J.; SAHA, I. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, Elsevier, v. 74, n. 1, p. 239–255, 2010. Cited on page 61.

MITTAL, S. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, ACM, v. 48, n. 4, p. 62, 2016. Cited 2 times on pages 25 and 29.

MUKKAMALA, M. C.; HEIN, M. Variants of rmsprop and adagrad with logarithmic regret bounds. In: *International Conference on Machine Learning*. [S.l.: s.n.], 2017. p. 2545–2553. Cited 2 times on pages 66 and 94.

NAIR, V.; HINTON, G. E. Rectified linear units improve restricted boltzmann machines. In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. [S.l.: s.n.], 2010. p. 807–814. Cited on page 26.

NANDAN, M.; KHARGONEKAR, P. P.; TALATHI, S. S. Fast svm training using approximate extreme points. *The Journal of Machine Learning Research*, JMLR.org, v. 15, n. 1, p. 59–98, 2014. Cited 3 times on pages 21, 28, and 29.

NEUMANN, J. V. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, v. 34, p. 43–98, 1956. Cited on page 1.

NISSSEN, S. *Fast Artificial Neural Network Library*. 2012. <<http://leenissen.dk/fann/wp>>. Accessed: 2017-06-26. Cited on page 38.

- PODOBAS, A.; MATSUOKA, S. Hardware implementation of posits and their application in fpgas. In: IEEE. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. [S.l.], 2018. p. 138–145. Cited on page 79.
- PRECHELT, L. et al. Proben1: A set of neural network benchmark problems and benchmarking rules. Technical Report 21/94, 1994. Cited on page 42.
- QIU, J. et al. Going deeper with embedded fpga platform for convolutional neural network. In: ACM. *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. [S.l.], 2016. p. 26–35. Cited on page 87.
- REED, D. A.; DONGARRA, J. Exascale computing and big data. *Communications of the ACM*, ACM, v. 58, n. 7, p. 56–68, 2015. Cited on page 2.
- RIEDMILLER, M.; BRAUN, H. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In: IEEE. *Neural Networks, 1993., IEEE International Conference on*. [S.l.], 1993. p. 586–591. Cited on page 63.
- ROSENBLATT, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, American Psychological Association, v. 65, n. 6, p. 386, 1958. Cited on page 6.
- RUMELHART, D. E. et al. A general framework for parallel distributed processing. *Parallel distributed processing: Explorations in the microstructure of cognition*, v. 1, p. 45–76, 1986. Cited on page 7.
- SAHIN, S.; BECERIKLI, Y.; YAZICI, S. Neural network implementation in hardware using fpgas. In: SPRINGER. *International Conference on Neural Information Processing*. [S.l.], 2006. p. 1105–1112. Cited on page 87.
- SAKR, C. et al. Understanding the energy and precision requirements for online learning. *Reduced version accepted for The 42nd IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2017) as “Minimum Precision Requirements for the SVM-SGD Learning Algorithm”*, 2016. ArXiv:1607.00669. Cited 3 times on pages 21, 28, and 29.
- SAKR, C. et al. Minimum precision requirements for the svm-sgd learning algorithm. In: IEEE. *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*. [S.l.], 2017. p. 1138–1142. Cited on page 21.
- SAVICH, A. W.; MOUSSA, M.; AREIBI, S. The impact of arithmetic representation on implementing mlp-bp on fpgas: A study. *IEEE transactions on neural networks*, IEEE, v. 18, n. 1, p. 240–252, 2007. Cited 2 times on pages 10 and 97.
- SCHMIDHUBER, J. Deep learning in neural networks: An overview. *Neural Networks*, v. 61, p. 85–117, 2015. Published online 2014; based on TR arXiv:1404.7828 [cs.NE]. Cited on page 9.
- SCHRAUDOLPH, N. N. A fast, compact approximation of the exponential function. *Neural Computation*, MIT Press, v. 11, n. 4, p. 853–862, 1999. Cited 2 times on pages 26 and 51.
- SIETSMA, J.; DOW, R. J. Creating artificial neural networks that generalize. *Neural networks*, Elsevier, v. 4, n. 1, p. 67–79, 1991. Cited on page 68.

- SRIVASTAVA, N. et al. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, v. 15, n. 1, p. 1929–1958, 2014. Cited 4 times on pages 15, 28, 29, and 95.
- SUTSKEVER, I. et al. On the importance of initialization and momentum in deep learning. In: *International conference on machine learning*. [S.l.: s.n.], 2013. p. 1139–1147. Cited on page 35.
- TIELEMAN, T.; HINTON, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, v. 4, n. 2, p. 26–31, 2012. Cited 2 times on pages 42 and 66.
- TORRES, V. A. et al. Combined weightless neural network fpga architecture for deforestation surveillance and visual navigation of uavs. *Engineering Applications of Artificial Intelligence*, Elsevier, v. 87, p. 103227, 1 2020. Disponível em: <<https://doi.org/10.1016/j.engappai.2019.08.021>>. Cited on page 18.
- TORRES, V. F.; TORRES, F. S. A comparative analysis of neural networks implemented with reduced numerical precision and approximations. In: *6th Brazilian Conference on Intelligent Systems*. [S.l.: s.n.], 2017. p. 193–204. Cited on page 26.
- TORRES, V. F.; TORRES, F. S. Resilient training of neural network classifiers with approximate computing techniques for hardware-optimised implementations. *IET Computers & Digital Techniques*, IET, v. 13, n. 6, p. 532–542, 2019. Disponível em: <<https://doi.org/10.1049/iet-cdt.2019.0036>>. Cited on page 92.
- TRELEAVEN, P.; PACHECO, M.; VELLASCO, M. Vlsi architectures for neural networks. *IEEE micro*, IEEE, v. 9, n. 6, p. 8–27, 1989. Cited on page 6.
- VALLE, M. Analog vlsi implementation of artificial neural networks with supervised on-chip learning. *Analog Integrated Circuits and Signal Processing*, Springer, v. 33, n. 3, p. 263–287, 2002. Cited on page 11.
- VENKATARAMANI, S. et al. Scalable-effort classifiers for energy-efficient machine learning. In: *ACM. Proceedings of the 52nd Annual Design Automation Conference*. [S.l.], 2015. p. 67. Cited 3 times on pages 22, 28, and 29.
- WAN, L. et al. Regularization of neural networks using dropconnect. In: *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*. [S.l.: s.n.], 2013. p. 1058–1066. Cited 3 times on pages 15, 28, and 29.
- WANG, X.; LEESER, M. Vfloat: A variable precision fixed-and floating-point library for reconfigurable hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, ACM, v. 3, n. 3, p. 16, 2010. Cited 2 times on pages 47 and 85.
- WILLIAMS, D.; HINTON, G. Learning representations by back-propagating errors. *Nature*, v. 323, n. 6088, p. 533–538, 1986. Cited on page 7.
- WU, G. et al. M2m: From mobile to embedded internet. *IEEE Communications Magazine*, IEEE, v. 49, n. 4, 2011. Cited on page 2.
- WU, S. et al. Training and inference with integers in deep neural networks. *Accepted Paper at the Sixth International Conference on Learning Representations*, 2018. ArXiv:1802.04680. Cited 3 times on pages 18, 28, and 29.

WU, Y. et al. Approximate computing of remotely sensed data: Svm hyperspectral image classification as a case study. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, IEEE, v. 9, n. 12, p. 5806–5818, 2016. Cited 3 times on pages 21, 28, and 29.

XILINX. *Floating-Point Operator v7.1 LogiCORE IP Product Guide*. [S.l.], 2017. Xilinx, Inc. Cited on page 47.

ZHANG, C. et al. Optimizing fpga-based accelerator design for deep convolutional neural networks. In: ACM. *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. [S.l.], 2015. p. 161–170. Cited 2 times on pages 87 and 97.

ZHANG, Q. et al. Approxann: an approximate computing framework for artificial neural network. In: EDA CONSORTIUM. *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. [S.l.], 2015. p. 701–706. Cited 3 times on pages 14, 28, and 29.

ZHANG, Q. et al. Approxit: An approximate computing framework for iterative methods. In: ACM. *Proceedings of the 51st Annual Design Automation Conference*. [S.l.], 2014. p. 1–6. Cited 3 times on pages 24, 28, and 29.

ZHU, J.; SUTTON, P. Fpga implementations of neural networks—a survey of a decade of progress. *Field Programmable Logic and Application*, Springer, p. 1062–1066, 2003. Cited on page 9.

Appendix

APPENDIX A – Additional Validation

The following plots compare the performances using the same methodology presented in the previous Chapters. Hyper-parameters are defined by search using the double precision reference implementation. The approximated solution, identified by **soft-ap** is compared to the reference (**double**) in the figures on the left (a) and to Google’s **bfloat16**, used in all variables, in the figures on the right (b).

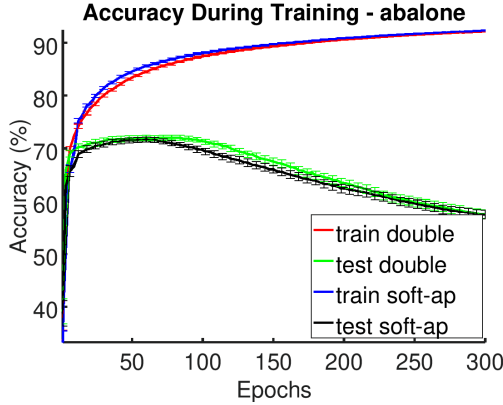
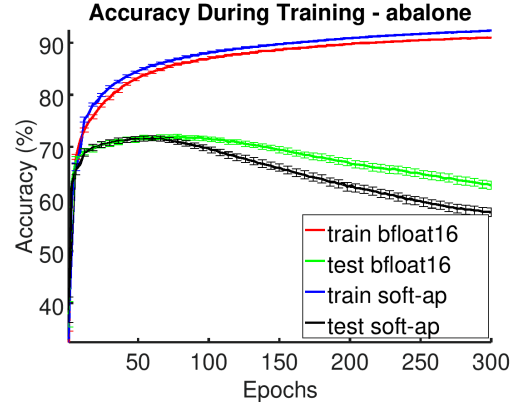
(a) Approx. FP16 \times Double precision(b) Approx. FP16 \times BFloat16

Figure 45 – Performance comparisons with the Abalone Dataset

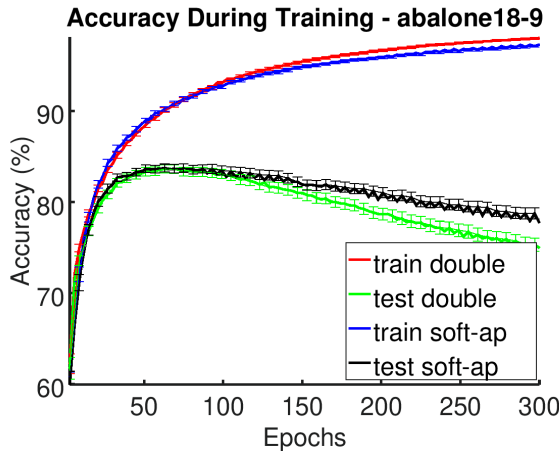
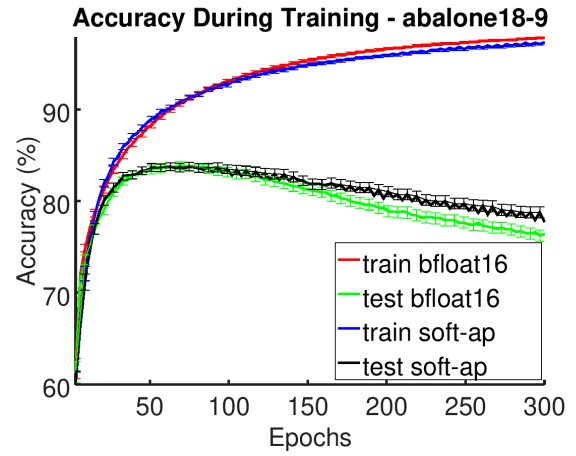
(a) Approx. FP16 \times Double precision(b) Approx. FP16 \times BFloat16

Figure 46 – Performance comparisons with the Abalone (18-9 variation) Dataset

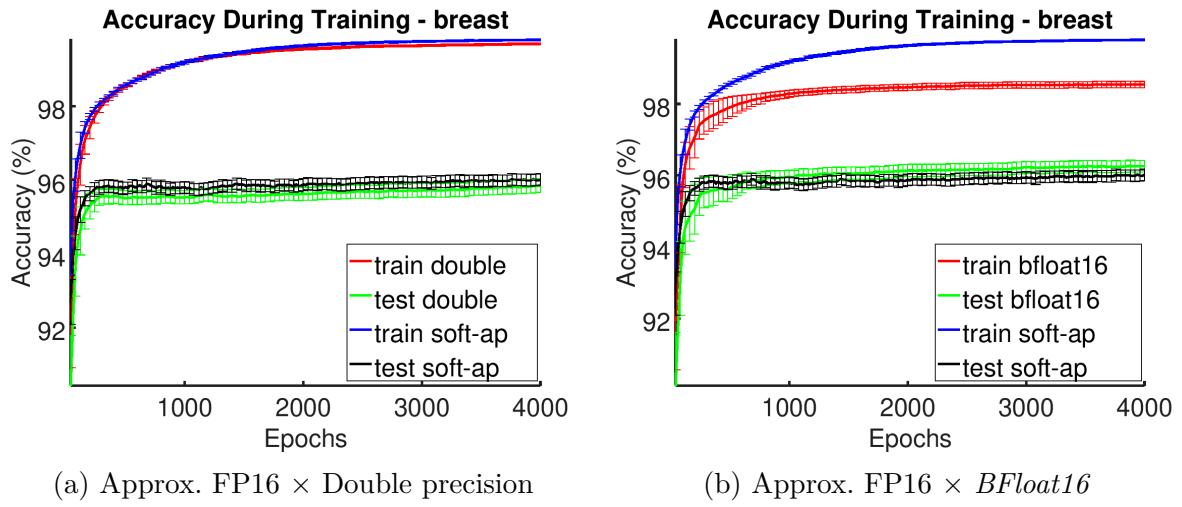


Figure 47 – Performance comparisons with the Breast Cancer Dataset

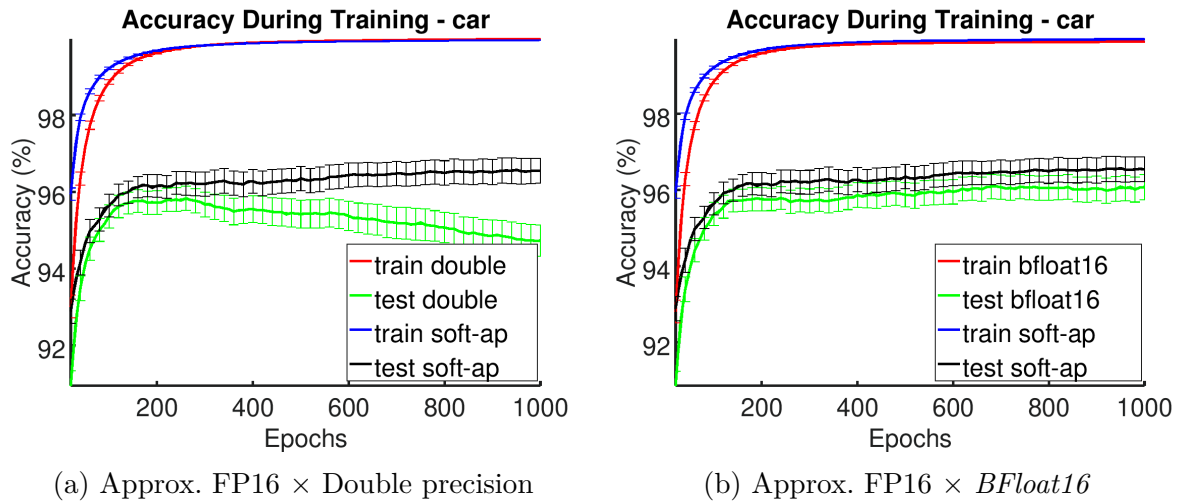


Figure 48 – Performance comparisons with the Car Dataset

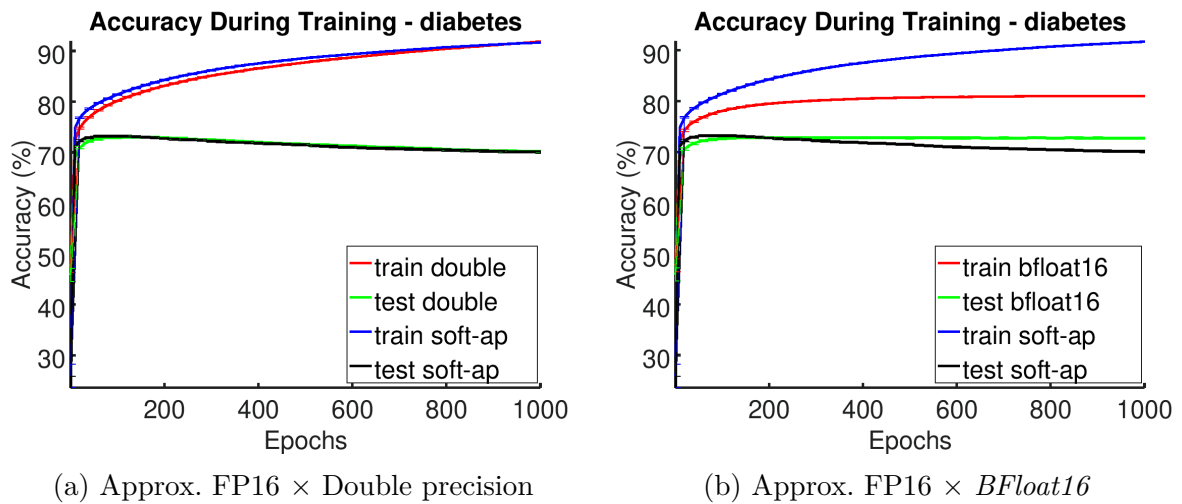


Figure 49 – Performance comparisons with the Diabetes Dataset

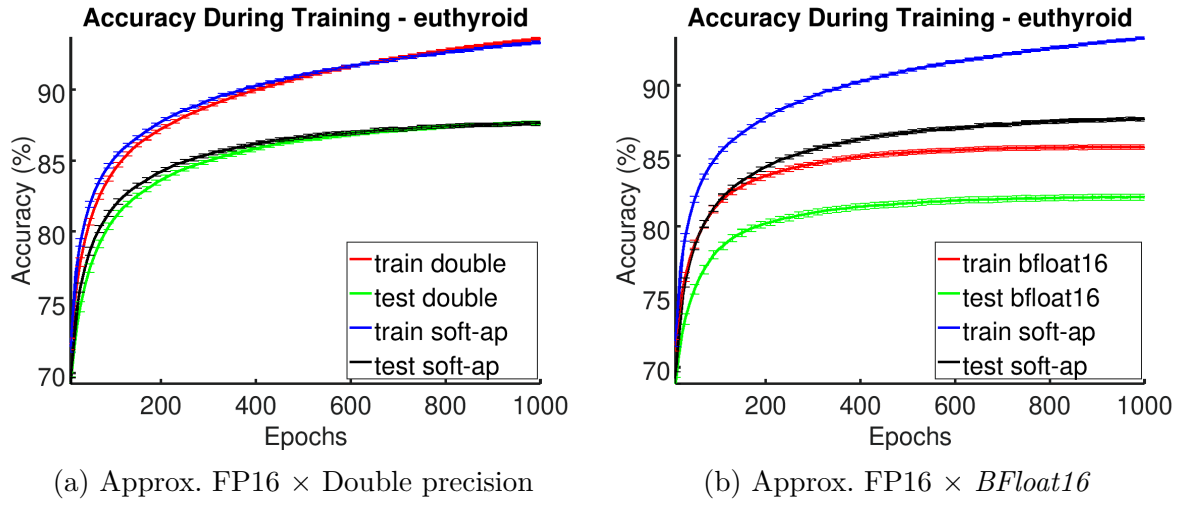


Figure 50 – Performance comparisons with the Euthyroid Dataset

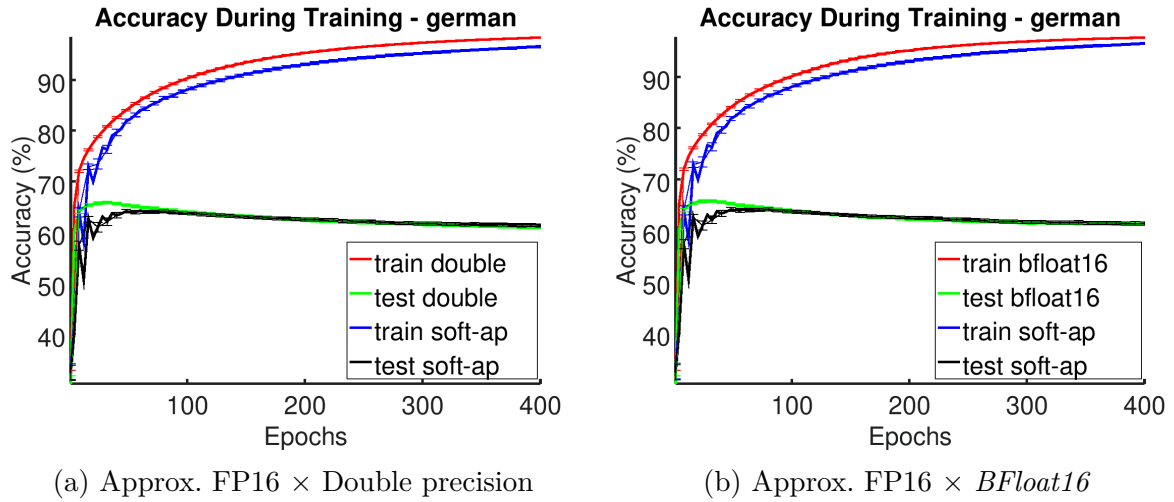


Figure 51 – Performance comparisons with the German Dataset

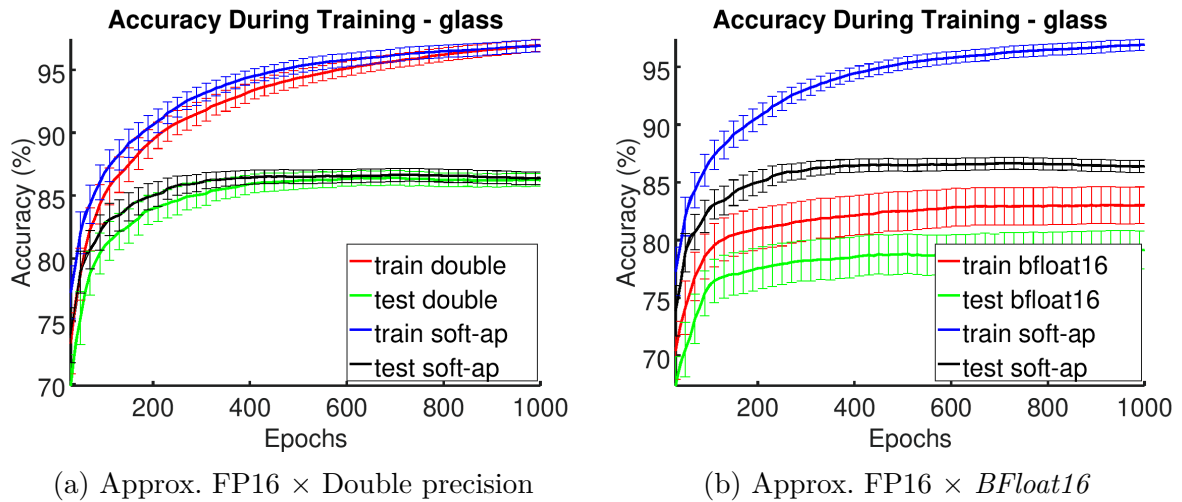


Figure 52 – Performance comparisons with the Glass Dataset

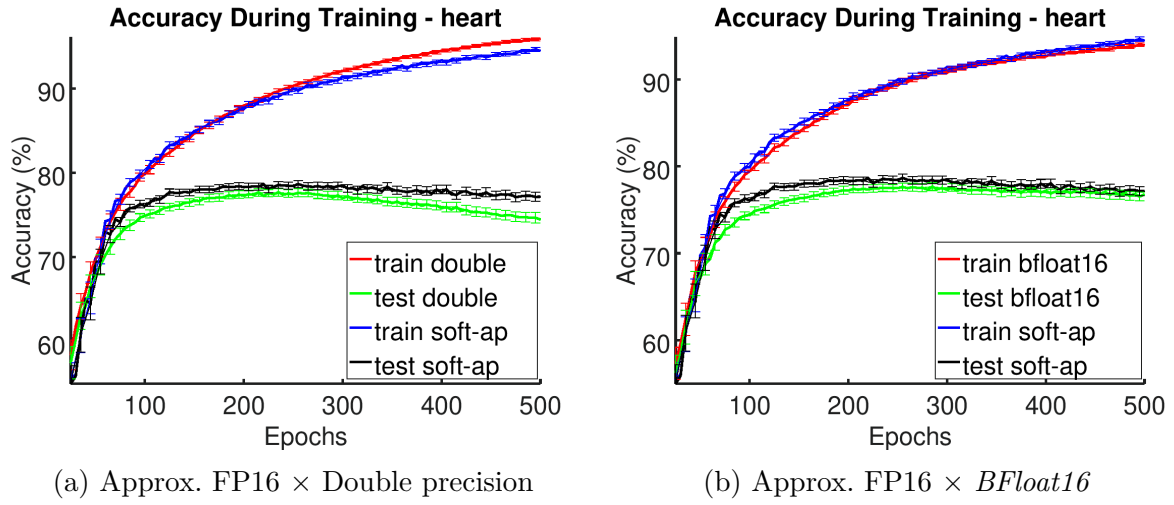


Figure 53 – Performance comparisons with the Heart Dataset

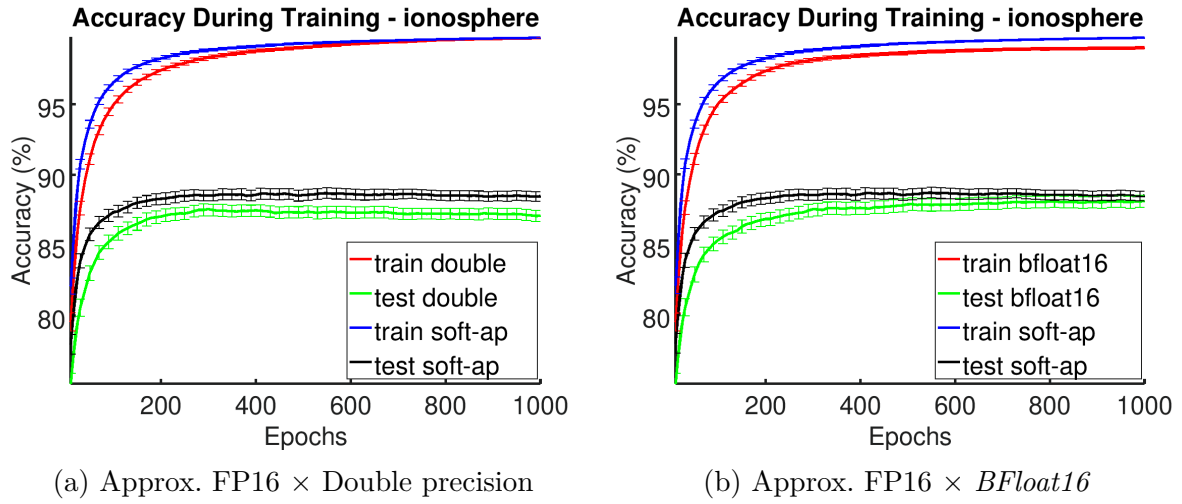


Figure 54 – Performance comparisons with the Ionosphere Dataset

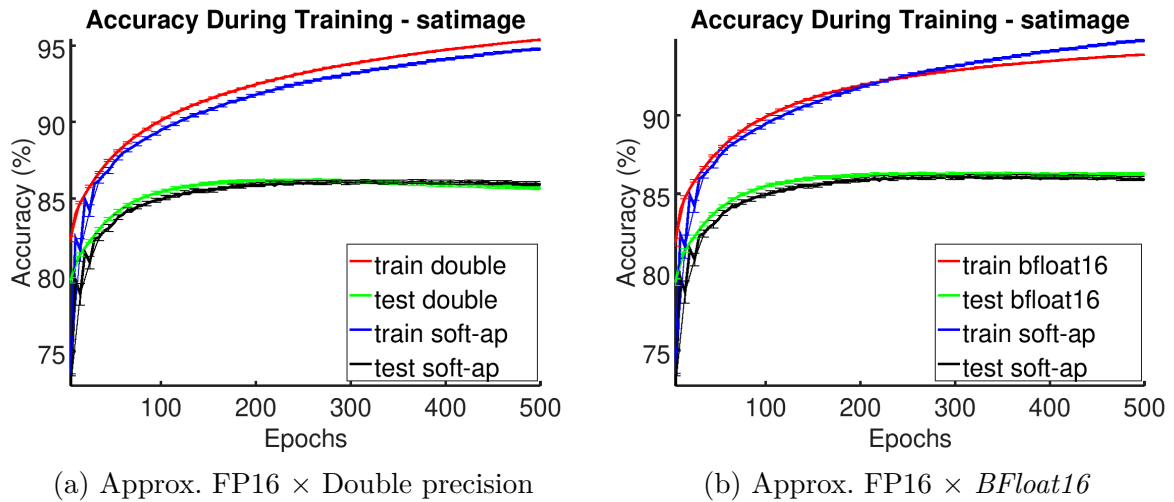


Figure 55 – Performance comparisons with the Satimage Dataset

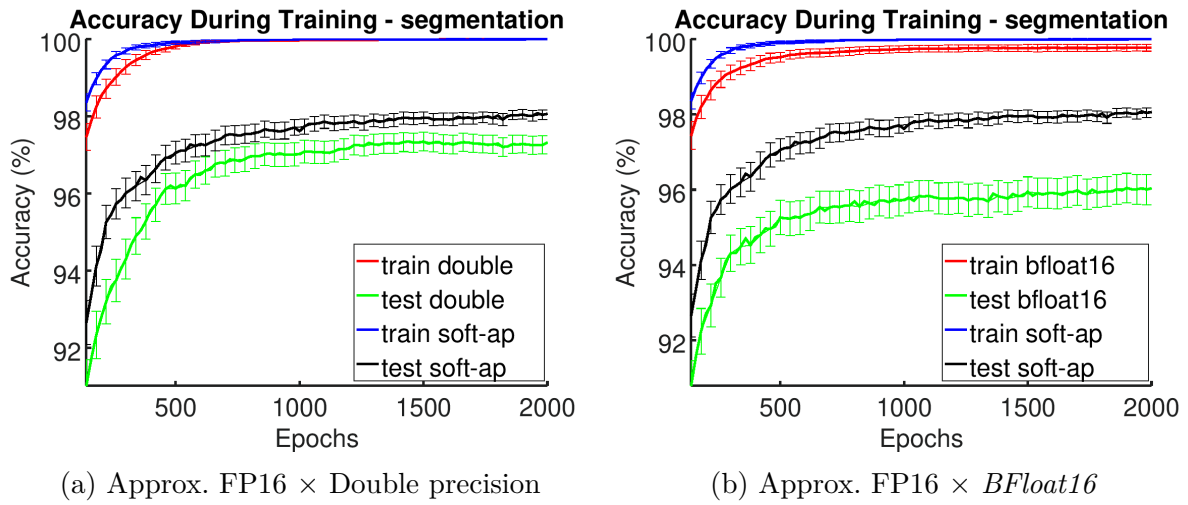


Figure 56 – Performance comparisons with the Segmentation Dataset

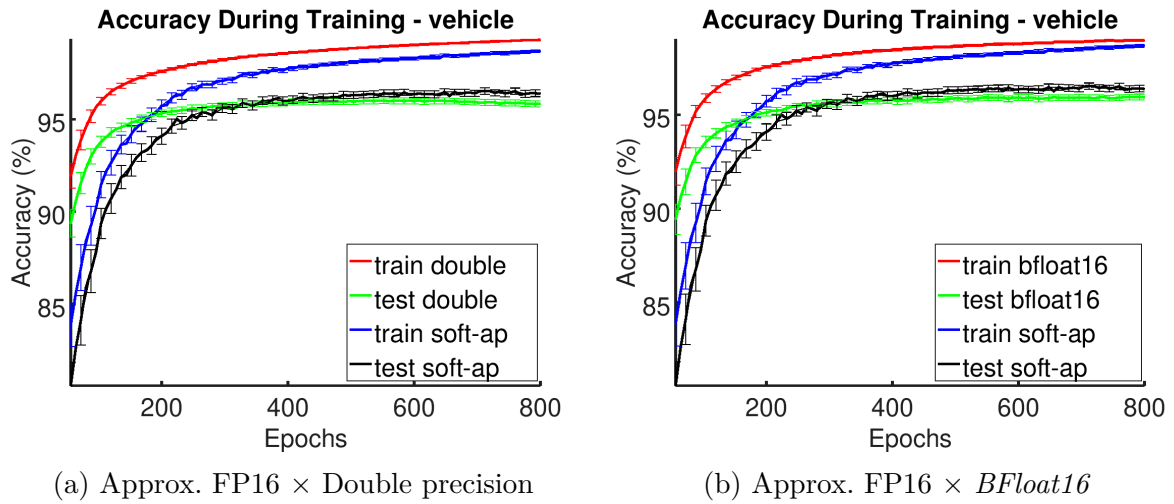


Figure 57 – Performance comparisons with the Vehicle Dataset

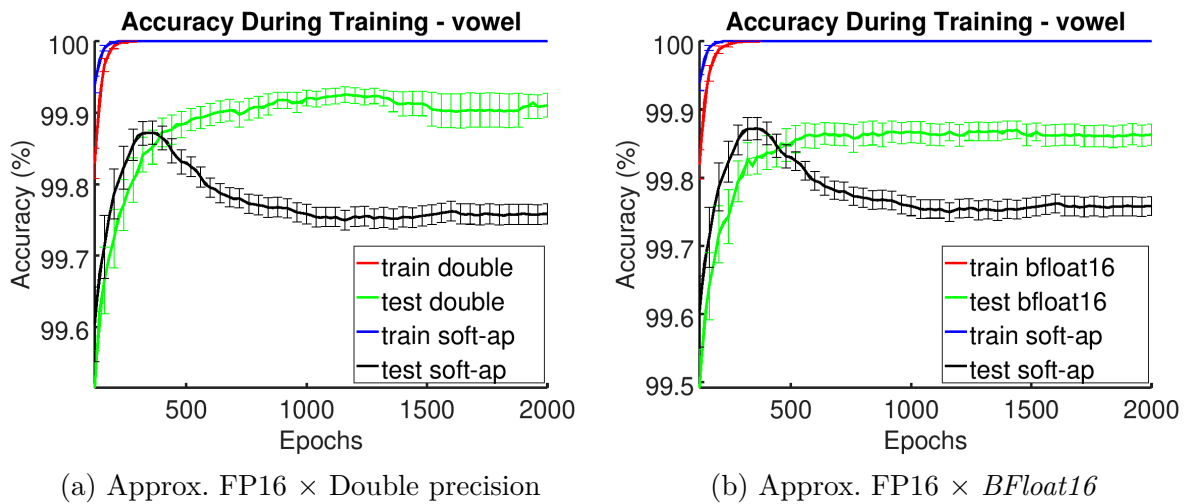


Figure 58 – Performance comparisons with the Vowel Dataset

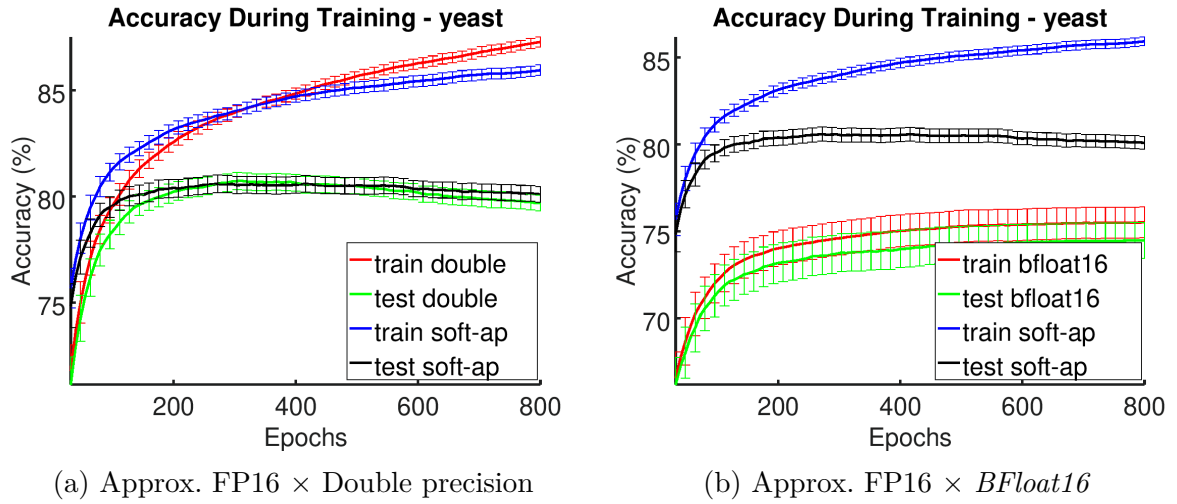


Figure 59 – Performance comparisons with the Yeast Dataset

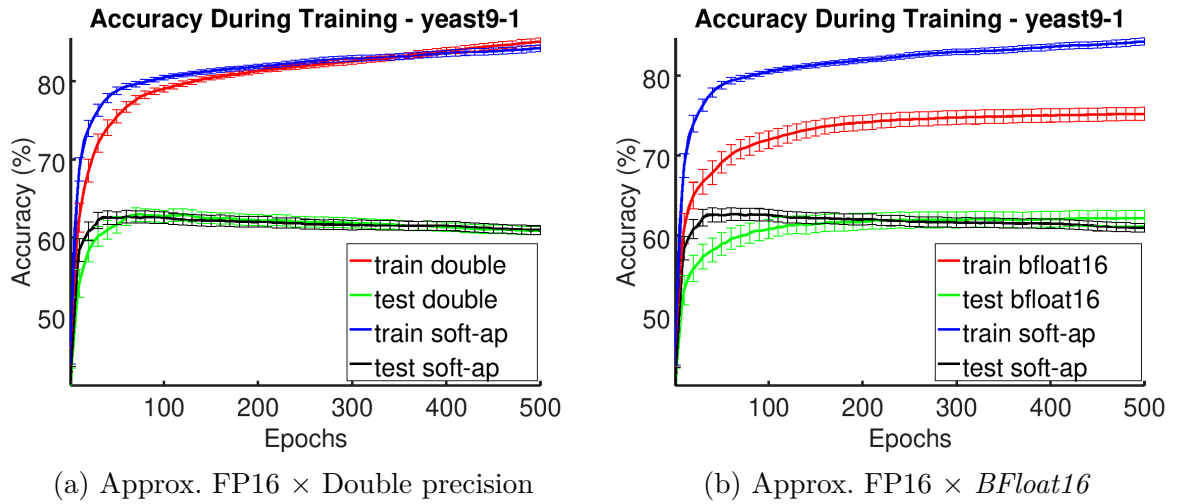
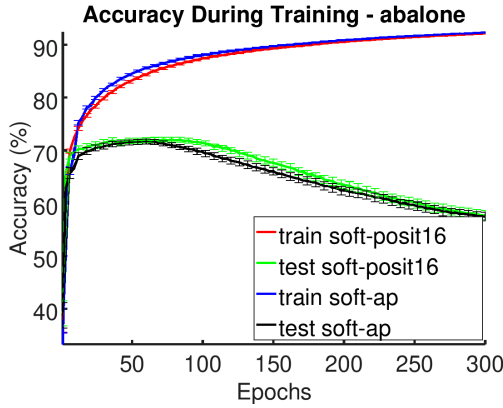


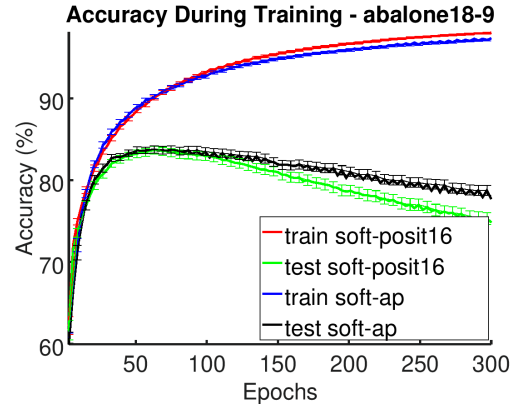
Figure 60 – Performance comparisons with the Yeast (9-1 variation) Dataset

APPENDIX B – Comparison with Posit

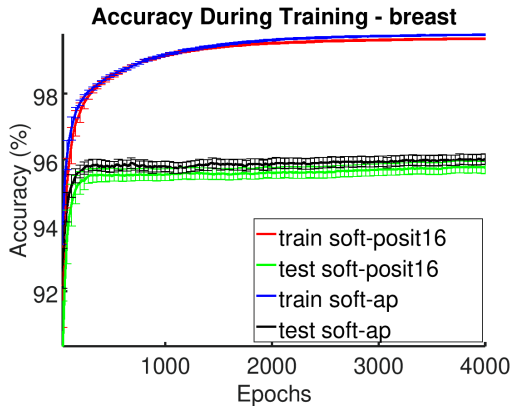
The following plots extend the comparison of the approximated FP16 method to another recently proposed format: Posit, in the variation with fixed size of 16 *bits*. In the following graphs, this format is identified as **soft-posit16**. Equivalently to the comparison with **bfloat16**, the memory usage in the case presented in this appendix requires the same amount of RAM used in the approximated solution, identified by **soft-ap**. The tests use the same methodology presented in the previous Chapters. Hyper-parameters are defined by search using the double precision reference implementation.



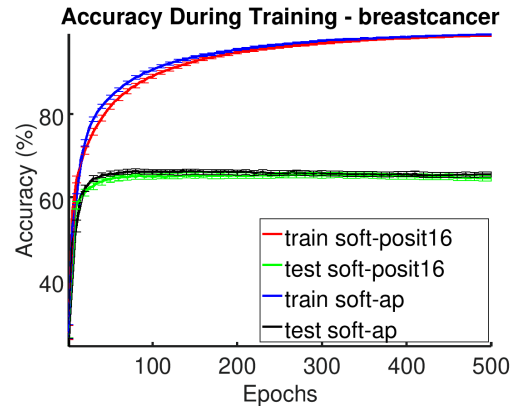
(a) Performance comparisons with the Abalone Dataset



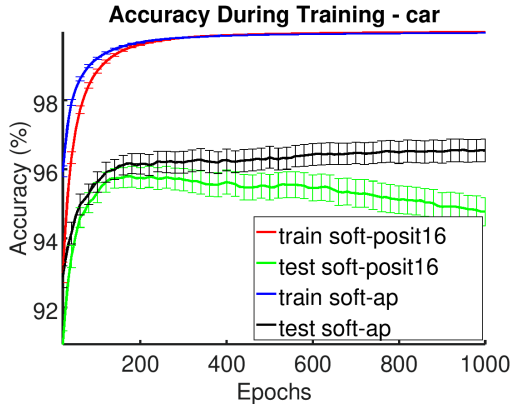
(b) Performance comparisons with the Abalone (18-9 variation) Dataset



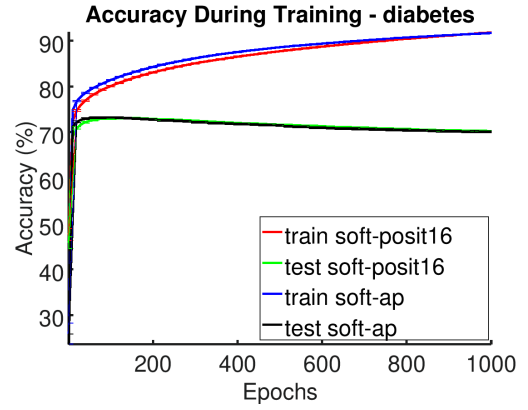
(a) Performance comparisons with the Breast Cancer Dataset (Diagnostic)



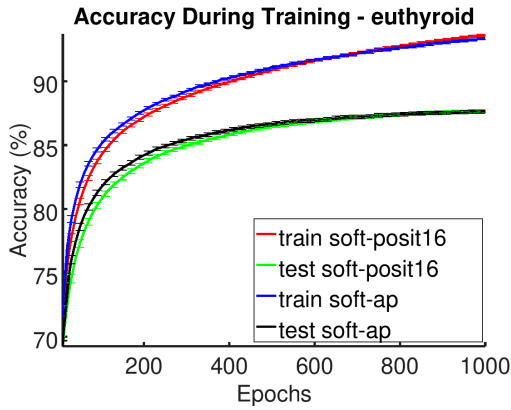
(b) Performance comparisons with the Breast Cancer Dataset (Prognostic)



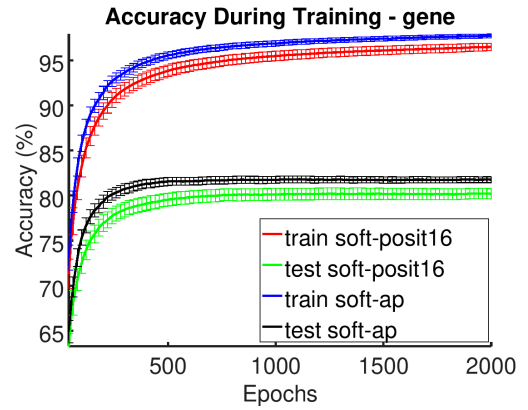
(a) Performance comparisons with the Car Dataset



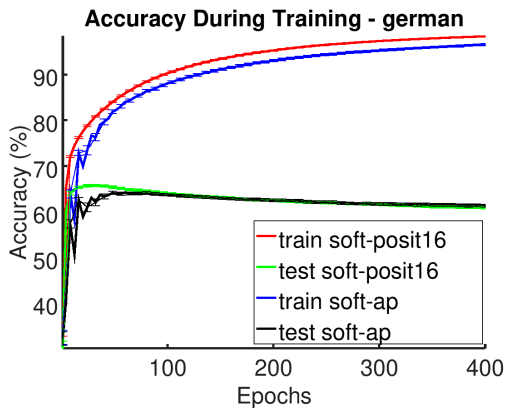
(b) Performance comparisons with the Diabetes Dataset



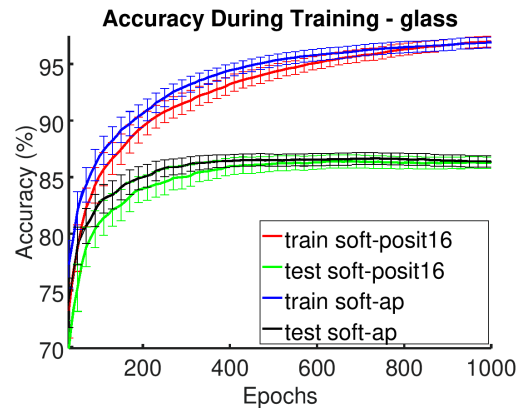
(a) Performance comparisons with the Euthyroid Dataset



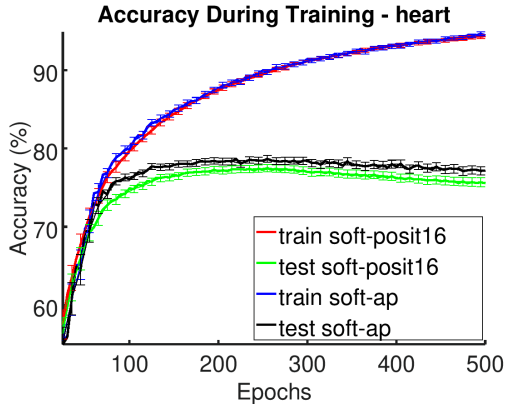
(b) Performance comparisons with the Gene Dataset



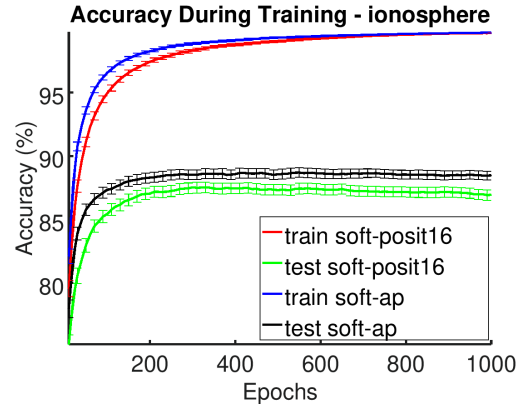
(a) Performance comparisons with the German Dataset



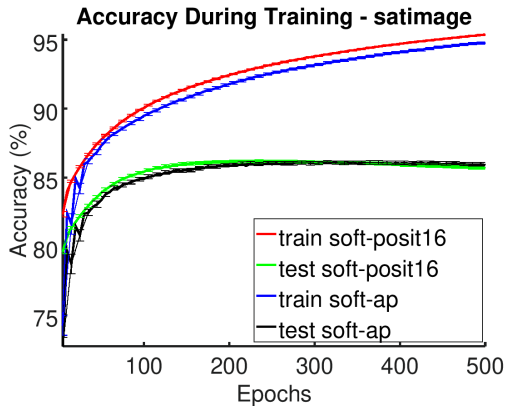
(b) Performance comparisons with the Glass Dataset



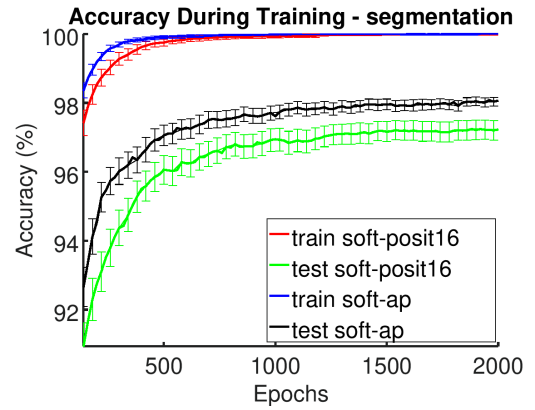
(a) Performance comparisons with the Heart Dataset



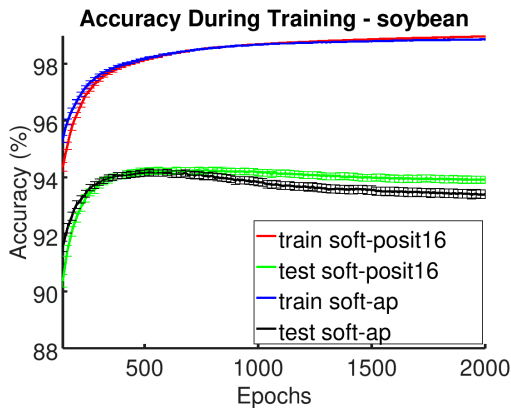
(b) Performance comparisons with the Ionosphere Dataset



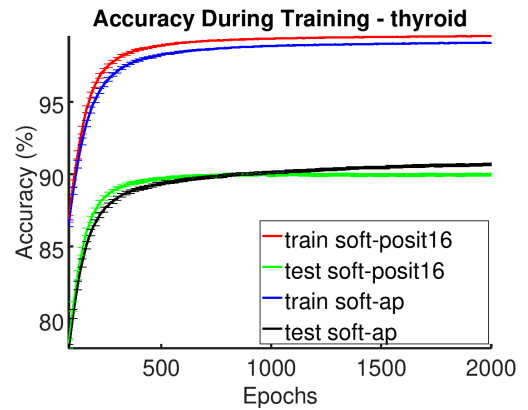
(a) Performance comparisons with the Satimage Dataset



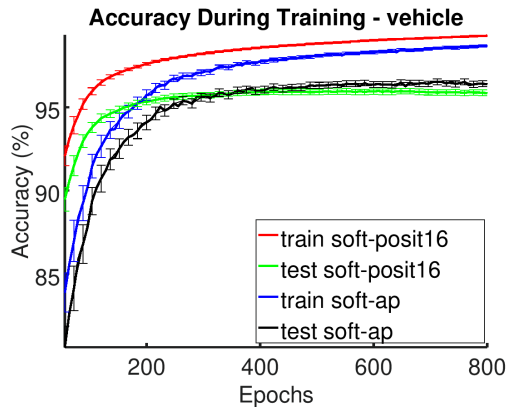
(b) Performance comparisons with the Segmentation Dataset



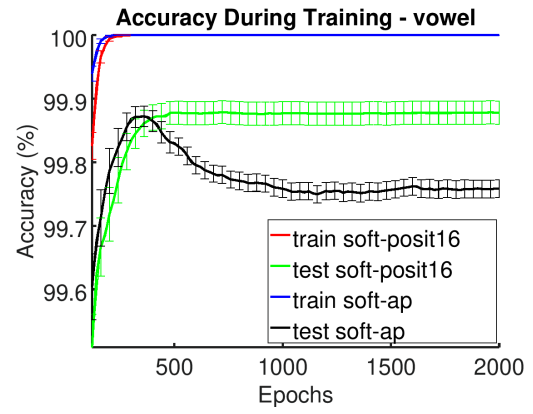
(a) Performance comparisons with the Soybean Dataset



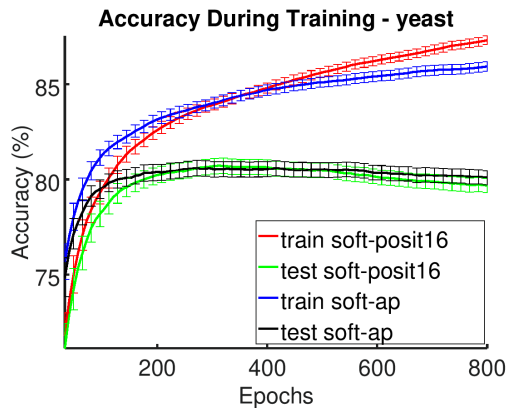
(b) Performance comparisons with the Thyroid Dataset



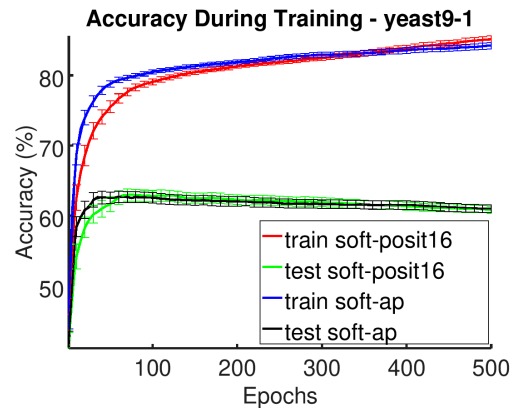
(a) Performance comparisons with the Vehicle Dataset



(b) Performance comparisons with the Vowel Dataset



(a) Performance comparisons with the Yeast Dataset



(b) Performance comparisons with the Yeast (9-1 variation) Dataset