Universitat de les Illes Balears

DOCTORAL THESIS
2021

COMPACT MACHINE LEARNING SYSTEMS WITH RECONFIGURABLE COMPUTING

Alejandro Morán Costoya

DOCTORAL THESIS
2021

Doctoral Programme of Electronic Engineering

COMPACT MACHINE LEARNING SYSTEMS WITH RECONFIGURABLE COMPUTING

Alejandro Morán Costoya

Thesis Supervisor: José Luis Rosselló Sanz
Thesis Supervisor: Vicente José Canals Guinand
Thesis Tutor: José Luis Rosselló Sanz

Doctor by the Universitat de les Illes Balears

Universitat de les Illes Balears

WE, THE UNDERSIGNED, DECLARE:

That the Thesis titled

Compact Machine Learning Systems with Reconfigurable Computing

presented by Sr. Alejandro Morán Costoya to obtain a doctoral degree, has been completed under the supervision of Dr. Jose Luis Rosselló Sanz and Dr. Vicente José Canals Guinand.

For all intents and purposes, we hereby sign this document.

Palma de Mallorca,

Sr. Alejandro Morán Costoya

Dr. José Luis Rosselló Sanz

Dr. Vicente José Canals Guinand

A mi familia

Agradecimientos

Para comenzar, me gustaría dar las gracias a mis directores de tesis. En primer lugar, al Prof. Josep Lluís Rosselló por haberme guiado desde el primer día y preocuparse de que tanto este como otros trabajos llegaran a buen puerto. En segundo lugar, al Dr. Vicente Canals por compartir su *know how* para resolver mis incontables dudas y aguantarme el día a día, haciendo posible que este trabajo finalmente haya visto la luz. Por otro lado, quiero agradecer al Prof. Miquel Roca sus consejos y a mis compañeros Christian y Erik nuestras fructíferas charlas. Además, deseo dar las gracias a mis padres, mi hermana y a Tania por haberme apoyado en todo momento.

Por último, reconocer mi agradecimiento por la financiación por parte del Ministerio de Ciencia e Innovación al proyecto TEC2017-84877-R, en el cual se enmarca el objeto de esta tesis.

Resumen

De manera similar a nosotros los humanos, las máquinas pueden *aprender* a partir de los datos e incluso superarnos en ciertas tareas específicas, lo cual es conocido como rendimiento sobrehumano. El proceso por el que una máquina construye conocimiento a partir de los datos se conoce como Aprendizaje Automático, que no es nada nuevo, dicho término fue acuñado por Arthur Samuel en 1959. Este enfoque difiere de la Inteligencia Artificial tradicional basada en el conocimiento (los llamados sistemas expertos), que fueron la opción predominante durante la década de 1980, ya que superaron a los enfoques de aprendizaje estadístico de entonces. Sin ánimo de quitar méritos a los pioneros del campo del Aprendizaje Automático gracias a quienes hoy somos capaces de construir máquinas *inteligentes* de última generación de forma relativamente sencilla, cabe destacar que una de las razones por las que el campo del Aprendizaje Automático no se popularizó y se convirtió en la opción preferida hasta los años 90, superando a los sistemas expertos, fue la falta de potencia computacional disponible para entrenar modelos complejos que trataran con grandes cantidades de datos. De hecho, es razonable pensar que el aumento de la potencia computacional ha sido una parte clave de la transición a lo que hoy conocemos como Aprendizaje Profundo, un subcampo del Aprendizaje Automático que se ocupa de las Redes Neuronales Artificiales Profundas.

Los modelos de Aprendizaje Profundo han demostrado ser capaces de superar a los humanos en ciertas tareas. Quizá el ejemplo más notable de los últimos años sea el acontecimiento en el que la Inteligencia Artificial AlphaGo, desarrollada por Google DeepMind, derrotó tres veces seguidas al jugador de Go número uno del mundo, Ke Jie, en una partida de Go de tres juegos. Aunque el evento marcó un antes y un después en la historia de la Inteligencia Artificial, desde el punto de vista del consumo energético de cada participante, no fue una batalla justa. La potencia del cerebro de Jie es de unos 20 W mientras que la disipada por AlphaGo es de unos 170 kW, es decir, el consumo de energía de AlphaGo sería unas 8500 veces mayor si ambos jugadores emplearan el mismo tiempo de juego. ¿Sería posible repetir un avance similar con una potencia disipada del orden de 20 W? Es probable que no veamos este cambio a corto plazo.

En este contexto, el objetivo de la tesis no es tan ambicioso en cuanto a mejorar la eficiencia energética o intentar desarrollar un sistema de Aprendizaje Profundo distribuido tan grande. En este trabajo, los modelos propuestos son mucho más pequeños y el objetivo es contribuir a la exploración de arquitecturas de hardware simplificadas altamente/totalmente paralelas hechas a medida y no basadas en una arquitectura de von Neumann, lo cual conlleva potenciales beneficios de eficiencia energética. En particular, se han propuesto varios diseños de FPGA que implementan el proceso de inferencia de varios modelos de Aprendizaje Automático y se han probado en un conjunto de bases de datos de referencia. Las implementaciones FPGA incluyen dos modelos de computación de reservorio basados en aritmética de punto fijo de baja precisión y una Red Neuronal de Función de Base Radial basada en Computación Estocástica. Además, se ha simulado y evaluado una Red Neuronal Convolucional basada en dos variantes diferentes de computación estocástica para diferentes precisiones de bits, ambas entrenadas utilizando un enfoque de cuantización consciente del entrenamiento.

Resum

De manera similar nosaltres els humans, les màquines poden *aprendre* a partir de les dades i fins i tot superar-nos en certes tasques específiques, el qual és conegut com a rendiment sobrehumà. El procés pel qual una màquina construeix coneixement a partir de les dades es coneix com Aprenentatge Automàtic, que no és res de nou, aquest terme ja va ser encunyat per Arthur Samuel en 1959. Aquest enfocament difereix de la Intel·ligència Artificial tradicional basada en el coneixement (els anomenats sistemes experts), que van ser l'opció predominant durant la dècada de 1980, ja que van superar als enfocaments d'aprenentatge estadístic de llavors. Sense ànim de treure mèrits als pioners de el camp de l'Aprenentatge Automàtic gràcies als quals avui som capaços de construir màquines *intel·ligents* d'última generació de forma relativament senzilla, cal destacar que na de les raons per les que el camp de l'Aprenentatge Automàtic no es va popularitzar i es va convertir en l'opció preferida fins als anys 90, superant als sistemes experts, va ser la falta de potència computacional disponible per entrenar models complexos que tractessin amb grans quantitats de dades. De fet, és raonable pensar que l'augment de la potència computacional ha estat una part clau de la transició al que avui coneixem com Aprenentatge Profund, un subcampo de l'Aprenentatge Automàtic que s'ocupa de les Xarxes Neuronals Artificials Profundes.

Els models d'Aprenentatge Profund han demostrat ser capaços de superar els humans en certes tasques. Potser l'exemple més notable dels últims anys sigui l'esdeveniment en el qual la Intel·ligència Artificial AlphaGo, desenvolupada per Google DeepMind, va derrotar tres vegades seguides a el jugador de Go número u de l'món, Ke Jie, en una partida de Go de tres jocs. Encara que l'esdeveniment va marcar un abans i un després en la història de la Intel·ligència Artificial, des del punt de vista de l'consum energètic de cada participant, no va ser una batalla justa. La potència del cervell de Jie és d'uns 20 W mentre que la dissipada per AlphaGo és d'uns 170 kW, és a dir, el consum d'energia de AlphaGo seria unes 8500 vegades més gran si els dos jugadors empressin el mateix temps de joc. Seria possible repetir un avanç similar amb una potència dissipada de l'ordre de 20 W? És probable que no vegem aquest canvi a curt termini.

En aquest context, l'objectiu de la tesi no és tan ambiciós pel que fa a millorar l'eficiència energètica o intentar desenvolupar un sistema d'Aprenentatge Profund distribuït tan gran. En aquest treball, els models proposats són molt més petits i l'objectiu és contribuir a l'exploració d'arquitectures de maquinari simplificades altament/totalment paral·leles fetes a mida i no basades en una arquitectura de von Neumann, la qual cosa comporta potencials beneficis d'eficiència energètica. En particular, s'han proposat diversos dissenys de FPGA que implementen el procés d'inferència de diversos models d'Aprenentatge Automàtic i s'han provat en un conjunt de bases de dades de referència. Les implementacions FPGA inclouen dos models de computació de reservori basats en aritmètica de punt fix de baixa precisió i una Xarxa Neuronal de Funció de Base Radial basada en Computació Estocàstica. A més, s'ha simulat i avaluat una Xarxa Neuronal convolucional basada en dues variants diferents de computació estocàstica per diferents precisions de bits, ambdues entrenades utilitzant un enfocament de quantització conscient de l'entrenament.

Abstract

Similar to us humans, machines can *learn* from data and even outperform us in certain specific tasks (a.k.a. superhuman performance). The process by which a machine builds knowledge from data is known as Machine Learning, which is nothing new, this term was already coined by Arthur Samuel in 1959. This approach differs from traditional knowledge-based Artificial Intelligence (the so called expert systems), which were the predominant choice during the 1980s since they outperformed statistical learning approaches back then. Without trying to take away merits to the pioneers of the Machine Learning field thanks to whom today we are able to build state-of-the-art *intelligent* machines in a relatively easy way, one of the reasons why the Machine Learning field did not become popular and the preferred choice until the 1990s, outperforming expert systems, was the lack of available computational power to train complex models dealing with large amounts of data. In fact, it is reasonable to think that the increase in computational power has been a key part of the transition to what we know today as Deep Learning, a subfield of Machine Learning dealing with Deep Artificial Neural Networks.

Deep Learning models have been shown to be capable of surpassing humans in certain tasks. Perhaps the most notable example in the recent years is the event in which the AlphaGo artificial intelligence developed by Google DeepMind defeated the world's number one Go player Ke Jie three times in a row in a three-game Go match. Although the event marked a before and after in the history of Artificial Intelligence, from the point of view of the energy consumption of each participant, it was not a fair battle. Jie's brain works with 20 W and AlphaGo's thermal power dissipation is about 170 kW, i.e. AlphaGo's energy consumption would be about 8500 times higher if both players spent the same playing time. Would it be possible to repeat a similar breakthrough with a power consumption in the order of 20 W? We are probably not going to see this change anytime soon.

In this context, the objective of the thesis is not nearly as ambitious in terms of improving energy efficiency or attempting to develop such a big distributed Deep Learning system. In this work, the proposed models are much smaller and the aim is to contribute to the exploration of simplified highly/fully parallel (custom) non von Neumann hardware architectures with potential energy efficiency benefits. In particular, several FPGA designs implementing the inference process of several Machine Learning models have been proposed and tested on a set of benchmark datasets. The FPGA implementations include two Reservoir Computing models based on low precision fixed-point arithmetic and a Radial Basis Function Neural Network based on Stochastic Computing. Additionally, a Convolutional Neural Network based on two different Stochastic Computing variants has been simulated and evaluated for different bit precision, both trained using a custom Training Aware Quantization approach.

Contents

Notation

This section is devoted to describe a list of math symbols and conventions so as to ensure coherency throughout the document. It is mostly inspired by the notation used in [1], [2].

Scalars, Matrices and Tensors

a, $\boldsymbol{a}$, $\boldsymbol{A}$, $\boldsymbol{\mathsf{A}}$	Scalar, vector, matrix and tensor, respectively
$\mathbf{1}$	Column vector of ones
$\boldsymbol{J}$	Matrix of ones
$\boldsymbol{\mathsf{J}}$	Tensor of ones
$\boldsymbol{I}$	Itentity matrix
a_i	ith element of a row or column vector $\boldsymbol{a}$
$A_{i,j}$	Element (i, j) (i.e. ith row and jth column) of a matrix $\boldsymbol{A}$
$\mathsf{A}_{i,j,k}$	Element (i, j, k) of a 3-D tensor $\boldsymbol{\mathsf{A}}$
$A_{i,:}$	ith row of $\boldsymbol{A}$
$A_{:,i}$	ith column of $\boldsymbol{A}$
$\mathsf{A}_{:,i,j}$	1-D slice of a 3-D tensor $\boldsymbol{\mathsf{A}}$
$\mathsf{A}_{:,:,j}$	2-D slice of a 3-D tensor $\boldsymbol{\mathsf{A}}$

Sets, Intervals and Graphs

$\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{C}$	Set of natural, integer, real and complex numbers, respectively
$\{a, b\}$	Set containing a and b
$(a, b]$	Interval of real numbers that range from a (not included) to b (included)
$\mathbb{Z} \cap [a, b]$	Interval of integer numbers that range from a to b
$\mathcal{G}$	Graph

Operations and Operators

$\boldsymbol{A}^\intercal$	Transpose operation applied to $\boldsymbol{A}$
$\boldsymbol{A}^+$	Moore-Penrose pseudoinverse of $\boldsymbol{A}$
$\boldsymbol{AB}$	Matrix multiplication of $\boldsymbol{A}$ and $\boldsymbol{B}$
$\boldsymbol{\mathsf{A}} \star \boldsymbol{\mathsf{B}}$	Discrete cross-correlation of $\boldsymbol{\mathsf{A}}$ with a kernel $\boldsymbol{\mathsf{B}}$
$\boldsymbol{\mathsf{A}} \odot \boldsymbol{\mathsf{B}}$	Hadamard product[1] of $\boldsymbol{\mathsf{A}}$ and $\boldsymbol{\mathsf{B}}$
$\boldsymbol{\mathsf{A}} \oslash \boldsymbol{\mathsf{B}}$	Hadamard division[1] of $\boldsymbol{\mathsf{A}}$ and $\boldsymbol{\mathsf{B}}$
$\lceil\cdot\rceil$	Scalar or elementwise[1] ceiling operation
$\lfloor\cdot\rfloor$	Scalar or elementwise[1] floor operation
$\lVert\boldsymbol{a}\rVert$	L^2 norm of $\boldsymbol{a}$
$\lVert\boldsymbol{A}\rVert$	Frobenius norm of $\boldsymbol{A}$
$\det(\boldsymbol{A})$	Determinant of $\boldsymbol{A}$

Probability Theory

$P(A)$	Probability of event A
$P(A \mid B)$	Probability of event A given B
$\mathrm{E}(X)$	Expected value of the random variable X
$\mathrm{var}(X)$	Variance of the random variable X
$\mathrm{std}(X)$	Standard deviation of the random variable X
$\mathrm{cov}(X,Y)$	Covariance of the random variables X and Y
$\mathrm{corr}(X,Y)$	Correlation of the random variables X and Y
$X \sim F$	Random variable X follows a distribution F
$A \perp B$	A and B are independent events.

Bitstreams

$\tilde{x}(t)$	Bitstream with activation probability p_x.
$\langle\tilde{x}(t)\rangle$	Arithmetic mean of $\tilde{x}(t)$.
$\tilde{x} \perp \tilde{y}$	Bitstreams $\tilde{x}(t)$ and $\tilde{y}(t)$ are uncorrelated.
$\tilde{x} \parallel \tilde{y}$	Bitstreams $\tilde{x}(t)$ and $\tilde{y}(t)$ are correlated.

[1]Elementwise (Hadamard) operations defined for tensors might be applied to the particular case of matrices and vectors.

Chapter 1

Introduction

The aim of this chapter is to present the motivation (Section 1.1), establishing the objectives (Section 1.2) and limitations (Section 1.3) of this thesis. Finally, the document organization is described in Section 1.4.

1.1 Motivation

Nowadays, the semiconductor industry faces what appears to be (or will be in the near future) the end of Moore's Law [3]. G.E. Moore initially predicted the number of transistors in a CPU would double every year based on observations from 1962 to 1966. However, he rectified the prediction in 1975 and based on the numbers from 1970 to 1975 said the number of transistors would double every two years, which is what is known as Moore's Law today. Although the prediction initially referred to transistor count in single chip high performance Central Processing Units (CPUs), it is also usually connected to other power laws in technology, e.g. industry revenues, transistor price or total chip power dissipation [4], [5]. Also, this concept can be extended to other computing devices such as Graphical Processing Units (GPUs) or Field Programmable Gate Arrays (FPGAs). It is not just an empirical rule, Moore's Law became a widely accepted goal for the semiconductor industry and it has even been argued that it is a *self-fulfilling prophecy* [6].

As regards transistor count, historical data is plotted on Fig. 1.1 for high performance CPU, GPU and FPGA devices and widely known chip designers to illustrate Moore's Law.

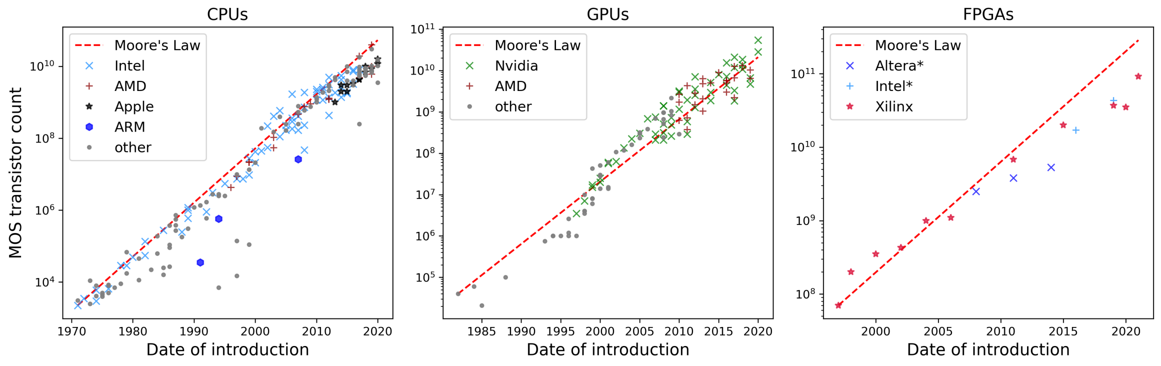


Figure 1.1: Moore's Law until 2020 for high performance CPUs, GPUs and FPGAs. *Intel® acquired Altera® at the end of 2015. Data source: https://en.wikipedia.org/wiki/Transistor_count.

Until now, the growing number of transistors in a single chip has been possible year after year because the transistor size[1] has been reduced thanks to the scientific and technological advances put in practice by the main manufacturers' Integrated Chip (IC) foundries.

Going smaller allows to fit more transistors in the same die area and increasing performance[2]. More specifically, doubling the transistor count every two years implies doubling CPU performance every 18 months, or at least that is what was supposed to happen. During the years it was so, it was a big deal because increasing performance was possible without impacting the overall area and therefore energy consumption remained nearly constant. This is known as Dennard scaling [7] and it ended around 2005–2007 mainly due to the MOSFET's leakage current problem [8]. Then, CPU designers tried to circumvent or minimize the problem by increasing the number of cores, which was an improvement, but far less efficient than if Dennard scaling had continued. Later, in 2012, Intel introduced the FinFET transistor [9], which is faster and more power efficient, and is building block of today's modern processors. Nevertheless, the rate of improvement in performance has been drastically reduced during the last ten years [10].

Notice during the years in which Dennard scaling was still alive, an IC design could be manufactured for the first time and manufactured again years later with the latest transistor technology, so that the last one would require less area and would be more energy efficient than the former. This was certainly a great deal for battery powered devices which did not require improving performance. However, it does not seem this growth rate will resume in the near future. There is not even a certainty that Moore's Law will hold in the near future.

In addition to the current difficulties faced by manufacturing processes, there is another challenge: the increasing demand of computational capacity and energy efficiency requirements for certain tasks, including algorithms related to Artificial Intelligence (AI) in general and Machine Learning (ML) in particular. A clear example to motivate the benefits these improvements can bring is the development of efficient Internet of Things (IoT) edge devices and simpler edge nodes with computing capabilities, which could meet lower power specifications. This is not trivial since these algorithms (typically inference for pattern recognition) need to process large amounts of data in a short time, requiring significant hardware parallelism to accelerate the process [11], which might require large chip area compared to e.g. a baseline 32-bit RISC-V architecture [12].

Recently, the state-of-the-art solution was to send data captured by sensor nodes to the cloud and wait for the server's response [13]. This server-dependent approach requires a significant amount of data transmission, which in turn results in a network congestion. Even though data transmissions with sufficient throughput would solve the problem, increasing throughput increases energy consumption. Moreover, there exist privacy issues related to sending sensor data directly to the cloud (e.g. sending images with people faces). The solution to this issue is to enable data inference and analysis capabilities at the edge, which are close to data sources. This approach is known as

[1]The transistor size is typically specified by its gate length. However, it is not always a representative number for the actual transistor dimensions.

[2]During the first 30 years of Moore's Law the increase in performance was not only due to the number of transistors, but also due to the fact that transistors with smaller gate lengths could switch faster.

Edge Computing (EC). Compared to the straightforward data transmission, energy consumption is potentially reduced since inference is done by low power edge devices, there is not a latency problem if the device is capable of managing real-time inference, and privacy is no longer an issue since raw data is not transmitted to the cloud, only metadata (e.g. date, location and event duration) and inference results (e.g. detection of chemical compounds in the air or gunshots or voice-based assistance) are sent.

This thesis is focused on prototyping FPGA pattern recognition inference solutions with potential applications in EC, either at edge nodes or as part of IoT devices.

1.2 Objectives

The main objective of this thesis is to explore unconventional highly parallel digital architectures focused on pattern recognition applications with potential improvements in the energy efficiency while minimizing potential accuracy degradation due to low precision computations. In this context, the research has the three following specific objectives:

- Design and development of highly/fully parallel Reservoir Computing (RC) and Artificial Neural Network (ANN) architectures presenting benefits in terms of energy efficiency compared to other research works.
- Adaptation and improvement of several standard ML algorithms to maximize accuracy in fixed-point and SC inference hardware.
- FPGA implementation and evaluation of the designs explored in this work.

1.3 Limitations

The main limitations of the hardware implementations is the lack of Very Large Scale Integration (VLSI) chips. All the designs proposed are FPGA based. Once an FPGA prototype is concluded, the VLSI equivalent implementation can be done based on the same digital design and would be much better in terms of area and energy efficiency. Moreover, in the case of the included SC implementations, there is another limitation, which is the direct applicability to more complex or higher dimensional input data than the used for testing purposes as it is not trivial because architectures are ad-hoc and fully parallel. Finally, it is worth highlighting dealing only with digital circuits is in fact an additional limitation when designs require a large number of registers to perform parallel calculations. Especially in the case of the SC, due to the simplicity of the internal logic, it would be much better in terms of area and power to integrate, e.g. build the logic inside a 4T, 6T or 8T Static Random Access Memory (SRAM) cells instead of using D-type flip-flops (FPGA registers) to hold numbers accessed in parallel. Here, NT refers to N transistors, the standard D-type flip-flop is 12T and requires more area than a SRAM memory cell[3].

[3]However, the number of transistors is not necessarily a good metric for area [14].

1.4 Structure of the thesis

The thesis contents are organized around six chapters as described below:

- Chapter 1. This chapter introduces the context in which the thesis is framed, establishing the objectives and limitations of the research carried out.

- Chapter 2. It is intended to introduce **Background** theory. It includes a selection of topics to facilitate the understanding of this work. In a nutshell, the background is splited in three main topics. The first topic is number representation and computation, including conventional and SC. The second topic is ML focused on ANNs and RC as well as several training methods. Finally, the last part of the background chapter is focused on the available options for prototyping energy efficient pattern recognition applications.

- Chapter 3. It describes the **Methodology** followed for each hardware implementations, going from an algorithmic description to the digital design simulation and finally the FPGA implementation.

- Chapter 4. It explores two RC inference hardware implementations, both are **Fixed-Point Implementations**. The first one is a ring-topology ANN trained for Audio Event Detection (AED) using the UrbanSound8K dataset [15] and the second one is based on Cellular Automata, trained to recognized handwritten digits from the MNIST dataset [16].

- Chapter 5. It includes two SC inference hardware implementations. Both are **Stochastic Computing Implementations** trained to recognize handwritten digits from the MNIST dataset. The first is a Radial Basis Function Neural Network (RBF-NN) and the second is a CNN. The main contribution is related to the quantization aware training adapted to benefit the SC CNN implementation.

- Chapter 6. Finally, **Conclusions and Future Work** are discussed, including potential improvements and this thesis results dissemination in international journals and conferences, as well as other publications not covered in this document.

This document contains four appendices too, which are conveniently referenced in the text and include the following content.

- Appendix A. It summarizes **Gradient Descent Optimization** algorithms utilized for training ANNs in this work.

- Appendix B. It contains a list of **Fixed-Point Arithmetic** operations.

- Appendix C. It introduces three different **Random Number Generation** approaches utilized in the SC implementations.

Chapter 2

Background

This background chapter is focused to explain three widely known topics, which are the cornerstones of the hardware implementations collected in the thesis and answer the following questions:

- **How to represent and compute numbers?**
  Section 2.1 introduces a generic view of number representation systems and the most common choices in computer arithmetic. Moreover, the particular case of SC is discussed in depth in section 2.2 due to its important role in this work.

- **How computers *learn* to recognize patterns?**
  Knowing how ML algorithms are implemented and work is fundamental if one aims to adapt certain aspects of well-known methods at the architectural and algorithmic levels. Section 2.3 presents an overview of similarities and differences between pattern recognition methods.

- **How to design energy efficient pattern recognition applications?** In order to explore power/energy consumption improvements in front of MCUs, CPUs or GPUs for an specific task, one should explore different approaches. In this context, the development of specific hardware is one of the options adopted by research groups and companies. Section 2.4 presents an overview of the energy efficiency design options that can be implemented on hardware, and more specifically in reconfigurable hardware (FPGA).

2.1 Number representation and arithmetic

Throughout this document, different number representations are mentioned and applied in practice. Since the understanding of these representations is crucial to comprehend the basics of how digital hardware implementations work.

Nowadays, the vast majority of processor's ALUs use weighted representations to store and compute numbers [17]. This weighted code is a dense representation in the sense that every code has a one-to-one mapping with a particular number, i.e. it is optimal for data storage. Although there are other dense representations that are non-weighted (e.g. gray code [18]) and thus optimal for data storage, operations with/between numbers are not trivial or require conversions to weighted codes.

Even though a dense representation is optimal for data storage, it does not need to be optimal for computation in terms of hardware complexity or energy consumption. In this context, the concept of unary processing is introduced.

2.1.1 Representation

A number a with finite precision might be stored as a code containing m elements a_i and r possible values per element, so that each one can take integer values from 0 to $r-1$. In the literature, m is typically called width and r radix or base. The number a is interpreted as the weighted sum of its elements, hence the name. The weights of each element in the sum depend on its position as depicted in Fig. 2.1, given by the following expression:

$$a = \sum_{i=0}^{m-1} \mathsf{a}_i r^i. \tag{2.1}$$

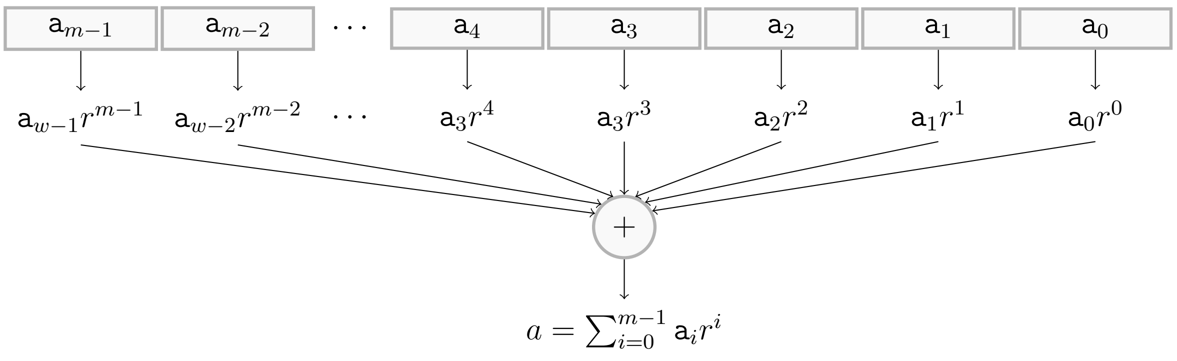


Figure 2.1: Unsigned fixed-point linearly weighted code of a number a with: code elements a_i, arbitraty radix r and width w, i.e. $a = (\mathsf{a}_{w-1}\mathsf{a}_{w-2}\ldots\mathsf{a}_1\mathsf{a}_0)_r$

For $r > 1$ weights represent jumps of r units going from left to right, so that every element adds an independent contribution to the final result. That is the reason why it is a dense representation.

One might ask which is the best radix choice for data storage in terms of theoretical representational efficiency R_{eff}. This question was answered by Stifler in 1950 [19] assuming r *need not to be an integral value and that the cost to represent each digit is proportional to* r [20]. So that the cost S to represent m elements of radix r is given by (2.2).

$$S = k_1 m r, \tag{2.2}$$

where k_1 is a constant. Given that $M = r^m$, M numbers can be represented and substituting $m = \frac{\ln M}{\ln r}$ into (2.2), yields:

$$S = k_2 \frac{r}{\ln(r)}, \tag{2.3}$$

where $k_2 = k_1 \ln(M)$ and S does not depend on m. In order to get the optimal r value of this function, its derivative is set to zero, as:

$$\left[\frac{dS}{dr}\right]_{r_{opt}} = k_2 \frac{\ln(r_{opt}) - 1}{\ln(r_{opt})^2} = 0 \longrightarrow r_{min} = e = 2.718\ldots \tag{2.4}$$

This means the optimal radix r_{opt} should be 3 for integer values. However, the relative difference with respect to $r = 2$ or $r = 4$ is small. In this case, the representational efficiency R_{eff} is defined as a ratio w.r.t. the minimum cost $S_{opt} \equiv S(r = e)$, i.e.

$$R_{eff} = \frac{\frac{1}{S}}{\frac{1}{S_{opt}}} = \frac{e\ln(r)}{r} \tag{2.5}$$

This representation efficiency is depicted in Fig. 2.2.

In Fig. 2.2, the case $r = 1$ has been ignored since its representational efficiency is quite poor. Taking the limit $r = 1$ in (2.5) yields $R_{eff} = 0$. Notice that for $r = 1$, the elements a_i could only take one value, so it is not possible to represent any number. On the other hand, if the elements a_i are allowed to be in two different states[1] (0 or 1) and the representation still sparse, then this is known as unary representation. This is the limit in which all positional weights are equal and element positions are no more relevant in such encoding scheme, i.e. any permutation of the elements a_i does not affect the value of a, which is never true for the traditional weighted representation with $r > 1$ and $M = r^m$ possible values per element. In addition, the decoding is not necessarily restricted to the sum of active elements, since other encoding/decoding schemes exist, mostly inspired by the brain [21]–[24].

Nowadays, ASIC technology is based on 2-level logic, from a practical point of view it makes sense to use radix 2 for data storage in modern digital computing devices. In this case elements are bits and the width is often referred to as the bit width. Under the assumptions in (2.2), note that the unary representation needs m bits to represent a number with M possible values, that is:

$$S = k_1 2^{M+1} \tag{2.6}$$

Therefore, the representational efficiency exponentially decreases with the number of unary bits:

$$R_{eff} = \frac{e \ln(M)}{2^{M+1}} \tag{2.7}$$

Even for $M = 2$ possible values, the efficiency is approximately $R_{eff} \approx 0.236$ and drops to a half of this value for $M = 4$ ($R_{eff} \approx 0.118$). Therefore, a unary code is not viable for data storage when compared to a dense representation.

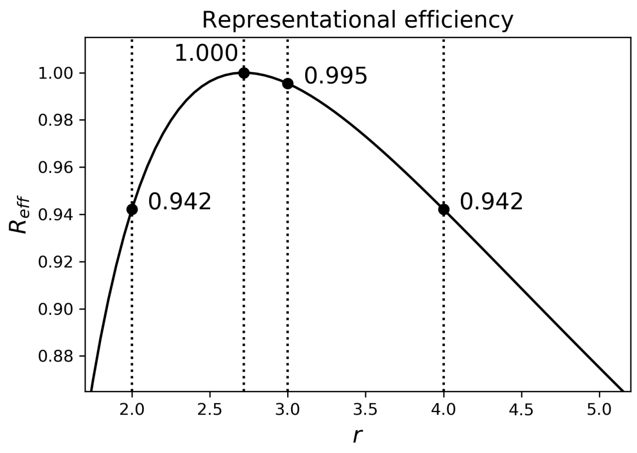


Figure 2.2: Graphical representation of the representational efficiency R_{eff}.

2.1.2 Arithmetic

So far, it has been discussed that radix 2 representation seems a reasonable choice in terms of representational efficiency and considering the nature of transistors, which are

[1]Imposing at least two different states is necessary for representation and computation since for $r = 1$, r^m is always 1.

the fundamental elements of the microelectronics industry. But, is it also better for data processing when compared to the unary scheme? there is not a one-off answer. It depends on the application and more specifically on the required precision. For example, unary processing strategies are not suitable to run numerical simulations, it requires negligible quantization error and are usually performed in double floating-point precision. Instead, if an application does not require high precision to achieve a desired model accuracy, e.g. inference in artificial neural networks, then some form of unary processing might be a good candidate.

Radix 2 arithmetic is typically done in the space domain, i.e. using combinational logic, and the result is instantaneous (or very fast). In the case of unary processing, computations might be translated to time domain. In particular, SC (section 2.2) provides a simple framework in terms of arithmetic logic complexity. Therefore, switching the computation paradigm would only make sense under certain circumstances. There are several aspects to evaluate:

- **Domain conversion**
  It is common to store input and parameter data as radix 2 numbers using e.g. SRAM and standalone register arrays. So that a translation or domain conversion between radix 2 and the unary code choice and viceversa is needed. However, in some cases the domain conversion might be less efficient than operating directly the source data. At the same time, it is possible to build highly parallel unary processing systems, suitable for near-sensor scenarios [25] based on SC to reduce data conversion hardware resources. In particular, they proposed to directly convert analog input data to stochastic bitstreams in order to increase energy efficiency in the first layer of SC CNNs.

- **Noise tolerance**
  Any integrated circuit signal might be exposed to noise from external physical sources, such as thermal noise, flicker noise, and shot noise [26]–[30]. Imagine a bit flip in a radix 2 representation, the error propagation across the different operations could be catastrophic. Instead, a bit flip in a unary bitstream would cause a very small error. Therefore, if an application requires noise tolerance, a unary processing technique such as SC or a deterministic variant should be a good solution [31].

- **Computational cost**
  Depending on the arithmetic operations and domain conversion resolution, a unary processing system might not be advantageous compared to the equivalent fixed or floating-point representation. There is a tredeoff between resolution and efficiency.

These aspects are discussed for SC in section 2.2. For the sake of completeness, it is worth mentioning that intermediate or mixed processing is possible, e.g. BURST [20]. Such work took ideas from binary weighted and SC processing schemes, which enables trade-offs in terms of circuit size and noise immunity.

2.2 Stochastic Computing

SC refers to a wide range of techniques in which continous (analog) or fixed point (digital) values are represented as random bitstreams[2] whose activation probability is proportional or related to these values. The strong point of such randomized representation resides in the simplicity of arithmetic operations between bitstreams. Arithmetic operations such as multiplication, absolute value difference, maximization or minimization can be performed by simple bit-wise operations.

The origin of SC dates back to the 1950s, introduced by J. von Neumann in his article *Probabilistic logics and the synthesis of reliable organisms from unreliable components* [32]. This work was based on R.S. Pierce's notes, inspired by the way nervous systems propagate information through pulse trains in a network of interconnected *organisms*, which process the information in such pulses. However, it was not until the late 1960s that the SC theory was completely developed, together with some early practical implementations [33].

In this period, two research laboratories working simultaneously on the SC principles came up with a similar approach. On the one hand, B.R. Gaines and J.H. Andrae implemented *A stochastic analog computer* [34] in 1965 when working at *Standard Telecommunications Laboratories* in England and later the researchers published a conference paper jointly with J.W. Esch [31] in April 1967. They focused on SC architectures suitable for AI and control algorithms. On the other hand, W.J. Poppelbaum and C. Afuso published *Noise-computer* [35] in 1965 at the University of Illinois (USA), reported digital/analog stochastic computer called Paramatrix, and later published a conference paper [36] in November 1967, also inspired by the ideas of J. von Neuman. Their research included programmable image processing SC architectures. It is also worth mentioning the contemporary work by S.T. Ribeiro [37] in 1964, who introduced a set of arithmetic units using width-modulated pulses as inputs, which are converted to pulse trains for computation and converted back to width-modulated pulses. In addition, this work, *Random-pulse machines* [38], published in June 1967 describes SC spatial integrators and matrix multipliers. By the end of the 1960s B.R. Gaines wrote a detailed book chapter entitled *Stochastic computing systems* [39], which summarizes the SC advances published up to that date.

There was an increasing interest in SC and its practical implementation during the 1970s. In fact, J.W. Esch built *A programmable analog computer based on a regular array of stochastic computing element logic* [40] in 1969. Finally, at the early 1980s, the SC research interest significantly dropped due to advances in integrated chip design and the advantages offered by conventional digital arithmetic over SC architectures in terms of versatility and precision.

Nevertheless, nowadays research interest in SC is growing [33], motivated by it's inherent noise-tolerant, e.g. [41], [42] and the increasing interest in energy efficient hardware architectures for NN inference, e.g. [43]–[45] and learning, e.g. [46], [47], as well as other highly parallelizable algorithms based on linear algebra, like the FFT [48].

This section is entirely devoted to introduce the foundations of SC, both from the theoretical and practical perspectives. In particular, implementations are introduced from a digital design point of view since all the designs explained in this document are targeted to FPGA platforms. It is not intended to cover all the literature in this regard, but to the end of the section the reader who is not familiar with SC will understand

[2]Bitstreams might be pseudorandom or even not random at all.

how it works and the reasons why it is an attractive framework for the aftermentioned applications.

At the architecture level, a system or subsystem based on SC is composed by three main blocks: stochastic number generator (SNG), probabilistic computing circuit and output decoder circuit, which are illustrated in Fig. 2.3. In this architecture, input data might be presented in any format, so it is first converted to stochastic bitstreams using SNGs. Then these bitstreams, which represent the original input data, feed the probabilistic computing logic in which simple SC elements based on the integration of SC bitstreams take place. Finally, the resulting bitstreams are decoded or converted to a convenient output format.

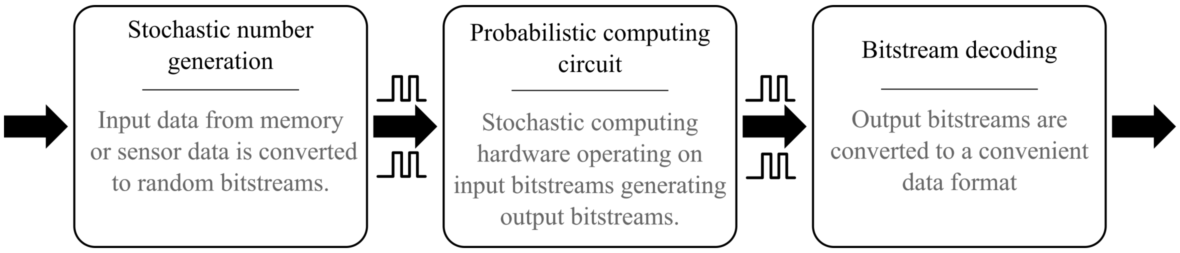

Stochastic number generation
Input data from memory or sensor data is converted to random bitstreams.
Probabilistic computing circuit
Stochastic computing hardware operating on input bitstreams generating output bitstreams.
Bitstream decoding
Output bitstreams are converted to a convenient data format

Figure 2.3: Generic SC architecture.

Since there exist several encodings to interpret bitstreams' information as numerical values, some of these are enumerated in Section 2.2.1. Then, bitstream generation as well as output bitstream decoding is explained in section 2.2.2, which we refer to as *domain conversion*. Next, several *operations* between bitstreams are listed in Section 2.2.3.

2.2.1 Bitstream coding

There exist many ways to encode a n-bit binary weighted signal to a sequence of 2^n bits. However, arithmetic operations between bitstreams might become complex depending on the bitstream coding.

All SC codes are entirely or partially based on representing quantities as one or more bitstreams whith activation probabilities proportional to the original quantities or related to a relationship between them.

Four different codings are described in this section: *unipolar*, *bipolar*, *sign-magnitude* and *extended*. The first three are directly related to the SC implementations proposed in this work and the fourth one (*extended*) has been included for completeness and to include a less trivial example compared to the first three codes, proposed by the supervisors of this thesis [49].

2.2.1.1 Unipolar

The unipolar coding was the initial approach to encode quantities as stochastic bitstreams [34], [35] and it is the easiest yet the most limited approach. An input quantity x is normalized in the range $[0, 1]$ so that it represents an activation probability p_x. Thus, the SC outputs are also restricted in the unit range. This approach is a good option when the task to be implemented does not require arithmetic operations between signed quantities. On the other hand, its main advantages are: few logic resource requirements, the fact that only one bitstream is required to represent each quantity and multiplication by zero yields exactly zero, which is not the case for other SC codings.

2.2.1.2 Bipolar

The bipolar coding was introduced as a method to represent a signed quantity with a single bitstream so that the outputs from operations between bitstreams might represent signed quantities too. In this case an input quantity x is normalized in the range $[-1, +1]$ and denoted as p_x^*, which is referred to as a bipolar variable. This bipolar variable is interpreted as an activation probability p_x via the change of variables (2.8). Despite this approach extends the unipolar code, the multiplication by zero does not yield exactly zero because $p_x^* = 0$ is represented by a bitstream with activation $p_x = 0.5$, which might not coincide with the number of active states divided by length of the bitstream. Also, even if the input bitstream has exactly 50% activation probability, the multiplicand[3] might be a bitstream with an odd number of active states since the resulting bitstream might not have exactly 50% activation probability.

$$p_x = \frac{p_x^* + 1}{2} \tag{2.8}$$

2.2.1.3 Sign-magnitude

In contrast to single bitstream unipolar and bipolar codings, the sign-magnitude (a.k.a. two-line bipolar) codings needs two bitstreams to represent a single number. In this case an input quantity x is normalized in the range $[-1, +1]$ and denoted by p_x^*, then p_x^* is decomposed according to (2.9).

$$p_x^* = \operatorname{sgn}(x)\operatorname{mgn}(p_x^*) \tag{2.9}$$

where $\operatorname{sgn}(x) = \operatorname{sgn}(p_x^*)$ is the sign ($+1$ if x is positive and -1 otherwise) and $\operatorname{mgn}(p_x^*)$ is the absolute value.

Since the main idea behind this convention is to represent the sign and magnitude as separate bitstreams, both quantities need to be converted to activation probabilities. Notice the sign is a bipolar variable can be represented as an activation probability via (2.8), so that the Heaviside function (2.10) represents this activation probability.

$$\mathcal{H}(x) = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.10}$$

As regards the magnitude or absolute value, since it is already an unipolar variable, the corresponding activation probability is given by $\operatorname{mgn}(p_x^*)$ or the simplified notation

$$\mathcal{M}(x) \equiv \operatorname{mgn}(p_x^*) \tag{2.11}$$

so that (2.9) can be rewritten as:

$$p_x^* = (2\mathcal{H}(x) - 1)\,\mathcal{M}(x) \tag{2.12}$$

where $\mathcal{H}(x)$ and $\mathcal{M}(x)$ are the two bitstream activation probabilities, which define a bipolar variable p_x^*. Multiplication by zero is exact without a significative hardware overhead and the needed computation time is generally reduced by 1/2 factor compared to the bipolar case. However, in digital systems signed values are conveniently encoded in its two's complement representation, so that decomposition in sign and magnitude requires additional operations, which is not a problem for parameters but can be an overhead for input data.

[3] In this case, the number that gets multiplied by zero.

2.2.1.4 Extended

The extended representation uses two bitstreams encoding two bipolar variables p_x^* and q_x^* to represent a quantity x, which does not need to be normalized since in this case the range is not bounded. This approach is based on representing x as the quotient of two bipolar bitstreams:

$$x = \frac{p_x^*}{q_x^*} \tag{2.13}$$

so that arbitrarily large numbers can be represented. Although this smart representation has obvious advantages in terms of input/output representability, but some problems arise in practice. Due to data quantization, as in the simple bipolar case, multiplication by zero is not exact. Also, when q_x^* is small and precision is limited, the representation of big positive and negative numbers is limited by the quantization too.

2.2.2 Domain conversion

In any of the previously enumerated codings, it is necessary to convert input variables to bitstreams with a certain activation probability. In particular, for the unipolar and bipolar cases, and according to notation defined in Table 2.1, an input variable x must be scaled in the interval $[0, 1]$ (unipolar) or $[-1, +1]$ (bipolar) to obtain the corresponding normalized input variable. An unipolar activation probability coincides with the normalized input p_x. In contrast, the bipolar scaled input p_x^* represents an activation probability $p_x = \frac{1}{2}(p_x^* + 1)$. Even though the value encoded by p_x is different, both scenarios are based on the conversion of each variable to a single bitstream. Given an activation probability p_x, regardless of its arithmetic meaning, it is converted to a bitstream $\tilde{x}(t)$ by comparison with a sequence following some kind of distribution, e.g. a random uniform sequence $\{R_{x,n} \sim U[0, 1), n = 0, 1, 2, \ldots N - 1\}$. These random events are updated at regular time periods[4] T. Therefore, if the observations of $\{R_{x,n}\}$ are $\{r_x(nT)\}$, being $r_x(t)$ a continous-time function updated at regular intervals T and the continous-time representation of the bitstream is generated by comparison with these random observations:

$$\tilde{x}(t) = \begin{cases} 1 & p_x > r_x(t) \\ 0 & \text{otherwise} \end{cases} \tag{2.14}$$

For a bitstream of length N, comparison 2.14 is updated each period T, so that the evaluation time is NT. From this definition, the number of active states converges to Np_x as N increases. This means a bitstream activation probability p_x is approximately recovered by counting active states in the bitstream. Since bitstreams have been defined as a continous (boolean) function of time, the recovered activation probability might be expressed using either continous or discrete time integration as reflected by (2.15).

$$p_x = \langle \tilde{x}(t) \rangle = \underbrace{\lim_{N\to\infty} \frac{1}{NT} \int_0^{NT} \tilde{x}(t)dt}_{\text{analog RC filter}} = \underbrace{\lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{N-1} \tilde{x}(nT)}_{\text{digital counter}} \tag{2.15}$$

[4] Although updates might happen at irregular time periods, using regular time intervals makes more sense for synchronous digital hardware implementation.

As highlighted in this expression, the physical implementation of the integral form may be implemented by an RC filter[5], while the discrete summation can be implemented by a digital counter.

At this point, it has been already described how to encode some quantity as a random bitstream and how to (approximately) recover it from its bitstream for unipolar and bipolar SC codings. In the case of the sign-magnitude convention, we need two bitstreams per variable and the sign bitstream $\tilde{\mathcal{H}}[x](t)$ is trivially obtained via:

$$\tilde{\mathcal{H}}[x](t) = \mathcal{H}(x) \;\; \forall t \in [0, NT] \tag{2.16}$$

while the magnitude bitstream obtained just like in the unipolar case. The magnitude of a bipolar quantity represents an activation probability, so that the magnitude bitstream $\tilde{\mathcal{M}}[x](t)$ is given by (2.14), but replacing p_x by $\mathcal{M}(x)$. Therefore, the sign-magnitude bitstream pair decoding is given by (2.17).

$$\begin{aligned} p_x^* &= \lim_{n\to\infty} \frac{1}{NT} \int_0^{NT} \left(2\mathcal{H}[x](t) - 1\right) \mathcal{M}[x](t)dt \\ &= \lim_{n\to\infty} \frac{1}{N} \sum_{n=0}^{N} \left(2\mathcal{H}[x](nT) - 1\right) \mathcal{M}[x](nT) \end{aligned} \tag{2.17}$$

Finally, notice the extended SC coding is also represented by two bitstreams, both wires represent bipolar variables that are encoded and decoded separately. Bipolar numerator and denominator bitstreams are independently obtained via (2.14) and are approximately recovered via (2.15).

2.2.3 Operations

Operations between stochastic bitstreams typically require simple digital gates, specially in the case of the multiplication. In general, both DSP and ML algorithms rely on linear algebra operations (multiplication, addition and subtraction) and other simple operations such as maximum, minimum and absolute value as well as non-linear functions used in ANN activations, e.g. ReLU or sigmoid. We can classify these operations in two groups depending on their complexity. These two groups are (a) combinational and quasi-combinational operations on bitstreams, and (b) Internal feedback operations. While (a) comprehends a list of simple SC operations, (b) refers to the implementation of intricate operations requiring memory elements and feedback. However, in this work we will refer to the literature on this topic and discuss only the SC summation operation.

Before introducing these operations, it is convenient to highlight the fact that some of them have strict bitstream correlation requirements. Typically, SC correlation between bitstreams is controlled at the generation stage. On the one hand, if a pair of input bitstreams $\tilde{x}(t), \tilde{y}(t)$ generated from activation ratios p_x, p_y by comparison with the same random sequence, i.e. $r_x(t) = r_y(t)$, then $\tilde{x}(t)$ and $\tilde{y}(t)$ are maximally correlated (or simply correlated for short) as depicted by blue time series in Fig. 2.4. On the other hand, if the same pair of input activation ratios is compared to independent random sequences $r_x(t), r_y(t)$, then the generated bitstreams are maximally uncorrelated (or uncorrelated for short) as depicted by blue and orange time series in Fig. 2.5.

[5]The principle is the same than for PWM to analog converters.

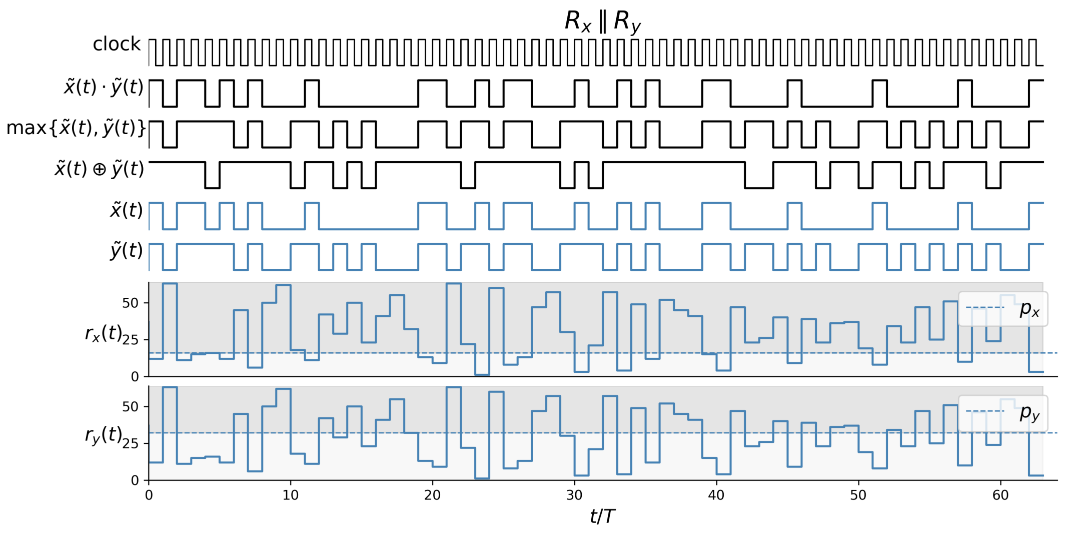


Figure 2.4: Example visualization of domain conversion for a pair of variables (x, y) generated by comparison with the same random number and some SC operations.

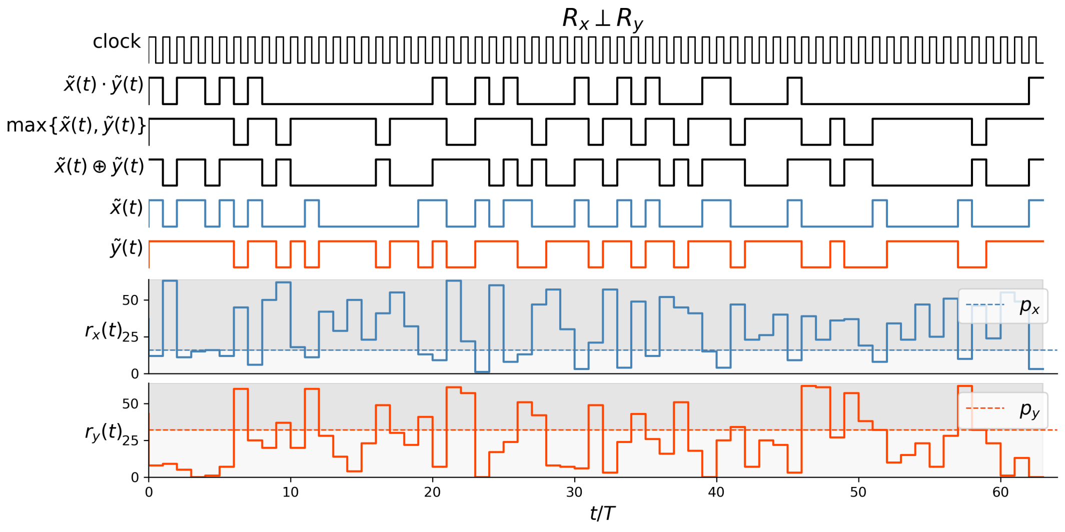


Figure 2.5: Example visualization of domain conversion for a pair of variables (x, y) generated by comparison with uncorrelated random numbers and some SC operations.

Therefore, if a bitstream results from a specific combination of input bitstreams, its mean activation ratio might depend on whether the inputs were correlated ($\tilde{x} \parallel \tilde{y}$) or uncorrelated ($\tilde{x} \perp \tilde{y}$). As an example, notice correlated AND, OR and XOR combinations depicted Fig. 2.4 result in different activation ratios compared to the equivalent uncorrelated operations depicted in Fig. 2.5.

2.2.3.1 (a) Combinational and quasi-combinational operations

Most of this kind of operations can be implemented with a single logic gate per pair of input bitstreams. Therefore, unipolar and bipolar basic operations are listed in Table 2.1. In this table, the most common operations are colored depending on whether input bitstreams are correlated (blue), uncorrelated (orange) or correlation does not matter (green). These operations are listed and explained below.

Table 2.1: Unipolar and bipolar SC notation and correlation-aware single gate equivalent operations in the normalized space. The most commonly used operations have been hightlighted with different colors depending on input correlation requirements. (Blue) correlated input bitstreams. (Orange) Uncorrelated input bitstreams. (Green) correlation does not matter.

		unipolar	**bipolar**
notation	original inputs	x, y	x, y
	scaling interval	$[0, 1]$	$[-1, +1]$
	scaled inputs	p_x, p_y	p_x^*, p_y^*
	activation probabilities	p_x, p_y	p_x, p_y
	SC bitstreams	$\tilde{x}(t), \tilde{y}(t)$	$\tilde{x}(t), \tilde{y}(t)$
gate, correlation	NOT, -	$1 - p_x$	$-p_x^*$
	AND, correlated	$\min\{p_x, p_y\}$	$\min\{p_x^*, p_y^*\}$
	AND, uncorrelated	$p_x p_y$	$\frac{1}{2}\left(p_x^* p_y^* + p_x^* + p_y^* - 1\right)$
	OR, correlated	$\max\{p_x, p_y\}$	$\max\{p_x^*, p_y^*\}$
	OR, uncorrelated	$p_x + p_y - p_x p_y$	$\frac{1}{2}\left(p_x^* + p_y^* - p_x^* p_y^* + 1\right)$
	XOR, correlated	$\|p_x - p_y\|$	$\|p_x^* - p_y^*\| - 1$
	XNOR, uncorrelated	$2p_x p_y - p_x - p_y + 1$	$p_x^* p_y^*$

- **Complementary**
  The complementary is the simplest operation, it takes an input bitstream and the output bit values are reversed, i.e. high values become low values and viceversa. If this operation is applied to a bitstream with activation probability p_x then the output activation probability becomes $1 - p_x$ without error. The complementary operation is achieved by a NOT logic gate as illustrated in Fig 2.6.

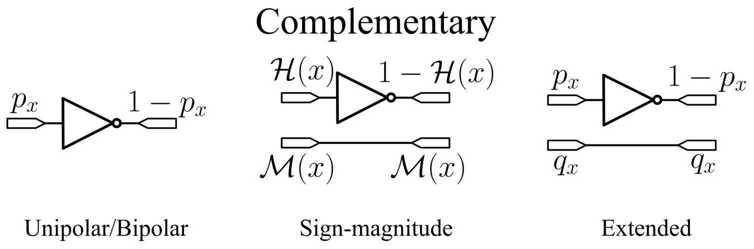


Figure 2.6: Digital logic schematic of the SC complementary operation and corresponding input and output activation probabilities for different SC codes. Since it consists of combinational logic applied to a single bitstream, the sequence from which the bitstream is generated does not play any role.

This operation is denoted as $\overline{\tilde{x}(t)}$, given the input bitstream $\tilde{x}(t)$, so that the output activation probability is

$$\langle \overline{\tilde{x}(t)} \rangle = \langle 1 - \tilde{x}(t) \rangle = 1 - \langle \tilde{x}(t) \rangle = 1 - p_x \tag{2.18}$$

If bitstreams come from unipolar variables, then the relation between input and output is directly obtained via (2.18). In contrast, if bitstreams come from bipolar variables then p_x^*, which is represented by a bitstream with activation probability p_x, it becomes $-p_x^*$. Therefore, the bipolar variable is related to its activation probability via (2.8), so that the input bitstream has the following activation probability:

$$p_x = \frac{1 + p_x^*}{2} \tag{2.19}$$

and accordingly, the output bitstream:

$$1 - p_x = \frac{1 - p_x^*}{2} \tag{2.20}$$

so, the complementary operation converts a bipolar variable p_x^* into $-p_x^*$.

- **Multiplication**
  The SC multiplication is far simpler than the conventional digital counterpart. In the unipolar case, two uncorrelated bitstreams $\tilde{x}(t)$ and $\tilde{y}(t)$, with activation probabilities p_x and p_y, results in an output bitstream with activation probability $p_x p_y$ by means of a simple AND logic gate. The AND logic gate outputs the high state only when both inputs are high, so if these inputs have been generated from independent random uniform sequences the probability of both inputs being high at a given time is $p_x p_y$, i.e.

$$\langle \tilde{x}(t) \cdot \tilde{y}(t) \rangle = \langle \tilde{x}(t) \rangle \langle \tilde{y}(t) \rangle = p_x p_y \tag{2.21}$$

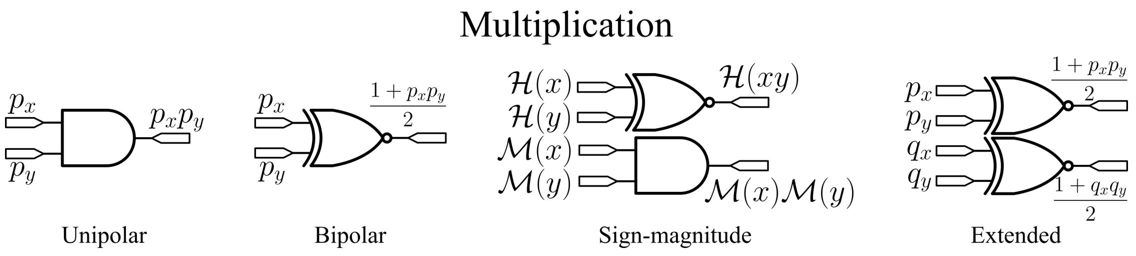


Figure 2.7: Digital logic schematic for the SC multiplication operation and corresponding input and output probabilities for different SC codes. All pairs of bitstreams that are inputs to the same logic gate must be maximally uncorrelated, except in the case of the sign-magnitude code, for which correlation does not matter for the sign bitstreams.

In contrast, if bitstreams come from bipolar variables, then the multiplication operation is obtained via an XNOR logic gate, that is:

$$\begin{aligned}
\langle \overline{\tilde{x}(t) \oplus \tilde{y}(t)} \rangle &= \langle \max \left\{ \tilde{x}(t) \cdot \tilde{y}(t), \overline{\tilde{x}(t)} \cdot \overline{\tilde{y}(t)} \right\} \rangle \\
&= \langle \max \{ \tilde{x}(t) \cdot \tilde{y}(t), (1 - \tilde{x}(t))(1 - \tilde{y}(t)) \} \rangle \\
&= \langle \tilde{x}(t) \cdot \tilde{y}(t) \rangle + \langle (1 - \tilde{x}(t))(1 - \tilde{y}(t)) \rangle \\
&= p_x p_y + (1 - p_x)(1 - p_y) \\
&= 1 + 2 p_x p_y - p_x - p_y \\
&= 2 \frac{(1 + p_x^*)(1 + p_y^*)}{4} - \frac{1 + p_x^*}{2} - \frac{1 + p_y^*}{2} + 1 \\
&= \frac{1 + p_x^* p_y^*}{2}
\end{aligned} \tag{2.22}$$

The multiplication digital SC designs for the different codings are depicted in Fig. 2.7, including the implementation for sign-magnitude and extended codings. Moreover, notice this implementation is also valid for the square operation if both inputs represent the same value and are uncorrelated. This can be achieved by delaying a single input bitstream (or bitstream pair) if self-correlations are not present. This special case is described by Fig. 2.8, where black squares represent time delays of at least one clock period, which can be achieved by flip-flops.

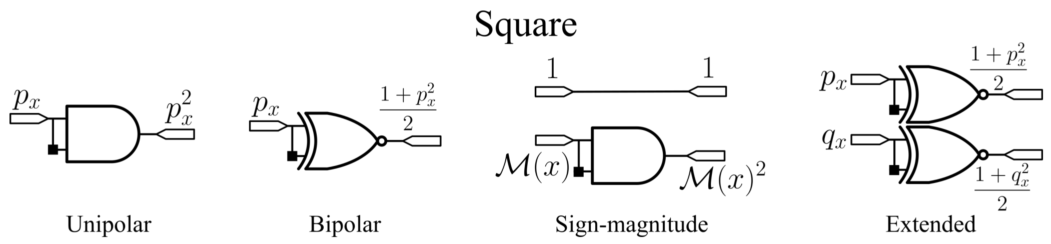


Figure 2.8: Digital logic schematic for the SC square operation and corresponding input and output probabilities for different SC codes. Input bitstreams must not present self-correlations in time.

- **Maximum and minimum**

  Both maximum and minimum SC operations require correlated input bitstreams, i.e. inputs generated by comparison with the same sequence. If this condition is met then the minimum operation requires a single AND gate in the unipolar and bipolar codings, so that:

$$\langle \tilde{x}(t) \cdot \tilde{y}(t) \rangle = \langle (\min \{\tilde{x}(t), \tilde{y}(t)\} \rangle = \min \{p_x, p_y\} \tag{2.23}$$

  Similarly, the maximum operation requires a single OR gate, so that:

$$\langle \max \{\tilde{x}(t), \tilde{y}(t)\} \rangle = \max \{p_x, p_y\} \tag{2.24}$$

  Notice $\min \{p_x, p_y\} = p_x$ or $\max \{p_x, p_y\} = p_x$ implies $\min \left\{p_x^*, p_y^*\right\} = p_x^*$ or $\max \left\{p_x^*, p_y^*\right\} = p_x^*$, respectively. Therefore, these operations are the same for the unipolar and bipolar codings and the corresponding SC digital implementation is depicted in Fig. 2.9.

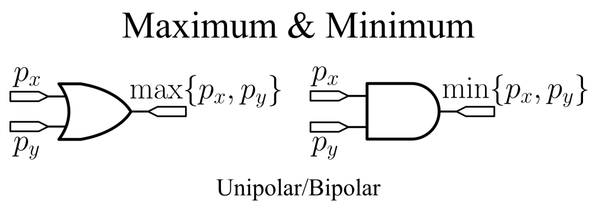


Figure 2.9: Digital logic schematic for the SC maximum and minimum operations applied to unipolar and bipolar bitstreams. Input bitstreams must be maximally correlated.

  It is worth highlighting both maximum and minimum operations can also be implemented by means of combinational logic in the case of sign-magnitude pairs of bitstreams. Apparently, these implementations do not present a clear improvement compared to other approaches.

- **Absolute value subtraction**

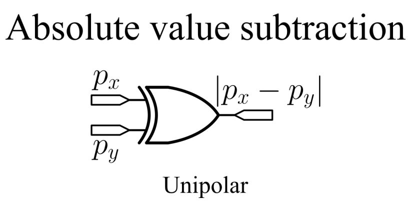


Figure 2.10: Digital logic schematic for the SC absolute value subtraction applied to unipolar bitstreams. Input bitstreams must be maximally correlated.

The SC absolute value subtraction operation is performed between pairs, which must be correlated and coded in the unipolar representation. The operation is done via a single XOR logic gate (see Fig. 2.10)

$$\begin{aligned}\langle \tilde{x}(t) \oplus \tilde{y}(t)\rangle &= \langle \tilde{x}(t)\,(1-\tilde{y}(t))\rangle + \langle \tilde{y}(t)\,(1-\tilde{x}(t))\rangle \\ &= -2\langle \tilde{x}(t)\cdot\tilde{y}(t)\rangle + \langle \tilde{x}(t)\rangle + \langle \tilde{y}(t)\rangle \\ &= -2\min\{p_x, p_y\} + p_x + p_y \\ &= |p_x - p_y|\end{aligned} \tag{2.25}$$

- **Average addition (MUX)**
  The classical SC average addition is accomplished by time-division multiplexing of the inputs to the output. The average addition of unipolar, bipolar, sign-magnitude or extended bitstream pairs is done via digital multiplexers and a random uniform selector signal, as depicted in Fig. 2.11. Notice the selector signal with activation probability 1/2 selects each input 50% of the time, so that the output has the average activation probability.

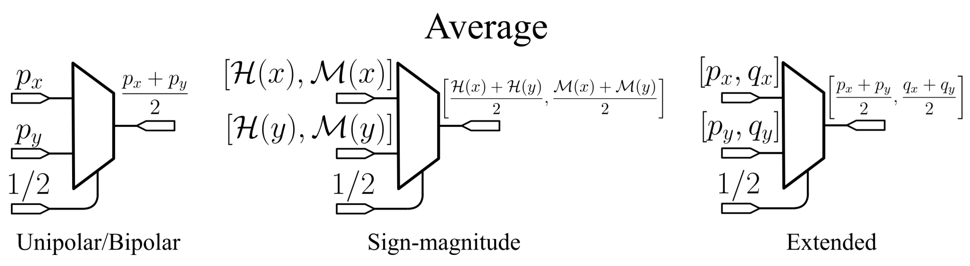

Average
p_x
p_y
$\frac{p_x+p_y}{2}$
1/2
Unipolar/Bipolar
$[\mathcal{H}(x), \mathcal{M}(x)]$
$[\mathcal{H}(y), \mathcal{M}(y)]$
$\left[\frac{\mathcal{H}(x)+\mathcal{H}(y)}{2}, \frac{\mathcal{M}(x)+\mathcal{M}(y)}{2}\right]$
1/2
Sign-magnitude
$[p_x, q_x]$
$[p_y, q_y]$
$\left[\frac{p_x+p_y}{2}, \frac{q_x+q_y}{2}\right]$
1/2
Extended

Figure 2.11: Digital logic schematic of the SC average operation using multiplexers. Correlation between bitstreams does not matter.

The same concept can be extended to multiple (n) inputs or pairs of inputs using an $n : 1$ multiplexer with an uniformly distributed selector signal containing integers ranging from 0 to $n-1$, so that each input is equally probable at the output, resulting in the average bitstream.

2.2.3.2 (b) Internal feedback operations

The set of internal feedback operations and its implementations are heterogeneous. In fact, many of these implementations are ad-hoc for a given application.

In particular, V. Canals collected many implementations of nonlinear functions approximations [50], e.g. division, hyperbolic tangent, sigmoid, exponentiation or Gaussian, among other. B. D. Brown *et al.* proposed to implement functions in the SC domain using finite state machines (FSMs) [51]. P. Li *et al.* proposed a general approach to implement functions from SC random bitstreams using finite state machines (FSMs) [52].

In particular, the accumulative parallel counter (APC), depicted in Fig. 2.12, should be introduced before the SC implementations presented in Chapter 5. In general, it takes m input bitstreams and counts how many of them are in the high state using a parallel counter, which outputs a $\lceil \log_2(m)\rceil$-bit binary weighted value and is accumulated over time each clock cycle, being N the maximum number of clock cycles, so that the register bit width has to be $\lceil \log_2(mN)\rceil$ at least. Therefore, the output is a

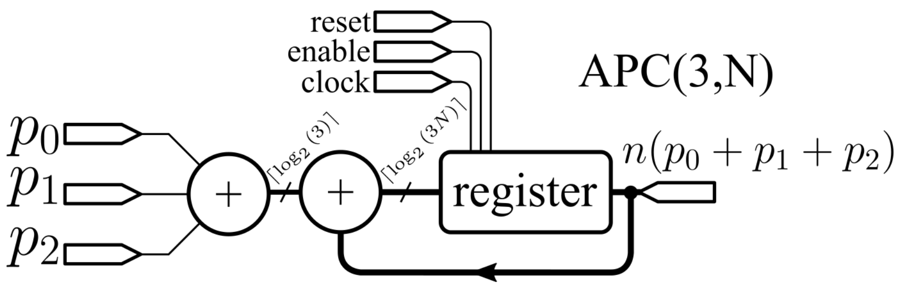


Figure 2.12: Accumulative parallel counter digital design.

radix-2 unsigned integer with mean value $n \sum_{i=0}^{m} p_i$. At the same time, n represents the current time step, which is equal to the final number evaluation cycles N.

The APC might be used to compute the average and convert it back to a stochastic bitstream for further computations, e.g. a three input bitstreams APC appointed as stochastic accumulative parallel counter (SAPC) is illustrated in Fig. 2.13.

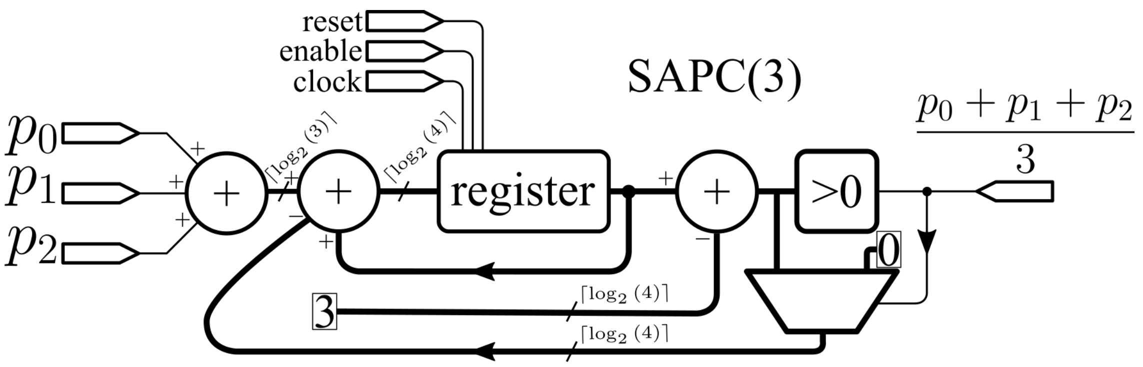

reset
enable
clock
SAPC(3)
p_0
p_1
p_2
$\lceil \log_2(3) \rceil$
$\lceil \log_2(4) \rceil$
register
>0
0
3
$\lceil \log_2(4) \rceil$
$\lceil \log_2(4) \rceil$
$\frac{p_0 + p_1 + p_2}{3}$

Figure 2.13: Accumulative parallel counter digital design incorporating an average bitstream output. Here the threshold value is equal to the number of input bitstreams to obtain the average activation probability.

The SAPC is more accurate than the SC average addition based on the multiplexer-based design, especially as the number of input bitstreams increases. In the multiplexer case, an input per cycle is evaluated, so that for m inputs the length of the output (average) bitstream is mN. However, in the case of the SAPC, every input bit is added each clock cycle and accumulated in the register. So, the length of the output bitstream is N assuming the same target precision obtained using the multiplexer approach.

However, the SAPC implementation is mainstream for unipolar and bipolar bitstream representations but not for sign-magnitude and extended codings.

Nevertheless, there exists a similar APC solution to sum numbers in the sign-magnitude representation, see Fig. 2.14. This figure depicts an APC adapted to integrate sign-magnitude bitstream pairs. This design is the same one illustrated in Fig. 2.12, but now inputs are two's complement streams representing either +1 or −1. At the same time, other works proposed the scaling-free SC addition [48], which outputs a bitstream not proportional to the addition or average operation and depends on the number of inputs[6].

[6]This was not a problem for the authors since they implemented the FFT, which can be decomposed in pipelined two-input operations, i.e. the number of inputs is always the same and their addition circuitry can be calibrated for that specific case.

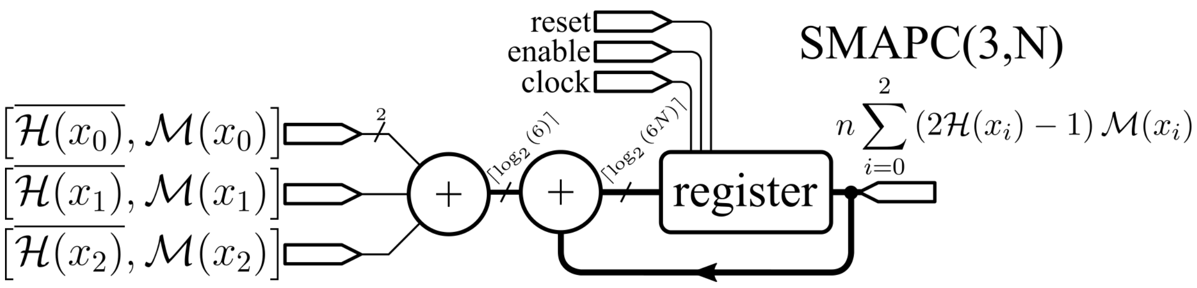

reset
enable
clock
SMAPC(3,N)
$n\sum_{i=0}^{2}(2\mathcal{H}(x_i)-1)\mathcal{M}(x_i)$
$[\mathcal{H}(x_0),\mathcal{M}(x_0)]$
$[\mathcal{H}(x_1),\mathcal{M}(x_1)]$
$[\mathcal{H}(x_2),\mathcal{M}(x_2)]$
2
$\lceil\log_2(6)\rceil$
$\lceil\log_2(6N)\rceil$
register

Figure 2.14: Accumulative parallel counter digital implementation adapted to integrate sign-magnitude bitstream pairs.

2.2.4 Error analysis

When converting an activation probability p_x to the corresponding bitstream $\tilde{x}(t)$ as described in subsection 2.2.2, the value of $\tilde{x}(t)$ at each time step follows Bernoulli random variable. Therefore, repeating the experiment N times results in an integer number of active states between 0 and N given by the random variable $N_{B,x} \sim B(N, p_x)$. So that $N_{B,x}$ follows a Binomial distribution with activation probability p_x and the probability of getting n *successes* in N independent experiments is given by (2.26). This equation holds as long as bitstreams are obtained by comparison with a sequence of independent and identically distributed uniform random numbers, which means $r_x(nT)$ could contain repeated values for different values of n.

$$P(N_{B,x} = n) = \binom{N}{n} p_x^n (1-p_x)^{N-n} \tag{2.26}$$

It can be shown that the corresponding mean and variance are given by:

$$\mathrm{E}\left(N_{B,x}\right) = N p_x \tag{2.27}$$

$$\mathrm{var}\left(N_{B,x}\right) = N p_x (1-p_x) \tag{2.28}$$

Notice (2.15) is equivalent to the expected value expression above, but expressed using a Bernoulli random variable instead of bitstream timeseries. Also, given the variance, it is possible to propagate the error through domain conversion stage and different SC operations. According to (2.27) and (2.28), recovering an activation probability p_x from the corresponding bitstream has a measurement deviation equal to std $\left(\frac{N_{B,x}}{N}\right)$, so that the measured activation probability assuming no intermediate SC operations is

$$\underbrace{\frac{1}{N}\sum_{n=0}^{N-1}\tilde{x}(nT)}_{\text{measurement}} = \underbrace{p_x \pm \sqrt{\frac{p_x(1-p_x)}{N}}}_{\frac{1}{N}\left(\mathrm{E}\left(N_{B,x}\right) \pm \mathrm{std}\left(N_{B,x}\right)\right)} \tag{2.29}$$

Fig. 2.15 illustrates how the standard deviation std $\left(\frac{N_{B,x}}{N}\right)$ depends on the number of Bernoulli experiments or bitstream length N. The left hand side figure shows mean and maximum standard deviation compared to the corresponding quantization error $(1/N)$. Since the measurement error also depends on p_x, the right hand side figure shows how the standard deviation depends on N and p_x. Notice the measurement error is maximum for $p_x = 0.5$ and there is no associated conversion error for $p_x = 0$ and $p_x = 1$.

In contrast, if bitstreams are obtained by comparison with a maximum length random sequence, random number repetitions are not allowed and the measurement error decreases to zero[7]. This idea was introduced in [53], inspired by Monte Carlo and quasi-Monte Carlo integration methods [54], [55].

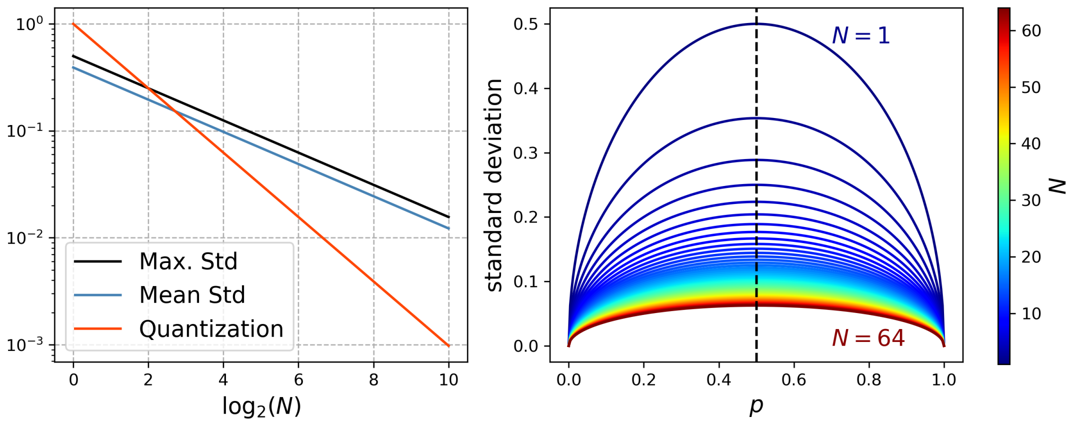


Figure 2.15: Binomial standard deviation.

The most relevant operations in this work are related to uncorrelated unipolar and bipolar bitstream multiplications. Since the error is different from zero for uncorrelated bitstream operations, the error analysis has been restricted to these two cases and domain conversion error is neglected. First, according to Table 2.1, for the unipolar multiplication:

$$\underbrace{\frac{1}{N}\sum_{n=0}^{N-1}\tilde{x}(nT)\cdot\tilde{y}(nT)}_{\text{measurement}} = \underbrace{p_x p_y}_{\text{ideal outcome}} \pm \underbrace{\sigma_{xy}}_{\text{unipolar product error}} \tag{2.30}$$

where the error term refers to the standard deviation, which might come from e.g. applying error propagation assuming p_x and p_y can be approximated by (2.29). However, carefully chosen random sequences to minimize the deviation term and a bitstream might represent an integer with no associated error. So, the error term is reduced compared to that propagated from the combined binomial distribution. In addition (2.30) result is easily extended to the sign-magnitude coding for a bitstream length equal to $N/2$. In practice N is finite and undesired correlations make the error greater than zero. For the bipolar product, the same reasoning is applied:

$$\left(\underbrace{\frac{1}{N}\sum_{n=0}^{N-1}\overline{\tilde{x}(nT)\oplus\tilde{y}(nT)}}_{\text{measurement}}\right)^{*} = \underbrace{p_x^* p_y^*}_{\text{ideal outcome}} \pm \underbrace{4\sigma_{xy}}_{\text{bipolar product error}} \tag{2.31}$$

The main difference compared to (2.30) is on the error term, which doubles compared the unipolar case because $\langle\overline{\tilde{x}(t)\oplus\tilde{y}(t)}\rangle = p_x p_y + (1-p_x)(1-p_y)$ and the variance for $(1-p_x)(1-p_y)$ is the same than for $p_x p_y$. Notice this error term is also doubled when the measurement is converted to the bipolar representation.

[7]Assuming no intermediate SC operations and input and output resolution are the same.

As an example to estimate σ_{xy} in (2.30) and (2.31), let $\tilde{x}(t)$ be a PWM-like bitstream so that $Np_x = \lfloor Np_x \rfloor = \sum_{n=0}^{Np_x-1} \tilde{x}(nT)$ or $Np_x = \lfloor Np_x \rfloor = \sum_{n=N(1-p_x)}^{N-1} \tilde{x}(nT)$, which can be easily generated by substituting the random sequence $r_x(t)$ by simple counter. Also, assume $\tilde{y}(t)$ has been generated by comparison with a uniform random number sequence $r_y(t)$, so that $Nr_y(t)$ is a maximum length sequence returning numbers from 1 to N and there is an associated standard deviation to the measure of Np_y. In this case the PWM-like bitstream $\tilde{x}(t)$ acts as a binary mask. The collision bitstream $\tilde{x}(t)\tilde{y}(t)$ activation rate becomes zero for $t \geq Np_xT$, i.e. the measurement is given by $\frac{1}{N}\sum_{n=0}^{Np_x-1} \tilde{y}(nT)$ because the maximum length sequence property is lost, i.e. the maximum random number value is greater than the sequence length. The expected value of this measurement is p_xp_y and its standard deviation is $\sigma_{xy} = \sqrt{\frac{p_y(1-p_y)}{Np_x}}$, as in (2.29) but with a different sequence length. Recall this was just an example and there are multiple possibilities depending on how the bitstreams are being generated, generally more difficult to derive analytically, so some authors choose to perform a simulation to characterize the error of their SC-based multipliers instead of providing a theoretical proof [56], [57]. In any case, the error is not uniform for different input activation probabilities, in particular, it is maximum when both activation probabilities are 50% and zero if one of the input activation probabilities is either 0 or 1. In our example $\sigma_{xy} \approx 0.044$ for $N = 256$, i.e. probabilities encoded as 8-bit unsigned numbers. Fortunately, this result can be improved using a maximum length random uniform sequence for $r_y(t)$ or, even better, for both $r_x(t)$ and $r_y(t)$.

Instead, two's complement multiplication between two $\lceil \log_2 N \rceil$-bit numbers x and y, and scaling to the same bit precision leads to an error given by $\frac{xy}{N/2} - \lfloor \frac{xy}{N/2} \rfloor$, which is equivalent to $\frac{xy}{N/2} \mod 1$. This means the maximum error value is only one unit below the ideal approximation and do not depend on the input values magnitude.

2.2.5 Noise tolerance

Usually, conventional arithmetic units operate m-bit numbers and the SC unipolar or bipolar counterpart would need bitstream lengths of $N = 2^m$ to achieve a similar precision on the output. Imagine an adverse environment in which our digital hardware suffers from bit flips. Let p_f be flipping probability of a 1 becoming 0 or viceversa within a clock period, which is considered equally likely for illustration purposes. Under these conditions, there would be an average of p_fm bit flips per number in the conventional binary weighted case and p_f2^m in the SC case. A bit flip in a binary weighted number could represent an error that ranges from 1 to 2^{m-1} depending on whether the bit flip happened on the less or most significative bit. However, since bitstreams are longer compared to the compact weighted representation, the maximum (and minumum) error per bit flip is always 1 and might be compensated by a second bit flip.

The main problem with TC's noise robustness relies on the standard deviation of the absolute error due to bit flips. Notice for every TC bit flip, there would be approximately m bit flips in a SC bitstream if p_f is identical for both cases. However, bitstream bit flips could be partially compensated with each other, e.g. if a high state is flipped and then a low state is flipped the impact to the number representation is effectively zero. In contrast, in the TC case, compensations are much less likely. In addition, this refers to the impact in single number representations, but the errors might be propagated through the whole TC or SC network. In this context, P. Li and D. J. Lilja demonstrated the superiority of SC over TC experimentally for a simple image segmentation algorithm [41].

Nevertheless, this advantage is only optimally exploited avoiding the interaction with external memory and controllers and receiving data to be processed directly from sensors.

2.2.6 Relative computational cost

The comparison between TC and SC for specific applications in terms of computational cost might depend on several factors. One of them is the number and type of operations needed for a specific application, e.g. low precision SC multiplications are much cheaper, but additions are not as efficient as the TC equivalent. So that we expect SC to be theoretically superior executing algorithms involving low precision matrix multiplications, convolutions or L2 distances, as well as nonlinear or piecewise linear activation functions. Another factor to consider is the current state of microelectronics technology. In order to account for these two factors, it would be reasonable to approximate the (static) energy cost S as something proportional to the area A and number of evaluations N needed:

$$S \propto AN \tag{2.32}$$

While in the case of TC, a parallel operation would be performed with one evaluation ($N = 1$), in the SC case it would need $Q = 2^m$ using an APC, being m the equivalent TC bit width if the operation is performed in the time domain. The impact of having a different number of evaluations is described by (2.33) for both cases.

$$S_{TC} \propto A_{TC} \qquad S_{SC}^{APC} \propto A_{SC}^{APC} 2^m \tag{2.33}$$

A very common ML operation is multiply-and-accumulate (MAC). In the case of TC, a MAC block is composed by a multiplier, adder and register. For this reason, the mean MAC area is proportional to the number of FA needed, which is ultimately related to the number of transistors needed. Assuming the multiplier, adder and accumulation need m^2 FA, $10m$ FA and $10m$ registers[8] respectively, for a maximum accumulation of $2^{10m} - 1$, the approximate cost would be:

$$S_{TC} \propto 36m^2 + 180m \tag{2.34}$$

which assumes a very optimistic case of 6 transistors per FA and 12 transistors per DFF [58]. The corresponding transistor counts are listed in Table 2.2.

In contrast, the equivalent SC MAC average size would be related to the number of transistors needed for a single multiplication plus the average number of transistors needed for every APC input. The (inverted) multiplication could be e.g. a single XOR logic gate composed by only 2 transistors. However, the APC is more tricky, it is composed by a PC and an accumulator. In the case of the 15-inputs APC, the PC would be composed by 11 FAs and the accumulator would be composed $m + \log_2(16)$ DFFs and the same number of FAs. Knowing that a DFF could be implemented with 12 transistors, the average APC cost would be:

$$S_{SC}^{APC} \propto (9.73 + 9.53m) \cdot 2^m \tag{2.35}$$

[8]Notice $10m$-bit adder and register might be an arbitrary choice. It could be both lower or higher depending on the specific application.

which also accounts the $8m$ transistors per comparator for signal conversion (that is m XOR and m FA). Notice this approximate calculation is similar for a MUX and simple binary counter instead of an APC. The cost averaged for every 15 input pairs (i.e. 15 MUX inputs) would be:

$$S_{SC}^{MUX} \propto (8 + 8m) \cdot 2^{m+1} \tag{2.36}$$

which accounts for 5 4-input MUX with 18-transistors per MUX [58] divided by 15 to get the average transistor count. The transistor count for the binary counter is not taken into account because the output of the MUX is already stochastic. The SC APC and MUX transistor counts are summarized in Table 2.3.

Table 2.2: TC CMOS MAC transistor count.

TC element	**transistor count**
$m \times m$-bit multiplier	$36m^2$
$10m$-bit adder	$60m$
$10m$ registers	$120m$
Total	$36m^2 + 180m$

Table 2.3: Bipolar SC CMOS MAC transistor count.

SC-APC element	**mean transistor count**	**SC-MUX element**	**mean transistor count**
comparator	$8m$	comparator	$8m$
multiplier	2	multiplier	2
APC	$7.73 + 1.53m$	MUX	6
Total	$9.73 + 9.53m$	**Total**	$8 + 8m$

The relation between SC and TC computational costs is represented in Fig. 2.16. The blue and orange lines represent the APC and MUX cases respectively. The APC implementation results clearly superior in all cases. As regards the comparison between the SC APC approach and TC, there exists a limit in which SC outperforms the TC approach in terms of computational cost. Since the relative computational cost has been defined as the quocient between S_{TC} and S_{SC}, TC is more efficient if this quocient is less than 1, and less efficient otherwise. Therefore, using the dotted black line in Fig. 2.16 as a reference, for $m \lesssim 5$ the SC approach would be a better choice in terms of computational cost for MAC operations.

Nevertheless, there are cases in which SC solutions with $m > 5$ might be more convenient than the equivalent TC, e.g.:

- Fully parallel TC systems require a lot of area compared to SC. For this reason TC implementations are mostly based on datapath and controller designs. So it is common that a fraction of available hardware resources remains active but is not used. In addition, parallel implementations require less controller overhead. Examples of this issue are instruction fetch and decode in a MCU or state transitions in FSMs.

- MAC hardware might be followed by an activation function, e.g. ReLU or sigmoid, or block reduction operations, e.g. max or average pooling. In this case the computational cost is more optimistic for SC than TC, but the exact difference in computational cost depends on the number of stacked SC stages as well as the hardware complexity and latency.

- If noise tolerance is a requirement, then one might choose to implement a SC system in front of a TC system.

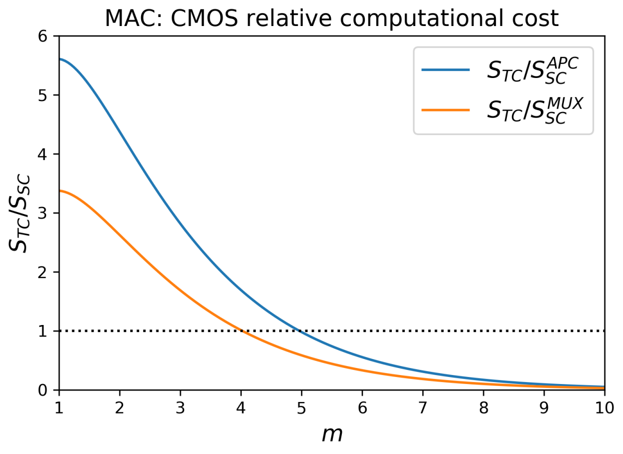


Figure 2.16: Ideal comparison between SC and TC in terms of computational cost based on CMOS transistor counts and number of needed iterations. Notice the dotted line represents the TC solution by itself. These results are only approximately valid for the specific case of the MAC operation.

2.3 Pattern Recognition

Pattern recognition refers to any process of signal identification. A signal might represent any object or abstract concept and is identified by predicting a label wich represents an arbitrary high level feature or decision.

The general pattern recognition pipeline is depicted in Fig.2.17. The sensor acquire raw data, which might be pre-processed. Then, feature extraction is used to create a higher level representation from pre-processed data. Finally, a decision is rendered based on some model.

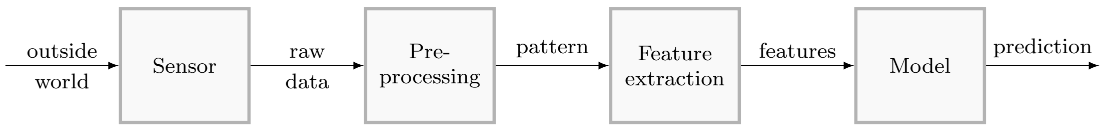


Figure 2.17: Generic pattern recognition pipeline.

Sensor and preprocessing stages might be very different depending on the type of input signals. For example, in the case of object recognition, input signals are most likely RGB pixel values from some CCD sensor and the pre-processing stage would be used to e.g. resize the image so that it matches the size expected by the feature extraction block. Instead, in the voice recognition case, raw data would be acquired by a microphone and the pre-processing state would be e.g. a noise removal algorithm.

Nowadays, some relevant pattern recognition applications are related to: weather forecasting [59], document recognition [16], voice recognition [60], cancer cell identification [61], fingerprint recognition [62], face detection [63] and recognition [64], object detection [65] or image segmentation [66] and captioning [67], among many others.

As regards the feature extraction and model stages, there are three commonly used strategies:

- Conventional ML: a pre-defined method is used to create feature maps, e.g. histogram of oriented gradients (HOG) for image data [68] or mel-frequency cepstrum (MFC) for audio data [69]. In these cases, after the feature extraction step, it is common to use a support vector machine (SVM) model [70], but other options like linear or logistic regression could also be good candidates for simpler tasks.

- End-to-end: feature extraction is part of the model. All parameters are learned from already seen data, e.g. backpropagation in Convolutional Neural Networks (CNNs). This approach requires more training data than the previous one if trained from scratch [71].

- Hybrid: as in the first case, a pre-defined method is used to create feature maps. A common example is audio classification with 2-dimensional CNNs, which typically require the mel spectrogram[9] as input data [72]. This approach might require less training data than the previous one.

2.3.1 Learning

There are several methods to accomplish learning in ML systems, as: Gradient-based learning, Moore-Penrose pseudoinverse and the use of Heuristic learning rules.

- Gradient-based learning: Nowadays, gradient-based learning is the most popular optimization strategy for NNs and might be applied to supervised, unsupervised and reinforcement learning [73]. The goal is to maximize/minimize a function depending on some parameters to be optimized. It is accomplished through an iterative algorithm, which might depend on some hyperparameters in order to converge to a local optimum. The main idea behind this approach is to update function parameters until convergence.

  Since there are a lot of possible gradient-based optimization algorithms, we summarized them in Fig. 2.18[10]. First, depending on whether the first or second derivative, an optimization method is first or second order, respectively.

  In the case of first order methods, the simplest method is known as gradient descent and only the first derivative is considered to carry out the parameter update. However, there are variations of this method which speedup convergence. Moreover, if all available data is used to perform every iteration it is said to be a full-batch method. On the contrary, if only a portion of the data is used, then it is a mini-batch method.

  In general, second-order algorithms use quasi-Newton methods since the Newton method would require computing the Hessian. The Hessian is too expensive to be computed at each iteration. This issue can be solved using a quasi-Newton

[9]The mel or mel-scaled spectrogram is just a part of the entire MFC pipeline, it does not involve the discrete cosine transform.

[10]Notice there also exist zero-order optimization methods, which have not been discussed since they are not gradient-based and do not provide relevant information to the reader regarding the work embodied in this document

approach, which approximates the Hessian to carry out the second order update faster, e.g. BFGS [74].

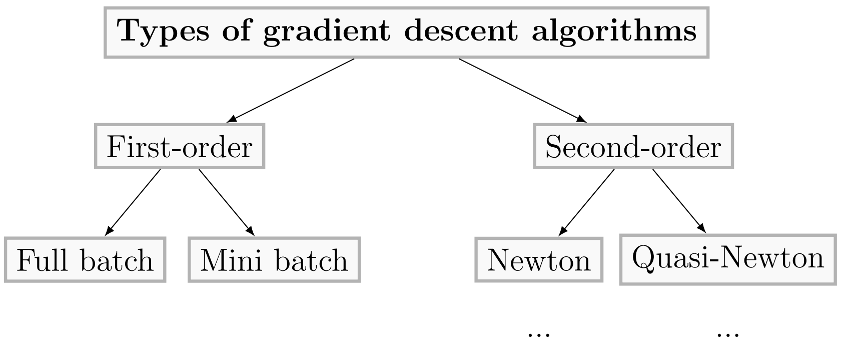


Figure 2.18: Common types of optimization algorithms based on gradient descent.

- Moore-Penrose pseudoinverse: The Moore-Penrose pseudoinverse is a closed-form solution to the multivariate least squares optimization problem.
- Heuristic learning rules: The heuristic learning approach is based on an *educated guess* of how the system should evolve. Typically, the solution is not guaranteed to be optimal but is sufficient to obtain a short-term goal or approximation to some problems. In some cases, heuristic methods converge much faster to a solution [75].

There exist a wide variety of ML models and some of them have been enumerated in Fig. 2.19. The ones related to our research are highlighted.

All ML models are based on a hypothesis[11] h parametrized by a set of parameters θ that maps some input dataset $\boldsymbol{X}$ to an approximation of the desired outputs $\hat{\boldsymbol{Y}}$, i.e.

$$\hat{\boldsymbol{Y}} = h_\theta\left(\boldsymbol{X}\right) \tag{2.37}$$

This hypothesis might be different between different methods, but it might also change how the parameterization θ is obtained depending on the training algorithm. In general, input and output data are represented by tensors. The notation used from here on assumes the first dimension of these input and output tensors is related to each dataset sample, while the rest of dimensions represent actual spatial or spatio-temporal dimensions for unstructured data. In this case there is not a pre-defined format or organization since individual values are meaningless. In contrast, structured data refers to relational databases in which data fits in fixed fields and columns. In this case individual values have a human-understandable meaning. Therefore, if input and/or output data are structured, there is a single spatial dimension and tensors in (2.37) can be represented by matrices in which each row represents a data sample and columns represent features.

2.3.2 Linear models

Linear models are the simplest models, the output of which is just a linear transformation of the input. That is:

$$\hat{\boldsymbol{Y}} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{B} \tag{2.38}$$

[11]The meaning of *hypothesis* in the ML literature is distinct (but related) to the same word in statistics (i.e. statistical hypothesis).

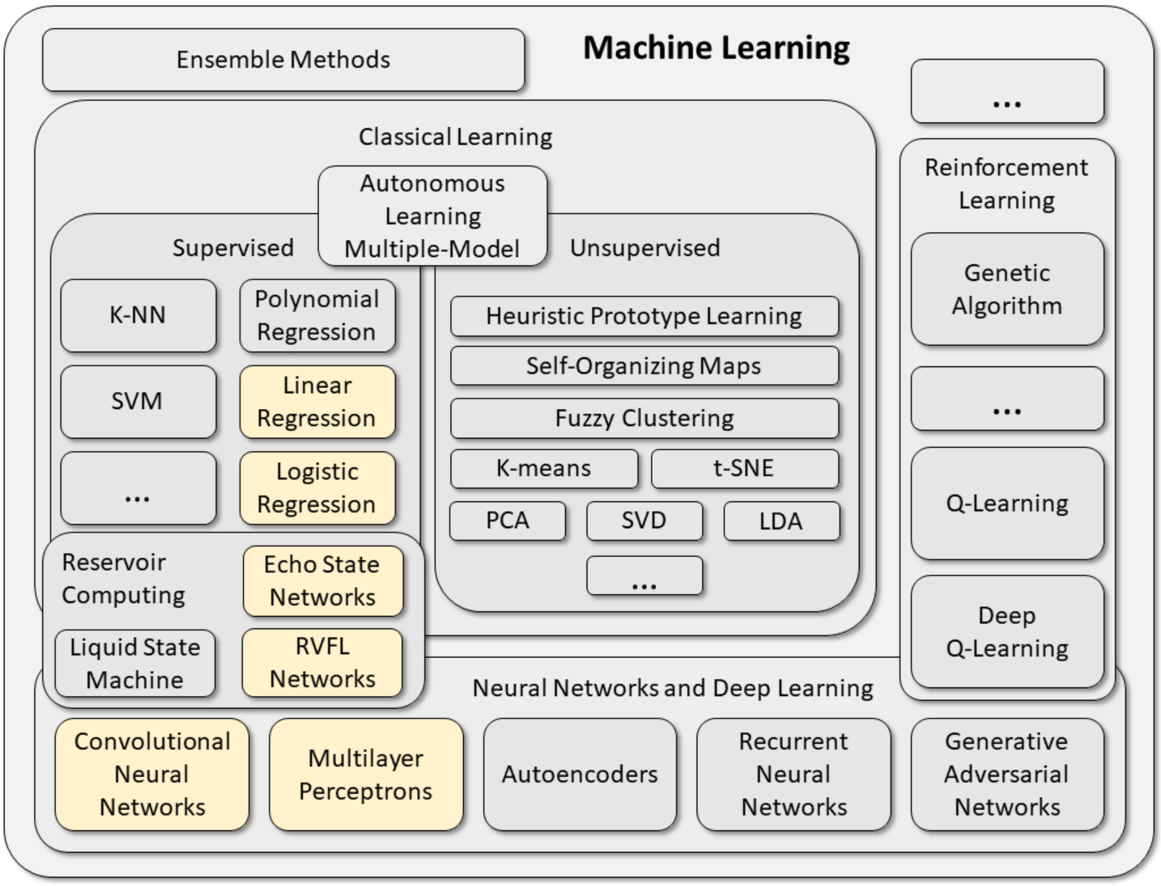


Figure 2.19: A map of ML methods. The ones highlighted in yellow have been needed for training or implemented in hardware.

where $\boldsymbol{W}$ is the weight matrix and $\boldsymbol{B}$ is the broadcasted bias vector $\boldsymbol{b}$. So that $\boldsymbol{W}$ and $\boldsymbol{b}$ are the parameters of the linear model. Notice this notation is equivalent to $\hat{\boldsymbol{Y}} = \boldsymbol{X}\boldsymbol{W}$ if $\boldsymbol{X}$ contains an additional column of ones. In this case $\boldsymbol{W}$ has an additional bias row.

Here we review two different linear models: linear and softmax regressions. Each of these can be used to solve regression and classification tasks, respectively. Linear and softmax regressions are convex optimization methods, i.e. are based on the minimization of a cost function with a single global minimum (convex function). If instead the method is based on the minimization of a cost function with multiple local minima (non-convex function), then it is referred to as a non-convex optmization method. Fig. 2.20 illustrates the difference between convex and non-convex functions for an ideal 1-dimensional cost function with the global minimum highlighted in red.

2.3.2.1 Linear regression

Linear regression is a statistical model that approximates linear relationships between input and target data via the minimization of a convex cost function. It is common to use different names for the same approach depending on the input (independent variables) and output (dependent variables) dimensionality. If the input and output are scalar values, then it is referred to as simple linear regression. If the input has more than one dimension but the output is a scalar, then it is referred to as multiple linear regression. Finally, in the general case, if both input and output are vectors, then it is referred to as multivariate linear regression. In the text we use the term

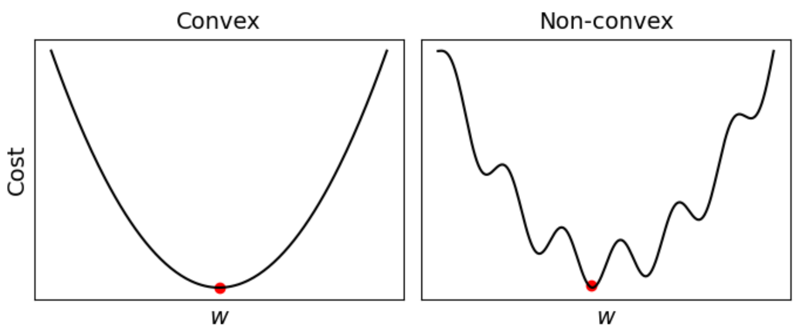


Figure 2.20: Convex vs non-convex 1-dimensional cost function. The global minimum is highlighted in red.

linear regression in its most general definition i.e. multivariate linear regression. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be row vectors denoting an arbitrary data sample and the corresponding target, respectively. Then, the loss function is the square error, given by (2.39).

$$L\left(\boldsymbol{x}\boldsymbol{W}, \boldsymbol{y}\right) = \sum_i \left(\boldsymbol{x}\boldsymbol{W}_{:,i} - y_i\right)^2 = \left(\boldsymbol{x}\boldsymbol{W} - \boldsymbol{y}\right)\left(\boldsymbol{x}\boldsymbol{W} - \boldsymbol{y}\right)^\intercal \tag{2.39}$$

which can be derived from the maximum likelihood estimation (MLE) method assuming the errors $\hat{\boldsymbol{Y}} - \boldsymbol{Y}$ are normally distributed. Since the goal is to minimize the square error of the whole dataset, then the cost function is defined as:

$$\begin{aligned} J\left(\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{Y}\right) &= \frac{1}{2m}\sum_i L\left(\boldsymbol{X}_{i,:}\boldsymbol{W}, \boldsymbol{Y}_{i,:}\right) + \overbrace{\frac{\lambda}{2m}\sum_{i,j} W_{i,j}^2}^{\text{L2 regularization}} \\ &= \frac{1}{2m}\|\boldsymbol{X}\boldsymbol{W} - \boldsymbol{Y}\|_F^2 + \frac{\lambda}{2m}\|\boldsymbol{W}\|_F^2 \end{aligned} \tag{2.40}$$

where m is the number of training samples and the L2 regularization term is optional and penalizes large weight values to reduce overfitting. Notice increasing λ increases regularization too. Therefore, the gradient of the cost is:

$$\nabla_{\boldsymbol{W}} J\left(\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{Y}\right) = \frac{1}{m}\left(\boldsymbol{X}^\intercal\boldsymbol{X}\boldsymbol{W} - \boldsymbol{X}^\intercal\boldsymbol{Y} + \lambda\boldsymbol{W}\right) \tag{2.41}$$

At this point, there are two possible methods to obtain the model parameters. An option is to solve (2.41) using normal equations by setting the gradient equal to zero so that:

$$\boldsymbol{W} = \left(\boldsymbol{X}^\intercal\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^\intercal\boldsymbol{Y} \tag{2.42}$$

Notice it is a convex optimization method because the term $\left(\boldsymbol{X}^\intercal\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}$ has a unique solution. It is also common in the literature to use the Moore-Penrose pseudoinverse $\boldsymbol{X}^+$ of $\boldsymbol{X}$, given by (2.43)[12], so that the weight matrix would be $\boldsymbol{W} = \boldsymbol{X}^+\boldsymbol{Y}$ for $\lambda = 0$.

$$\boldsymbol{X}^+ = \left(\boldsymbol{X}^\intercal\boldsymbol{X}\right)^{-1}\boldsymbol{X}^\intercal \tag{2.43}$$

Another option is to minimize the cost (2.41) via (first order) gradient descent using either full batch or mini-batch methods. The simplest iterative process is described by

[12]Here we assume $\boldsymbol{X}$ is a real matrix. If $\boldsymbol{X}$ is a complex matrix, then transpose operations would be replaced by conjugate transpose operations.

Algorithm 1 for the (2.41) cost gradient. Where α the learning rate, which might be decreased over iterations.

Algorithm 1: Simple full batch gradient descent.

```
Input: Learning rate and initialized weight matrix
Result: Optimized weight matrix
repeat
    $\boldsymbol{W} := \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J\left(\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{Y}\right)$;
until convergence;
```

There are also more effective update rules for first order gradient descent algorithms, such as momentum [76] or Adam [77], which have been included in Appendix A.2 and A.3 respectively. It is also possible to find the optimal parametrization using quasi-Newton methods, such as L-BFGS [78].

2.3.2.2 Logistic and Softmax regression

Logistic regression is a statistical model that approximates linear relationships between input and target data via the minimization of a convex cost function. However, its interpretation is fundamentally different. In the case of linear regression the goal is to approximate a continuous variable while in this case the goal is to model a binary categorical output, i.e. label "0" or "1". Therefore, a sigmoid or logistic function is applied to the linear output, so that the it is interpreted as the probability of an input vector $\boldsymbol{X}_{i,:}$ belonging to the class with label "1", i.e.

$$\hat{\boldsymbol{y}} = \sigma\left(\boldsymbol{X}\boldsymbol{w} + \boldsymbol{b}\right) \tag{2.44}$$

where σ is the sigmoid function (2.45) and the effect of the bias could be included in the weight matrix by adding a column of ones to the input vector, so we will use $\hat{\boldsymbol{y}} = \sigma\left(\boldsymbol{X}\boldsymbol{w}\right)$ to simplify notation.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.45}$$

The logistic regression loss function for a single sample $\boldsymbol{x}$ and target y, derived from the MLE, assuming the dependent variable follows a binomial distribution is given by the binary cross-entropy (2.46)[13] [79].

$$L(\sigma\left(\boldsymbol{x}\boldsymbol{w}\right), \boldsymbol{y}) = -y \log\left(\sigma\left(\boldsymbol{x}\boldsymbol{w}\right)\right) + (1-y)\log\left(1 - \sigma\left(\boldsymbol{x}\boldsymbol{w}\right)\right) \tag{2.46}$$

Therefore, the cost function with L2 regularization is:

$$J\left(\boldsymbol{X}, \boldsymbol{w}, \boldsymbol{y}\right) = \frac{1}{m}\sum_{i}\left(-y_i \log\left(\sigma(\boldsymbol{X}_{i,:}\boldsymbol{w})\right) + (1-y_i)\log\left(1 - \sigma\left(\boldsymbol{X}_{i,:}\boldsymbol{w}\right)\right)\right) + \frac{\lambda}{2m}\|\boldsymbol{w}\|^2 \tag{2.47}$$

Since the derivative of the sigmoid function is $\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$, then the gradient of the cost is given by (2.48).

$$\nabla_{\boldsymbol{w}} J\left(\boldsymbol{X}, \boldsymbol{w}, \boldsymbol{y}\right) = \frac{1}{m}\boldsymbol{X}^{\intercal}\left(\sigma(\boldsymbol{X}\boldsymbol{w}) - \boldsymbol{y}\right) + \frac{\lambda}{m}\boldsymbol{w} \tag{2.48}$$

[13] Cross-entropy is also known as Kullback-Leibler distance or relative entropy.

Moreover, logistic regression generalization to multiple classes is known as softmax regression and the derivation is very similar. Now the model is:

$$\hat{\boldsymbol{O}} = s(\boldsymbol{X}\boldsymbol{W}) \tag{2.49}$$

where the nonlinear function (2.45) converting logits to probabilities would be generalized by (2.50).

$$s(\boldsymbol{Z}) = \frac{e^{\boldsymbol{Z}}}{\boldsymbol{1}^\intercal e^{\boldsymbol{Z}}} \tag{2.50}$$

The number of rows of $\boldsymbol{Z}$ is equal to the number of possible outputs (different classes) and different rows represent different dataset samples.

Assuming mutually exclusive classes and targets following a multinomial distribution, MLE yields the loss function (2.51) for a single sample $\boldsymbol{x}$ and the corresponding one-hot target $\boldsymbol{o}$, which is the general form of the discrete cross-entropy [79].

$$L(s(\boldsymbol{x}\boldsymbol{W}), \boldsymbol{o}) = -\boldsymbol{o} \log s(\boldsymbol{x}\boldsymbol{W})^\intercal \tag{2.51}$$

so that the cost function with L2 regularization is:

$$J(\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{O}) = -\frac{1}{m}\boldsymbol{O} \log s\left(\boldsymbol{X}\boldsymbol{W}\right)^\intercal + \frac{\lambda}{2m}\|\boldsymbol{W}\|_F^2 \tag{2.52}$$

Since the derivative of (2.50) is $\nabla_{\boldsymbol{Z}} s(\boldsymbol{Z}) = s(\boldsymbol{Z})(\boldsymbol{I} - s(\boldsymbol{Z}))$, then the cost function gradient is:

$$\nabla_{\boldsymbol{W}} J(\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{O}) = \frac{1}{m}\left(\boldsymbol{X}^\intercal \left(s(\boldsymbol{X}\boldsymbol{W}) - \boldsymbol{O}\right)\right) + \frac{\lambda}{m}\boldsymbol{W} \tag{2.53}$$

Notice the logistic regression expression is recovered for the two-class case. Since it is not possible to express the optimal weights as an analytical function of $\boldsymbol{X}$ and $\boldsymbol{W}$, logistic and softmax regressions cannot be solved via normal equations. So, the optimal weights are obtained by first or second order gradient descent methods. So the simplest full batch method would be Algorithm 1. So that after training it converges to the optimal because the cost function is convex [80].

It is worth clarifying some works use linear regression for classification tasks [81], [82], but the least squares loss (2.40) is not optimal for discrete output values. However, it is possible to apply the linear hypothesis to a classification task by defining a threshold y_{th}, i.e.

$$\hat{y}_i = \begin{cases} 1 & h_\theta(\boldsymbol{X}_{i,:}) > y_{th} \\ 0 & h_\theta(\boldsymbol{X}_{i,:}) \leq y_{th} \end{cases} \tag{2.54}$$

where the threshold value y_{th} would be equal to 0.5 for logistic regression because the model represents a probability, which is not true for linear regression, so that y_{th} needs to be tuned to reach optimum results.

In the linear regression case the threshold value y_{th} needs to be tuned to reach optimum results, meanwhile for logistic regresion y_{th} it is assumed to be 0.5. A 1-dimensional binary class example is illustrated in Fig. 2.21, which assumes y_{th} for both cases. Notice samples between the vertical dashed lines would be misclassified by the linear regression method.

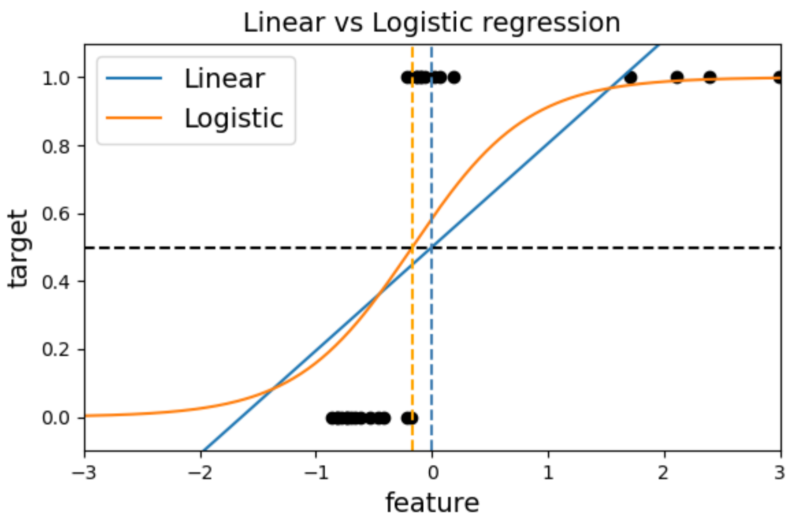


Figure 2.21: Example 1-dimensional comparison between linear and logistic regression results. Blue and orange dashed vertical lines are the decision boundaries for linear and logistic regression respectively.

2.3.3 Learning in Artificial Neural Networks

ANNs are computational models partially inspired by the structure and behavior of the brain. Such inspiration or analogy with real neural systems comes from the fact that brains are composed by interconnected excitable units that receive, compute and transmit information. In this context, there exist different classification of ANN types depending on the neuron architecture [83] or network connectivity [2], which is also applicable to DL models [73].

Real neurons communicate with each other through spikes. These spikes are abrupt changes in the membrane potential called action potentials, which travel from each neuron terminal buttons to the dendrites of the neurons connected to those buttons. The dendrites extend from the soma, which collects the contribution of all excitatory and inhibitory incoming spikes to the membrane potential and generates an action potential every time some threshold voltage is reached. Finally, the action potential wave is transmitted from the soma to the terminal buttons though the axon.

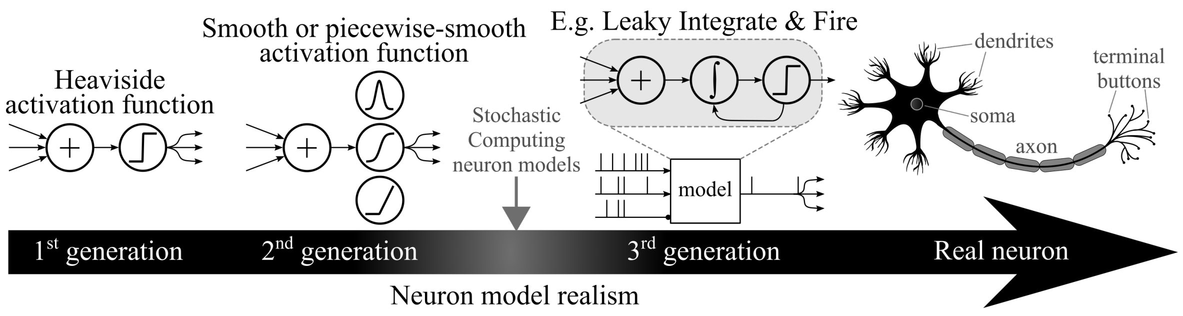

Heaviside activation function
Smooth or piecewise-smooth activation function
Stochastic Computing neuron models
E.g. Leaky Integrate & Fire
model
dendrites
soma
axon
terminal buttons
1st generation
2nd generation
3rd generation
Real neuron
Neuron model realism

Figure 2.22: Generations or types of artificial neuron models and their similarity to a biological neuron.

Generally, the behavior of artificial neurons used in AI applications is only partially inspired on the biological counterpart, or at least this has been the trend observed so far [73].

Three neuron model generations are depicted in Fig. 2.22 depending on how similar the model is compared to a biological neuron:

- **1st generation**
  Assume average rate coding for incoming spikes, so that inputs are represented by static quantities. The role of the soma is replaced by a weighted sum of these inputs, followed by a Heaviside activation function. An ANN composed by these neurons is capable of modelling any boolean function.

- **2nd generation**
  As for 1st generation neurons, inputs are represented by static quantities. In this case the role of the soma is replaced by a weighted sum of the inputs followed by a smooth or piecewise-smooth activation function. In practice, multiple activation functions can be used, e.g. radial basis function (RBF), sigmoid or rectifiers. An ANN composed by these neurons is capable of modelling any continuous function.

- **3rd generation**
  There is a wide variety of 3rd generation models and all of them assume input and output spike dynamics. Since the amplitude of an action potential is independent of the amount of current that caused the event [84], i.e. it does not carry relevant information in the amplitude but in the spike frequency or even specific firing times, so the specific shape of the action potential is not taken into account for AI applications [85], [86]. Also, spiking-neuron models including biophysically meaningful measurable parameters are in general reserved for computational neuroscience studies [87], [88].

Nowadays 2nd generation neurons are the preferred AI choice due to three main reasons. First, since inputs and activations are represented by quantities, existing CPUs and GPUs can be used to perform the corresponding floating point operations at high speeds. Second, the derivative of the activation functions is not zero and exists in (almost) the entire range of real numbers, allowing a relatively easy implementation of error back-propagation algorithms [76], which is not true for the Heaviside activation utilized in 1st generation neurons and is much more intricate in the case of 3rd generation neurons [89]. Third, in contrast to 1st generation neurons, a sufficiently large ANN is capable of modelling any continuous function no matter how complicated it is.

Regarding the connectivity of ANNs, it is common to distinguish two types of networks: feedforward and recurrent. A feedforward NN (FFNN) is a directed graph with no internal connectivity loops. An example of FFNN is depicted in Fig. 2.23 in which information flows from the input layer to the output layer with no feedback loops, so that it represents a static function. In contrast, recurrent neural networks (RNN) are represented by directed graphs in which (self) feedback loops are allowed. An RNN represents a discrete dynamical system in which the inputs and outputs can change every time step, so that present outputs depend on previous inputs, as depicted in Fig. 2.24.

In the following sections, two types of FFNN and its corresponding learning strategies are discussed. These two approaches are Prototype learning (section 2.3.3.1) and Backpropagation in Multilayer Perceptrons (section 2.3.3.2). In addition, the Reservoir Computing (RC) framework (section 2.3.3.3), which includes a simplified learning strategy for RNNs, is also discussed.

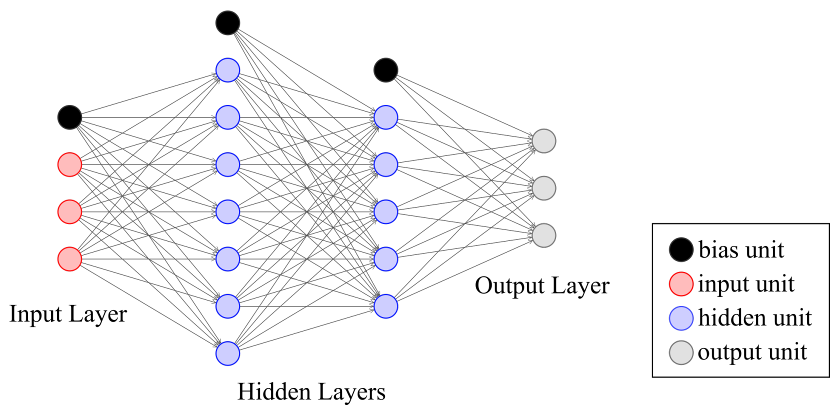


Figure 2.23: Example fully connected feedforward neural network architecture.

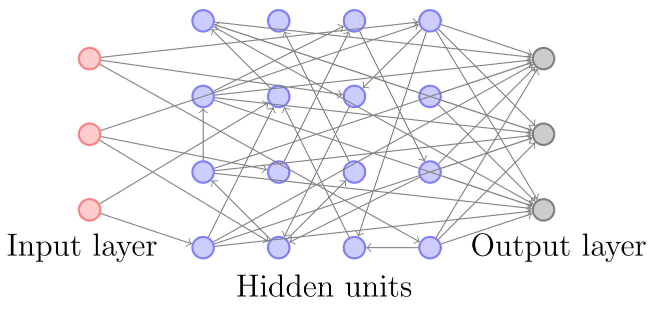


Figure 2.24: Example Recurrent neural network architecture.

2.3.3.1 Prototype learning

This kind of training strategy is typically applied to ANNs with a single hidden layer in which each neuron activation is a radial basis functions (RBFs) [90]. This type of network is often referred to as Radial Basis Function Neural Network (RBF-NN) and prototype learning is not the only training option. There is a particular type of RBF-NN in which input-to-hidden weights are initialized randomly and remain fixed, while hidden-to-output weights are trained using a linear model. RBF-NNs based on this simplified learning approach are known as Random Vector Functional Link Neural Networks (RVFL-NN)[14] [94], [95]. This concept can be extended to multiple hidden layers and any kind of activation function. Notice the RVFL-NN approach is similar to the Echo State Network (ESN) concept (see section 2.3.3.3), except for the fact that an ESN is an RNN architecture.

Nevertheless, single hidden layer RVFL-NNs are not necessarily the best option for neither accuracy nor explainability, which is the main strength of prototype learning approaches. Prototype learning approaches adjust input-to-hidden parameters, resulting in accuracy improvements compared to random initialization. Once input-to-hidden parameters are learned, these parameters tend to represent a set of *meaningful* templates of the input data, referred to as prototypes. Suppose a single hidden layer RBF-NN with already learned input-to-hidden weights $\boldsymbol{W}^{[1]}$. These parameters are

[14]As a curiosity, the reader might find in the literature some research papers citing a controversial paper by G.B. Huang, who introduced the term Extreme Learning Machine (ELM) [91], an idea similar to RVFL-NN published ten years later without giving proper credit to the original work. Since it is not our intention to judge the intentionality of this fact or discuss whether there are significative differences between ELMs and RVFL-NNs or not, we invite the interested reader to investigate such controversy [92], [93].

prototypes because after training tend to represent templates that are similar to input data for a sufficiently large number of hidden units. Each row $W^{[1]}_{i,:}$ represents a prototype associated to the i-th hidden node and each hidden node *resonates* for large enough similarities between input and prototype. A simple example is depicted in Fig. 2.25, in which the three hidden nodes detect upright triangles ($W^{[1]}_{:,0}$), squares ($W^{[1]}_{:,1}$) and circles ($W^{[1]}_{:,2}$), respectively.

Each hidden node represents an RBF activation function g, which depends on a distance (e.g. Euclidean distance) between the input and each prototype. The i-th hidden activation is given by $g\left(\left\|\boldsymbol{x} - W^{[1]}_{:,i}\right\|\right)$. Finally, these activations are processed by a linear model.

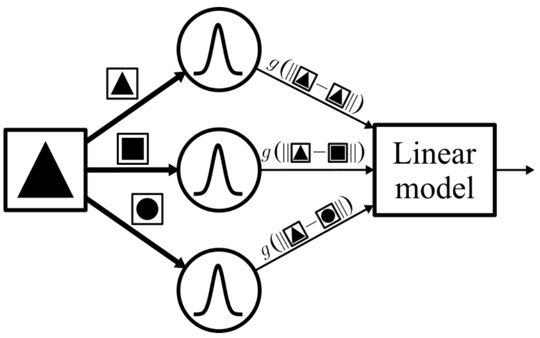


Figure 2.25: RBF-NN example. This is an idealized example and figure distortions and size are not taken into account by the model.

Suppose input $\boldsymbol{x}$ in Fig. 2.25 is an upright triangle exactly equal or very similar to the one assigned for the first prototype, so that the first hidden node would return a high value and the other two should be low, or at least lower.

The default choice for g is the Gaussian kernel (2.55) function. Where γ is an additional hyperparameter to be tuned, but γ^2 is usually related to the inverse of the number of hidden units, i.e. Gaussian function narrows as the number of hidden units increases to avoid multiple matches and thus increasing overfitting. The hyperparameter γ is assumed to be constant for different hidden units, which is not necessarily the case.

$$g(z) = e^{-\gamma z^2} \tag{2.55}$$

There exist multiple methods to obtain the prototypes. The straightforward approach is to randomly select input samples from the training set as prototypes, but performance can be substantially improved by applying unsupervised learning techniques such as K-means [96], [97], as reported by B. Scholkopf *et al.* [98]. K-means is the unsupervised pre-training approach utilized in our RBF-NN implementation (Section 5.1). The algorithm describes a clustering method that generates K prototypes or centroids which split data samples in K different groups. So that each input sample belongs to the the closest centroid[15].

The same class of input data might be referred to more than one centroid. Therefore, once prototypes are found the linear model is trained using labeled training data. This fact is illustrated in Fig. 2.26 for 2-dimensional input data.

[15]The closest centroid depends on the distance metric; however, assume Euclidean distance by default.

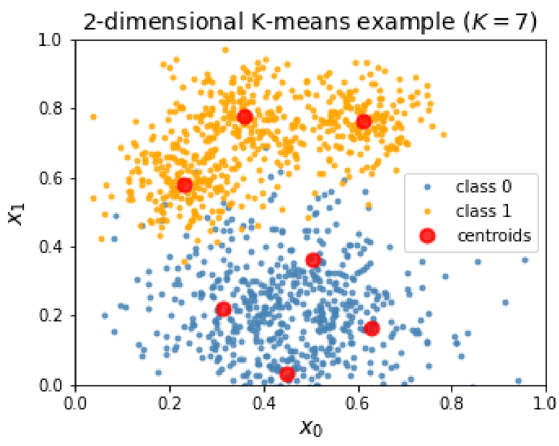


Figure 2.26: Example synthetic 2-dimensional data representing 2 classes and corresponding centroids obtained using K-means algorithm with $K = 7$.

Nevertheless, K-means is not the only option for clustering data or obtaining prototypes, e.g. Expectation-Maximization [99]. Additionally, in this work the learning process has been splitted in two different stages: unsupervised prototype learning and supervised learning based on fixed prototypes. However, it is possible to obtain even better accuracy using Autonomous Learning Multiple-Model First-order (ALMMo-1) systems, which do not require two separate training stages, and learn from streaming input data [100].

2.3.3.2 Backpropagation in Multilayer Perceptrons

A multilayer perceptron (MLP) is a network of interconnected neurons organized in layers and is the most common example of FFNNs. In this type of architecture, layers are connected sequentially, so that the i-th layer is only (fully) connected to the $(i+1)$-th layer, as illustrated in Fig. 2.23. These architectures are typically trained via a backpropagation algorithm, which extends the linear models ideas introduced in Section 2.3.2.

First, suppose the network has been already trained and parameters (i.e. weights and bias) do not need to be further modified. In this case the FFNN would be ready to be used for inference, which refers to the fact that parameters are now constant and the model has been deployed to solve certain task, see section 2.4.1 for further details about inference. Inference in FFNN is the process of manipulating input to obtain the output referred to as forward propagation. The MLP forward propagation is as follows, each layer connected to the next one represents an additional matrix multiplication between input and weight values followed by an elementwise activation function. Suppose multiple input samples organized in rows represented by the matrix $\boldsymbol{X}$ and results in activations $\boldsymbol{A}^{[L]}$ for layer L. The input layer activations are $\boldsymbol{A}^{[0]} = \boldsymbol{X}$, so that the MLP forward propagation is defined by (2.57), given definition (2.56).

$$\boldsymbol{Z}^{[L]} \equiv \boldsymbol{A}^{[L-1]}\boldsymbol{W}^{[L]} + \boldsymbol{B}^{[L]}, \tag{2.56}$$

$$\boldsymbol{A}^{[L]} = g^{[L]}\left(\boldsymbol{Z}^{[L]}\right), \quad L = 1, 2, \ldots L_f \tag{2.57}$$

where $g^{[L]}$ is the activation function for all units in layer L and L_f is the number of layers not including input layer. The forward propagation process is defined by the iteration of (2.57) from $L = 1$ to $L = L_f$.

Now, suppose the MLP has been initialized with random weights and it has to be trained to map input data $\boldsymbol{X}$ to output $\boldsymbol{O}$. Here is where backpropagation comes into play, performing a forward propagation and then, according to some cost function J, propagate the error back layer by layer while calculating the corresponding parameter updates. This process is conceptually the same as for linear or softmax regression, i.e. a parameter variation given the previous activation.

The backpropagation starts from the output layer, i.e. $L = L_f - 1$, and the goal is to first calculate the error associated to the output layer in order to propagate it layer by layer. This errors are denoted as $\delta\boldsymbol{A}^{[L]}$ since these represent the desired corrections for $\boldsymbol{A}^{[L]}$ based on some cost function J. Then, the correction for the output layer is:

$$\delta\boldsymbol{A}^{[L_f]} \equiv \nabla_{\boldsymbol{A}^{[L_f]}} J\left(\boldsymbol{A}^{[L_f]}, \boldsymbol{O}\right) \tag{2.58}$$

Once $\delta\boldsymbol{A}^{[L_f]}$ is known, the rest of the backward propagation is systematic. The resulting iterative relations in the backward propagation are as follows:

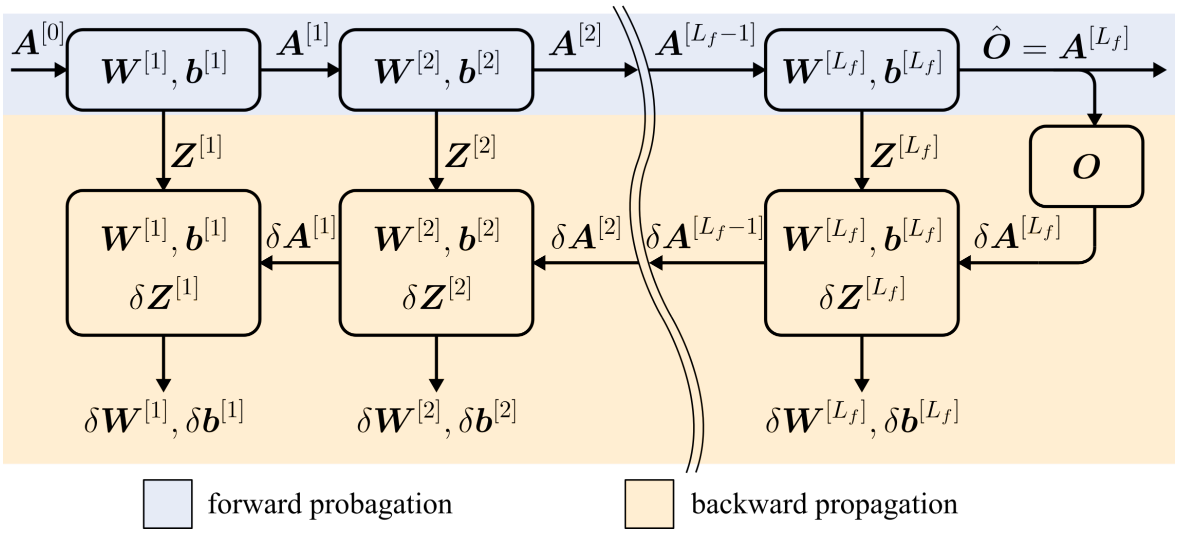
$$\delta\boldsymbol{Z}^{[L]} = g^{[L]'}\left(\boldsymbol{Z}^{[L]}\right)\delta\boldsymbol{A}^{[L]} \tag{2.59}$$

$$\delta\boldsymbol{W}^{[L]} = \frac{1}{m}\boldsymbol{A}^{[L-1]\intercal}\delta\boldsymbol{Z}^{[L]} \tag{2.60}$$

$$\delta\boldsymbol{b}^{[L]} = \frac{1}{m}\boldsymbol{1}^{\intercal}\delta\boldsymbol{Z}^{[L]} \tag{2.61}$$

$$\delta\boldsymbol{A}^{[L-1]} = \delta\boldsymbol{Z}^{[L]}\boldsymbol{W}^{[L]} \tag{2.62}$$

If the cost function J is not scaled with the number of samples m, then the weight variation is computed as in (2.53) layer by layer. The entire process scheme involving both forward and backward propagation has been shown in Fig. 2.27.



$\boldsymbol{A}^{[0]}$
$\boldsymbol{W}^{[1]}, \boldsymbol{b}^{[1]}$
$\boldsymbol{A}^{[1]}$
$\boldsymbol{W}^{[2]}, \boldsymbol{b}^{[2]}$
$\boldsymbol{A}^{[2]}$
$\boldsymbol{A}^{[L_f-1]}$
$\boldsymbol{W}^{[L_f]}, \boldsymbol{b}^{[L_f]}$
$\hat{\boldsymbol{O}} = \boldsymbol{A}^{[L_f]}$
$\boldsymbol{Z}^{[1]}$
$\boldsymbol{Z}^{[2]}$
$\boldsymbol{Z}^{[L_f]}$
$\boldsymbol{O}$
$\delta\boldsymbol{Z}^{[1]}$
$\delta\boldsymbol{A}^{[1]}$
$\delta\boldsymbol{Z}^{[2]}$
$\delta\boldsymbol{A}^{[2]}$
$\delta\boldsymbol{A}^{[L_f-1]}$
$\delta\boldsymbol{Z}^{[L_f]}$
$\delta\boldsymbol{A}^{[L_f]}$
$\delta\boldsymbol{W}^{[1]}, \delta\boldsymbol{b}^{[1]}$
$\delta\boldsymbol{W}^{[2]}, \delta\boldsymbol{b}^{[2]}$
$\delta\boldsymbol{W}^{[L_f]}, \delta\boldsymbol{b}^{[L_f]}$
forward probagation
backward propagation

Figure 2.27: Forward (blue) and backward (orange) propagation schemes for a MLP batch/mini-batch iteration.

Backpropagation methods involve computationally expensive matrix multiplication calculations and depending on the layer sizes and network deepness. However, it is the most widely extended approach to train ANNs, including MLPs, CNNs and RNNs because of its superior performance on supervised learning tasks with large datasets.

Nowadays researchers and engineers have many open source resources to create ANNs without worrying about the backpropagation implementation. The programmer would only specify the network architecture, so that the forward propagation (Fig. 2.27, blue) is completely defined, and the cost function, as well as the corresponding hyperparameters related to regularization strategies and iterative update methods like Adam. Once it is defined, the backward propagation (Fig. 2.27, orange) is also already inferred behind the scenes using automatic differentiation techniques [101], which is the approach implemented by most deep learning frameworks like PyTorch [102], [103] or Tensorflow [104].

2.3.3.3 Reservoir Computing

Reservoir Computing (RC) is a generalization of the concepts of Echo State Network (ESN) and Liquid State Machine (LSM), proposed by Jaeger [105] and Maass [106], respectively. In both cases, the authors propose the use of RNNs to solve tasks requiring certain memory and a supervised learning process, which is much simpler than the well-known backpropagation trough time [107]–[109].

RNNs consists of several input units, connected to the hidden units and these are connected between them and to the output layer[16], just like in Fig. 2.24. While in the most general case all model parameters are learned, Jaeger and Maass proposed to randomly initialize model weights and modify only those weights connected to the output units, as depicted in Fig. 2.28.

The main difference between Jaeger (ESN) and Maass (LSM) works lies in the fact that ESNs are composed by 2nd generation neurons and LSMs are composed by 3rd generation (spiking) neurons. Since they obtained similar conclusions independently, both authors are considered the fathers of RC.

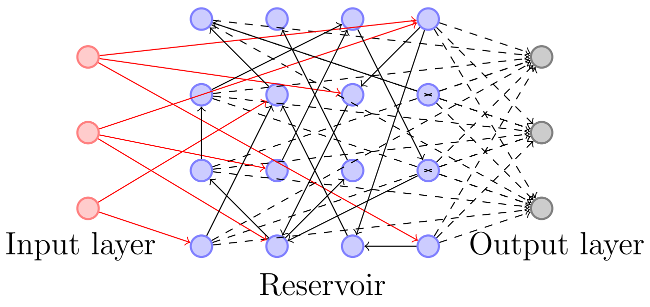


Figure 2.28: Echo State Network or Liquid State Machine scheme. Three blocks: Input layer, reservoir and the output layer compose the network. During the training process, solid line arrows correspond to weights are not modified and dashed line arrows correspond to trainable weights, i.e. only the reservoir-to-output weights are modified.

In the case of ESN and LSM, hidden nodes are referred to as *reservoir*. However, a reservoir can be any discrete or continous dynamical system. Fig. 2.29 shows a generic scheme in which a reservoir or dynamical system is perturbed by an (optionally encoded) input signal. Then, these perturbations change the reservoir state, which are fed to a readout model to make a prediction. Just like in the ESN and LSM cases, the only part in which parameters are optimized is this readout layer. In most cases the readout is a linear transformation. In addition, one could use a compressed representation of the reservoir state instead of feeding the whole reservoir state vector

[16]Input units could also be connected to the output ones and here we assume there is no feedback from the output to the reservoir.

to the readout model, e.g. Jin and Li applied principal component analysis (PCA) to compress a LSM reservoir state [110].

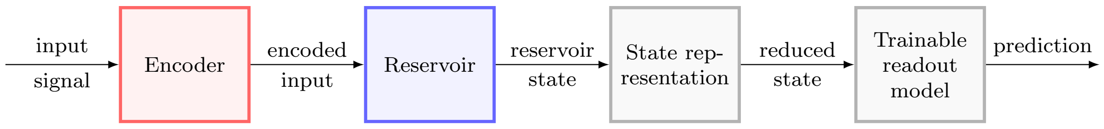


Figure 2.29: Reservoir Computing scheme.

Generally, an RC system has the following properties:

- **Approximation**
  Let $(\boldsymbol{x}, \boldsymbol{y})$ be the input and output data, respectively. It must be fulfilled that for very similar input samples $\boldsymbol{x}$ and $\boldsymbol{x} + \delta\boldsymbol{x}$, the responses are also very similar, i.e.
$$(\boldsymbol{x} \rightarrow \boldsymbol{y}) \Leftrightarrow (\boldsymbol{x} + \delta\boldsymbol{x} \rightarrow \boldsymbol{y} + \delta\boldsymbol{y}) \tag{2.63}$$

- **Separation**
  Even though the approximation property is met, the distance between reservoir states must be sufficiently large to distinguish between two similar input samples belonging to different classes or categories in a classification task.

- **Memory**
  Since recurrent networks present feedback between their nodes, the reservoir response to an input signal retains information from previous inputs, which might be reflected in the output layer. The reservoir stored information is progressively lost as the system evolves in time, i.e. the memory referred to an input received at certain time t_0 is lost as the difference $t - t_0$ grows.

These characteristics are common to other RNNs, which may exhibit even longer-term memory, e.g. long short-term memory (LSTM) blocks [111] or gated recurrent units (GRUs) [112]. The RC systems main advantage is the training simplicity compared to full backpropagation. Everything related to the input encoding, reservoir and (optionally) the reservoir state representation (see Fig. 2.29) are hyperparameters, which need to be explored via e.g. grid search, random search or heuristic rules [113]. The only parameters modified directly by training are those of the readout model.

In order to have a qualitative understanding on how the RC approach works, let the input data be M-dimensional and the (reduced) reservoir state be represented by an N-dimensional vector. If $N > M$, the input is being projected to a higher dimensional feature space which contains information from previous inputs. If the encoder, reservoir and state representation are such that the N features are linearly independent[17], then increasing N also increments the probability of the task being linearly adjustable or separable. In this case, it is always possible to fit the training data if N is sufficiently large. This is the reason why linear readout models, like linear or logistic/multinomial regression described in section 2.3.2, are the most common choice in the RC framework.

The main hyperparameter tuning approach in ESN or LSM networks does not account for any encoder nor state representation stages (Fig. 2.29). In this case the whole training process is as follows [105], [114]. First, an sparse and random input-to-reservoir weight matrix $\boldsymbol{W}^{in}$ and internal reservoir weight matrix $\boldsymbol{W}^{reservoir}$ are

[17]Notice this does not necessarily mean the reservoir performance is optimal.

initialized and left constant[18]. Second, the network is fed with input training data $\boldsymbol{U}$, obtaining the corresponding activation states $\boldsymbol{X}$. Third, the readout model is trained using either the generated reservoir activation states $\boldsymbol{X}$ or both input and activation states, denoted by $\boldsymbol{Z} = (\boldsymbol{U}|\boldsymbol{X})$. Finally, the network is tested on new data and its performance is evaluated using some metric, e.g. accuracy or mean square error (MSE), and the whole process is repeated for different initializations using e.g. grid search or random search strategies.

In general, the random initialization of weight matrices is restricted to some target spectral radius. So that each initialization is restricted to some pair of spectral radius $\rho(\boldsymbol{W}^{in})$ and $\rho(\boldsymbol{W}^{reservoir})$. The spectral radius is defined as the largest eigenvalue of the corresponding matrix, that is:

$$\rho(\boldsymbol{W}) = \max_i \|\lambda_i\| \tag{2.64}$$

where the eigenvalues $\lambda_i \in \mathbb{C}$ are the solutions for λ of (2.65).

$$\det(\boldsymbol{W} - \lambda \boldsymbol{I}) = 0 \tag{2.65}$$

Nevertheless, the random nature of weight matrices makes convenient to repeat the whole process for each spectral radius pair in order to obtain average performance metrics. Taking into account other hyperparameters related to these matrices is a common practice, e.g. sparsity in $\boldsymbol{W}^{in}$ and $\boldsymbol{W}^{reservoir}$, or the strength of reservoir self-feedback loops when considered separately (a.k.a. leaking rate [114]).

Moreover, different physical reservoirs have been proposed, as: including digital and analog circuits, nonlinear electronic networks, opto-electronic, optical or quantum [115]–[121], among others, see e.g. [122] or [123] for an up to date detailed list of physical implementations. As a curiosity, it is even possible to meet the RC conditions with just a bucket filled with water as reservoir [124]. If the reservoir is a physical or computational dynamical system, the hyperparameter search is not based on the weight matrices' spectral radius as in the case of ANNs, but on the design space exploration.

This thesis includes two RC FPGA implementations. Firstly, Section 4.1 presents a simplified ESN approach with a constrained ring-topology architecture applied to audio event detection (AED). Secondly, Section 4.2 presents an RC system based on a cellular automata (CA) adapted to time-independent input samples, and applied to a handwritten digit recognition benchmark.

2.4 Energy efficient inference hardware

2.4.1 Inference

After training a model to solve certain task, it is freezed, i.e. the model parameters are fixed to values providing the highest performance on the validation set. Therefore, it is possible to train a model with massive datasets on a GPU and deploy it to lower power or faster devices for inference purposes. So, the main reason to deploy a model with inference-only capabilities is to improve energy efficiency. Such advantage might be exploited either at servers or low power devices. Energy efficiency results in speedups

[18]It is, however, very common to exploit the unsupervised learning capabilities of spiking neurons in LSMs to tune the internal reservoir connectivity in an unsupervised manner.

with the same power at the server level and longer battery durations for low power devices.

The deployment of an ML system[19] consists on the obtention of a freezed model and its evaluation on test set. Such process is described in Fig. 2.30. First, the model parameters are optimized using e.g. gradient descent and validation data. This is usually repeated for different initializations and training hyperparameters, as well variations for different models. Then, an unseen set of samples is used for testing purposes and sometimes compared across different models. Under certain criteria, which is not necessarily accuracy, the best model is freezed (its architecture and learned parameters are stored). Until this step, everything is usually programmed in a general purpose computer, based on CPU and/or GPU. The next step is to verify the predicted test performance matches the inference device performance, which might be checked by direct device measurements. Finally, once the device has been successfully evaluated on the test set, it is ready to be used in real life applications.

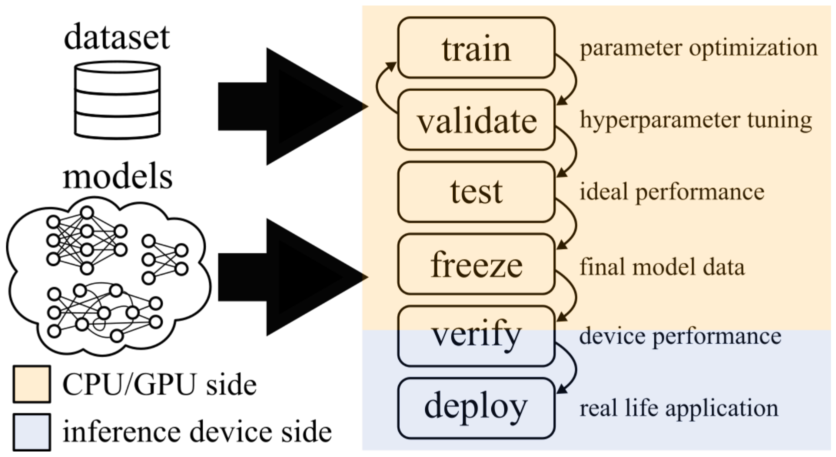


Figure 2.30: High level view of the ML deployment process.

Energy efficiency improvements might be reached without impacting arithmetic operations' precision using e.g. new technologies [125], computer architectures [126], [127] or application specific architectures [128], [129]. However, energy efficiency improvements can be achieved by enabling a tradeoff between simplicity and global performance. Simplicity is a wide concept and allows many options to simplify a model. The main idea is to simplify the model as much as possible with no or little impact on the model performance. Examples of simplification are: weight pruning, compression, quantization and approximate arithmetic [130]–[132], which might rely on non-deterministic computations [133].

This thesis was focused on digital technologies and FPGA in particular and our improvements rely on ANN model simplifications. In particular, our contribution includes parameter and operation quantization, as well as SC, which might introduce non-deterministic results under certain conditions. However, other model simplifications such as weight pruning and compression are out of the scope of this work, which are not necessarily incompatible with the proposed hardware implementations.

[19]ignoring a potential manual feature engineering work

2.4.2 Dedicated hardware

When it comes to improving energy efficiency by simplification, there exist multiple available options such as reducing parameter and operation precision, modifying computational elements... In particular, those platforms based on digital logic are put in the spotlight. Other approaches based on analog designs are feasible too and can achieve higher energy efficiency. Its development is generally slower and its hardware implementation is not easily scalable, partly due to industry's long-time focus on digital systems miniaturization, which was not possible to the same extent for analog circuitry.

Perhaps, this scenario may change for ML in the future in favor of analog chips or other emerging technologies such as quantum computing, making them more viable for the industry. On the other hand, general purpose CPUs and GPUs are designed to be precise and flexible. This fact makes these solutions less energy efficient than specific hardware solutions, which restrict the hardware topology, data paths and precision. Thus achieving better parallelism in the same area at the cost of losing the general purpose functionality. In order to handle these restrictions with today's technology, the most appropriate solution is to develop a custom ASIC. Usually, this is the best commercially available option for optimizing power, performance and area (PPA). So, in terms of PPA efficiency (i.e. energy efficiency and area), ASIC is the preferred choice.

Nevertheless, a custom ASIC cannot be modified or reconfigured to meet new specifications or correct any error. As an intermediate solution, FPGAs allow the designer to deploy reconfigurable digital systems. An FPGA is certainly not as flexible and easy to use as a GPU and even less than a CPU, but it is far less restrictive than an ASIC. This fact makes FPGAs optimal for rapid prototyping and testing of new designs, typically without requiring long coverage and timing simulations in addition to steps in the workflow for a specific technology (e.g. 22 nm) to finally achieve the tape-out. Although some workflow steps are common to FPGAs, it is mostly automated by design software tools because the FPGA architecture already is fixed. This flexibility over ASICs is not for free. FPGAs are typically restricted to the synchronous digital designs. Also, assuming the same technology, the same design in FPGA is less efficient in terms of PPA than the ASIC counterpart.

In summary, the main types of devices in decreasing order of flexibility and programming abstractions, and increasing order of PPA efficiency are as follows: CPU, GPU, FPGA and ASIC. Also, due to the restrictive ASICs' flexibility, FPGAs have been the choice for this thesis, which explores several design prototypes.

Following the topic at hand, Chapter 3 enumerates the generic steps to follow, from an inference model architecture to the final FPGA implementation.

Chapter 3

Methodology

The methodology described in this chapter refers to the set of methods related to the FPGA implementations, starting from an ML algorithm, which inference part is adapted to be implemented in hardware. This process involves many subprocesses, some of them are carried out using libraries and Electronic Design Automation (EDA) tools, while others are done using custom tools. All these steps are related to each other, indicated in Section 3.1 (General workflow). It should be noted that we include all processes related to the designs implemented in this thesis. Specific implementation details are discussed in the following sections. Section 3.2 introduces software and hardware resources utilized to accomplish the steps enumerated in the general workflow. Then, Section 3.3 describes the strategies followed to design, simulate and debug a system to be implemented in FPGA. Finally, section 3.4 enumerates a list of several hardware communication protocols utilized to evaluate the inference models based on a test set.

3.1 General workflow

A summary of the steps to build an inference ML model is summarized in Fig. 3.1.

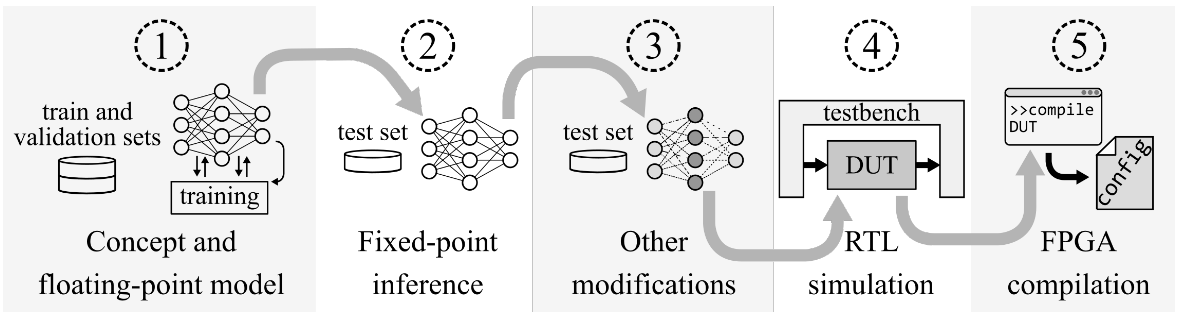


Figure 3.1: Design methodology workflow used in this thesis for hardware implementation.

The description of each step (1 to 5) is summarized below.

1. **Idea and floating-point model**
   Depending on the nature of the task one is aiming to implement, some models might be a better choice, e.g. RNN for text processing and timeseries prediction or CNN for image classification. On the other hand, there are tasks that can be properly solved with different models, e.g. tasks like sound or EEG classification, which might be solved using RNN, CNN or even a combination of both.

However, in this thesis one of the goals is to create innovative models that at the same time can be implemented in an FPGA. FPGA devices limit the model's parallelism as well as and the total number of inputs, outputs and parameters. The corresponding designs might incorporate benefits with respect to other approaches, either referred to its FPGA implementation or by a potential ASIC. Therefore, the level of parallelism and number of model parameters is limited by the target hardware architecture.

Moreover, depending on the potential accuracy degradation due to quantization (see step 2), the final result (accuracy) might be improved by taking into account the precision of the target hardware model. This process is often referred to as Quantization Aware Training (QAT). During training, it takes into account the fact that input, output, some arithmetic operations, intermediate results and parameters are typically low precision in hardware. As a result, the training process provides already quantized parameters corresponding to certain input and arithmetic operation precision. Nevertheless, quantization can be also done a posteriori at the expense of accuracy degradation.

2. **Fixed-point inference**
   If QAT is applied in step 1, the inference model does not need to be modified and this step simply refers to reporting a result (e.g. accuracy) based on a test set. Sometimes it is also convenient to store some intermediate results for debugging.

3. **Other modifications**
   If the architecture is not strictly based on conventional arithmetic, then computations do not match exactly those of the previous step (step 2). In this case other modifications before HDL simulation might be applied. For example, in our case some designs are based on SC, so that potential arithmetic errors need to be modeled. In this particular case, the SC inference process is simulated. As in step 2, a result is reported based on validation data and SC parameters are explored to minimize the differences between pure fixed-point inference and SC.

4. **RTL simulation**
   HDL code can describe FPGA and ASIC behaviour or any hardware architecture, which is used to synthesize our Design Under Test (DUT) and the corresponding (non-synthesizable) testbench. Then, a Register Transfer Level (RTL) functional simulation is performed, i.e. ideal, cycle accurate behaviour of the circuit is verified. This means that timing issues due to e.g. combinational delays or asynchronous behavior is not modeled at this stage.

5. **FPGA compilation**
   Some details or nomenclature might be slightly different when it comes to FPGA compilation from different vendors. In general, the workflow contains the following steps[1] (5.1 to 5.8).

   5.1 IP Generation

   5.2 Analysis & Synthesis

   5.3 Fitter (Place & Route)

[1]This list has been taken from the different compilation processes utilized in the Intel® Quartus® Prime software and other design suites like Vivado® are slightly different.

5.4 Fast Forward Timing Closure Recommendations

5.5 Timing Analysis

5.6 Power Analysis

5.7 Assembler

5.8 EDA Netlist Writer

It is also possible to include in-circuit debugging tools before compilation and analyze signals and registers once the configuration file has been downloaded, using e.g. SignalTap II Logic Analyzer or In-System Memory Content Editor tools available in Intel® Quartus® Prime. Finally, when the designer has verified the FPGA works as expected, in-circuit debug resources are removed. These debug resources are not needed anymore and the whole compilation process is repeated without them in order to obtain the final configuration file and download it to the FPGA.

3.2 Software and hardware

Different programming languages, frameworks and EDA softwares have been utilized depending on the target step in Fig 3.1. The floating-point model training, fixed-point inference and other modifications (steps 1, 2 and 3) are carried out using Python together with PyTorch or TensorFlow programming frameworks and common linear algebra libraries like NumPy. Once the inference model specifications are well established, the DUT VHDL code and the corresponding testbench is created and simulated (step 4) using ModelSim*-Intel® FPGA Edition software. Finally, the compilation processes (step 5) are carried our within the Intel® Quartus® Prime FPGA design software.

On the other hand, the list of Intel®/Altera® FPGAs utilized in this thesis is presented in Table 3.1.

Table 3.1: Utilized FPGA design kits and some specifications.

Design kit	FPGA	ALM count	HPS	PCIe
Terasic® DE10-Nano	Cyclone® V 5CSEBA6U23I7	41,500	✓	✗
Terasic® DE5-Net	Stratix® V 5SGXEA7N2F45C2	234,720	✗	✓
Gidel® Proc10A	Arria® 10 10AX115N2F45E1SG	427,200	✗	✓

3.3 Simulation and debug

Step 4 (RTL simulation) in Fig. 3.1 consists of synthesizing and accurately simulating the VHDL functional description. Therefore, this description is actually designed for a specific hardware architecture since there are infinitely many possible ways to process the step 3 simplified inference model. Roughly, the possible ways in which the inference algorithm operations can take place are: sequentially, fully parallel or a combination of both. The latter is the most reasonable in most cases. The main synthesizable design that implements the desired inference model behaviour is referred to as DUT.

It is defined so that input and output buses are utilized to control and send/receive data to/from memories and register maps. Then, to test the correctness of the DUT, it has to be externally excited and its outputs asserted against the expected ones or stored for a further analysis. This is where the testbench comes into play. In our case the testbench is a separate VHDL description which is in charge of reading predefined stimulus file(s) and writing signals and registers in output file(s). The output files can contain internal signals and registers recorded for visual inspection. Finally, output files are processed by a separate program to check its correctness.

Once FPGA compilation (step 5) is done, the resulting configuration file is loaded to the FPGA. After the configuration, the design is debugged by visualizing some signals with SignalTap II Logic Analyzer and verifying whether the behavior is as expected. The debugging process might require multiple iterations, i.e. modify, compile, configure, debug and repeat.

3.4 Communication interfaces

The final goal is to evaluate the test set on the FPGA and store results to calculate the desired performance metric, e.g. accuracy. A communication protocol between the FPGA and an external source is required. Depending on the selected FPGA development board the preferred communication interface and top level design might be different. This means our design hierarchy requires additional logic to enable communication between the data source and the proposed inference design.

In this thesis, we use the Avalon® Interface, which is the default choice in the Platform Designer (formerly Qsys) GUI, a Quartus® Prime tool that enables interconnection between different components such as CPU cores, Direct Memory Access (DMA), memories, peripherals, custom logic, etc. In fact, it is a tool for System-on-Chip (SoC) design and HDL generation. A relatively simple SoC example hierarchy is depicted in Fig. 3.2, including top level components (gray) and interconnect (blue). This figure may include other additional SoC components as streaming microphone, camera sensor, HDMI controllers and other soft cores. Most frequently used components are usually available as IP cores and there is no need to design each one from scratch. These IP cores can be included in the design with the Platform Designer tool and the designer can simply interconnect components in the GUI[2]. If a custom component has to be added to the design, the designer can define and include it. In this case the custom logic block depicted in Fig. 3.2 would contain the already simulated and debugged inference model HDL description, which is interconnected to the rest of components using the Platform Designer GUI. The tool automatically generates the top level design, which includes all components and the corresponding interconnect logic.

At this point, the whole system could be simulated using libraries incorporating Bus Functional Models (BFMs) provided by the design software. However, we skip this step and simply compile it, configure the FPGA and verify its functionality. In order to verify the functionality, extra interface software development is needed.

[2]Even though the Platform Designer tool is our preferred choice for Terasic® FPGA boards, the Gidel® ProcA10 has been set up using the Gidel® ProcWizzard application, which is similar to Platform Designer, but it takes into account the specific on-board memory resources and their PCIe drivers.

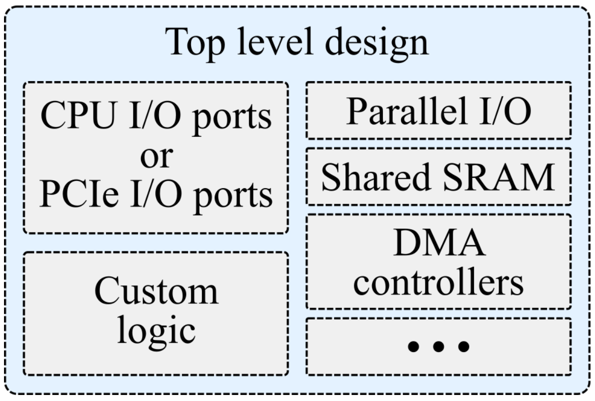


Figure 3.2: Simple SoC interconnect and example components. (Gray) IP and custom components. (Blue) Interconnect logic.

Going back to Table 3.1, notice the DE10-Nano is a SoC FPGA which contains HPS and FPGA integrated in the same die and the board to not have a PCIe interface. In contrast, the other two FPGA boards (DE5-Net and ProcA10) do not integrate a processor but incorporates PCIe ports to enable fast communication between a host PC and FPGA. In addition, the DE5-Net and ProcA10 have different PCIe drivers and corresponding C++ libraries. Therefore, different software interfaces are needed for each board. Once the communication between host and FPGA has been checked, the test set can be evaluated by our custom inference model.

Chapter 4

Fixed-Point Implementations

Multiplications are expensive operations in terms of area and energy efficiency compared to other arithmetic fixed-point operations. Modern FPGAs include adders inside ALMs as well as integrated SRAM and Digital Signal Processing (DSP) units, which are typically used to implement fixed-point multipliers. In fact, modern FPGAs integrate numerous highly efficient and optimized DSP units[1]. These units integrated in the FPGA fabric make fixed-point implementations the preferred choice for pattern recognition applications.

Even though floating-point inference acceleration based on FPGA is also a feasible choice to increase energy efficiency and throughput compared to general purpose GPUs [134], it is in low precision fixed-point implementations where FPGA acceleration really makes a difference. Application specific [135], [136] and more general purpose architectures based on systolic arrays [131] bring benefits in this regard. Although these modifications to simplify the model are especially common in the DL field, here they are applied to smaller ANNs, with simpler training, based on RC.

In particular, this chapter is focused on two kinds of architecture. The first is the so called ring topology ESN, which is introduced in section 4.1 and a low precision hardware implementation for Audio Event Detection (AED). The results of this research have been published in an international journal in collaboration with Endura Technologies[2] [137]. The second RC implementation is described in Section 4.2 different from the conventional ESN approach, which fits better in the more general RC definition since it is based on CA. Also, it does not present feedback connections between data samples, so that it could be described as a feed-forward architecture and therefore shares similarities with a multilayer RVFL. This research resulted in an international journal article too [138].

4.1 Ring topology Echo State Networks

[1]Modern low-end FPGAs integrate on the order of 10–100 DSP units and the high-end ones on the order of 1000–10,000.

[2]Endura Technologies (`http://enduratechnologies.com/`) describes itself as "a R&D service company providing development services of state of the art, disruptive solutions in power management, artificial intelligence, and audio applications [...]".

4.1.1 Contribution

In this section, we present a simple hardware-optimized ring topology ESN circuit design, which presents high energy efficiency capacities that fulfill low power requirements for edge intelligence applications. As a proof of concept, we used the proposed design to implement a low power AED application in FPGA. The obtained results show that the proposed approach may provide good accuracy with low power characteristics. In fact, our approximation shows an energy consumption below the micro-joule per inference. The proposed system is therefore optimal for edge applications requiring ML capabilities, e.g. near sensor computing, in which energy efficiency and accuracy are the key issues.

4.1.2 Related work

RC systems can be hardware optimized [139] using a ring topology [140], so that neuron's fan-in is drastically reduced. Along with this ring topology, the reservoir connectivity may also be optimized by selecting specific weights so that only simple shift-and-add operations are performed at each reservoir neuron instead of computationally expensive MAC operations. In this context, there exist previous works regarding FPGA implementations focusing on the so-called single-node reservoir based on only one physical node [141] which can represent a ring topology by time division multiplexing with an input mask [142], [143] and nonlinearities with feasible electronic and optical implementations. Moreover, RC hardware implementations have been previously applied to spoken digit recognition [118], [143]. In this context, our contribution in front of previous publications on FPGA implementations stand out in the training method and the digital implementation. The first contribution is that training is performed on a per frame basis using log-mel energies as the input features. Also, the node states are register-based instead of stored in RAM and the reservoir implementation is fully parallel with a simple nonlinearity at each node. This optimized RC model has demonstrated to provide good accuracy and energy efficiency characteristics for time series forecasting or equalization problems [139], [144], [145].

In this context, and inspired by the aftermentioned works, we developed a feasible methodology for low power AI applications, it is based on a simplified ring topology RC system. Such system is tested on an AED task, showing to be a feasible alternative that may fit edge computing applications' low power requirements. For example, if an end user is interested in identifying a potential dangerous situation if the system detects, e.g. gun shots or people screaming. AED or audio tagging system could filter environmental sound and detect specific audio classes. Initially, the RC system is trained and evaluated on a 2-class dataset to determine whether there is a gun shot or not in a 4 s audio slice. Secondly, a 10-class dataset, the Urban Sound 8K [15], is also used for training and evaluation purposes.

4.1.3 Theoretical foundations

Based on the *standard* foundations of the ESN framework summarized in Section 2.3.3.3, the structure and precision of the reservoir is modified in order to reduce logic resources and power consumption. First, since this section is dealing with an ESN, the system evolves in discrete time steps and its dynamics are governed by the input distribution. The ESN dimensionality and connectivity matrix $\boldsymbol{R}$ are defined by (4.1).

$$
\begin{aligned}
\boldsymbol{R} &= \left(\boldsymbol{W}^{in} | \boldsymbol{W}^{reservoir}\right) \\
&= \begin{pmatrix} v_{0,0} & \cdots & v_{0,M-1} & r_{0,0} & \cdots & r_{0,N-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ v_{N-1,0} & \cdots & v_{N-1,M-1} & r_{N-1,0} & \cdots & r_{N-1,N-1} \end{pmatrix}
\end{aligned} \tag{4.1}
$$

This connectivity matrix contains the input-to-reservoir weights $v_{i,j}$ and the reservoir adjacency matrix[3] defined by the weights $r_{i,j}$. In the single sequence case, let the M-dimensional input data and N-dimensional reservoir state at time step k be denoted by the row vectors $\boldsymbol{u}_k$ and $\boldsymbol{x}_k$, respectively. In this case input and the full reservoir state vector feed the readout model. So, for a single sequence state, the readout model inputs at time step k is conveniently defined as:

$$
\boldsymbol{z}_k = (\boldsymbol{u}_k | \boldsymbol{x}_k) \tag{4.2}
$$

If these row vectors ($\boldsymbol{u}_k$, $\boldsymbol{x}_k$ and $\boldsymbol{z}_k$) are aranged as matrices in which different rows represent different time steps, for L_s time steps, then (4.2) is equivalent to (4.3).

$$
\boldsymbol{Z} = (\boldsymbol{U} | \boldsymbol{X}) \tag{4.3}
$$

then the reservoir states stored in matrix $\boldsymbol{X}$ evolve according to (4.4).

$$
X_{k,:} = f(Z_{k-1,:}\boldsymbol{R}^{\intercal}), \quad k = 1, 2, 3, \ldots, L-1; \qquad\qquad X_{0,:} = \boldsymbol{x}_{initial} \tag{4.4}
$$

where f is a nonlinear elementwise function and $\boldsymbol{x}_{initial}$ is conveniently chosen. All the components of this row vector are initially set to zero for simplicity. Since the main target of this work is to obtain a hardware friendly design, several major simplifications are made. The ESN ring topology can achieve competitive performance on certain prediction tasks such as NARMA [147], the Santa Fe Laser [148] or nonlinear channel equalization [149], just as Tino and Rodan reported in *Minimum complexity echo state networks* [140].

In addition, the input-to-reservoir and internal (ring topology) reservoir weights are restricted in order to reduce the hyperparameter search, circuit size and complexity. On the one hand, input-to-reservoir weights are ternary, so that $v_{i,j} = \xi_{i,j} v$ with constant v and $\xi_{i,j} \in \{-1, 0, +1\}$ are chosen randomly with a given degree of sparsity and equal probability of being negative (-1) and positive $(+1)$. On the other hand, the internal reservoir weights are set to a constant value r, i.e. $r_{i,j} = r$, which is referred to as the nonlinearity strength. Therefore, the simplified connectivity matrix is given by (4.5).

$$
\boldsymbol{R} = \begin{pmatrix} v\xi_{0,0} & \cdots & v\xi_{0,M-1} & 0 & \cdots & \cdots & 0 & r \\ v\xi_{1,0} & \cdots & v\xi_{1,M-1} & r & 0 & \cdots & \cdots & 0 \\ v\xi_{2,0} & \cdots & v\xi_{2,M-1} & 0 & r & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ v\xi_{N-1,0} & \cdots & v\xi_{N-1,M-1} & 0 & \cdots & 0 & r & 0 \end{pmatrix} \tag{4.5}
$$

[3] An ESN reservoir can be represented by a directed graph and the adjacency matrix indicate whether each node is connected to each other and itself or not, as well as the weight between adjacent nodes [146, Chapter 2].

Furthermore, the nonlinear activation function (4.4) is chosen to be a simple piece-wise linear function, often referred to as hard tanh or HTanh activation in the literature [150], defined as

$$\mathrm{HTanh}(x) \equiv \max(-1, \min(x, 1)) \tag{4.6}$$

If the v and r values are restricted to inverse powers of 2 (i.e. $v = 1/2^n$ with $n = 0, 1, 2, 3, \ldots$), then multiplications can be replaced by shift right and add operations. Fig. 4.1 presents the digital fixed-point implementation of each individual node implemented in this work. This simplified activation function enables highly parallel reservoir architectures since there is no need to implement CORDIC algorithm [151] or Taylor series expansion, which use more hardware. In contrast, we propose simpler fixed-point operations listed in Appendix B to avoid the use of conventional digital multipliers within the reservoir implementation. Thus reducing area and power. In order to give an intuition about the area requirements, Table 4.1 lists the FPGA resource utilization for the 8 and 16-bit adder, multiplier and maximum blocks. Notice the multiplier is substantially more resource demanding than an adder or maximum block, making the reservoir's fully parallel implementation not feasible.

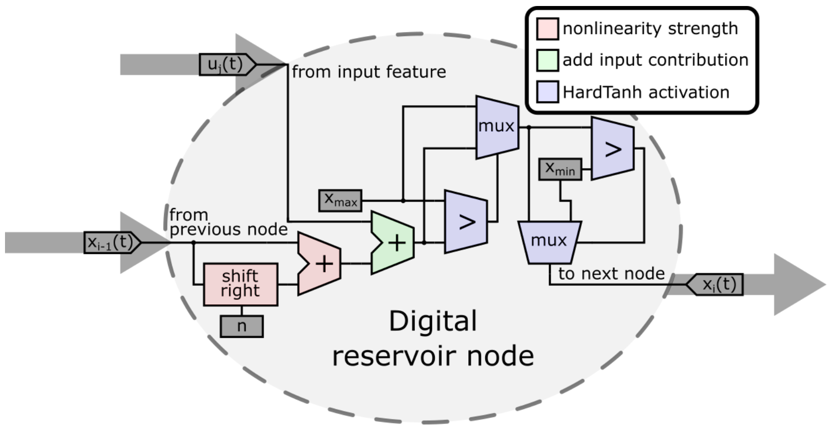


Figure 4.1: Digital node hardware architecture for the ring topology reservoir.

Table 4.1: Number of 8-bit hardware lookup tables required to perform an 8-bit and 16-bit binary operation. Data obtained from Quartus Prime software for a Cyclone V FPGA device.

	Adder	**Multiplier**	**Maximum**
8-bit	8 LUTs	91 LUTs	15 LUTs
16-bit	16 LUTs	171 LUTs	26 LUTs
Ratio	1	11	1.7

In this context, a similar reservoir architecture has been applied to 1-dimensional prediction tasks showing similar results as other research groups for the same datasets [139], [145]. Table 4.2 summarizes the FPGA works in this field (*FPGA*) compared to other technologies (*Numerical, Optoelectronic* and *Analog circuit*). Notice most hardware results are quite similar and all of them are within a similar order magnitude.

In this work the results from [139], [145] have been extended to $M > 1$, and applied to AED for $M = 64$ inputs. In addition to dimensionality, the other main difference is that it is no longer a question of predicting time series, but of a classification task.

Table 4.2: Several 1-dimensional timeseries prediction results using hardware implementations based on the ring topology reservoir approach. The corresponding datasets are Mackey Glass [152], Santa Fe laser data [148] and nonlinear channel equalization (NCE) for different signal-to-noise raitos [149].

		Implementation							
		Ref. [140]		Ref. [153]		Ref. [117]		Ref. [139]	
Dataset	**Metric**	**Numerical**	N	**Optoelectronic**	N	**Analog circuit**	N	**FPGA**	N
Mackey Glass	$\log_{10}(RMSE)$	-	-	-1.24	615	-	-	-1.74	300
Santa Fe dataset	$NMSE$	0.008	200	0.106	388	0.031	400	0.079	200
NCE (16dB)	SER	-	-	0.05	246	-	-	0.035	27
NCE (20dB)		-		0.013		-		0.0095	
NCE (24dB)		-		0.007		-		0.007	
NCE (28dB)		-		0.0025		-		0.00055	

Sound data is preprocessed and features represent filtered frequency channels so that data is organized as a set of sequences containing a log-mel [154] spectrogram for each sound sequence. Feature extraction in this section is obtained from a bit accurate hardware simulation and the main parameters of this implementation are listed in Table 4.3, see e.g. [155] for hardware implementation details.

Table 4.3: List of relevant log-mel spectral feature extraction parameters.

Parameter	**Value**
Audio sampling rate	16 kHz
Window size	512
Window overlap	352
Mel bins	64
Output bit width	8

Since AED is a classification task, the objective is to assign a single label to each sequence. There are two approaches to handle this problem when using RNNs regarding the network memory and sequence length. If the sequence is short enough, after being processed by a relatively simple RNN, it would be enough to perform a training based on the state of the network at the end of the sequence. The last time step must contain information about all the input sequence.

However, it can also be the case that the sequence is longer than the number of time steps influencing the final state of the RNN. Therefore, it is possible that the final state does not contain information related to the event contained in some interval of the sequence. A possible solution is to consider all the time steps to train the network.

To get an idea of how many frames are needed to train a network based on the final state of each sequence, we can take the example of the LSTM, which has a memory capacity on the order of 20 previous entries[4]. This problem can also be solved using attention models [156] but the approach requires the use of LSTMs and additional RNN structures as well as fully connected nodes and hyperbolic tangent activations,

[4]RC models have similar memory capacity but it is typically evaluated on 1-dimensional data [105], which is not the case. To the best of our knowledge, there are no research works evaluating optimal memory capacity as a function of input data dimensionality for RC models. However, any multidimensional input can be arranged as a 1-dimensional input array, so that the memory capacity is less than in the 1-dimensional case and, in the worst case, inversely proportional to the number of input timensions.

which is far more complex than the proposed approach. In fact, it is also common to tackle AED problems combining either CNN [72] or both CNN and RNN [157] to overcome the memory problem.

Since the motivation of the work is to keep the model as simple as possible, we use all sequence frames. The main disadvantage is that sequences must be labeled on a per-frame basis and further postprocessing is needed to set a criterion to assign a label to the whole sequence based on predicted frame labels. Training and postprocessing methods are described in detail in Section (4.1.4).

4.1.4 Methods

The system consists of three main blocks, as illustrated in Fig. 4.2. These three blocks correspond to: feature extraction, ESN and postprocessing steps. As mentioned above, this section describes training and postprocessing. In addition, the evaluated datasets and statistical performance metrics are also introduced in this section.

Moreover, individual calculations are done with the best known set of model hyper-parameters, obtained using random search for r and v, and input-to-reservoir weights are random uniform with certain degree of sparsity.

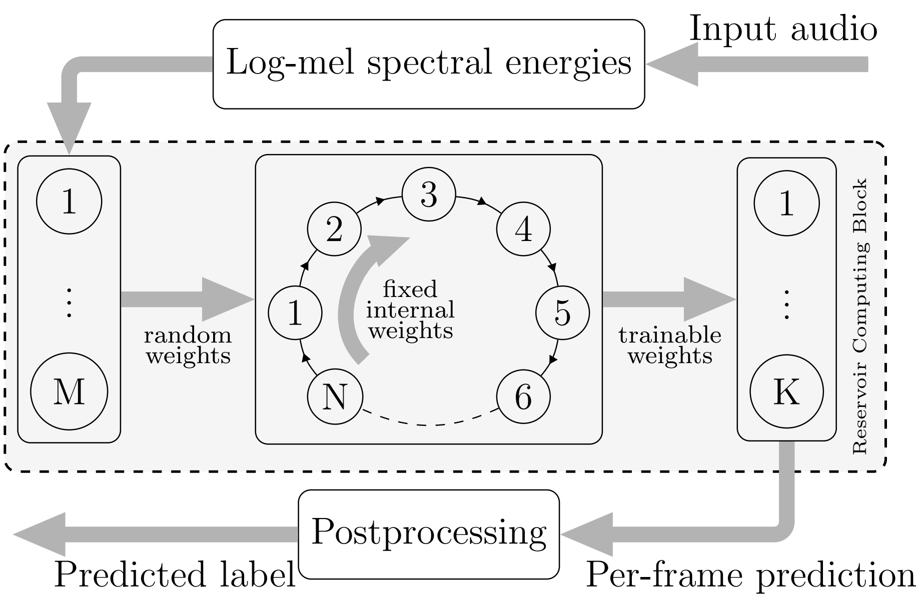


Figure 4.2: Ring topology RC applied to log-mel spectral energies. The input audio is first pre-processed to obtain the log-mel features, so that a linear combination of these features are inputs to the reservoir, which in turn computes a higher dimensional nonlinear mapping in the time domain. Finally, the reservoir states are linearly combined to obtain a meaningful readout, which is post-processed to improve performance.

4.1.4.1 Training on a per-frame basis

Before training, all sound waves are converted to log-mel spectrograms. Fig. 4.3 (top, middle) shows an example both sequences. Notice according to the feature extraction parameter choice in Table 4.3, every second of sound wave sampled at 16 kHz is represented by 100 feature frames. As the example shown in Fig. 4.3, dataset samples contain 400 frames[5] As regards the labeling strategy, training on a per-frame basis

[5]The actual duration might be shorter but empty space is filled with silence to simplify matrix notation.

refers to the fact that individual time steps in a sequence are labeled separately. The readout layer provides a real-time response, i.e. the output may change every time step, and the supervised training method includes real-time labeling. Every single log-mel frame is therefore labeled according to its corresponding value as depicted in Fig. 4.3 (bottom).

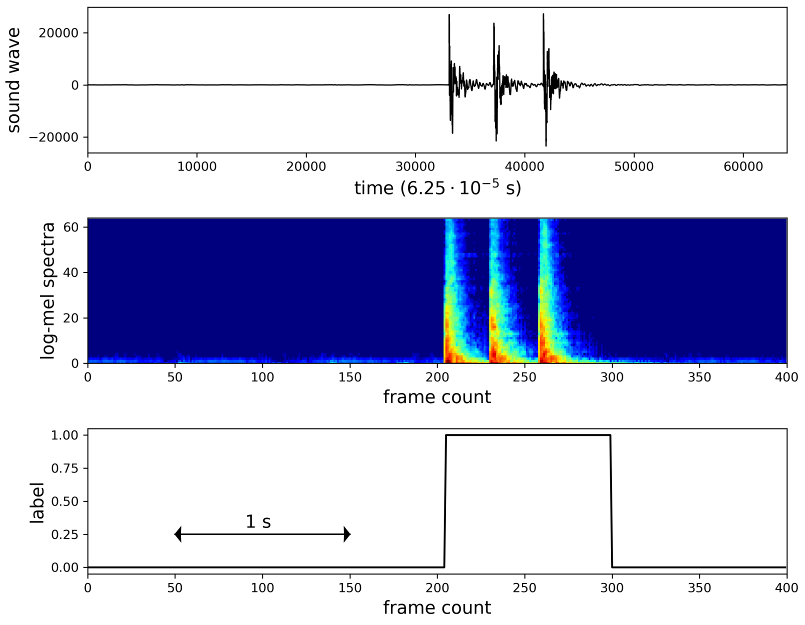


Figure 4.3: Audio, features and corresponding label assigned on a per-frame basis.

Since all frames are labeled, a linear model can be used to map each reservoir state (concatenated with the current input frame) to real-time category scores. However, there is a difference compared to the time series prediction task defined by (4.3) and (4.4). In this case the dataset is composed by a collection of sequences and the reservoir states are frozen to be reinitialized after each sequence, i.e. it is set to zero in this case.

4.1.4.2 Postprocessing

Suppose the concatenated inputs and reservoir states are given by the tensor $\boldsymbol{Z}$, so that $Z_{i,j,:}$ contains the inputs and reservoir states for the j-th frame of the i-th sequence. So that the already trained readout layer provides an Immediate Category Score (ICS) $\boldsymbol{Y}$, such that $Y_{i,j,:}$ are the readout logits corresponding to the j-th frame from the i-th sequence. An example containing dog barks (DB) is shown in Fig. 4.4 (top) together with the correspondig ICS sequence (bottom). It is an example extracted from an UrbanSound8K multiclass dataset, described below (4.1.4.4). In this specific case the audio contains (correctly detected) dog barks (red, DB) and people speaking in the background, which is identified by the model as children playing (green, CP), a reasonable mistake given the low complexity of the model. The problem is the fact that the whole sequence is weakly labeled, i.e. even though several events could be identified, there is only one label assigned to each sequence (*dog bark* in this case). So, based on an output ICS sequence like this, which unique category should be assigned to the whole sequence? In this case, what has been done is to assign a label to each frame based on the highest ICS whenever it is grater than some threshold value y_{th} or not. Whenever the ICS winner is greater than y_{th}, the corresponding score associated to this

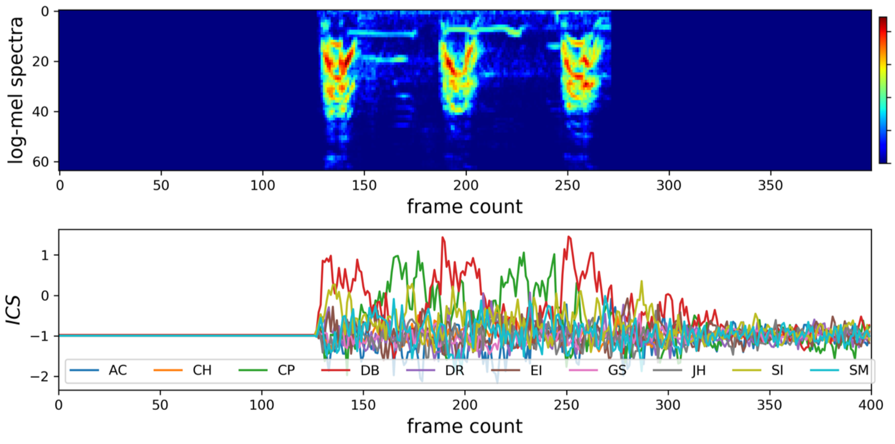


Figure 4.4: Example mel features and immediate category scores.

winner category accumulates 1, and 0 otherwise. This threshold-based Accumulated Category Score (ACS) is denoted by $\boldsymbol{S}$. The ACS associated to the j-th class from the i-th sequence is therefore given by (4.7), where $L_{tran.}$ is the number of time steps needed to skip transient reservoir dynamics, which is set to $L_{tran.} = 5$, L_i is the i-th sequence length and $\mathcal{H}$ is the Heaviside step function.

$$S_{i,j} = \sum_{k=L_{tran.}}^{L_i} \mathcal{H}\left(y_{th} + \mathsf{Y}_{i,k,j}\right) \tag{4.7}$$

These ACSs are finally utilized to assign a label for each sequence, grouped in vector $\boldsymbol{l}$ and given by (4.8), where $\boldsymbol{d}$ are reduction factors, which are additional hyperparameters needed to handle single sequence data imbalance. It contains as many components as there are classes.

$$l_i = \underset{j}{\text{argmax}} \left\{ \frac{S_{i,:}}{d_j} \right\} \tag{4.8}$$

For example, if a sequence contains a single and brief gunshot but additional overlapping audio is identified as another class and all $\boldsymbol{d}$ components are equal, the whole sequence would be misclassified. Instead, $\boldsymbol{d}$ components are chosen according to the typical single event length, it helps to reduce misclassifications due to single events shorter than others since the decision (4.7) is based on accumulation.

Therefore, in Fig. 4.4 the predicted sequence label would be *dog bark* if the threshold is set to e.g. $y_{th} = 0$. However, in other cases, even if $\boldsymbol{d}$ is correctly chosen, the fact that more than one source of sound is present and one of them is much more frequent and above threshold than the actual sequence ground truth label would cause a missclassification.

4.1.4.3 Gunshot detection database

The gunshot detection database contains a custom collection of *ambient* and *gunshot* audio waves. However, for the frame-base labels, a third *silence* class is introduced, created to take into account frames not containing either *ambient* or *gunshot* data. The file count for sequence labels is shown in Table 4.4, with a total of 727 files.

Table 4.4: Gunshot dataset wave file counts. It contains 727 files combining both *ambient* and *gunshot* audio data.

	Ambient	**Gunshot**
Train	400	111
Test	113	103
Total	513	214

4.1.4.4 Multiclass audio dataset

The UrbanSound8K dataset [15] contains ten different classes of urban sounds, listed in Table 4.5. It contains 8732 labeled audio files with a duration that is less or equal than 4 s combining background and foreground audio samples. The dataset classes are listed in Table 4.5 in an environment containing additional street noise and sounds such as people speaking. All wave files are preprocessed to obtain their corresponding log-mel spectrograms, which are used for training and testing, following the 10-fold cross-validation experimental setup explained in [15]. Therefore, results presented for this dataset are averaged over the 10 possible train/test combinations.

Table 4.5: UrbanSound8K labels, abbreviation and meaning.

Label	**Abbreviation**	**Meaning**
0	AC	Air conditioner
1	CH	Car horn
2	CP	Children playing
3	DB	Dog bark
4	DR	Drilling
5	EI	Engine Idling
6	GS	Gunshot
7	JH	Jackhammer
8	SI	Siren
9	SM	Street music

4.1.4.5 Statistical performance metrics

In order to evaluate the goodness of a model, several statistical performance metrics are introduced depending on the specific task to evaluate. In the gunshot detection case, silence is excluded from the evaluation since it is part of the frame label options but not class of interest. It is a binary classification task. In this kind of applications it is desirable to be able to minimize the number of gunshots detected as ambient without penalizing accuracy at all, i.e. it is better to trigger false alarms and detect a gunshot that turned out not to be a shot than to classify it as ambient in order to be sure all detected gunshots are in fact gunshots. So, Table 4.6 metrics have been utilized to evaluate a possible tradeoff between these two behaviours while maintaining reasonable accuracy. *Overall accuracy* is usually referred to as just *accuracy* and the other two, *true positive rate* and *true negative rate*, are also referred to as *sensitivity* and *specificity*.

Table 4.6: Statistical performance metrics. (TP) True positives. (TN) True negatives. (FP) False positives. (FN) False negatives.

Metric	Expression
Overall accuracy	$\frac{TP+TN}{TP+FP+FN+TN}$
True Positive Rate	$\frac{TP}{TP+FN}$
True Negative Rate	$\frac{TN}{TN+FP}$

In contrast, for the multiclass AED UrbanSound8K dataset, rather than focusing on sensitivity or specificity, the goal is to compare obtained results in terms of accuracy using a pre-defined 10-fold cross-validation for model performance evaluation, as suggested by the original UrbanSound8K reference [15]. It consists of averaging test accuracy results obtained for different pre-defined test set choices. In addition to accuracy (i.e. top-1 accuracy), top-2 accuracy[6] is also evaluated to justify the fact that in our particular frame-base inference setup missclassification is not necessarily missdetection.

4.1.5 Results

This section reports inference performance metrics obtained from the proposed ESN implementation applied to input datasets described in Section 4.1.4 (Methods). As explained in Section 4.1.3, to be able to evaluate model performance from the audio datasets, which contain audio files, log-mel spectral features are obtained from a hardware simulation. These features are 8-bit 64-dimensional features per frame. The RC subsystem is implemented in an Intel Cyclone V FPGA and the postprocessing stage is performed offline, i.e. the computation of (4.7) and (4.8) is done externally.

On the one hand, results associated to the gunshot dataset are summarized in Table 4.7. These results correspond to a fairly small reservoir size composed by 192 nodes with 8-bit activations and readout parameters. The best result obtained is for $y_{th} = 1$. The other two options are also shown for comparison purposes. From left to right, $y_{th} = 1.11$ corresponds to a 100% specificity, but in this case the goal would be to maximize sensitivity as long as accuracy is not degraded, so that $y_{th} = 1.0$ would be a better choice, since it improves both accuracy and sensitivity. However, further decreasing y_{th} involves accuracy and specificity degradation with no additional sensitivity benefits, so that $y_{th} = 1.0$ is the best option.

On the other hand, results associated to the UrbanSound8K dataset have been obtained in terms of both top-1 and top-2 accuracy, which are represented in Fig. 4.5. Individual data points correspond to accuracy averaged over the 10-fold cross-validation train/test set combinations. Here, training has been performed over 250,000

[6]Top-N accuracy is computed by interpreting as correct those predictions for which the ground truth is one of the N most likely categories.

Table 4.7: Performance metric values (in %) for an ESN reservoir with 192 nodes, 8-bit input, activation and parameters, for different postprocessing thresholds. The selected category factors are: $= d_0 = 64$ (silence), $d_1 = 64$ (ambient), $d_2 = 1$ (gunshot).

	Threshold (y_{th})		
Performance metric	1.11	1.0	0.9
Accuracy	91.6	96.3	94.9
Sensitivity	82.5	94.2	94.2
Specificity	100	98.2	95.6

randomly selected frames and corresponding labels in each of the 10 training sets per data point. Then, reduction factors are chosen so that they are inversely proportional to the (frame-base) training occurrences for each class. As an example, suppose class A has a training samples (i.e. frames with label A), then its reduction factor is chosen to be proportional to $a/250,000$. From Fig. 4.5, one can observe the model is able to fit training data with an increasingly large reservoir. However, the improvement is not translated into additional test set accuracy. Top-1 test accuracies are similar or slightly higher than the ones obtained using decision trees (J48) and k-NN (k=5) methods applied to the same problem [15][7] and using the same 4 s slice duration.

In this case, the difference in accuracy is not (or not only) due to an overfitting problem related to the amount of data used for training, it is a postprocessing problem. A way of thinking about this is to suppose the RC system is detecting the actual ground truth category but it is not as frequent as it should be to be correctly classified. As an example, suppose there were only one dog bark (DB) and the rest is people talking confused with children playing (CP) in Fig. 4.4 and the corresponding reduction factor is such that (4.7) divided by the reduction factors $\boldsymbol{d}$ yields highest score for CP, does it mean DB was not detected even though the whole sequence is not correctly labeled? The answer is no, DB would have been detected. Therefore, top-2 accuracy is also represented in order to detect whether the ground truth corresponds at least to the second highest score category, which means the correct class ICS was above threshold at some time interval.

In addition to results related to model performance, reservoir's energy efficiency and number of required readout MAC operations are also considered and presented in Fig. 4.6. Notice there are two different vertical axis with MACs per inference at the left hand side (blue) and energy efficiency in nJ/inference at the right hand side (orange). MAC operations are proportional to the number of classes, sequence length and reservoir size. These numbers are obtained for 10 classes and 400 frames per inference. As regards energy efficiency, it refers only to the reservoir subsystem and is proportional to its size and sequence length, which is fixed to 400.

Compared to relatively small CNNs oriented to audio processing, the number of MAC operations is much higher [72], [158] compared to the proposed model. While the order of magnitude of MAC operations per inference is over 1000M for the case of CNNs, the ESN implementation needs between 10M and 50M MACs per inference (see Fig. 4.6). Nevertheless, this represents a decrease of about 15% accuracy in the 12288 nodes reservoir model compared to state-of-the-art CNN results on the UrbanSound8K

[7]The exact numbers are not available since these results were represented in a graphical format.

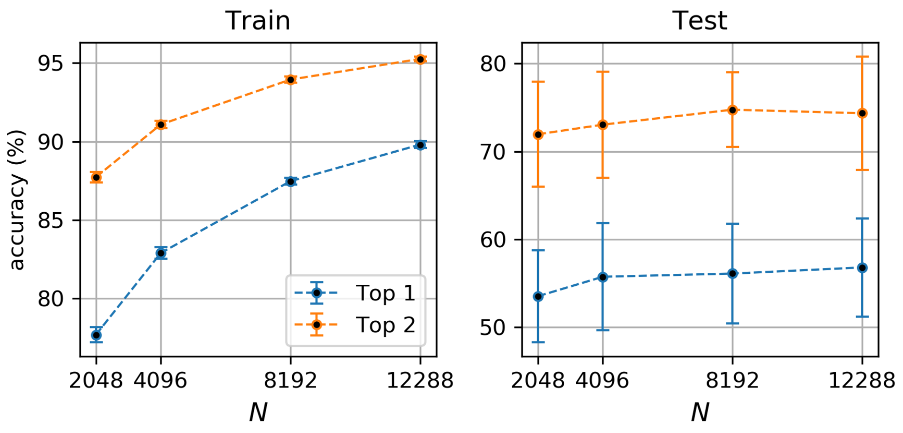


Figure 4.5: UrbanSound8K training (left) and testing (right) mean top-1 and top-2 accuracies, as well as corresponding standard deviation, obtained from 10-fold cross-validation. Reduction factors have been selected so that they are inversely proportional to the (frame-base) training set occurrences for each class.

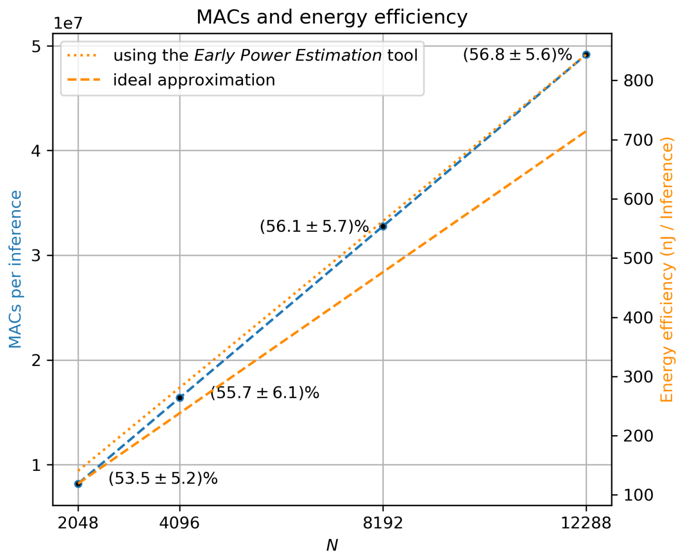


Figure 4.6: Number of needed MAC operations per inference (left axis) and ESN reservoir efficiency (right axis) with corresponding test set top-1 accuracies for each data point. The ideal energy efficiency approximation corresponds to n 64-node reservoirs, with $n=N/64$.

dataset [158] without data augmentation.

As regards energy efficiency, there is a huge gap compared to existing low power sound recognition systems. In particular, real-time bird sound recognition was implemented in an ARM Cortex-M4F based TI TivaC TM4C1294NCPDT microcontroller [159] in which the classification stage consumes 35 mJ per inference at 50 MHz clock frequency, which is about 40× the energy consumption per inference for the 12288 nodes reservoir (worst case).

4.1.6 Summary

This design was motivated by the increasing demand in low-power pattern recognition at the edge, which enables processing capabilities to increase energy efficiency and reduces large sensor data transfers to the cloud and solves related privacy issues while maintaining reasonable accuracy.

Even though the results are not comparable to those obtained by CNNs for the UrbanSound8K dataset [158], it is equivalent or slightly better than other ML methods [15]. Also, top-1 classification accuracy could be possibly improved by modifying the postprocessing stage and the model might be better suited to audio event tagging or used to filter most environmental data, transmitting a signal indicating a class of interest is present in the audio whenever it is detected, i.e. its ICS is the highest and above threshold.

Moreover, the model is based on a fully parallel hardware architecture designed to reduce energy consumption in very specific tasks related to AED. The fact it is targeted to a very specific set of applications and further fixed-point optimization and simplifications in terms of parameter, internal storage and computation in the reservoir makes it at least about 40× more energy efficient than a low power software solution applied to bird sound recognition [159]. Therefore, it is an attractive candidate for a variety of battery-powered scenarios, such as always-on inference, which could also benefit from mixed-signal implementations [160] or audio event data co-processing as part of an SoC [161], suitable for smart watches or smart sensors. Furthermore, it is not necessarily limited to audio classification and potential ML use cases might include audio tagging, monitoring of physiological data [162] or channel equalization [163].

4.2 Reservoir Computing and Cellular Automata

It has been shown how RC can be applied to classify and detect certain types of multidimensional time series, in particular, it has been applied to features extracted from audio files. Notice the extracted features might be interpreted as grayscale images, so that a similar approach might be useful for classifying images by processing image rows or columns sequentially. Nevertheless, the low complexity ring topology reservoir and further simplifications in bit width and activation function are great for reducing energy consumption but makes it uncompetitive in terms of accuracy when applied to image classification compared to more complex randomly connected reservoirs and DL approaches [164], even if these images are small compared to image sizes handled today in the DL field [165]. Therefore, in order to enable comparisons with other works, the proposed methodology has been evaluated on the well-known MNIST dataset.

4.2.1 Contribution

Recently, CA spatio-temporal evolution has been proposed as a feasible way to obtain reservoir states to implement RC systems in which the automaton rule is fixed and the training is performed using a linear model. Based on this idea, this work contributions are threefold. First, a new pattern recognition system has been proposed and analyzed. Second, a systematic method to evaluate and discover the best model is proposed, based on a reduced design space, smart initial conditions and data augmentation (DA). Third, an FPGA implementation associated to the final inference model has been developed. The pattern recognition system is based on feature expansion generated by Elementary Cellular Automata (ECA) and classification by means of a linear model. The model is potentially capable to handle pattern classification and exploits 2-dimensional spatial correlations. It has been applied to handwritten digit recognition from grayscale images, obtaining competitive results in terms of processing time, resource utilization, power and inference accuracy, which makes it a good candidate for edge AI. Even though the model is applied to grayscale images, it could be extended to a higher number of spatially-correlated problem dimensions.

4.2.2 Related work

There are multiple examples of CA being applied to image analysis and processing following knowledge-based approaches [166]–[169], which are actually not related to this work since the proposed model is data-driven rather than knowledge-based. A data-driven image saliency detection algorithm was proposed to exploit local similarity using bayesian optimization combined with CA to obtain saliency maps [170]. Their proposed CA forward propagation is generic, optimized and applied to superpixels, while here CA is restricted to the ECA search space, not optimized, applied to single bits and targets classification tasks. Moreover, multiple attractor CA (MACA) have been used to classify 1-dimensional bit strings [171], [172], working as an associative memory and processing is restricted to limit cycle dynamics, which is trained using a genetic algorithm. Another application is this line is CA vector quantization [173], which is also related to the MACA approach. In contrast, the system proposed in this thesis implements RC based on CA (ReCA).

ReCA systems incorporate transient and chaotic or pseudorandom temporal evolution of the automata. Therefore, the fundamental difference compared to associative memory approaches is the fact that dynamics are not restricted to fixed points and limit cycles.

Following developments related to ReCA [174]–[179] , the CA spatio-temporal evolution is used as the reservoir states. The original idea was first proposed by Yilmaz [174], whose work was extended to deep RC models by Nichele and Molund [178]. As regards the implementation described in this thesis, in the ReCA system the CA states take input data as its initial state and the spatio-temporal volume is first reduced and then interpreted by a readout linear model. An exhaustive study of the resulting accuracy for different ECA rules arranged in a specific configuration is carried out. After this analysis, the most accurate rule is finally selected and implemented in FPGA using simple digital circuitry. In this particular case, rule 90 provides the highest validation fitting for the MNIST dataset. Interestingly, rule 90 was already proposed by Yilmaz for metric learning as a computationally efficient alternative to RBFs in support vector machines.

Oliveira et al. described an attempt based on the application of multiple 2-dimensional CA rules with training based on a genetic algorithm [180], but limited to using only 1350 training and 150 test examples in which each pixel is represented by a single bit. In contrast, in this work CA is composed by many simple 1-dimensional CA, all automata follow the same rule and the entire MNIST is considered. MNIST images are not thresholded to a single bit pixel representation and the training approach is analogous to that of RC or RVFL-NN rather than based on genetic algorithms.

Although this is not the first time in which CA are implemented in digital hardware for research purposes (see e.g. [181]), by the time the proposed model implementation was published [138], to the best of our knowledge it was the first time in which a full ReCA system oriented to pattern recognition had been hardware-implemented and tested. In addition, the workflow to explore design space and the model itself include novel ideas.

Disclaimer

There exist other interesting papers connected to a greater or lesser extent to the content described in this thesis. However, those research works were published after ours [138].

The most significant contribution related to this topic is a conference paper that reproduces a quite similar model and cleverly extends our contribution eliminating all multiplications, achieving similar accuracy [182]. It proposes a different readout based on random forest applied to single bit features, as well as low-cost FPGA implementations based on 3-line buffer processing and simple ECA units for different model variations. Moreover, it implements feature pruning by removing CA iterations that are less relevant to the classification task to further optimize resource utilization. Nevertheless, there is a subtle difference between our model and what they consider to be a reproduction of our model, which has been clarified in next section✣.

Another interesting yet quite different approach which is also worth highlighting is the self-classifying MNIST (software) implementation because it is the first work in which end-to-end differentiable neural CA are trained for classification purposes [183].

4.2.3 Theoretical foundations

The general ideas behind CA were first introduced by J. Von Neumann [184] and later revisited by S. Wolfram. The latter introduced ECA and their naming convention [185] described below. In addition, it is also described how ECA are used to build the proposed classifier.

4.2.3.1 Elementary Cellular Automata

In a nutshell, ECA are deterministic and the simplest class of 1-dimensional CA providing a rich and complex dynamic behavior. ECA are discrete dynamical systems with synchronous updates, each of which is composed by a 1-dimensional string of cells with two possible states evolving according to simple rules. These rules dictate how the string of binary cells should evolve based on cell states at the current time step and limited to self and nearest neighbor interaction. Since self and nearest neighbor interaction are represented by a group of three binary cells, a rule must be applicable for each of the $2^3 = 8$ possible configurations. Therefore, a rule specifies the next cell state for each configuration, resulting in a total of $2^8 = 256$ possible ECA rules. According to the standard naming convention proposed by [185], these rules are labeled from 0 to 255 based on the binary weighted code extracted from the next cell state for each of the 8 possible configurations. These configurations are in the following order, from the most significative (MSB) bit to the less significative bit (LSB): 0.111_2, 0.110_2, 0.101_2, 0.100_2, 0.011_2, 0.010_2, 0.001_2 and 0.000_2. The 8 possible groups of previous state configurations indicated from left to right are depicted in Fig. 4.7 for rule 86.

Over time, researchers have established several criteria to classify ECA in different classes according to their properties. Perhaps the most intuitive classification criteria is the one introduced by S. Wolfram [186], who established four classes according to

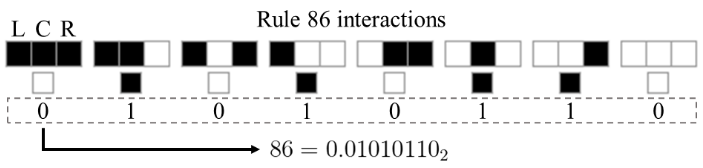


Figure 4.7: Rule 86 interactions according to notation described in the text. Self interaction is defined by the previous center (C) state, and left (L) and right (R) previous states define nearest neighbour interactions.

the dynamical system evolution skipping transient dynamics and for random initial conditions. These classes are: *uniform* (class 1), *periodic* (class 2), *chaotic* (class 3) and *complex* (class 4). The different classes have been illustrated in Fig. 4.8, depicting four examples per class. It is worth mentioning other classification criteria have been proposed in the literature, e.g. including additional classes to Wolfram's criteria [187], based on topology [187] or power spectra [188]. This work follows the original Wolfram's classification proposal [186] since it is sufficient to qualitatively justify the obtained results.

4.2.3.2 Model

Features are represented by numeric quantities, which can always be expressed in its raddix-2 weighted representation. A simple way to perform feature expansion is to map input features to input cells setting the initial state of an automaton. Therefore, an n-dimensional input vector $\boldsymbol{u}$ represented by m-bit components would be mapped to different ECA cells holding individual bits, so that there must be at least $n \cdot m$ binary cells. Then the ECA is evolved according to a given rule, generating a spatio-temporal collection of binary states, which might include initial states. Then, depending on whether input data is temporally correlated to previous inputs or not, feedback operations are introduced in order to incorporate memory capabilities. Depending on whether memory is incorporated or not, the system structure is more similar to either RC or RVFL-NN, respectively. Also, notice another option to achieve the memory property would be to choose the CA size (i.e. number of cells) much greater than the number of input bits.

However, the proposed architecture is focused on time independent inputs, so that the system resembles a RVFL-NN structure rather than a generic RC approach. Nevertheless, since RC contemplates any physical or computational dynamical system, it is better defined as a memoryless RC approach. In fact, one could interpret an M-step CA evolution as a locally connected feedforward network with M equally sized layers as illustrated by Fig. 4.9. Notice this architecture is different from a conventional FFNN or multilayer RBF-NN with these characteristics, since cell state activations are not restricted to nonlinearities applied to weighted sums or euclidean distances.

Moreover, unstructured input data might present correlation across more than one spatial dimension, i.e. it might be conveniently represented by a matrix or tensor, which have a rank greater than one. MNIST dataset contains 28×28 grayscale images, so that an input image is naturally represented by a matrix. Therefore, a group of inputs is conveniently represented by an $m \times b \times h \times w$ boolean tensor $\boldsymbol{U}$, where m is the number of grayscale images and h and w are the image height and width, respectively. In addition, each pixel is explicitly splitted into b bits. As already described, the straightforward approach would be to simply assign each bit to a different ECA cell and generate the automaton evolution. Nevertheless, this straightforward approach

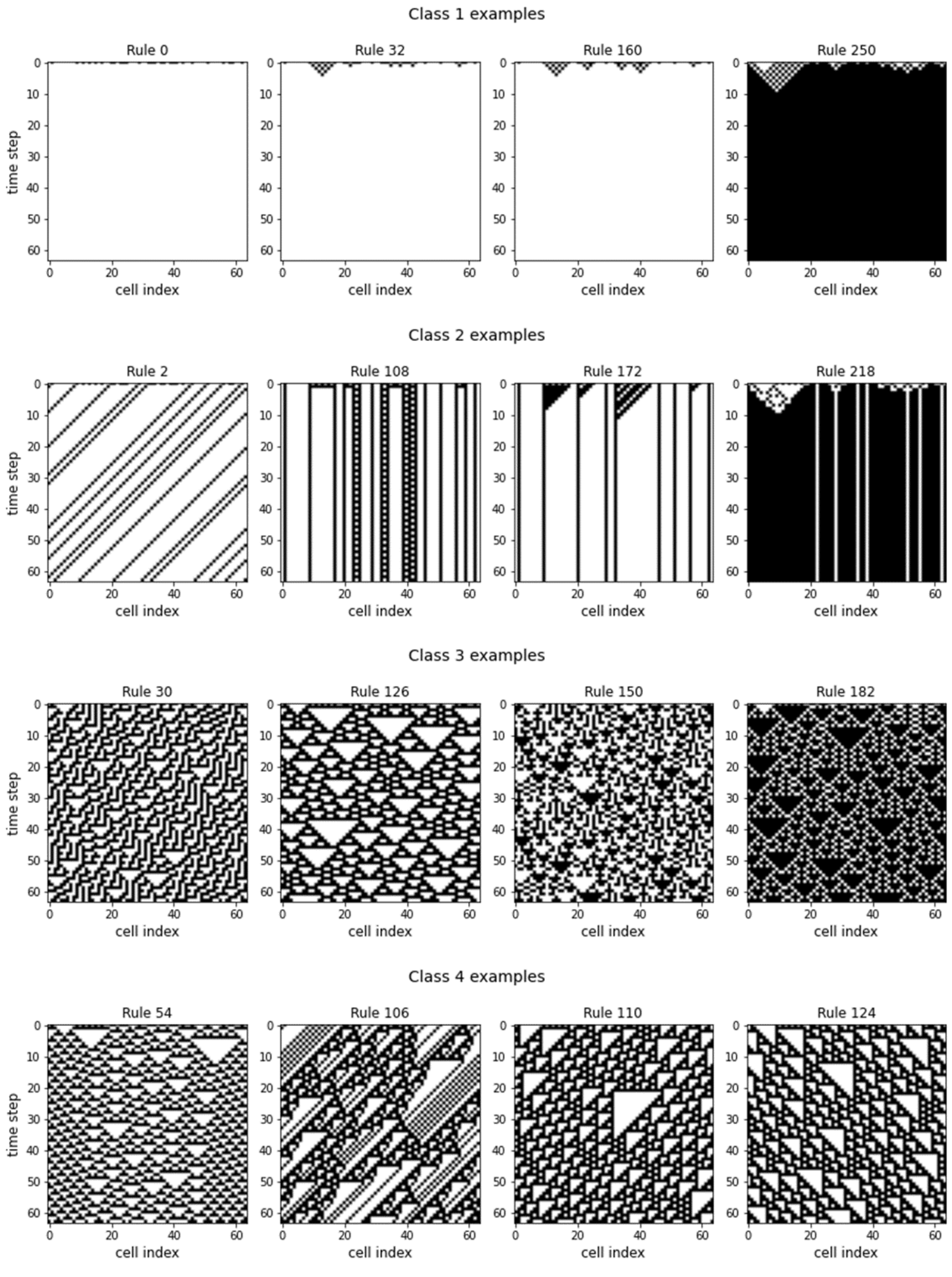


Figure 4.8: Example ECA rules and temporal evolution for each of the four classes defined by S. Wolfram.

does not exploit 2-dimensional spacial correlation in a grayscale image. The system would be naturally evolved with a 2-dimensional CA with 2^b possible states per cell, where b is the number of bits required to represent a pixel value and is equal to 8 (a byte per pixel) in most cases, including MNIST samples. The problem with many-state 2-dimensional CA is the vast design space exploration requirements in order to find

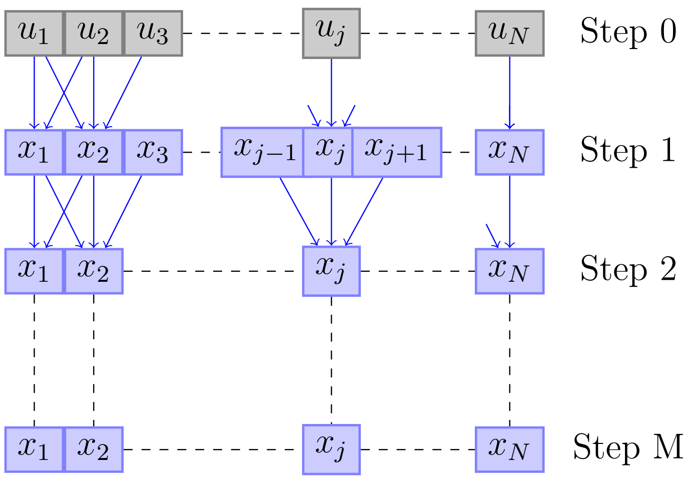


Figure 4.9: 1-D cellular automaton network structure with self and two nearest neighbors interaction.

an appropriate reservoir rule. For example, for 2-dimensional 256-state with self and four nearest neighbor interaction CA rule search space, the best candidate would be one out of 256^{256^5}. However, it would make sense to use rotational invariant rules[8], which would reduce the search space to 256^{256^2} but it is still not feasible and would be necessary to further delimit the search space. For this reason, the search space is delimited to ECA rules, which are applied to rows and columns independently for each bit layer.

Suppose the CA spatio-temporal evolution is represented by the rank-4 tensor $\boldsymbol{X}$, such that $X_{i,j,:,:,:}$ represents the j-th CA iteration for the i-th sample. The proposed evolution scheme, i.e. obtaining $X_{:,i-1,:,:,:}$ from $X_{:,i,:,:,:}$, is as follows. There is no interaction across different bit layers and two different CA coexist, one of them is iterated row by row ($\boldsymbol{X}^r$) and the other column by column ($\boldsymbol{X}^c$), with no interaction between different rows or columns. Then both CA evolutions are combined into a single feature expansion tensor $\boldsymbol{X}$. At the beginning, all states are initialized with the input data

$$X^r_{:,0,:,:,:} = X^c_{:,0,:,:,:} = X_{:,0,:,:,:} = \boldsymbol{U} \tag{4.9}$$

Let g^k be the function g applied k times, e.g. $g^3(\cdot) = g(g(g(\cdot)))$. So that

$$\begin{aligned} X^r_{i,j,:,:,:} &= g_r\left(X^r_{i,j-1,:,:,:}\right) = g^j_r\left(X_{i,0,:,:,:}\right), \\ X^c_{i,j,:,:,:} &= g_c\left(X^c_{i,j-1,:,:,:}\right) = g^j_c\left(X_{i,0,:,:,:}\right), \quad 1 \le j \le M \end{aligned} \tag{4.10}$$

where the function $g_r : \{0,1\}^{b\times h\times w} \to \{0,1\}^{b\times h\cdot w}$ iterates binary layers' rows from the most to the less significative bit, containing $b{\cdot}h$ ECA of length w, and $g_c : \{0,1\}^{b\times h\times w} \to \{0,1\}^{b\times h\times w}$ does so with the columns, containing $b\cdot w$ ECA of length h. Finally, these two boolean tensors obtained independently are combined using a bitwise XOR function

$$\boldsymbol{X} = \boldsymbol{X}^r \oplus \boldsymbol{X}^c \tag{4.11}$$

where this XOR combination is not an arbitrary choice, it is based on the fact that XOR or XNOR do not change the mean number of high or low states, which is not the case for other logical operations. For example, a bitwise AND/OR operation would effectively

[8]If the image is rotated 90, 180 or 270 degrees the CA evolution contains exactly the same information.

turn 25% high/low states into low/high states, which negatively affects classification results. As regards the discrepancy between the proposed model and the one A. López et al. [182] refer to as a reproduction of it, the difference is that ECA rows and columns are updated separately via (4.10) and joined together via (4.11).

Notice the same method can be generalized to higher dimensions to classify e.g. RGB images, following the same convention used for rows and columns in equation 4.10, and combining both contributions using a multiple input XOR[9] as in 4.11.

The resulting space-time evolution represented by $\boldsymbol{X}$ is coded as an integer by combining states across the pixel bit dimension, from the most to the less significative, denoted by $\boldsymbol{X}'$

$$\boldsymbol{X}' = \sum_{i=0}^{b-1} 2^i X_{:,:,i,:,:} \tag{4.12}$$

which does not require any additional hardware if the bit width dimension is coded in its radix-2 weighted representation. This is illustrated in Fig. 4.10 (Input & iterations). Furthermore, to obtain invariance under small translations and reduce the number of readout model parameters, max-pooling is applied as a feature reduction strategy, also illustrated in Fig. 4.10. Assuming both h and w are multiples of 2 (as is the case of MNIST samples), max-pooling is applied to 2×2 windows with stride 2 and zero padding. Notice this step is a particular (arbitrary) state representation (see Fig. 2.29 in Section 2.3.3.3) borrowed from DL, introduced by [189].

After feature reduction H is applied to CA iterations in the grayscale representation $\boldsymbol{X}'$, the readout model processes it and assigns a label per input sample. A linear model has been chosen, so assume the feature reduction operation returns flattened max-pooling operations, i.e. $H : \mathbb{N}^{m \times (M+1) \times h \times w} \to \mathbb{N}^{m \times \frac{(M+1) \cdot h \cdot w}{4}}$, so that inference is given by (4.13). The full inference process is depicted in Fig. 4.10 for a single MNIST sample.

$$\hat{\boldsymbol{O}} = s\left(H\left(\boldsymbol{X}'\right)\boldsymbol{W} + \boldsymbol{B}\right), \tag{4.13}$$

$$\hat{y}_i = \underset{j}{\operatorname{argmax}} \left\{\hat{O}_{i,j}\right\} \tag{4.14}$$

As a final remark on this particular ReCA model, the same ECA rule is utilized for row and column iterations, for all bit significances, i.e. g_r and g_c in (4.10) are based on the same ECA rule. Then, the search space is reduced to the 256 possible ECA rules.

4.2.4 Methods

At this point, the proposed model has already been defined. So, in this section the training and validation process is described. The goal is to choose a particular (optimal) ECA rule for a specific classification task. Therefore, first a systematic workflow has been defined to find the best option within the model constraints and taking into account additional techniques such as data augmentation (DA). Finally, the hardware based on the best rule is described.

4.2.4.1 Training workflow

The flow chart in Fig. 4.11 summarizes all steps followed to train and validate the model described in previous section. This chart includes several steps that are fixed to

[9] The multiple input XOR operation is equivalent to modulo-2 addition.

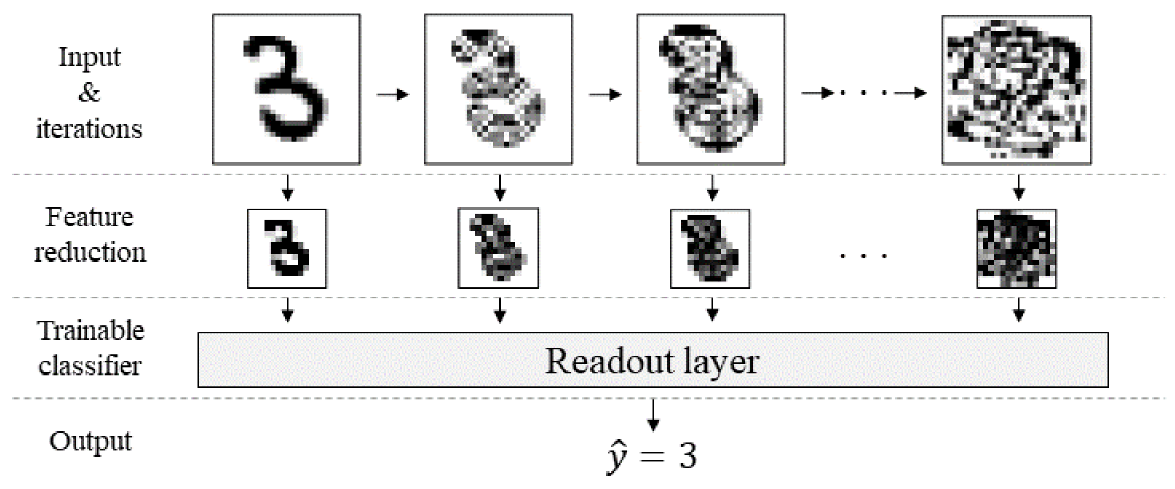


Figure 4.10: Scheme of the proposed classifier applied to a MNIST sample.

specific values or conditions listed in Table 4.8.

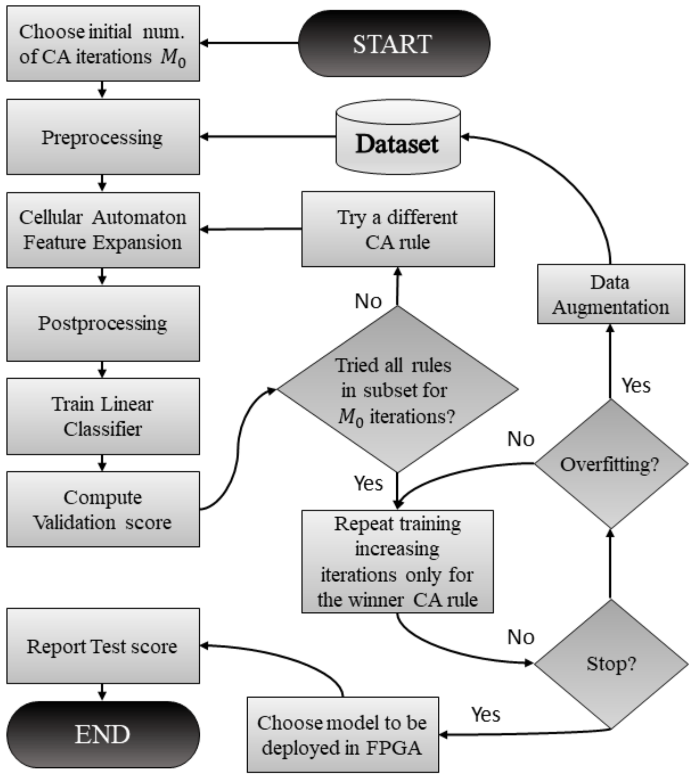

START
Choose initial num. of CA iterations M_0
Preprocessing
Dataset
Cellular Automaton Feature Expansion
Try a different CA rule
Postprocessing
No
Data Augmentation
Tried all rules in subset for M_0 iterations?
Yes
Train Linear Classifier
No
Overfitting?
Compute Validation score
Yes
Repeat training increasing iterations only for the winner CA rule
No
Stop?
Report Test score
Yes
END
Choose model to be deployed in FPGA

Figure 4.11: A flow chart of the proposed systematic procedure to obtain the best model.

The first step is to choose a sufficiently large number of CA iterations M_0 to try for the first time and check the performance for a subset of ECA rules. It is not necessary to try all 256 ECA rules since some of them are symmetric and it does not make sense to try Class 1 ECA rules without transient evolution. Choosing an appropriate value for M_0 is important because it is used to discover which is the winner rule based on some score metric, e.g. validation set accuracy in this case. On the one hand, if M_0 is too small, it is possible to end up with a bad decision, e.g. selecting a Class 1 rule that returns the best results for M_0 iterations, but these results are the same for $M = M_0+1$ iterations because it has converged to an uniform state. On the other hand, if M_0 is too large, overfitting might be a problem. One way to rule out the overfitting problem is to check that the same solution (ECA rule) is reached using one less iteration.

Table 4.8: Design choices for this work.

Flow chart feature	This work
Initial CA iterations M_0	10
Dataset	MNIST
Preprocessing	*None*
CA subset	Combinations of ECA rules given by (4.10) and (4.11)
Feature reduction	Max-pooling (2×2 with stride 2 and zero padding)
Readout model	Maximum entropy classifier
Score metric	Accuracy
Stop condition	$M = 16$
Overfitting condition	No improvement in validation score
Data augmentation method	Elastic distortions ($\alpha_d = 30$, $\sigma_d = 5$)
Data augmentation per step	100% of training set

Once the best rule candidate has been identified, training is repeated for an increasing number of iterations ($M > M_0$) in order to obtain better results at the expense of a more complex model, increasing its size and trainable parameters at each additional CA iteration. However, if as M is increased the model complexity is increased too, there will be overfitting at some point unless the validation score reaches its maximum value, e.g. 100% accuracy. An option to mitigate this problem is to use DA techniques. In this case elastic distortions [190] increase the training set size by a 100% every time it is applied. Nine different elastic distortions for the same parameters are depicted in Fig. 4.12 for illustration purposes. This DA method is controlled by two parameters, one denoted as α_d, which quantifies the amount of displacement per pixel and the other is denoted as σ_d, which is the standard deviation of a Gaussian kernel that is applied to convolve the image. Both parameters are specified in Table 4.8.

4.2.4.2 Hardware inference model

The proposed hardware design is fixed to a single rule since it uses hardwired ECA rules to reduce resource utilization. The implementation has been outlined in a simple manner in Fig. 4.13. Since ECA rules are synthesized with simple logic functions, it is feasible to implement in parallel registers holding the state of each automaton along with the logic necessary to compute the next state. In fact, most logic resources are allocated to implement registers holding ECA states, depicted in green in Fig. 4.13a and 4.13f, resulting in the combined state (4.11), depicted in blue. The raddix-2 weighted interpretation of this combined state, illustrated in Fig. 4.13b, is actually not a logical operation. The reduced state vector is obtained by applying a 2×2 max-pooling operation. Each 2×2 window depicted in Fig 4.13b (or $2 \times 2 \times 8$ in 4.13a) results in a single 8-bit signal representing the reduced state vector component in 4.13c. If the 4 components in a 2×2 window are denoted as a, b, c and d, then each output is given by $\max\{a, b, c, d\}$. So, max-pooling operations are also implemented by

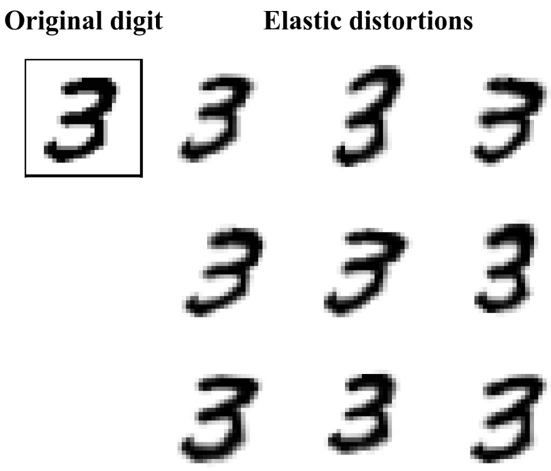


Figure 4.12: Example of 9 elastic distortions applied to an MNIST digit using $\alpha_d = 30$ and $\sigma_d = 5$.

relatively simple logic functions[10] and therefore implemented in parallel. In contrast, the logits contribution (Fig. 4.13d) is not computed with fully parallel logic since it would require too many multipliers. It has been serialized using 40 multipliers (four multipliers per MNIST class) as well as additional control logic. The results are then accumulated in registers storing the logits iteration by iteration (Fig. 4.13e), so that the results are valid when all iterations are done.

In addition, as documented in the results section below, the best choice is Rule 90 and is the one implemented in this work. Each row or column that can be updated with this rule, implemented by the Rule 90 Processing Unit (R90PU), described in Fig. 4.14. The R90PU integrates an L-bit register and simple boolean logic. Since the rule is 0.01011010_2, it does not include self interaction and returns a high state when left and right nearest neighbors are different, so that the next state is obtained with an XOR logic function of the nearest neighbors.

The digital design has been implemented with a VHDL code defining three $28 \cdot 28 = 784$ arrays of 8-bit registers: the first to store initial state, the second to store row-wise iterations and the third to store row iterations, requiring a total of 2352 (8-bit) registers for these arrays. Since it is also necessary to store each 14×14 (8-bit) reduced state vector, as well as 10 (8-bit) registers for the logits contribution and 10 (16-bit) additional registers to store inference results. All of them add up to a total of 2578 (8-bit) registers. Control and accumulation hardware resources are negligible compared to rule 90, max-pooling and multiplier logic. Parallel CA implementation requires $2 \cdot 8 \cdot 28 = 448$ R90PUs, with 26 XOR gates per R90PU and $8 \cdot 784 = 6272$ additional XOR gates for bit-wise combination of row and column CA. Adding up all gates related to this part results in a total of $17,920$ XOR gates. Each 2×2 parallel max-pooling requires 3 (8-bit) comparators and 3 (16-to-8) multiplexers, i.e. 588 (8-bit) comparators and 588 (16-to-8) multiplexers are needed (and described in VHDL) for the max-pooling applied to the whole grayscale image. Finally, since multipliers are more expensive in terms of area, it is not feasible to implement all of them in parallel (each iteration requests 1960 MAC operations), so only 40 MACs are implemented in

[10]At least much simpler than multipliers.

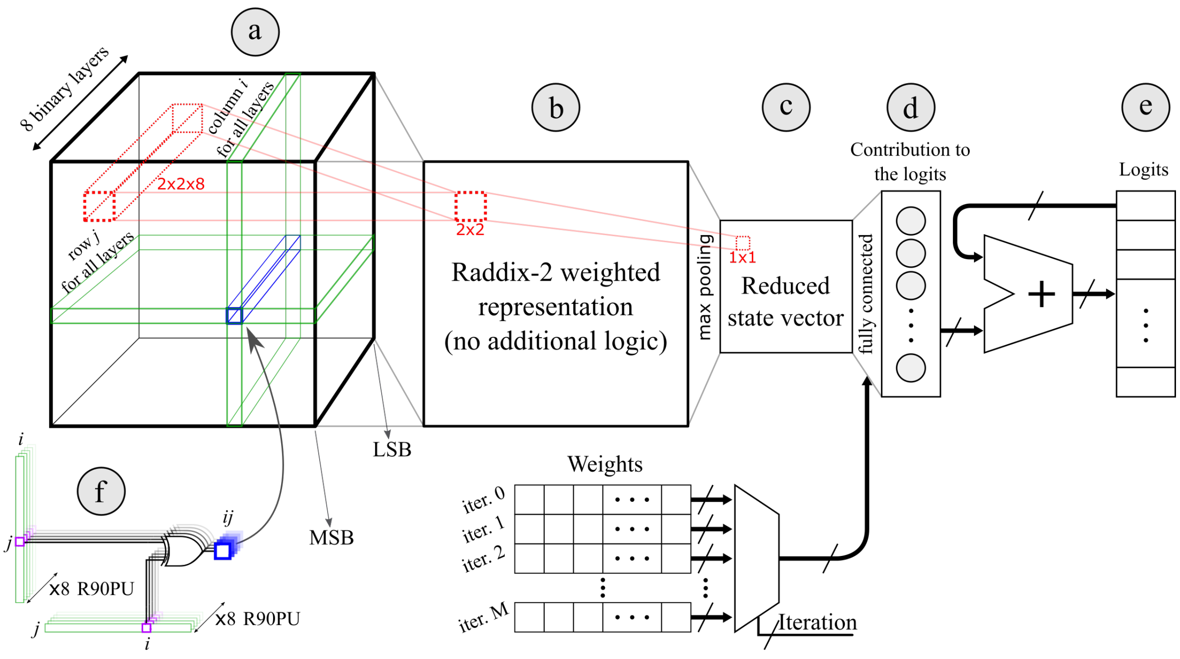


Figure 4.13: A scheme of the hardware implementation of the proposed classifier. (a) 3D boolean tensor representation of 2D grayscale data, this tensor representation is organized in 2D layers from the most significative bits (MSB) to the less significative bits (LSB) of the grayscale image. (b) The 8 binary layers are interpreted as unsigned integers. (c) Max pooling filter applied to the raddix-2 representation of the current iteration. (d) Contribution to the logits is computed using the pre-stored weights obtained offline. (e) Current contribution to the logits is accumulated until the final iteration is reached. (f) Rows and columns are iterated independently using rule 90 and combined to obtain an updated 3D boolean tensor using a bitwise XOR operation. This process is repeated until the final iteration is reached. Once the contibution of the final iteration is accumulated, the logits are valid data.

parallel and iterated 49 times per iteration using few additional control logic. Notice using 40 multipliers is an arbitrary choice since 40 is the maximum number less than 112 by which 1960 is divisible, with 112 being the number of available DSP blocks in the selected FPGA.

4.2.5 Results

As described in the previous methods section, the first step is to discover which is the best ECA rule candidate to fit the MNIST dataset. This process is carried out using $M_0 = 10$, which is sufficiently large to avoid vanishing and periodic dynamics after the

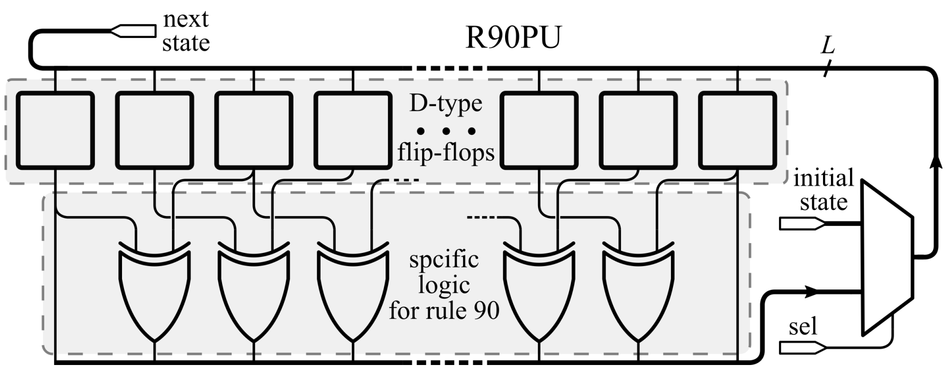


Figure 4.14: Digital scheme of the rule 90 processing unit (R90PU). The present state is indexed from 1 to L ($L = 28$ in the case of the MNIST digits for both height and width automata) and rule 90 is implemented by computing the XOR of the nearest neighbours except for the first and last elements (for which we consider fixed boundary conditions). The state register is composed by synchronous D-type flip-flops with an enable signal.

transient evolution, and it is sufficiently low to avoid overfitting. After following all steps indicated in Fig. 4.11, the final inference model is retrained to obtain low precision two's complement parameters. Finally, the test accuracy is verified and additional results related to the FPGA implementation are reported too.

4.2.5.1 Software exploration

A first screening through ECA rules using fixed boundary conditions[11] was done, excluding symmetrical (both left/right and color symmetries) and trivial Class 1 rules without transient states. The top-30 ECA rules with lowest validation error are reported in Fig. 4.15, indicating the corresponding training errors as well. These results present a clear relationship with the corresponding class of ECA rule:

- **Class 1**: Once a uniform state is reached, adding more iterations does not improve precision. The only Class 1 rule with a large enough transient evolution before reaching the uniform state that appears in the top-30 is rule 168. It is an exception providing better results than three of the four Class 4 rules included in the search, i.e. it is better than rules 41, 106 and 110, which are not even in the top-30, but it provided slightly worse results than rule 54. However, in general Class 4 rules outperform Class 1 rules.

- **Class 2**: There are two different cases for this class of ECA rules. The first case is similar to Class 1, once the evolution reaches periodic[12] dynamics with short periods, i.e. period-1 or period-2, the accuracy is no longer improved as the number of iterations increase, e.g. rule 44 with 5.8% associated validation set error. The second case are ECA rules that have higher precision and reach periodic dynamics too. In this case the maximum period is on the order of the automaton length ($L = 28$) and the number of transient states is larger compared to previous cases, e.g. rule 14, the fourth best result in Fig. 4.15.

- **Class 3**: Since each automaton is finite, at some point the behaviour will become cyclic, but the period is much higher than Class 2 ECA rules and the evolution appears to be random. In addition, dynamics are more complex than in previous cases, so that finding linear dependencies between the the new iteration and linear combinations of previous ones is less likely. Thus providing potentially higher accuracy, e.g. top-3 best results: rules 90, 126 and 18.

- **Class 4**: As for Class 3 ECA rules, since each automaton is finite, the evolution must become cyclic after some large period. These rules form areas of repetitive structures that interact in complicated ways. Therefore, it is possible to obtain linearly independent new iterations for a long period, but it does not need to be the base for each new iteration, since repetitive structures, despite not being exactly the same throughout iterations, they may have certain linear dependence with previous rules due to aligned structures. For this reason, direct application of Class 4 rules do not provide better results compared to Class 3 and some Class 2 rules.

[11]Since no difference in terms of accuracy has been observed when comparing fixed and periodic boundary conditions, the actual implementation was done using the former.

[12]Periodic includes period-1, which is also counts as Class 2 ECA, it is different from Class 1 because uniform means that all states are either high or low.

Interestingly, most of the top-30 better validation results are Class 2. However, 10 of them are Class 3 and there are 11 Class 3 ECA rules discarding symmetries. Unlike chaotic rules, periodic rules more common, there are a total of 65 but only 19 of them are in the top-30. In fact, the most remmarkable result is that the top-3 rules are chaotic. From these results, rule 90 is selected to be implemented in hardware since it is the one with lowest validation error. Moreover, it has a fairly simple digital implementation, already discussed in the methods section.

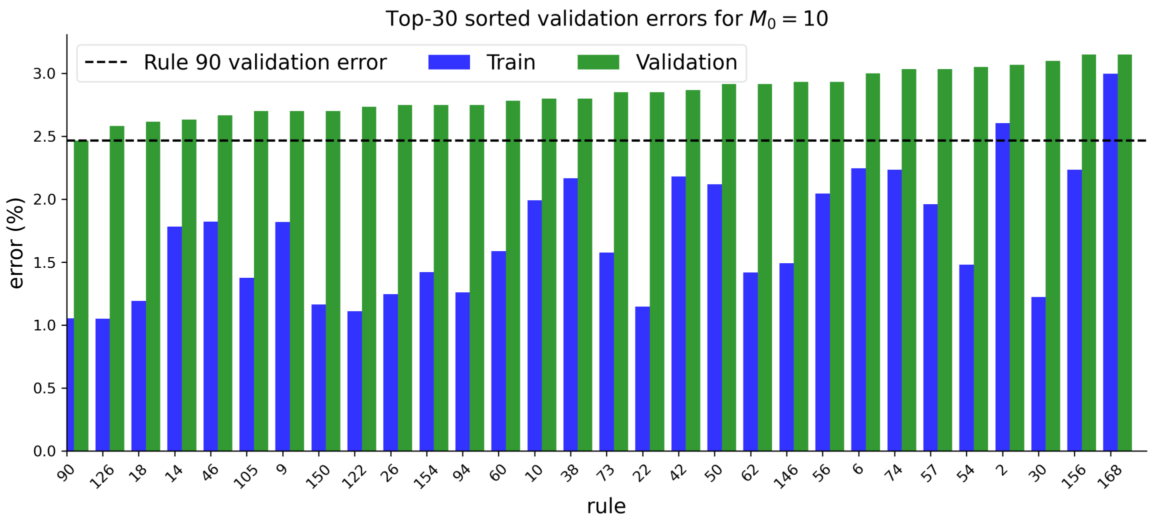


Figure 4.15: Performance of the proposed architecture using $M_0 = 10$ iterations of different, non-symmetric elementary cellular automata rules. Rules with vanishing or period 2 and temporal evolution without transient state are not shown in this figure.

As indicated by the Fig. 4.11 flow chart, after checking different ECA rules and selecting rule 90, the training process continues for an incremental number of iterations in order to improve accuracy. Fig. 4.16 shows training (blue circles), validation (green triangles) and test (red stars) results for up to $M = 16$ iterations. Notice different background tones represent different DA stages, so that at the beginning, and until $M = 10$ (w/o DA), the training set is not augmented. Then data augmented twice, first for $M = 11$ (DA $\times 2$) and then for $M = 14$ (DA $\times 3$). Therefore, each Fig. 4.16 region has a different training set size; from left to right: $55,000$, $110,000$ and $165,000$, respectively.

Therefore, the digital design depicted in Fig. 4.13 accounts for 16 iterations of rule 90 with hardwired readout weights. The difference between the hardware and software models is the weight bit width and format. While in the software case weights are in 32-bit floating point precision, in the digital design these weights have been restricted to 8-bit two's complement numbers. Despite this, a very similar accuracy has been achieved by re-training the model with direct fake quantization after each gradient descent iteration, using Adam optimization technique (see Appendix A). Finally, the model evaluation on the test set reported 1.92% classification error in the test set. Hyperparameters related to Adam optimization are listed in Table 4.9. In this context, it is important to highlight that more accurate and advanced QAT approaches exist in the literature, which is further explored for SC CNNs in Chapter 5. However, for this shallow readout model direct fake quantization after each optimization step has been sufficient to recover floating-point results.

An example optimization process is presented in Fig. 4.17, showing the error for

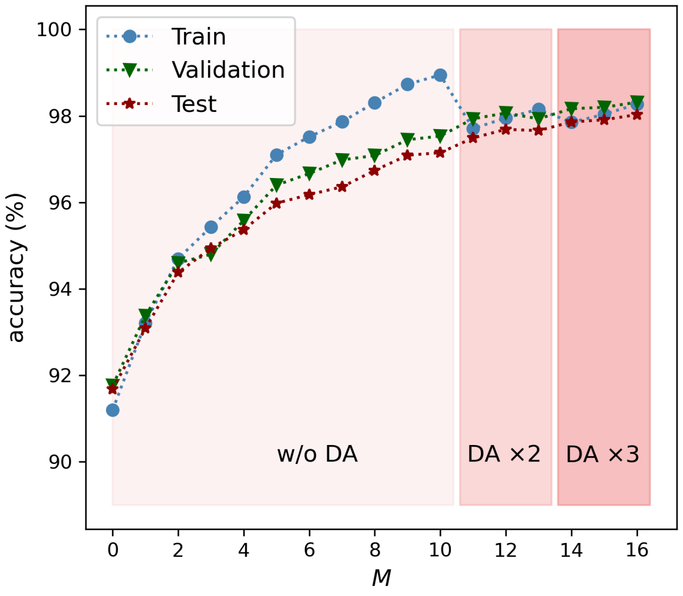


Figure 4.16: MNIST training, validation and test accuracy, using rule 90 as described by (4.10) and (4.11), adding 55000 additional training samples when needed, illustrated by the filled rectangles. (w/o DA) without data augmentation. (DA ×2) 110000 total training samples. (DA ×2) 165000 total training samples. The regularization parameter is $C = 0.04$ and the maximum number of limited memory BFGS iterations is fixed to 1000.

each training mini batch (blue) as well as the whole validation set (green) as a function of the number of elapsed mini batch steps. After a large number of steps, poor performance peaks might appear due to 8-bit weight restrictions. However, by stopping the training process when the validation accuracy is equal or higher than the one obtained in Fig. 4.16 for $M = 16$, the obtained test results are almost the same.

4.2.5.2 FPGA metrics

Additionally, the present implementation has been compared to other FPGA research works implementing popular CNN models in terms of accuracy, latency, maximum performance, power, efficiency, power-delay product, logic utilization and DSP blocks, which are listed in Table 4.10. *Accuracy* refers to test set classification accuracy, with

Table 4.9: This table specifies model hyperparameters for the 8-bit weight model. (a) In reality there are 17 because the initial state is included. (b) Implemented using Tensorflow's AdamOptimizer class with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. (c) Mini batches are chosen at random for each optimization step.

Hyperparameter	Value
ECA rule	90
CA iterations[a] (M)	16
Optimization method	Adam[b]
Learning rate	0.00012
Batch size[c]	17,000
Distortions per image	3

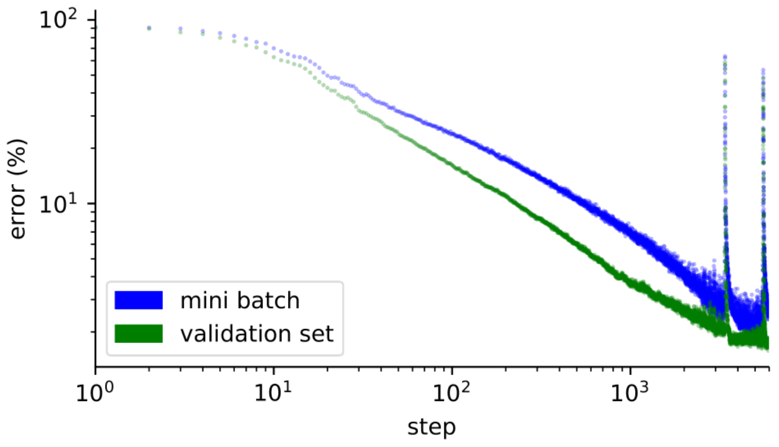


Figure 4.17: Example training error evolution obtained using direct fake quantization.

(w/) and without (w/o) DA. *Clock freq.* is the global clock frequency, which might change depending on the development board or whether the implementation is making use of a phase-locked loop (PLL) block. *Latency* is the time required to compute a single inference. *Max. performance* is reported as the the number of kiloinferences per second (KIPS) normalized by the global clock frequency. *Power* is the thermal power dissipation related to the inference model implementation. *Efficiency* is the number of kiloinferences (KI) per Joule invested. *Logic utilization* is the number of FPGA building blocks needed, reported either in terms of LUT and FF or ALM, which depends on vendor's hardware architecture and design software. *DSP blocks* is related to the number of required multiplier or MAC units, e.g. in this case the FPGA is making use of 40 DSP blocks because the design needs 40 parallel MAC blocks. Notice the proposed implementation is faster than most conventional CNN implementations and reports competitive accuracy while maintaining low power consumption and resource utilization.

Table 4.10: Metrics extracted from the proposed FPGA implementation using 8-bit precision weights compared to some previous works.

Model	**Accuracy (%)**		**Clock freq.**	**Latency**	**Max. performance**	**Power**	**Efficiency**	**Power-delay**	**Logic**	**DSP**
	w/o DA	**w/ DA**	**(MHz)**	**(ms)**	**(KIPS/MHz)**	**(W)**	**(KI/J)**	**product (mJ)**	**utilization**	**blocks**
Ref. [191]	99.52	-	100	4-6 (approx.)	0.002	-	-	-	36.4K LUT + 41.1K FF	8
Ref. [192]	98.62	-	100	26.37	0.00038	-	-	-	14.8K LUT + 54.1K FF	20
Ref. [193]	98.32	-	150	0.0034	1.96	26.2	11.22	0.0891	182.3K ALM	20
This work	97.10	98.08	50	0.020	1.00	0.289	173	0.00578	22.6K ALM	40
Ref. [194]	96.80	-	150	0.0254	0.262	-	-	-	51.1K LUT + 66.3K FF	638
Ref. [195]	96.33	-	100	0.924	0.011	-	-	-	16.1K LUT + 6K FF	12

4.2.6 Summary

A simple memoryless ReCA model composed by many ECA implementing the same rule has been introduced and evaluated for the MNIST dataset. A careful analysis based on a custom workflow is described in Fig. 4.11, indicating the best model variant is the one based on rule 90, achieving 1.92% test set classification error on FPGA implementation. Moreover, this digital design is compared to other works, all related to CNN FPGA implementations for the MNIST dataset. From the results reported in Table 4.10 it is concluded that the proposed implementation achieves competitive accuracy, low latency, very high energy efficiency and relatively low resource utilization compared to other works. Although the obtained accuracy is still not as good as that of state-of-the-art floating-point models, it represents the first proof of concept for the successful applicability of ReCA systems in image classification, both at software and hardware level. Moreover, its energy efficiency makes it a perfect candidate for edge applications, which usually require relatively low area and high energy efficiency. Further work in this line might be related to simple RGB image classification, such as CIFAR-10 [196], or time dependent data, such as ECG [197] or EEG [198]. Another option is to consider higher dimensional CA and/or larger CA neighbothood, performing weight binarization [199], substituting the readout by a random forest with binary features [182], or even including unconventional arithmetic hardware based on SC in the readout layer to reduce the number of logic elements and DSP blocks for larger images while maintaining a fully parallel architecture.

Chapter 5

Stochastic Computing Implementations

In the previous chapter, simplifications applied to the inference algorithm were proposed and studied in order to enable highly parallel digital implementations, preserving the traditional fixed-point logic. In contrast, in the present chapter, what is proposed is not to simplify the inference model itself, but the way in which arithmetic operations are implemented. For this purpose, SC elements are implemented, which are already introduced in Chapter 2. It allows implementing common arithmetic operations with low-cost hardware compared to its fixed-point counterpart, even including potential benefits in terms of energy efficiency in the case of multiplication operations, which are at the core of most ML and DL inference algorithms.

Therefore, the implementation of SC systems mimicking or approximating fixed-point operations of the original algorithm, together with simplifications to reduce the required parameter accuracy in the model, allows such a system to be highly or even fully parallel.

As far as the content of this chapter is concerned, two well-known algorithms implemented with SC elements are studied. The first is the RBF-NN, for which we investigate how to reduce the error associated with the autocorrelation of stochastic bitstreams using a RNG, and also introducing a new type of APC and radial basis activation function. Secondly, the SC CNN implementations and more specifically the SC LeNet-5 [16] is discussed, since it is one of the most widely used CNNs for the MNIST task. However, it should be noted that in the latter case the hardware implementation has been done by C.F. Frasser and our contribution is related to the deployed weight quantization technique. In addition, two possible mathematical models for mapping a conventional CNN to SC are presented.

Moreover, throughout this chapter several RNG methods utilized for bitstream generation are utilized, including on-chip (LFSR and rng_n1024_r32_t5_k32_s1c48) ROM based approaches, documented in Appendix C.

5.1 Radial Basis Function Neural Networks

5.1.1 Contribution

There are three main contributions related to the proposed SC RBF-NN. First, there is a contribution to the SC paradigm in general, being the most remarkable a new APC design with non-local and spatially extended memory, which returns an output bitstream. Second, the classical LFSR utilized to generate pseudorandom bitstreams is substituted by a high quality RNG to avoid autocorrelation effects in the SC blocks. Third and most remarkable, a parallel SC RBF-NN design is implemented and tested on FPGA, taking advantage of the other two contributions. Results are compared to the equivalent fixed-point software version and simulated SC version in which activations are exactly computed.

5.1.2 Related work

This is not the first time an RBF-NN is implemented using SC arithmetic since Y. Ji et al. already proposed a fully parallel SC architecture for RBF-NNs [200]. However, the contribution described in this work is differs from it in several aspects.

First, Y. Ji et al. architecture exploits the fact that a product of exponentials results in the sum of its exponents, i.e.

$$\prod_i e^{-p_{d;i}^2} = e^{-\sum_i p_{d;i}^2} \tag{5.1}$$

which is an smart approach to save resources since SC multiplication (AND gate with multiple inputs) is much simpler than addition (APC or MUX). However, this computation is accurate only if the bitstreams representing $p_{d;i}^2$ are uncorrelated. One might think if there were as many different random numbers as there are $p_{d;i}^2$ components, then the problem would be solved. Although this solution can solve the problem for a reduced number of inputs, implementing many RNGs in parallel can be costly in terms of logic resources. Also, if the number of inputs is huge, there might be partially correlated bitstreams, increasing the error in the final result. This issue arises because each bitstream representing $p_{d;i}^2$ is obtained from the XOR between a pair of correlated signals, which feeds a FSM to compute the bitstream representing each $e^{p_{d;i}^2}$ by exploiting self decorrelation, resulting in signals with common self-correlation patterns induced by such FSM. Also, for a high number of inputs, it is more likely to introduce correlations if one is using e.g. LFSRs as RNGs. To avoid this issue, in this work Euclidean distance bitstreams are computed explicitly and then pass through an RBF activation, using a single shared RNG.

In addition, the RBF activation is computed by a feed-forward digital circuit instead of a 2-dimensional FSM and the proposed design is applied to several pattern recognition tasks, including the MNIST, which is significantly higher dimensional than the tests done in [200].

5.1.3 Theoretical foundations

The main idea is to take the RBF-NN model introduced in the Background chapter (Section 2.3.3.1) as starting point and then describe an equivalent inference model based on SC. As a remainder, recall the NN has only one hidden layer with pattern-matching units based on Euclidean distances followed by RBF activations. So, the activation on the j-th squared Euclidean distance due to the i-th input sample is given by (5.2).

$$D_{i,j} = \|X_{i,:} - W^h_{:,j}\|^2 \tag{5.2}$$

So, hidden activations are given by (5.3), where it is conveniently assumed that each RBF activation is given as a function of the squared distance, which makes the SC model notation easier. Also, the simplest model case is assumed, i.e. $\gamma_j = \gamma \; \forall_j$.

$$A_{i,j} = e^{-\gamma_j D_{i,j}} \tag{5.3}$$

These hidden activations are linearly combined to generate outputs (5.4). Then, for classification tasks the predicted label is given by (5.5).

$$\hat{\boldsymbol{O}} = s\left(\boldsymbol{A}\boldsymbol{W}^o + \boldsymbol{B}\right), \tag{5.4}$$

$$\hat{y}_i = \underset{j}{\operatorname{argmax}} \left\{ \hat{O}_{i,j} \right\} \tag{5.5}$$

Also, remember the parameters $\boldsymbol{W}^h$, $\boldsymbol{W}^o$ as well as the hyperparameters K and γ are obtained offline. First the prototypes $\boldsymbol{W}^h$ are obtained via the K-means algorithm. Then, prototypes are quantized and used to obtain the activations. Once this is done, the readout part is trained taking into account that the objective is to obtain a fixed-point inference model. For this purpose, training is performed using direct fake quantization as in Section 4.2.5.

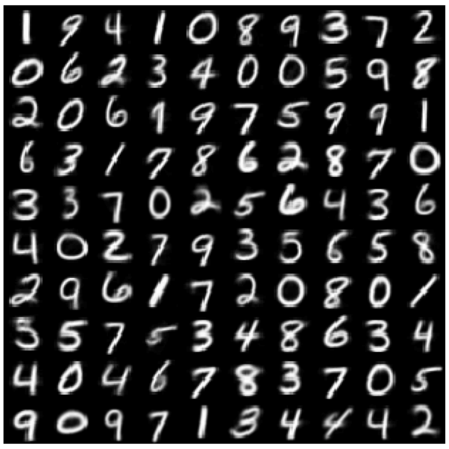


Figure 5.1: Example prototypes (100 out of 1023) obtained using K-means algorithm on the MNIST training set.

5.1.3.1 An equivalent unipolar/bipolar SC model

If input data samples $\boldsymbol{X}$ are normalized in the unit range, these features can be expressed as activation probabilities $\boldsymbol{P}_X = \boldsymbol{X}$. Since prototypes must be represented in the same scale as input data because $\boldsymbol{X}$ and $\boldsymbol{W}^h$ represent the same quantities, then prototypes can also be expressed as activation probabilities $\boldsymbol{P}_{W^h} = \boldsymbol{W}^h$. Therefore (5.2) and (5.3) are equal in terms of activation probabilities. However, intermediate

SC bitstreams representing squared Euclidean distances are normalized by the number of inputs, so that (5.2) is conveniently rewritten as:

$$P_{D;i,j} = \frac{\|P_{X;i,:} - P_{W^h;:,j}\|^2}{d} \tag{5.6}$$

where d is the number of input features and $\boldsymbol{P}_D$ represents an activation probability equal to the squared Euclidean distances scaled by the number of inputs, so that the results are in the unit range. Which implies that (5.3) can be rewritten as:

$$P_{A;i,j} = g\left(P_{D;i,j}\right) \approx e^{-\gamma_{eff} P_{D;i,j}} \tag{5.7}$$

where g is a custom RBF activation function and $\gamma_{eff} = \gamma d$, so that $\boldsymbol{P}_A \approx \boldsymbol{A}$. Up to this point, everything is interpreted in the unipolar SC coding. The corresponding SC operations required for (5.6) are:

$$\tilde{S}_{i,j,:}(t) = \left(\tilde{X}_{i,:}(t) \oplus \tilde{W}_{:,j}(t)\right) \cdot \tilde{S}_{i,j,:}(t-T) \tag{5.8}$$

where $\tilde{\boldsymbol{S}}(t)$ contains bitstreams representing each squared difference needed to compute Euclidean distances and inputs and prototypes must be maximally correlated, i.e. generated by the same random sequence, so that:

$$\tilde{X}_{i,j} \parallel \tilde{W}^h_{j,k} \; \forall_{i,j,k} \tag{5.9}$$

which means pairs of components related to the same feature index are correlated. In this implementation all components share the same random number for simplicity. From this expression, scaled Euclidean distances could be computed as:

$$P_{D;i,j} \approx \frac{1}{Nd} \sum_{n=0}^{N-1} \sum_{k=0}^{d-1} \tilde{S}_{i,j,k}(nT) \tag{5.10}$$

however, intermediate domain conversion is avoided by using APC blocks which directly return bitstreams $\tilde{\boldsymbol{D}}(t)$.

Next, the RBF activations (5.7) are computed by a more complex set of operations. The particular implementation might be either programmable or hardwired and performs the following parametrized computations for input and output bitstreams denoted as $\tilde{x}(t)$ and $\tilde{y}(t)$, respectively.

$$a(t) = \sum_{n=0}^{15} \tilde{x}(t - nT), \tag{5.11}$$

$$\tilde{a}_i(t) = \mathcal{H}\left(a(t) - \alpha_i\right), \;\; i = 0, 1, 2, 3, \tag{5.12}$$

$$\tilde{y}(t) = \max\left\{\max\left\{\max\left\{\tilde{a}_0(t)\tilde{c}_0(t), \tilde{a}_1(t)\overline{\tilde{c}_0(t)}\right\}\tilde{c}_1(t), \tilde{a}_2(t)\overline{\tilde{c}_1(t)}\right\}\tilde{c}_2(t), \tilde{a}_3(t)\overline{\tilde{c}_2(t)}\right\} \tag{5.13}$$

where $\tilde{c}_i$ are coefficient bitstreams. Notice (5.13) might seem complicated but is composed by three 2-to-1 multiplexers and its implementation is further discussed after the SC model formulation. Ideally, if $a(t)$ is computed from a much larger line buffer it would be constant in time and proportional to the input activation p_x. So that $\tilde{a}_i(t)$ in (5.12) would be constant too and integration in time would result in piecewise function, result of the superposition of four different step functions. However, the input line buffer is finite, which introduces some standard deviation in $a(t)$ and smoothens the boundaries between active and inactive regions in each step function. Therefore, integration in time of the output $\tilde{y}(t)$ would return a weighted superposition of smooth step functions and its shape depends on coefficients α_i and c_i.

As regards the readout part, it can be implemented using either bipolar, sign-magnitude or extended representations. Since the readout layer is fine-tuned using intermediate activations extracted from the actual hardware implementation, it is not necessary to work with high precision parameters, which would also contribute to increase logic resources. So, the SC extended representation is discarded. Also, there are two main reasons to choose bipolar instead of sign-magnitude coding. First, in the last layer small errors due to variations near the bipolar zero (50% activation probability) are not be propagated further. Second, sign-magnitud is a good choice to reduce evaluation time by a half at expenses of slightly higher resource utilization, which does not make sense in this case because the readout is computed in parallel and less overall evaluation time would impact precision because hidden (unipolar) activations would have less evaluation time too. Therefore, only the bipolar readout layer architecture has been considered. That is:

$$P^*_{\hat{O};i,j} = \frac{1}{K}\sum_{k=0}^{K} P^*_{A;i,k} P^*_{W^o;j,k} \approx \left(\frac{1}{NK} \sum_{n=0}^{N-1} \underbrace{\sum_{k=0}^{K}}_{\text{APC}} \underbrace{\overline{\tilde{A}_{i,k}(nT) \oplus \tilde{W}^o_{k,j}(nT)}}_{\text{bipolar product}} \right)^* \tag{5.14}$$

where $k = 0$ is reserved to the bias, i.e. $P_{A;:,0} = \mathbf{1}$, and bitstream pairs must be maximally uncorrelated

$$\tilde{A}_{i,j} \perp \tilde{W}^o_{j,k} \; \forall_{i,j,k} \tag{5.15}$$

In practice all readout weight bitstreams are generated from the same random number, which is different from the one used to generate input and prototype bitstreams.

5.1.3.2 Hardware description

In order to explain how previous operations are described by parallel digital logic circuits, several small block diagrams are introduced to describe RBF and linear units. Finally, these building blocks are put in the context of the circuit describing the RBF-NN.

Each scaled Euclidean distance is computed by adding d-dimensional groups of squared subtractions, as described by (5.8). The SC design utilized to obtain squared differences for a pair of bitstream vectors is illustrated in Fig. 5.2. Then, each group of d squared differences are inputs to a new APC block, as a result it returns a bitstream with the average activation probability.

As regards the APC design, a similar idea was already introduced in Section 2.2.3 for the SAPC (Fig. 2.13). The general case is depicted in Fig. 5.3. Notice when the threshold value is equal to a power of 2, the SAPC can be simplified by substituting the comparison block by simply choosing the MSB. In order to increase the bandwidth, for large input sizes d, smaller SAPCs are connected in a feed-forward tree structure,

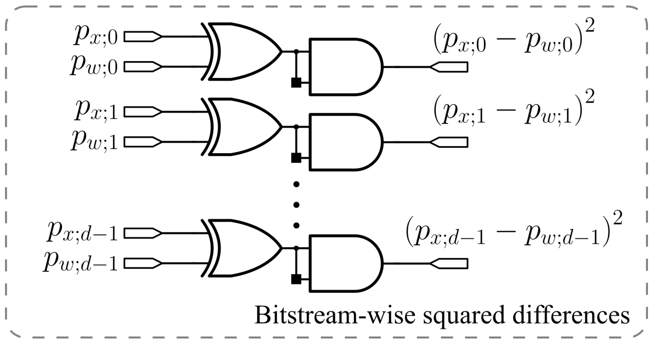


Figure 5.2: Bitstream-wise squared differences block diagram. It operates on two bitstream vectors of size d and requires correlated input pairs, so that each of the $2d$ input pulses might be generated from the same random sequence.

see e.g. Fig. 5.4, which describes an SAPCN for 32 input bitstreams. This approach is necessary to avoid timing issues with large critical paths when the number of input bitstreams is large as in the case of MNIST ($d = 784$). The main reason for this is the use of resource optimized parallel counters composed by full and half adders, which increase the critical path length as the number of inputs increases.

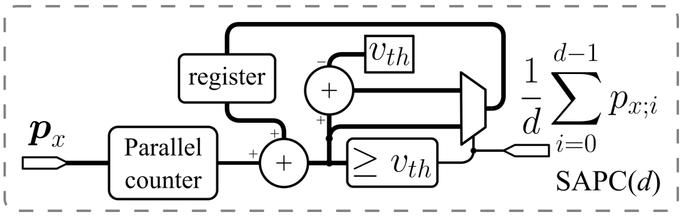


Figure 5.3: SAPC block diagram for an arbitrary number of input bitstreams.

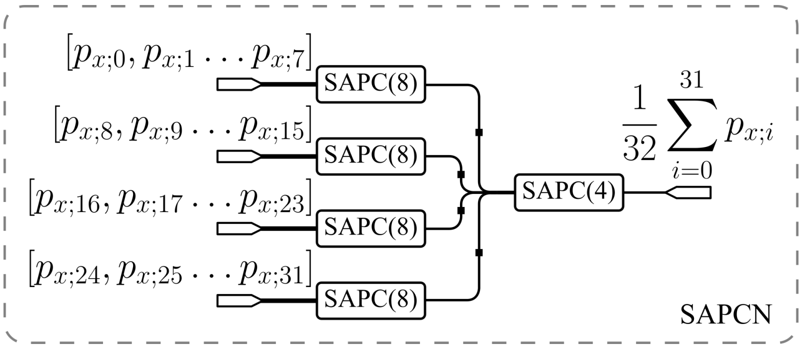

$[p_{x;0}, p_{x;1} \cdots p_{x;7}]$
SAPC(8)
$[p_{x;8}, p_{x;9} \cdots p_{x;15}]$
SAPC(8)
$[p_{x;16}, p_{x;17} \cdots p_{x;23}]$
SAPC(8)
$[p_{x;24}, p_{x;25} \cdots p_{x;31}]$
SAPC(8)
SAPC(4)
$\frac{1}{32}\sum_{i=0}^{31} p_{x;i}$
SAPCN

Figure 5.4: Example SAPCN block diagram for 32 input bitstreams.

After each scaled squared Euclidean distance is obtained, the RBF activation is computed by the Fig. 5.5 circuit, which implements logic operations in (5.11), (5.12) and (5.13). Notice this circuit needs to be parametrized correctly depending on the required γ_{eff} (see (5.7)).

At this point we have all the necessary ingredients for the RBF unit and still need to define the hardware required to implement the bipolar readout linear unit. The bipolar multiplications described by (5.14) in the readout layer are calculated through several parallel circuits such as the one shown in Fig. 5.6. Then, linear readout bitstreams are obtained using SAPCNs too, and bitstream-wise multiplications (Fig. 5.6) instead of squared differences as inputs (Fig. 5.2), without further activations.

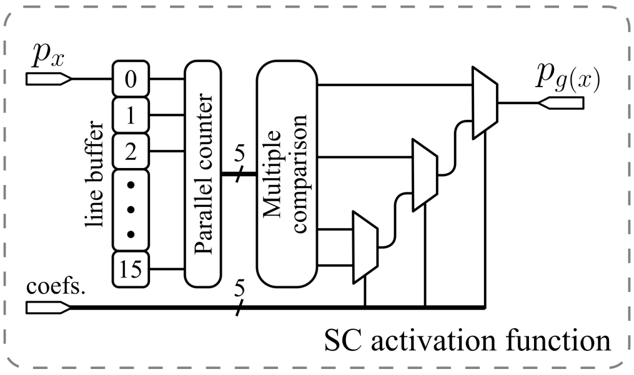


Figure 5.5: SC RBF activation block diagram.

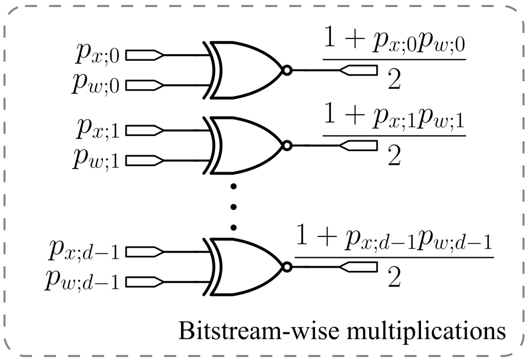

$p_{x;0}$
$p_{w;0}$
$\frac{1+p_{x;0}p_{w;0}}{2}$
$p_{x;1}$
$p_{w;1}$
$\frac{1+p_{x;1}p_{w;1}}{2}$
$p_{x;d-1}$
$p_{w;d-1}$
$\frac{1+p_{x;d-1}p_{w;d-1}}{2}$
Bitstream-wise multiplications

Figure 5.6: Bitstream-wise multiplications block diagram. It operates on two bitstream vectors of size d and requires uncorrelated input pairs, so that each of the $2d$ input pulses might be generated from a different, uncorrelated random sequence.

Both SC RBF and linear readout units are constructed by the components described above, as illustrated in Fig. 5.7. So the next level of abstraction in the RBF-NN design already provides a global view of the implemented circuit, as shown in Fig. 5.8, which also includes input and output registers as well as domain conversion components. In addition, there is a global counter for setting up the number of integration steps and a flag rises when the inference is done.

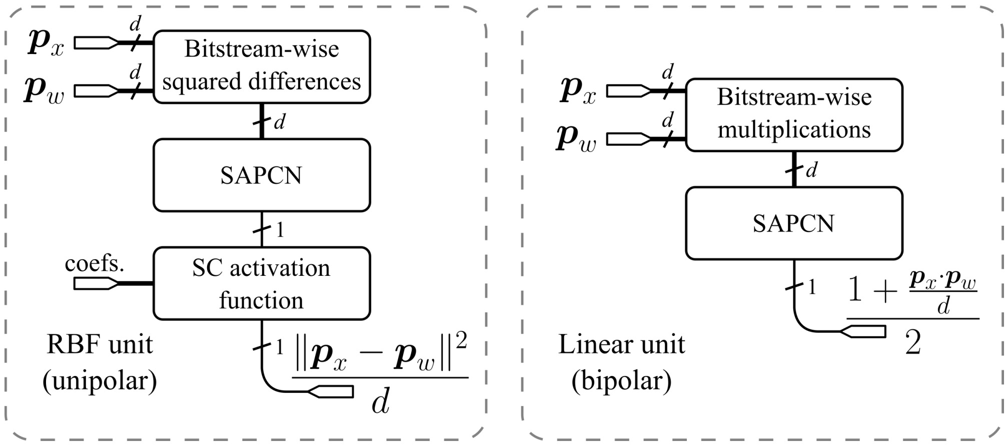

$\boldsymbol{p}_x$
$\boldsymbol{p}_w$
d
Bitstream-wise squared differences
SAPCN
1
coefs.
SC activation function
RBF unit (unipolar)
$\frac{\|\boldsymbol{p}_x - \boldsymbol{p}_w\|^2}{d}$
Bitstream-wise multiplications
Linear unit (bipolar)
$\frac{1+\frac{\boldsymbol{p}_x \cdot \boldsymbol{p}_w}{d}}{2}$

Figure 5.7: High level block diagram description of SC RBF and linear units utilized in the SC RBF-NN design.

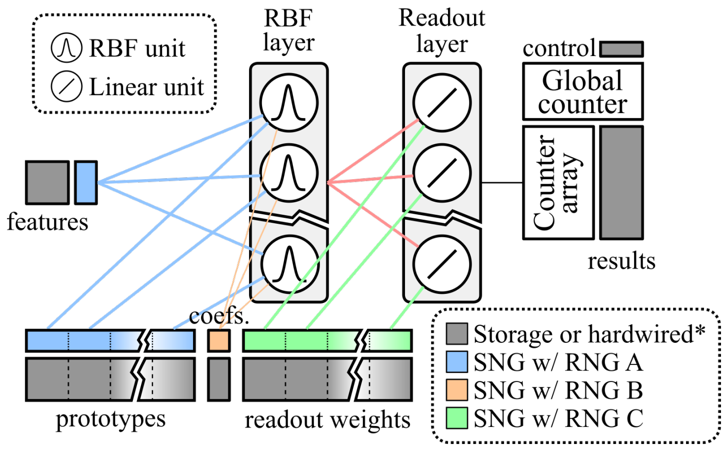


Figure 5.8: Fully parallel RBF-NN block diagram description. Wires with the same color coming from SNG arrays represent indicate whether the resulting bitstream vectors with maximally correlated components, i.e. different colors indicate a different random number driving the SNG array. (*) The fully parallel FPGA implementation utilizes hardwired prototypes, coefficients and readout weights to save logic resources.

5.1.4 Methods

The goal is to replicate the RBF-NN inference model using SC arithmetic in the form of a fully parallel design as shown in Fig. 5.8. Due to the relevant role of autocorrelation in random number generation, we pay special attention to its impact on distance calculations. Therefore, the first step has been the choice of an RNG, comparing the LFSR with another one that has better statistical properties.

The most common option for on-chip random number generation in SC designs is based on the generation of maximum length sequences, mainly with LFSRs. Most SC applications consist of replacing conventional multiplications by a few AND (unipolar) or XNOR (bipolar) logic gates and the multiplications are usually between pairs of different quantities and less frequently for squaring operations. This means that a good seeds' choice for a pair of LFSRs minimizes the error made, approaching the computational accuracy of the equivalent fixed-point operation. In the case of the SC square operation, what is done is to delay in time the corresponding bitstream.

The reader might think a possible solution is to create a second bitstream from the same quantity, which is generated from a different seed. However, this would involve a large amount of additional logical resources and in our case this is not possible since in our case bitstreams do not come directly from SNGs (5.2, square operations occurr after absolute value difference operations). Another possible solution is to incorporate a seed choice based on several clock cycle delays instead of just one, so that the hardware overhead is feasible as long as the required delay is not too high. More delay clock cycles results in more D-type flip-flops, one per each squaring operation and per additional clock cycle delay. Finally, if this solution is not satisfactory, one can change the RNG by another one that presents much less autocorrelation from one cycle to the next, as done in this study.

Before changing the LFSR to another RNG, it was analyzed whether it would make sense to use more than one delay cycle in the circuits implementing the squaring operations. Several numbers of delay cycles were analyzed and represented in Fig. 5.10, where the mean absolute percentage error (MAPE) and the corresponding standard deviation are shown as a function of the number of delay cycles and the required correction factor at the output (hardware scaling factor in Fig. 5.10). Ideally, the scaling factor should be $1/255 \approx 0.00392$ so that the correction does not have to be applied a posteriori. The best option would be the 3-cycle delay (green) because the minimum MAPE and its deviation are lower than for 1 and 2 cycles, and almost the same as for 4, 5 and 6 cycles. Nevertheless, from this graph it can be seen that increasing the delay does not make a big difference in the error. In order to avoid possible problems related to this issue and the impossibility to modify the scaling factor as bitstreams flow into the parallel RBF-NN logic, the LFSR has been replaced by another RNG.

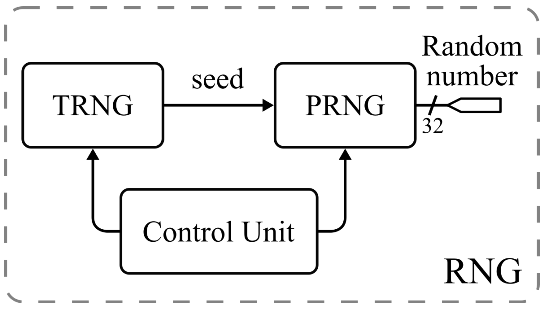


Figure 5.9: Block diagram of the RNG module.

To guarantee the randomness of bitstreams, especially with regard to minimizing autocorrelation, a good quality RNG is needed. A True Random Number Generator (TRNG) specifically designed for FPGA was introduced in [201], which seems a reasonable choice since it passes NIST and DIEHARD statistical tests and requires few logical resources. However, the TRNG implementation is based on ring oscillators, which have a high power consumption when implemented in FPGA [202]. Therefore, from a power consumption point of view, it makes more sense to use a Pseudo-Random Number Generator (PRNG) in this case. These two types of RNGs (TRNG and PRNG) are combined in such a way that the TRNG generates a seed value to set up the PRNG and then the former is switched off, as illustrated in Fig. 5.9. The corresponding HDL code has been developed by Prof. Luis Parrilla (University of Granada), who contributed to this work. The RNG is composed of a TRNG to generate a random seed based on 50 ring oscillators with 3 inverters per oscillator, and the PRNG reproduces the rng_n1024_r32_t5_k32_s1c48 design from [203]. The TRNG initialization needs 64 clock cycles and 1024 additional cycles are required to generate the seed. After the seed is generated, the PRNG returns a 32-bit random number per cycle, which is splitted into 4 bytes. In this application only 3 of them are actually utilized to generate bitstreams as indicated by SNGs depicted in Fig. 5.8 (blue, orange and green SNGs).

Once the RNG choice has been justified, the K-means pretrained parameters are loaded as prototypes. Using an architecture similar to Fig. 5.8 in which there is no output layer and the counter array is connected the RBF units (red wires) to create a new feature set from training data directly from the hardware. Therefore, instead of using readout weights obtained from exact features, these weights are obtained by fitting a linear model to the feature set obtained from training data in order to minimize the error due to small discrepances between the fixed-point and SC models. The model differences are mainly due to discrepancies between the SC RBF hardware activation and (5.7), and not so much due to slight variations in the distance calculations. These are only present for relatively small SC integration times. In this sense, the output layer is fine-tuned to adapt to the true shape of the SC activation function and accounting for random fluctuations related to short evaluation times. This method has been utilized

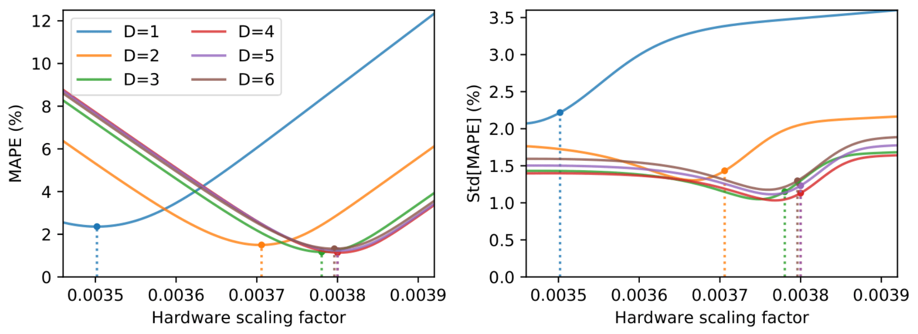


Figure 5.10: MAPE and the corresponding standard deviation with respect to the equivalent fixed-point implementation for scaled SC Euclidean square distances. The distances are evaluated on the MNIST training set and the RNG is an 8-bit LFSR. Different colors represent different delay values (D) for the computation of the SC square operation. The dotted vertical lines correspond to the optimum scaling factor for each delay value.

to evaluate different datasets, including different integration periods for the MNIST dataset.

In addition to MNIST, the design has been also tested with lower dimensionality datasets to check its performance. These datasets are summarized below:

- Iris flower [204]. It is probably the best known dataset in the ML community and one of the simplest classification problems based on real world data. The goal is to classify 4-dimensional features (sepal length, sepal width, petal length and petal width) in three different classes corresponding to a specific type of iris plant (setosa, versicolour or virginica), which cannot be done by a simple linear model because one of the three classes is not linearly separable from the other two. This small dataset contains 50 instances per class.

- Banknote authentication [205]. It consists of 4-dimensional features that are claimed to be obtained from banknote-like images via feature extraction based on a Wavelet Transform tool. It is a binary classification benchmark containing 762 data samples for class 0 and 610 for class 1.

- Breast cancer Wisconsin (diagnostic) [206]. Features are computed from image data of a breast mass and describe 32 different features of cell nuclei in the image. It is a binary classification benchmark which has 569 data samples and the goal is to predict whether the breast mass is malignant (212 observations) or benignant (357 observations).

- Optical recognition of handwritten digits [207]. Tiny 32×32 bitmaps containing handwritten digits are divided into 4×4 nonoverlapping windows, so that the output is an 8×8 5-bit map in which each number represents the bit count in the corresponding 4×4 window. The 64-dimensional digits range from 0 to 9 and the dataset contains 5620 data samples.

5.1.5 Results

The aftermentioned datasets have been evaluated by the RBF-NN model with 8-bit prototypes and readout weights. In Table 5.1 both fixed-point and SC results are reported. Notice in the SC case the number of evaluation cycles is $N = 511$, and in the particular case of $K = 1023$, results do not correspond to those directly obtained from the fully parallel FPGA implementation. Instead, in the $K = 1023$ case, a programmable RBF-NN SC design containing 31 parallel prototypes is utilized to iteratively obtain the results. These results are the same than for a hardware simulation model of the fully parallel design. As can be seen in the table, all the results are close (or equal) to those obtained by fixed-point inference. Moreover, the results are also close to those obtained in [16] (1000 RBF + linear classifier), which utilizes floating-point precision for the parameter storage, computation of distances and activations, obtaining 96.4% accuracy for the MNIST dataset.

Table 5.1: Accuracy percentage comparison between fixed-point and SC inference models. The number of evaluation cycles for al SC results is $N = 511$. (a) Fully parallel hardware. (b) Co-processing with 31 programmable hidden units.

Dataset	d	K	**Fixed-point**	**SC**
Iris	4	17	97.33	97.33
Banknote	4	30	99.63	99.51
Breast cancer	32	34	97.34	97.34
Digits	64	48	96.95	96.89
MNIST[a]	784	255	94.11	94.05
MNIST[b]	784	1023	96.25	96.2

In the MNIST case ($d = 784$), it is also reported how the number of evaluation cycles N affect test accuracy (see Fig. 5.11a), obtaining almost no improvement for $N \geq 511$ compared to $N = 511$ when the number of prototypes is sufficiently high. Therefore, from an energy efficiency point of view, $N = 511$ is the best choice since it corresponds to the lowest evaluation time with almost zero ($\sim 0.05\%$) accuracy degradation in front of $N = 8191$. Differences are small because the output layer was fine-tuned by training with the activation values calculated with the SC model, which allows the model to take into account variability related to bitstream randomness and the specific activation function shape. Moreover, Fig. 5.11b represents the expected growth in logic utilization, which should grow linearly with d and the number of parallel (physical) prototypes. Since programmable RBF units need parameter storage in FPGA registers, which represents a serious constraint in FPGAs due to the limited number of embedded registers. It requires more resource utilization per RBF unit (red) than the hardwired fully parallel version (blue).

To conclude this section, Table 5.2 has been included too, which lists several metrics related to the FPGA implementation, comparing it with the memoryless ReCA model. These metrics come from Table 4.10. Notice the last column provides (FP-BNN model) high maximum performance and energy efficiency numbers while maintaining good accuracy, so it is included here too for comparison purposes.

5.1.6 Summary

This work is focused on the implementation of a fully parallel RBF-NN based on SC, highlighting the importance of avoiding self correlation in bitstreams when computing square operations using a high quality RNG. Moreover, a new SAPC and activation function designs are introduced. The SAPCN could be used for any other ANN, but in this case it maps input bitstreams to an accurate output bitstream with a mean

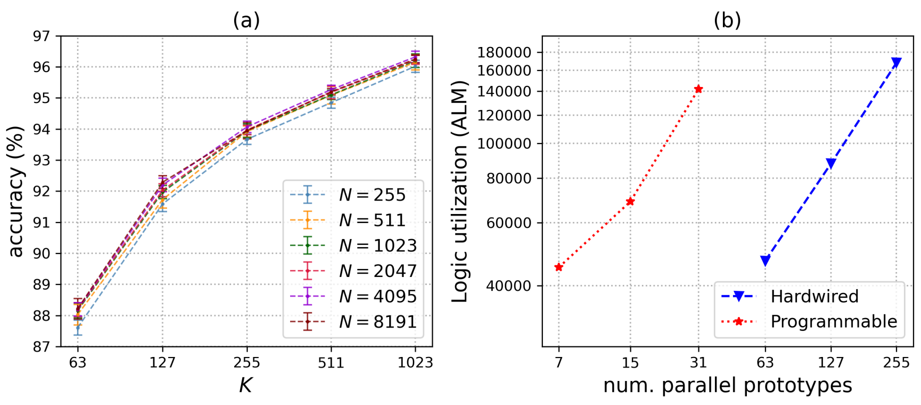


Figure 5.11: MNIST accuracy results and FPGA resource utilization. (a) Accuracy obtained from the stochastic RBFNN implementation for different number of kernels and evaluation times. (b) FPGA resource utilization for both programmable and hardwired architectures with $d = 784$. Here $K = 511$ and $K = 1023$ results are obtained from simulations since fully parallel designs do not fit on the FPGA.

Table 5.2: Comparison between SC based RBFNN, ReCA and an energy efficient CNN implementation. (a) Fully parallel hardwired with $K = 255$ and $N = 511$, tested on FPGA. (b) Fully parallel with $K = 1023$ and $N = 511$, not tested on FPGA; power and hardware utilization are extrapolated and accuracy is obtained from an equivalent SC simulation model.

	Hardwired[a]	Hardwired[b]	ReCA (Section 4.2)	Ref. [193]
Model	SC RBF-NN	SC RBF-NN	Memoryless ReCA	FP-BNN
Accuracy (%)	94.1(2)	96.2(1)	98.08	98.32
Clock frequency (MHz)	50.0	50.0	50.0	150.0
Latency (ms)	10.5	10.5	0.0	3.4
KIPS	95.2	95.2	50.0	294.0
Max. performance (KIPS/MHz)	1.90	1.90	1.00	1.96
Power (W)	5.90	23.6	0.289	26.2
Energy efficiency (KI/J)	16.14	4.03	173	11.22
Logic utilization (ALM)	168K	673K	22.3K	182.3K
DSP blocks	0	0	40	20
On-chip memory (bytes)	0	0	0	NA

activation probability. At the same time, this is the input of the RBF activation. This SC approach works fine for several datasets, obtaining similar results compared to the fixed-point implementations.

As listed in Table 5.2, the final RBF-NN FPGA implementation is not optimal in terms of accuracy, performance, logic resources nor energy efficiency for the MNIST classification task. In fact, there are other implementations resulting in higher accuracy, which are smaller, faster and more energy efficient based on other computation paradigms [138], [193].

Nevertheless, the proposed inference model is almost equivalent to its floating-point counterpart in the MNIST case [16], even with 8-bit parameters for relatively small evaluation times when the readout layer is fine tuned to account for real hidden layer activations. The main result is the successful usage of the proposed SAPCN and RBF activation units, achieving results equivalent to those obtained by a fixed-point model. Also, there is room for several variations or improvements such as lower precision (or simply binary) prototypes, different kernel functions or smaller evaluation times. These modifications would represent an improvement in terms of logic utilization and energy efficiency if accuracy is preserved. So that it might serve as the baseline for comparison with further works incorporating architecture improvements.

5.2 Convolutional Neural Networks

The origin of the ideas that gave rise to what we know today as CNN dates back to the 1950s and 1960s. Back then, Hubel and Wiesel concluded that the visual cortex of the cat and monkey contains neurons that fire individually because they are related to small areas of the visual field. Each of these fields, the contents of which cause a single neuron to fire, is known as a receptive field, which vary in size and each neuron has its own receptive field [208]. These receptive fields overlap in different regions of the visual field and are very similar for spatially correlated neurons. Moreover, there are two types of cells in charge of this task depending on its size and complexity of the pattern detected by the receptive field: simple and complex [209]. While simple cells essentially detect edges with different orientations, complex cells have more connections, i.e. its receptive fields are larger, and detect more complex patterns.

These advances in the field of neurophysiology inspired K. Fukushima, who introduced in 1980 the main CNNs components used today. K. Fukushima proposed a new network architecture for pattern recognition unaffected by shift in position: *neocognitron* [210]. Both supervised and unsupervised local parameter optimization approaches were proposed, including different types of cells in a biologically inspired self-organizing structure. However, nowadays the backpropagation supervised training is the best option.

Backpropagation in combination with an explicit weight sharing training approach together with an idea similar to a CNN, known as time-delay neural network (TDNN) was first introduced by A. Waibel et al. some years later and applied to phoneme recognition [211]. The TDNN input was a 16-coefficient mel spectrogram and the network architecture can be described as a 2-dimensional CNN with a kernel width including several time steps and kernel height equal to the number of mel coefficients, i.e. 16 in that case, and time integration of the last convolution outputs. Therefore, the architecture is invariant to time shifts by construction. Similar ideas were applied to image processing applications too.

Perhaps, one of the most successful advances in this context was by Y. LeCun et al. in 1989 [212], who built an small CNN to classify handwritten numbers based on backpropagation with weight sharing based Fukushima's ideas, including multi-channel 2-dimensional convolution operations and average pooling for downsampling purposes. Over the years, the computational capacity growth made it possible to increase the number of ANN hidden layers. In 1998, Y. LeCun et al. introduced the LeNet-5 [16], a CNN with which they broke the MNIST accuracy record. The paper also includes comparisons with a wide variety of alternative ML algorithms and how to use the inference mechanism to recognize documents.

A slightly modified version of the so called LeNet-5 is depicted in Fig. 5.12. While the original model uses average pooling and sigmoid hidden activations, here it is introduced with max-pooling filters after each convolution and hidden activations are ReLUs. In this figure, the notation $c\,@\,h \times w$ refers to the number of channels c, height h and width w at each stage. Even though there are 7 different layers represented in Fig. 5.12, the CNN is called LeNet-5 because max-pooling layers are not parametrized and CNN convolution layers usually include multiple multi-channel convolution (or simply convolution in a DL context) followed by pooling operations. Similar architectures along with the application of backpropagation-based training are the foundation of DL applied to computer vision. In fact, a lot of the breakthroughs with great impact in the DL area are based on essentially the same type of architectures for larger images and deeper networks. Fortunately, researchers managed to accelerate the training process using GPUs, allowing its application to process higher resolution input RGB images [213] and networks with more and more parameters and layers [214], [215]. Many advances related to CNNs are quite recent, both at the level of network architecture and implementation of ever deeper networks as well as at the level of parameter optimization, but these are topics beyond the scope of the work described here.

This section is focused on the hardware simplification of operations and memory required for the inference process in a CNN. In this context, low precision fixed-point inference is currently a hot topic in the DL field and thanks to the advances in QAT approaches, these changes do not lead to significant degradation in overall accuracy [150], [199]. The ANN training process focusing on obtaining fixed-point parameters and activations by the end of the backpropagation process is known as quantization aware training (QAT) and a specific algorithm is proposed. Furthermore, two different SC descriptions of the CNN inference are described and simulated in order to evaluate the quantization algorithm performance. All experiments are on the LeNet-5 architecture described in Fig. 5.12, including results obtained from a fully parallel SC design proposed and implemented in FPGA by C.F. Frasser et al., including VLSI implementation metrics too [216].

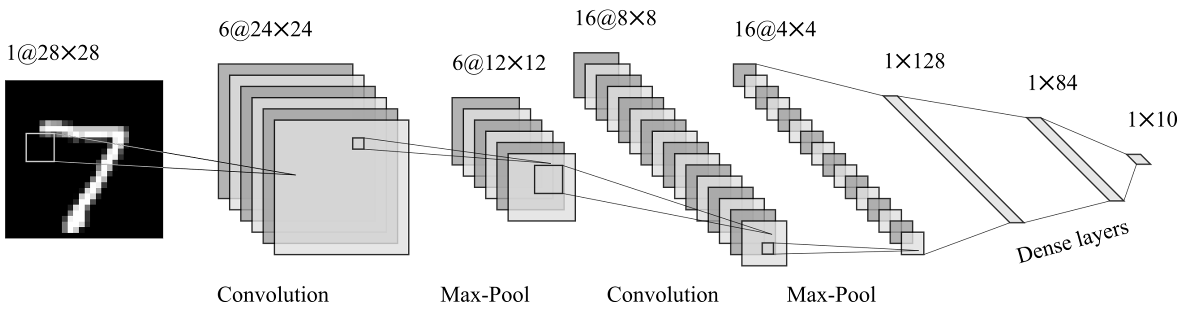


Figure 5.12: LeNet-5 neural network architecture.

5.2.1 Contribution

The contribution of this section arises from the need to create a quantization scheme for an already implemented 2-dimensional CNN design based on SC [216]. For this purpose, and inspired by existing QAT methods adopted when targeting low precision fixed-point hardware [150], [199], a similar technique is developed to obtain fixed-point parameters suitable for fully parallel SC implementations.

Therefore, two different SC inference models based on the LeNet-5 CNN are evaluated on the MNIST test set with quantized parameters and intermediate operations. One of the simulation models is based on the bipolar SC coding and the other on the sign-magnitude coding. Finally, the inference results are reported too, comparing it with other SC and fixed-point implementations. It should be noted that results related to FPGA implementation and potential VLSI implementation are not part of this thesis.

5.2.2 Related work

There are no quantization aware backpropagation training methods oriented to SC models. However, the proposed method is inspired by existing approaches targeting fixed-point inference models, but existing approaches currently applied to obtain fixed-point models might work as well under certain modifications. Some of those approaches explore low bit width quantization, with [217] or without [218] including it in the training process, while others are more focused on especial cases, including binary [150], [199], [219], [220] and ternary [221]–[223] weighted CNNs. In both cases, it might make sense to use different bit precisions for parameter storage and arithmetic operations. The content presented in this section utilizes these ideas and introduces some new simplifications to scale CNN signals layer-wise instead of filter-wise. Moreover, these scales are restricted to powers of two to meet the SC hardware specifications.

In SC, the parameter bit width does not matter as long as it is less or equal than $\lfloor \log_2 N \rfloor$, where N is the bitstream length, and ideally, increasing N smoothly increases precision in arithmetic operations. Here a $\lceil \log_2 N \rceil$-bit bipolar signal refers to the ideal maximum representational precision in intermediate operations obtained from the interaction between bipolar bitstreams with length N or sign-magnitude bitstreams with length $\lfloor N/2 \rfloor$. In this context, we proposed a quantization method that takes into account the SC operations imprecisions, the closer the parameters are to their limit values ($+1$ and -1), the more accurate the multiplications are. In the bipolar coding, the bitstreams representing 0 values increases the error propagated to activations in the next layer. Since our hardware implementation is in fact based on bipolar coding, this effect is also sought to be minimized. The SC formulation and simulations presented here are mainly inspired by an existing hardware implementation [216], extending it to the sign-magnitude SC coding. There exist other SC CNN implementations in the literature [45], [57], [224], [225], but their approach to implement SC blocks performing (multi-channel) 2-dimensional convolutions, max-pooling and activation function is based on approximate pooling and activation operations. Instead, in our approach the SC error source comes from the stochastic multiplications inside the convolution sub-blocks, then max-pooling and ReLU are computed by comparing correlated bitstreams, so that the result is evaluated without any error. Therefore, besides potential bitstream generation issues, the proposed SC architectures are more accurate than previous SC implementations for a generic number of parameter and signal bit width.

5.2.3 Theoretical foundations

To understand how a CNN works it is necessary to know the convolution operation details. Convolutions can be discrete or continuous and applied to one or more spatio-temporal dimensions. Since it is one of the main ingredients of CNNs, the first part of this section has briefly reviewed this concept. Moreover, assuming that the reader is already familiar with the inference mechanism of a fully-connected FFNN or MLP, which is described in Section 2.3.3. In addition, the way in which inference happens in a CNN is also reviewed. We also focus on model quantization, which occurs along the training process, i.e. during backpropagation (see Section 2.3.3.2).

Finally, and to conclude with the theoretical foundations of this research, two SC-based models are described, one based on bipolar coding and the other on sign-magnitude coding, as well as the digital block diagrams for the corresponding SC equations.

5.2.3.1 Convolution operation

The convolution operation is usually introduced as a continuous mathematical operation between two functions g_x and g_w denoted by the operator $*$, so that $(g_x * g_w)$ is the function resulting from the convolution and the simplest case is 1-dimensional convolution. Formally, the integral of the product of both functions is calculated, with one of them reversed and shifted to obtain the resulting function, i.e.:

$$(g_x * g_w)(t) = \int_{-\infty}^{\infty} g_x(t-\tau) g_w(\tau) d\tau, \quad g_x, g_w : \mathbb{R} \to \mathbb{R} \tag{5.16}$$

The same type of operation can be applied to a finite number of points. So in the one-dimensional case, instead of functions g_w and g_w, one can refer to the arguments $\boldsymbol{x}$ and $\boldsymbol{w}$ as row vectors, so that:

$$\boldsymbol{z} = \boldsymbol{x} * \boldsymbol{w}, \quad z_i = \sum_{j=0}^{f-1} x_{si+j} w_{f-1-j} \tag{5.17}$$

where $\boldsymbol{w}$ is usually referred to as kernel or filter, s is the stride and f the filter size. A discrete convolution can be either valid, same or full. Both same and full variants can be achieved by padding zeros to the vector being convolved and performing the valid convolution on already padded data with the same filter. Therefore, $\boldsymbol{z}$ has $\lfloor \frac{h-f}{s} \rfloor + 1$ elements because (5.17) describes a valid convolution.

In practice, the filters are model parameters obtained during the training process, so that it does not matter whether the filter parameters are reversed in order to perform each multiplication. In fact, CNNs implement cross-correlations instead of convolutions, the only difference between these two operations being whether the filter is rotated or not. Similarly to (5.16) and (5.17), the cross-correlation operation is denoted by $\star$ and given by (5.18) in the continuous case and (5.19) in the discrete case.

$$(g_x \star g_w)(t) = \int_{-\infty}^{\infty} g_x(t+\tau) g_w(\tau) d\tau, \quad g_x, g_w : \mathbb{R} \to \mathbb{R} \tag{5.18}$$

$$\boldsymbol{z} = \boldsymbol{x} \star \boldsymbol{w}, \quad z_i = \sum_{j=0}^{f-1} x_{si+j} w_j \tag{5.19}$$

The same concept can be generalized to multiple input channels and filters, which is how a 1-dimensional CNN layer would operate. As an example, Fig. 5.13 illustrates how a single channel pattern (top left) is cross-correlated with multiple filters (top right), obtaining the cross-correlation results (bottom left). In this example the input pattern is the superposition of two amplitude modulated sinusoidal waves at 20 and 30 Hz represented by 512 input vector elements. Filters are pure sinusoidal waves with linearly increasing frequencies and 64 elements per filter. As a result, cross-correlation amplitudes (bottom right) are higher at 20 and 30 Hz, making them easier to be detected. In fact two filters would be sufficient to detect these two frequencies. Nevertheless, if the target is to accurately detect any frequency, many different filters are needed. The best approach is to use the Discrete Fourier Transform (DFT) since the Fast Fourier Transform (FFT) algorithm is a much more efficient approach.

The cross-correlation concept can be extended to an arbitrary number of dimensions. However, it is sufficient to introduce the 2-dimensional cross-correlation, which

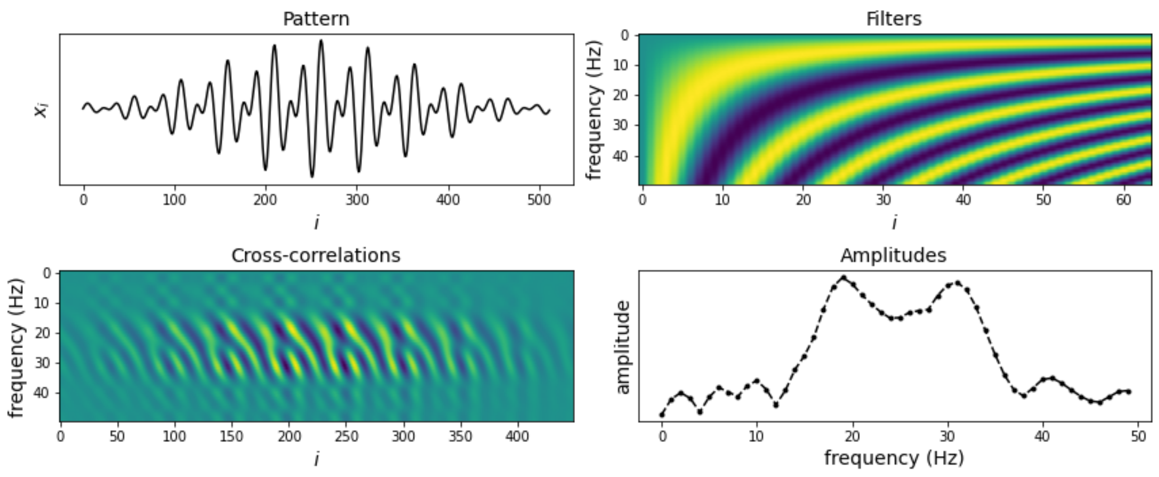


Figure 5.13: Example 1-dimensional discrete cross-correlation for a single input channel and multiple filters.

is the core of the LeNet-5 and by extension modern computer vision and audio processing based on DL. The 2-dimensional continuous operation extends (5.18) to cross-correlation applied to 2-variable functions, that is:

$$(g_X \star g_W)(u, v) = \int_{-\infty}^{+\infty} \int_{-\infty}^{\infty} g_X(u + \tau_0, v + \tau_1) g_W(\tau_0, \tau_1) d\tau_0 d\tau_1, \quad g_X, g_W : \mathbb{R}^2 \to \mathbb{R}^2 \tag{5.20}$$

so, in analogy with (5.19), the valid discrete 2-dimensional cross-correlation operates on the pattern $\boldsymbol{X}$ and filter $\boldsymbol{W}$ matrices, given by:

$$\boldsymbol{Z} = \boldsymbol{X} \star \boldsymbol{W}, \quad Z_{i,j} = \sum_{i_0=0}^{h-1} \sum_{i_1=0}^{w-1} X_{si+i_0, sj+i_1} W_{i_0,i_1} \tag{5.21}$$

where $\boldsymbol{X}$ is an $h \times w$ matrix, usually representing a byte map with height h and width w, respectively. While Fig. 5.13 shows the result of applying various filters in the 1-dimensional case, the Fig. 5.14 shows it for the 2-dimensional case taking a grayscale image as input. Notice how filters applied to image data work as edge or primitive shape detectors, e.g. horizontal lines, diagonal lines or specific vertex shapes. This is the base of how a CNN works. The composition of several concatenated cross-correlation layers combined to other (simpler) operations allows the trained network to recognize features with a higher level of abstraction. In fact, a CNN detects increasingly complex and larger shapes as a network grows in depth.

The operations performed in 2-dimensional CNN layers are multi-channel cross-correlations, which is a particular case of 3-dimensional cross-correlation in which the number of channels is equal to the filter depth, so that the filter is not strided across the channel dimension. Therefore, assuming rank-3 tensors X, W and resulting matrix $\boldsymbol{Z}$, the discrete, valid, multi-channel and 2-dimensional cross-correlation (from now on, simply cross-correlation for short) is given by:

$$\boldsymbol{Z} = \mathsf{X} \star \mathsf{W}, \quad Z_{i,j} = \sum_{i_0=0}^{c-1} \sum_{i_1=0}^{h-1} \sum_{i_2=0}^{w-1} \mathsf{X}_{i_0, si+i_1, sj+i_2} \mathsf{W}_{i_0,i_1,i_2} \tag{5.22}$$

An example is the case of RGB images (three channels), since each of the applied filters must have three channels ($c = 3$). This is necessary even if we are dealing with

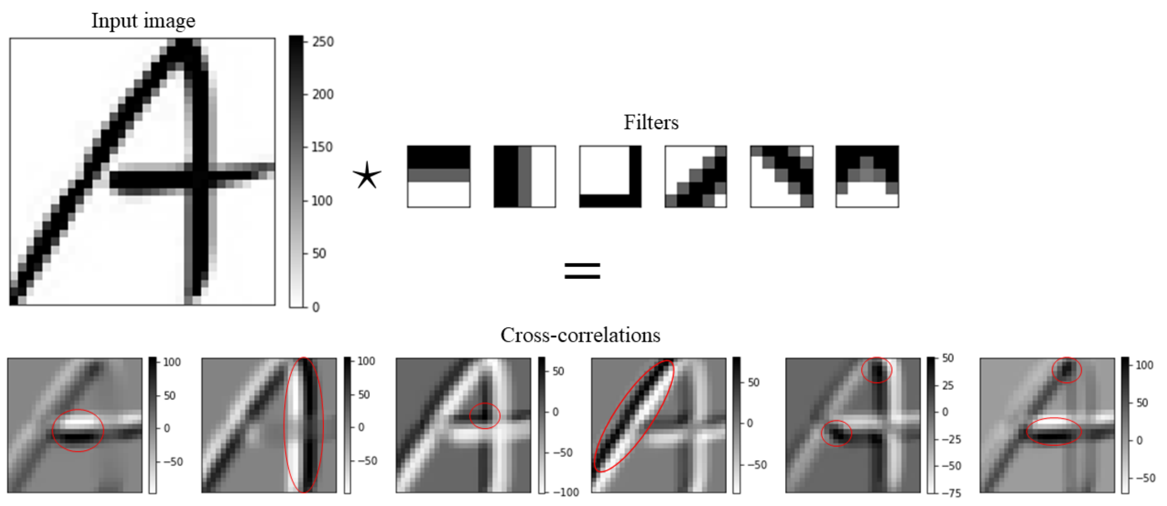


Figure 5.14: Example 2-dimensional discrete cross-correlation for a 32×32 image with a single input channel and multiple 5×5 filters. The highest resulting pixel values are highlighted in red.

grayscale ($c = 1$) images and there is more than one convolutional layer, as is the case of LeNet-5 (Fig. 5.12). In the general case, each tensor filter is $c \times f_1 \times f_2$, so blocks of the same size are obtained by sliding a cubic window through the $c \times h \times w$ input tensor $\boldsymbol{X}$ with stride s. This input tensor represents multi-channel data with two spatial dimensions and cubic windows are not strided across channels. For each filter to be applied, each $c \times f_1 \times f_2$ block is multiplied element-wise by the filter of the same size and all resulting elements are added together (dot product) as described by (5.22), this can be generalized to the multiple filter case. In the single filter case, results are represented by a $\left(\lfloor\frac{h-f_1}{s}\rfloor + 1\right) \times \left(\lfloor\frac{w-f_2}{s}\rfloor + 1\right)$ matrix $\boldsymbol{Z}$, however, if d different filters have to be applied, it might be a good option to use a more compact notation in which results are organized in a $d \times \left(\lfloor\frac{h-f_1}{s}\rfloor + 1\right) \times \left(\lfloor\frac{w-f_2}{s}\rfloor + 1\right)$ tensor and a single $d \times c \times f_1 \times f_2$ filter tensor $\boldsymbol{W}$ containing all d filters, so that resulting cross-correlations are given by:

$$Z_{i,:,:} = \boldsymbol{X} \star W_{i,:,:,:} \tag{5.23}$$

The Fig. 5.14 example has a single-channel input $\boldsymbol{X}$ that would be a $1 \times h \times w$ tensor to which $d = 6$ different $1 \times 5 \times 5$ filters are applied, i.e. the filters can be described by a $6 \times 1 \times 5 \times 5$ tensor $\boldsymbol{W}$. Each single filter cross-correlation results in a slice $Z_{i,:,:}$, so that all results are grouped in a $d \times \left(\lfloor\frac{h-f_1}{s}\rfloor + 1\right) \times \left(\lfloor\frac{w-f_2}{s}\rfloor + 1\right)$ tensor $\boldsymbol{Z}$.

5.2.3.2 Forward propagation

In order to keep the notation as similar as possible to the corresponding implementation, tensor dimensions are the same as in PyTorch. The specific dimension sizes are indicated in Table 5.3, where c is the number of input channels, d is the number of output channels, h is the image height, w is the image width, f_1 is the filter height and f_2 is the filter width.

The layer L cross-correlation output, denoted by the tensor $\boldsymbol{Z}^{[L]}$ is given by the cross-correlation operation (5.22) between activations from previous layer $\boldsymbol{A}^{[L-1]}$ and filters $\boldsymbol{W}^{[L]}$:

$$Z^{[L]}_{i,:,:} = \boldsymbol{A}^{[L-1]} \star W^{[L]}_{i,:,:,:} \tag{5.24}$$

After a convolution operation, it is a common practice to apply $q \times q$ max-pooling and ReLU. Both operations are applied in layer L and given by (5.25) and yield the output

Table 5.3: Tensor types and the corresponding ranks and dimension sizes consistent with this document.

Tensor type	rank	sizes
input from previous layer	3	(c, h, w)
cross-correlation filters	4	(d, c, f_1, f_2)
resulting cross-correlations	3	(d, h, w)
valid convolution w/ stride s	3	$\left(d, \left\lfloor \frac{h-f_1}{s} \right\rfloor + 1, \left\lfloor \frac{w-f_2}{s} \right\rfloor + 1\right)$
$p \times p$ pooling w/ stride p	3	$\left(d, \frac{1}{p^2}\left(\left\lfloor \frac{h-f_1}{s} \right\rfloor + 1\right), \frac{1}{p^2}\left(\left\lfloor \frac{w-f_2}{s} \right\rfloor + 1\right)\right)$

activations in layer L.

$$A_{i,j,k}^{[L]} = \max\left\{0, \max_{a,b} Z_{i,qj+a,qk+b}^{[L]}\right\}, \quad a, b \in \mathbb{Z} \cap [0, q-1] \tag{5.25}$$

Applying (5.24) and then (5.25) is illustrated in 5.15 and these two operations result in the most common type of CNN layer. The typical forward propagation in a CNN concatenates one or more CNN layers, combined with simple MLP architectures at the end, as is the case of the LeNet-5 architecture depicted in Fig. 5.12. Each convolution[1], max-pooling and activation[2] sequence is a CNN layer. Therefore, the LeNet-5 is composed by 2 CNN layers and a 3-layer MLP. Despite convolutional and fully connected layers are part of a standard CNN, fully connected layers are not discussed. Notice that a fully connected layer can be described as a cross-correlation in which the input and each filter have the same shape, so that the stride $s = 0$, $f_1 = h$, $f_2 = w$ and d is the number of units.

5.2.3.3 Quantization aware training

Inference in SC systems does not necessarily require a QAT method as long as the resulting model accuracy is not affected at all. However, this is not our case. So the idea arose to apply a relatively simple quantization algorithm to improve the results in order to bring them closer to the floating-point model results. Quantization is quite common in the literature, especially in point-fixed models. Here the idea is very similar since the arithmetic operations are carried out via SC logic but parameters need to be stored in a fixed-point format as in previous case.

Understanding how backpropagation works is helpful since the training carried out with quantized weights requires the utilization of this algorithm too. The standard training approach is the same as for the MLP since cross-correlations can be expressed as matrix multiplications by manipulating feature and filter tensors [226]. The main difference is related to the error propagation through max-pooling operations, since the gradient does not propagate through non-maximal values. Moreover, the max operation is locally linear (proportional with slope 1) with respect to each maximum value through which the error is propagated. So, it does not require any additional operation other than addressing the gradient descent only to the maximum arguments, for each window $p \times p$.

[1]It is actually cross-correlation, but in the DL literature these terms are used interchangeably because filter weights are trainable and both operations become equivalent if filters are flipped.

[2]Activations are implicit after each max-pooling or dense layer.

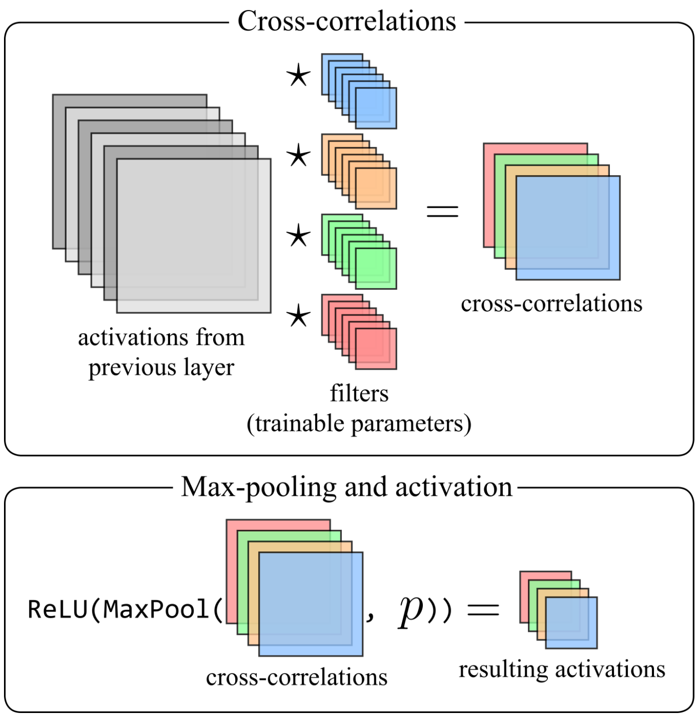


Figure 5.15: Example illustrating tensor shapes in a typical CNN layer, including both cross-correlation and max-pooling followed by a ReLU activation.

As regards QAT, there are several approaches and the one selected is just another quantization algorithm. It is important to differentiate between two training approaches. If a training algorithm is intended to be implemented on a device (on-device learning), in which the model parameters must always be in a fixed-point format, then intermediate parameter updates must be fixed-point too. Instead, if the training algorithm is intended to be run on a server or desktop PC for subsequent implementation of the inference on another device (off-device learning), in which the model parameters can be stored in a floating-point format during training and converted to a fixed-point format at the end of the process. In this work, the model is trained in floating-point on a desktop PC and these parameters are forced to converge to integer values by the end of the process.

Here, a step towards integer quantization is done after every backpropagation step as described by Algorithm 2. Forward, backward and optimization steps are done as usual, the innovation is that each optimization step is potentially followed by an step towards quantization.

5.2.3.4 An equivalent bipolar SC inference model

The equivalent SC inference model implemented in this work requires all input and intermediate calculations to be normalized in the range $[-1, +1]$, since these numbers represent activation probabilities with the bipolar change of variables already applied. So the first step is to convert (5.24) and (5.25) to quantities in the bipolar range. The cross-correlation outputs are denoted by $\boldsymbol{P}_Z^{*[L]}$ and are given by (5.26). These quantities are mostly proportional to the original cross-correlation outputs.

$$\boldsymbol{P}_Z^{*[L]} = \boldsymbol{s}^{[L]} \odot \mathrm{HTanh}_{r_{th}} \left(\boldsymbol{P}_A^{*[L-1]} \star \boldsymbol{P}^{*[L]}_{W;i,:,:,:} \right) \tag{5.26}$$

Algorithm 2: NN quantization step after each backpropagation step

```
Input: Input mini-batch data samples $\boldsymbol{X}$ and targets $\boldsymbol{O}$, current weights
       $\{\boldsymbol{W}^{[1]}, \dots, \boldsymbol{W}^{[L_f]}\}$, a boolean applyQuant to control whether to apply
       an step towards quantization or not, and hyperparameters as well as
       other arguments (denoted as optimizerArgs[$i$] and quantizerArgs[$i$],
       respectively, for layer $i$)
Result: Updated weights
/* obtain intermediate linear transformations and predicted output */
$\left\{\{\boldsymbol{Z}^{[1]}, \dots, \boldsymbol{Z}^{[L_f]}\}, \hat{\boldsymbol{O}}\right\} \leftarrow$ forwardPropagation($\boldsymbol{X}$, $\{\boldsymbol{W}^{[1]}, \dots, \boldsymbol{W}^{[L_f]}\}$);
/* obtain weight errors based on some cost function by propagating errors
   backwards */
$\{\delta\boldsymbol{W}^{[1]}, \dots, \delta\boldsymbol{W}^{[L_f]}\} \leftarrow$ backwardPropagation($\hat{\boldsymbol{O}}$, $\boldsymbol{O}$, $\{\boldsymbol{Z}^{[1]}, \dots, \boldsymbol{Z}^{[L_f]}\}$);
/* weight update for all layers */
for $i \leftarrow 1$ to $L_f$ do
    optimizerStep($\boldsymbol{W}^{[i]}$, $\delta\boldsymbol{W}^{[i]}$, optimizerArgs[$i$]);
end
/* weight manipulation for all layers */
if applyQuant then
    for $i \leftarrow 1$ to $L_f$ do
        quantizerStep($\boldsymbol{W}^{[i]}$, quantizerArgs[$i$]);
    end
end
```

Here $s_j^{[L]}$ are appropriate scaling factors. Although this is a relatively general form for these (output filter-wise) scaling factors, for simplicity, assume a common scaling factor for a given layer, that is $s^{[L]} = s_j^{[L]}\ \forall j$. Also, the HTanh ensures the outputs are in the interval $[-1, 1]$[3]. As regards the fused max-pooling and ReLU outputs, bipolar quantities are denoted by $\boldsymbol{P}_A^{*[L]}$ and given by: (5.27).

$$P^{*[L]}_{A;i,j,k} = \max\left\{0, \max_{a,b} P^{*[L]}_{Z;i,qj+a,qk+b}\right\}, \quad a, b \in \mathbb{Z} \cap [0, q-1] \tag{5.27}$$

From (5.26) and (5.27), it is also possible to formulate a fixed-point forward propagation hardware model with quantized activations, inputs and weights. Instead, the goal is to formulate a bipolar SC model. The straightforward approach would be to compute the equivalent SC of (5.26), then convert quantities back to the SC domain and compute the equivalent of (5.27). However, to formulate the SC model in this work we apply the equivalent (5.27) for layer $L-1$ and then (5.26), so that no intermediate domain conversions are required to perform this sequence of operations (first fused max-pooling and ReLU, then cross-correlation). Therefore, (5.27) and (5.28) would be our reference expressions expressed with bipolar quantities.

$$P^{*[L]}_{Z;i,j,k} = s_i^{[L]}\mathrm{HTanh}_{r_{th}}\left(\sum_{i_0=0}^{c-1}\sum_{i_1=0}^{f_1-1}\sum_{i_2=0}^{f_2-1} P^{*[L-1]}_{A;i_0,sj+i_1,sk+i_2} P^{*[L]}_{W;i,i_0,i_1,i_2}\right), \quad a, b \in \mathbb{Z} \cap [0, q-1] \tag{5.28}$$

[3]Applying ReLU after HTanh function results in a saturated ReLU operation and the saturation threshold depends on scales $s_j^{[L]}$.

In order to describe the SC equations, the original tensors listed in Table 5.3 are now bitstreams, i.e. boolean functions of time. These bitstreams are denoted with a tilde and evaluated each clock period T. According to section 2.2, the bipolar multiplication is implemented by the XNOR of the uncorrelated inputs and the maximum is obtained by the logical disjunction (OR gate) of the correlated inputs. Then (5.28) can be approximated by (5.29) and (5.30) for large N values.

$$\tilde{A}^{[L-1]}_{i,j,k}(t) = \overbrace{\underbrace{\max\left\{\tilde{p}_{\frac{1}{2}}(t), \max_{a,b} \tilde{Z}^{[L-1]}_{i,qj+a,qk+b}(t)\right\}}_{(q^2+1)\text{-input OR logic gate}}}^{\text{Fused max-pooling and ReLU}} \tag{5.29}$$

$$P^{[L]}_{Z;i,j,k} \approx \frac{s^{[L]}_i}{N}\text{ReLU}_{r_{th}}\left(\underbrace{\sum_{n=0}^{N-1}\sum_{i_0,i_1,i_2}}_{\text{APC}} \overline{\tilde{A}^{[L-1]}_{i_0,sj+i_1,sk+i_2}(nT) \oplus \tilde{W}^{[L]}_{i,i_0,i_1,i_2}(nT)}\right), \quad a,b \in \mathbb{Z}\cap[0,q-1] \tag{5.30}$$

The resulting quantities approximate $P^{[L]}_{Z;i,j,k}$, which represents a probability, ready to be converted to stochastic bitstreams again to feed the next layer. This probability would be converted to the equivalent bipolar variable under the change of variables $\boldsymbol{P}^{*[L]}_Z = 2\boldsymbol{P}^{[L]}_Z - \boldsymbol{J}$.

The bitstream $\tilde{p}_{\frac{1}{2}}(t)$ is the result of converting the bipolar zero to a stochastic bitstream, which has 50% activation probability. Moreover, in (5.30) correlation and decorrelation of bitstreams play an important role, as described by C.F. Frasser et al. [216]. Fused max-pooling and ReLU operate with correlated bitstreams, meanwhile multiplications require uncorrelated bitstreams, as reflected by (5.31).

$$\tilde{p}_{\frac{1}{2}} \parallel \tilde{Z}^{[L]}_{i,j,k} \ \forall_{i,j,k}, \qquad \tilde{p}_{\frac{1}{2}} \perp \tilde{W}^{[L]}_{i,j,k,l} \ \forall_{i,j,k,l} \tag{5.31}$$

Then, obtaining the bitstream representing the largest value becomes trivial using a simple (q^2+1)-input OR gate. Also, each multiplication requires an XNOR logic gate. Equivalently, these two operations can be concatenated using a (q^2+1)-input NOR and an XOR gate.

5.2.3.5 An equivalent sign-magnitude SC inference model

We take (5.27) and (5.26) as the starting point. Since bipolar quantities take values in the interval $[-1, +1]$, which can be directly mapped to the sign-magnitude SC inference model. In order to simplify the final expression, let us first define sign and magnitude as separate variables $\text{sgn}(\boldsymbol{X})$ and $\text{mgn}(\boldsymbol{P}^*_X)$ that fully determine the bipolar equivalent $\boldsymbol{P}_X$ from a tensor $\boldsymbol{X}$, that is:

$$\boldsymbol{P}^{*[L]}_X = \text{sgn}(\boldsymbol{X}) \odot \text{mgn}(\boldsymbol{P}^*_X), \tag{5.32}$$

The sign function can be expressed in terms of the (elementwise) Heaviside function $\mathcal{H}$:

$$\text{sgn}(\boldsymbol{X}) = \frac{1}{2}\left(\mathcal{H}(\boldsymbol{X}) + \boldsymbol{J}\right) \tag{5.33}$$

In addition, for the sake of simplicity it is also convenient to define the following short notation:

$$\mathcal{M}(\boldsymbol{X}) = \mathrm{mgn}(\boldsymbol{P}_{\boldsymbol{X}}^{*}) \tag{5.34}$$

with the corresponding bitstreams denoted as $\tilde{\mathcal{M}}[\boldsymbol{X}](t)$.

Using this notation, the fused max-pooling and ReLU applied to a $q \times q$ pairs of sign-magnitude bitstreams representing the activations in layer $L-1$ are given by (5.35), which are the input bitstreams to the next cross-correlation in layer L.

$$\tilde{\mathcal{M}}\left[A_{i,j,k}^{[L-1]}\right](t) = \overbrace{\max_{a,b}\left(\mathcal{H}\left(Z_{i,qj+a,qk+b}^{[L-1]}\right) \cdot \tilde{\mathcal{M}}\left[Z_{i,qj+a,qk+b}^{[L-1]}\right](t)\right)}^{q^2 \text{ AND gates and } q^2\text{-input OR gate}}, \quad a,b \in \mathbb{Z} \cap [0, q-1] \tag{5.35}$$

Since multiplication operations involving the sign and magnitude can be decoupled and operated separately, signs are interpreted as bipolar variables that remain constant in time, representing either -1 or $+1$ and operated accordingly. That is, the multiplication of two numbers a and b with the same sign results in a positive number $c = ab$, i.e. $\mathcal{H}(c) = 1$, in contrast, if the sign of a and b is different, then c is negative, i.e. $\mathcal{H}(c) = 0$. At the same time, decoupled magnitudes are simply multiplied as unipolar bitstreams. Therefore, the cross-correlation operation in layer L is described by:

$$P_{Z;i,j,k}^{[L]} \approx \frac{s_i^{[L]}}{N}\mathrm{HTanh}_{r_{th}}\left(\underbrace{\sum_{n=0}^{N-1}\sum_{i_0,i_1,i_2}}_{\text{APC}}\left(2\overbrace{\left(\overline{\mathcal{H}\left(A_{i_0,sj+i_1,sk+i_2}^{[L-1]}\right) \oplus \mathcal{H}\left(W_{i,i_0,i_1,i_2}^{[L]}\right)}\right)}^{\text{addition (1) or subtraction (0)}} - 1\right)\right.$$
$$\left. \cdot \overbrace{\tilde{\mathcal{M}}\left[A_{i_0,sj+i_1,sk+i_2}^{[L-1]}\right](nT) \cdot \tilde{\mathcal{M}}\left[W_{i,i_0,i_1,i_2}^{[L]}\right](nT)}^{\text{unipolar product}}\right) \tag{5.36}$$

In this expression, magnitude multiplications are unipolar, sign multiplications are bipolar and each resulting bitstream pair contributes to the parallel count, which can be implemented by a parallel counter (PC). Each bitstream pair contribution for a single time step n can be either positive or negative, being the parallel count a signed quantity too. The result is then accumulated for N clock cycles to approximate (5.28), which can be implemented by an APC that accounts for two's complement (signed) quantities.

5.2.3.6 Hardware description

An appropriate logic and block diagram description is helpful for simulation and hardware implementation purposes. The aim of the following block diagrams is to present the overall system structure according to previous equations. The main ingredients for both SC models are the scalar product and fused max-pooling and ReLU operations for a given $p \times p$ window. Both components are depicted in Fig. 5.16 and 5.18 for the bipolar and sign-magnitude cases.

On the one hand, Fig. 5.16 presents the digital design implementing fused max and ReLU SC operations, which is the basic building block for the fused max-pooling and ReLU for bipolar and sign-magnitude cases, given by (5.29) and (5.35), respectively.

Both blocks operate with correlated bitstreams and the activation probabilities are equivalent to (5.27) if the $p \times p$ window is unrolled as a p^2 vector. In the fused max and ReLU bipolar case, the implementation is based on Section 2.2.3 demonstrations, while for the sign-magnitude case the principle is similar but with additional AND logic gates so that for each sign-magnitude bitstream pair, a negative sign bit inhibits the magnitude bitstream. Since the sign bit remains constant, correlation between magnitude bitstreams is not lost after each AND gate. Following the same notation as in Fig. 5.16, the output activation probabilities in Fig. 5.17 are $p_z = \max\{p_{x;0}, p_{x;1}, \ldots, p_{x;p^2-1}, 0.5\}$ for the bipolar ReLU-max (BRM) and $\mathcal{M}(z) = \max\{\mathcal{M}(x_0), \mathcal{M}(x_1), \ldots, \mathcal{M}(x_{p^2-1}), 0\}$ for the sign-magnitude ReLU-max. As depicted in Fig. 5.17, organizing these blocks according to the corresponding tensor shape enables the fully-parallel computation of bitstreams representing all left hand side tensor elements in (5.27). This block diagram depicted in Fig. 5.17 is referred to as bipolar ReLU-max-pool (BRMP) or sign-magnitude ReLU-max-pool (SMRMP) depending on whether the coding is bipolar or sign-magnitude, respectively.

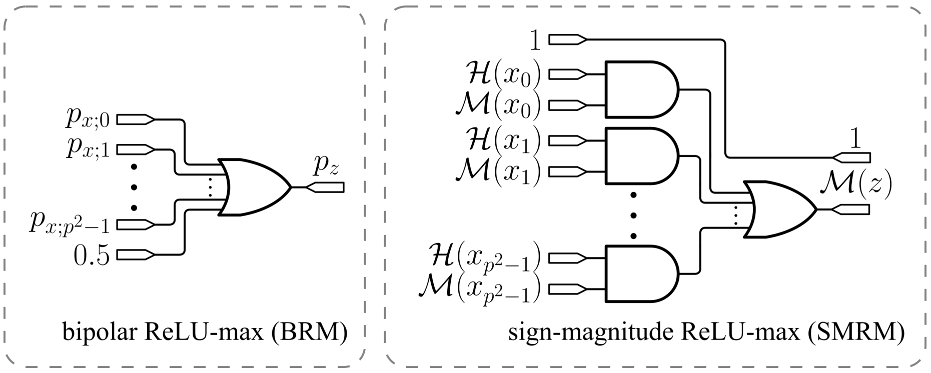


Figure 5.16: Fused max and ReLU SC logic for both bipolar (left) and sign-magnitude (right) bitstream number representations.

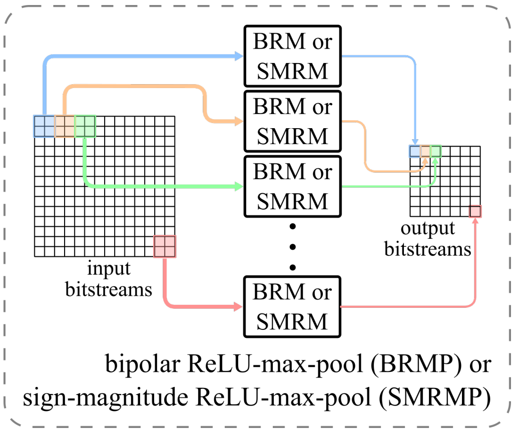


Figure 5.17: SC CNN fused ReLU and max-pooling block diagram for either bipolar bitstreams or sign-magnitude bitstream pairs. Notice in this example $p = 2$ and the input bitstreams represent a 14×14 channel with 2 spatial (or spatio-temporal) dimensions, so that the outputs represent a downsampled 7×7 channel.

On the other hand, The dot product implementation is similar to that proposed for RBF-NN, but with different APC blocks. Fig. 5.18 depicts the digital design for the SC dot product, which is an element of the cross-correlation block. Each cross-correlation output can be interpreted as the dot product between the unrolled $c \times f_1 \times f_2$ input block and the corresponding filter. Therefore, combining bipolar dot product (BDot) or sign-magnitude dot product (SMDot) with BRMP or SMRMP blocks (Fig. 5.17) allows to implement an complete convolutional layer. Fig. 5.19 depicts the digital design for a single convolutional layer, where input and output data signals are bitstreams. Notice only one intermediate APC array per layer is needed, since output bitstreams are the next layer inputs without additional domain conversion.

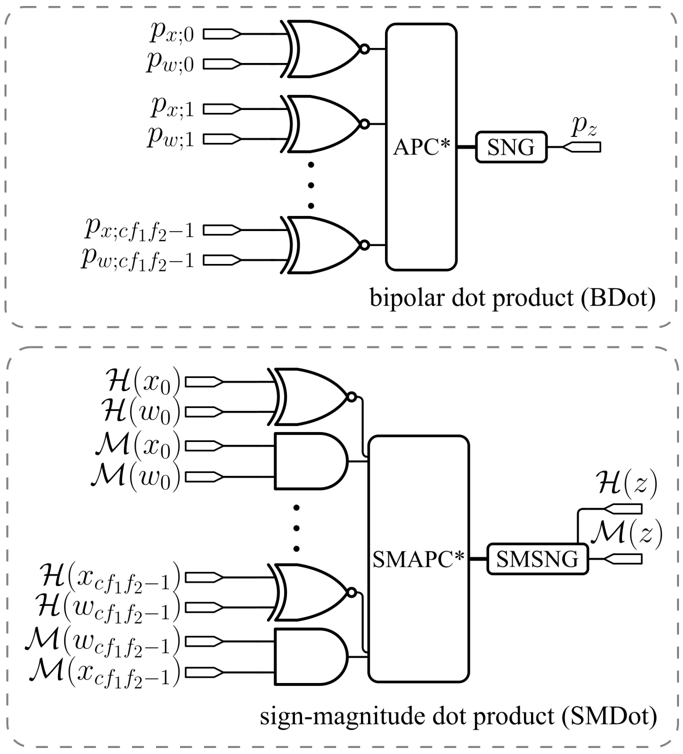


Figure 5.18: Dot product SC logic for both bipolar (left) and sign-magnitude (right) bitstream number representations. (*) The depicted APC and SMAPC have a registered output to enable pipelining trough different layers and provide binary weighted numbers that are already apropriately scaled by a power of 2.

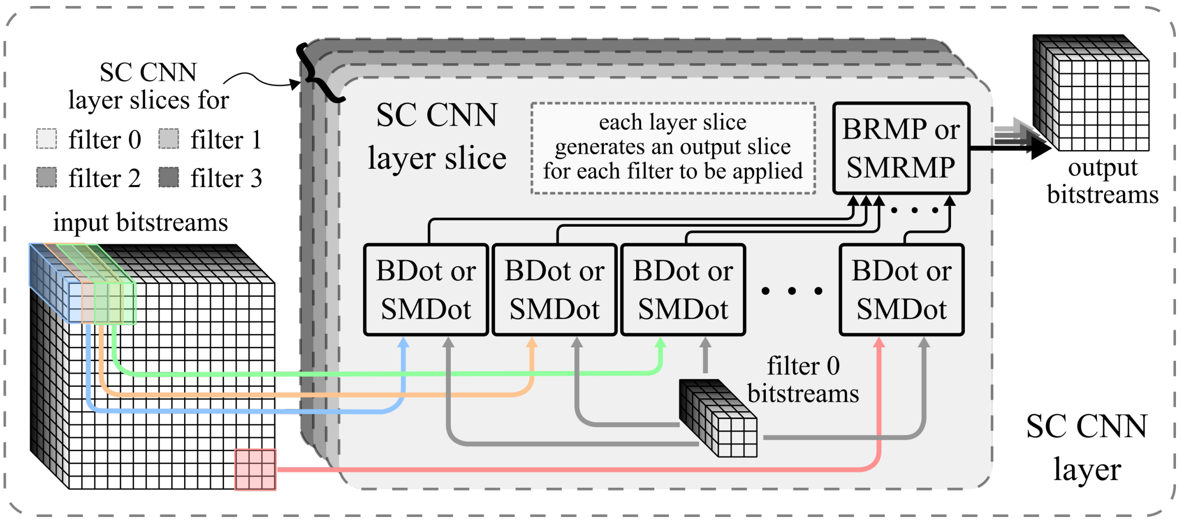


Figure 5.19: SC CNN layer block diagrams for either bipolar bitstreams or sign-magnitude bitstream pairs. Notice in this example $d = 4$, $c = 6$, $h = w = 16$, $s = 1$, $p = 2$.

5.2.4 Methods

This section describes the quantization method, as well as the general workflow once the final parameters have been obtained. So far, a general introduction on model quantization has been described (Algorithm 2) and the equations and hardware blocks required to evaluate inference in a SC CNN layer have also been presented.

The proposed quantization method is built on the idea that trained ANN weights follow a normal distribution[4]. Therefore, there are a few large weights and most of them are concentrated around zero. The main issue with normally distributed weights is that linear quantization results in large overall quantization relative errors since values near zero are much more frequent than weight values beyond two or three times the standard deviation. To tackle this problem, there are two main quantization methods[5]: applying a nonlinear quantization scheme or modifying the final weight distribution to obtain better linear quantization error.

The nonlinear quantization approach reduces the model size since there is no need to store floating-point weights but codebooks of them instead. However, in this work this approach is discarded because the intermediate operations require high precision, e.g. floating-point multiplications. In fact, we work with SC bitstreams that can only represent integers, so it is not the most appropriate method for our case. Instead, modifying the weight distribution in order to reduce relatively large errors related to quantization can result in a model accuracy closer to that obtained in floating-point. This is done during training using labeled data, which allows more control over the resulting weight distribution. Optionally, the impact of quantizing activations could be taken into account, but in this work it is not considered since the weight distribution manipulation method contributes to reduce the inherent error to bitstream multiplications.

From simulation results [57], we know the SC multiplication error is maximum when input and parameter bitstreams have 50% activation rate and are generated from uncorrelated maximum length random uniform sequences. For any input activation probability, the maximum *multiplication* error occurs when the parameter bitstream has 50% activation probability. Since weights are normally distributed after a standard ANN training, this effect becomes relevant for the bipolar SC multiplication since the bipolar zero corresponds to 50% activation probability. At the same time, the most common weight values are distributed around zero, which contributes to increase the error too. In contrast, multiplying an input bipolar quantity p_x^* by another bipolar quantity p_w^* representing either a -1 or +1 does not have any associated error if the input bitstream is generated from a maximum length sequence. Similarly, in the sign-magnitude case, the SC multiplication result is exact as long as the weight magnitude probability $\mathcal{M}(w)$ is either 0 or 1 and the input magnitude probability $\mathcal{M}(x)$ is generated from a maximum length sequence.

Therefore, a more SC hardware-friendly weight distribution could take into account the following tips in order to reduce the error:

- As done in fixed-point models, weight clipping between $-n_{lim}$ and $+n_{lim}$ can be set to avoid long normal distribution tails, i.e. outlier weight values, which would

[4]This does not only happen by the end of the training process. The weights are initialized so that they follow a normal distribution with a specific mean and variance to accelerate learning, but what is interesting here is the distribution in the final state.

[5]The literature on this subject is extensive but the vast majority of approaches can be classified into one of the two methods mentioned in the text or a combination of both [227].

increase linear quantization errors. In this case, a small variance is related to small quantization errors.

- Modifying the weight distribution so that values are more evenly distributed in order to reduce the relative error associated to smaller values due to quantization. In addition, this reduces large relative errors in bipolar SC multiplications because parameters that were near zero are spread out.
- Increasing the number of exact multiplication results can be accomplished by increasing the number of weights represented by bipolar quantities equal to -1 and $+1$. These values correspond to 0 and 1 activation rates in the bipolar coding, respectively. In turn, these values correspond to a magnitude activation rate equal to 1 in the sign-magnitude coding, since the computation between sign bits is done separately.

These three tips can be achieved using a simple weight update rule, which is as follows. Let the symmetric clipping function be:

$$\text{clip}_{n_{lim}}(x) = \text{HTanh}_{n_{lim}}(x) \tag{5.37}$$

so that applying it to a weight distribution increases the frequency of the limit weight values $\pm n_{lim}$. Suppose the weight distribution has mean zero and standard deviation σ, so that n_{lim} is defined to be proportional to σ, i.e. $n_{lim} = n_\sigma \sigma$.

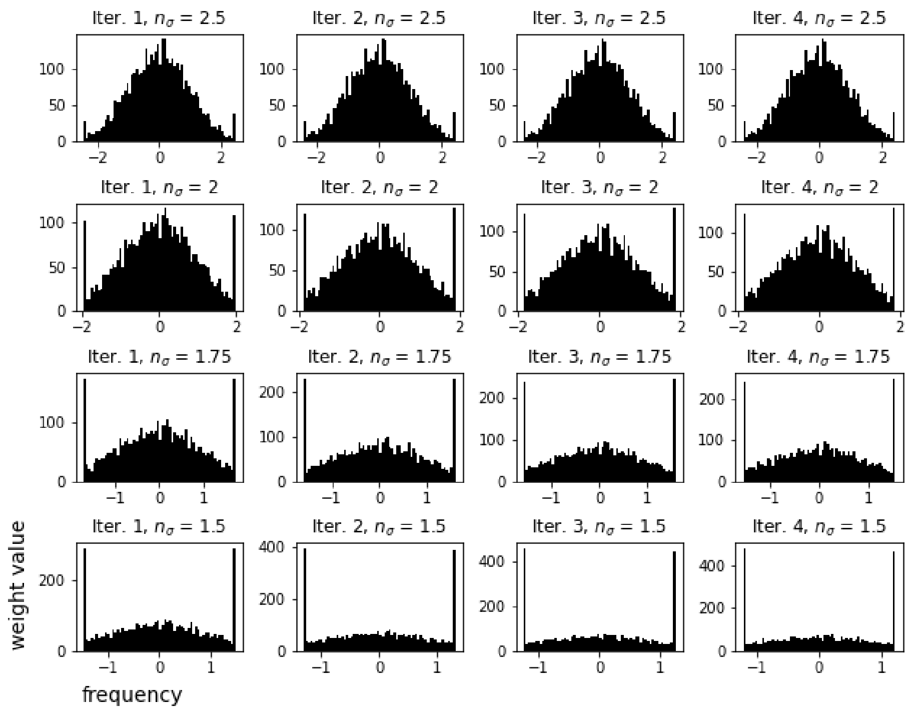


Figure 5.20: Multiple weight distributions after applying several 1, 2, 3 and 4 weight clipping iterations (columns) for different n_σ values (rows) utilized to update n_{lim} based on the distribution standard deviation.

As the original weight distribution is transformed from normal to pseudo-uniform, the variance decreases and the number of weight values within a given number of standard deviations increases. The first column of plots in Fig. 5.20 shows this weight clipping effect applied to a normal weight distribution with mean 0 and variance 1. In addition, it is also shown how repeatedly applying this weight clipping transformation is equivalent to choosing slightly lower n_σ values for sufficiently small values since the standard deviation of the resulting distribution is smaller and n_{lim} decreases. In Fig. 5.20, (5.37) is repeatedly applied for several iterations without including weight updates. However, if this clipping transformation is applied during training, weights are slightly different from one iteration to the next, so that the clipped distribution is slightly different because new weight values beyond $\pm n_{lim}$ might appear, i.e. weight clipping might affect training performance if n_σ is too small. Therefore, there is a tradeoff between how uniform is the resulting distribution transformation and training performance. That is, the lower n_σ, the lower the quantization error and training performance compared to that obtained without weight clipping.

In general, for sufficiently high bit precision (e.g. 7 or 8-bit weight values) post-training quantization might be good enough for the weight clipping method described above, for either fixed-point or SC models. However, a lower bit precision might require exact weight values at the end of the training process to reduce weight quantization errors to zero. Assuming the maximum absolute value of weights $\boldsymbol{W}$ is n_{lim}, quantization to $2n_{values}$ different integer weights is given by:

$$\boldsymbol{W}_q = \left\lfloor \frac{\boldsymbol{W}\, n_{values}}{n_{lim}} \right\rceil \frac{n_{lim}}{n_{values}} = \boldsymbol{W}_{int} \frac{n_{lim}}{n_{values}} \tag{5.38}$$

so that n_{values} is the number of different positive weight values and the resulting integer weights are $\boldsymbol{W}_{int}$ the scaled version is $\boldsymbol{W}_q$. Notice this is known as restricted range quantization because it assumes the same number of positive and negative values, which is different from rounding to the nearest two's complement number because it includes an additional negative number. While restricted quantization returns values in the range $[-n_{lim}, n_{lim}]$, so that the scale is common for both negative and positive numbers, rounding to the nearest two's complement representation requires full range quantization to represent values in the range $[-n_{lim} - 1, n_{lim}]$. In this work, only restricted range quantization is considered since full range quantization results in higher accuracy loss for small bit widths.

Directly quantizing weight values after each training epoch might not work. If weight updates are smaller than $\frac{1}{2n_{lim}n_{values}}$, then no update occurs in practice. To overcome this issue, most ANN quantization approaches use floating-point weights to account for small weight updates and the quantized version is used for the forward propagation. This approach is better than direct weight quantization in DNNs and can be improved if the backward propagation is modified to include quantization errors. However, in this work a simple heuristic rule is applied to evolve weight values so that they converge to integer values at the end of the training process. A quantized weight tensor $\boldsymbol{W}_q$ is obtained from $\boldsymbol{W}$ via (5.38), so that weights are updated as

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \varepsilon\left(\boldsymbol{W}_q - \boldsymbol{W}\right) \tag{5.39}$$

where the rounding rate ε controls how close is each updated weight to its quantized version.

Both weight clipping and the update towards quantized weight values are applied after each training epoch as described by Algorithm 3. This algorithm includes an additional step to obtain layer scaling factors as powers of 2. However, this additional step decreases performance and we observed that it is better to explore different scaling factors experimentally to find the best combination. Moreover, notice convergence to quantized weight values is not guaranteed unless $\varepsilon = 1$, which corresponds to simplest quantization scheme. Therefore, both n_σ and ε are tuned after every epoch to speed up training[6] and gradually reduce quantization errors. In order to accomplish gradual adaptation towards quantization with reduced errors, n_σ is monotonically decreased from a maximum to a minimum value, and ε is monotonically increased from 0 to 1. At the beginning, large n_σ values do not impact training performance and if ε is small at the beginning, quantization errors are small. In contrast, for a sufficiently large number of epochs, n_σ is near its minimum value and ε has increased. In this case, a small n_σ modifies the weight distributions so that quantization errors are smaller even if ε has been increased.

[6]Based on experience, the smaller n_σ, the slower the training process compared to standard back-propagation training without quantization using the same hyperparameters.

Algorithm 3: Proposed quantization step for 3 bits or more

```
Input: Weights $\boldsymbol{W}$, enabled width factor $n_\sigma$, rounding rate $\varepsilon$, a flag (flag)
       indicating the weight scale has been converted to a power of 2,
       maximum ReLU value $r_{th}$, number of integer values $n_{values}$
Result: Updated weights
flag must be initialized to 0 and there must be a different flag and $n_{lim}$ if the
  algorithm is applied to multiple weight matrices every quantization step, i.e.
  different layers require a different flag and $n_{lim}$;
/* the limit value is chosen to be proportional to the weight's standard
   deviation and depends on the enabled width factor $n_\sigma$                    */
$n_{lim,next} \leftarrow n_\sigma \operatorname{std}(\boldsymbol{W})$;
if $\boldsymbol{W}$ are not the output layer weights then
    if flag then
        $n_{lim,next} \leftarrow n_{lim}$;
    else if $(|n_{lim,next} - n_{lim}| < \epsilon)$ and $(n_\sigma < n_{\sigma,th})$ then
        /* round to the nearest power of 2                                  */
        $n_{lim,next} \leftarrow \frac{r_{th}}{2^{\lfloor \log_2 \left( r_{th}/n_{lim,next} \right) \rceil}}$;
        flag $\leftarrow 1$;
    /* update limit value                                                   */
    $n_{lim} \leftarrow n_{lim,next}$;
end
/* the weight value's range is restricted between $-n_{lim,next}$ and $+n_{lim,next}$   */
$\boldsymbol{W} \leftarrow \texttt{clip}(\boldsymbol{W}, -n_{lim,next}, +n_{lim,next})$;
/* nearest integer weights assuming restricted range quantization           */
$\boldsymbol{W}_q \leftarrow \left\lfloor \frac{\boldsymbol{W}\, n_{values}}{n_{lim,next}} \right\rceil \frac{n_{lim,next}}{n_{values}}$;
if this is the last epoch then
    /* the objective is to obtain integer weights at the end of the training
       process, i.e.  after the last epoch                                  */
    $\boldsymbol{W} \leftarrow \boldsymbol{W}_q$;
else
    /* real weights get closer to the rounded version, with greater or lesser
       intensity depending on the monotonically increasing rounding rate $\varepsilon$  */
    $\boldsymbol{W} \leftarrow \boldsymbol{W} + \varepsilon\,(\boldsymbol{W}_q - \boldsymbol{W})$;
end
```

In our experiments n_σ is exponentially decreased between a maximum and minimum value and ε follows a logistic function, so that it is nearly 0 at the beginning and very close to 1 by the end of the training process. The parameter functions utilized for training are represented in Fig. 5.21 for 60 epochs.

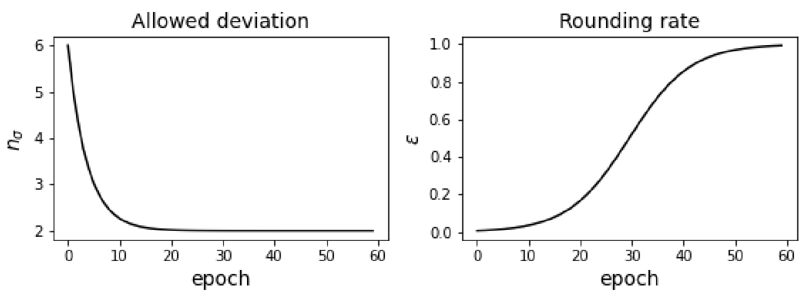


Figure 5.21: Example n_σ and ε hyperparameter sequences for 60 epochs.

The training scheme described above is applied to both SC CNN models, which reproduce a LeNet-5 (see Fig. 5.12) architecture simulating concatenated SC CNN layers described in previous section (Fig. 5.19), as well as SC FC layers. Results have been evaluated in terms of test set accuracy for different signal and parameter bit widths. The models describe bit accurate parallel SC logic, which has been implemented and tested in FPGA within the research group, including its corresponding VLSI synthesis in collaboration with another research group. This is a fully parallel SC LeNet-5 implementation based on the bipolar coding. Therefore, several hardware performance, area, power and energy efficiency metrics are included too.

5.2.5 Results

This section includes several results related to the SC LeNet-5 simulations as well as results on the bipolar FPGA implementation and VLSI synthesis.

Simulations reveal that the SC sign-magnitude provides almost the same accuracy results than the equivalent fixed-point implementation and the SC bipolar models require large bitstream periods in order to obtain similar performance. These results are summarized in Fig. 5.22. In this figure signal bit width refers to the inputs, outputs and intermediate activations[7] maximum resolution. Different weight bit widths (rows) correspond to different models while different signal bit widths (columns) are obtained using the same parametrization for a constant weight bit width, i.e. intermediate signal quantization is not taken into account during the training process. As regards layer-wise scaling constants, these have been tuned to maximize validation set accuracy. However, the main and most time-consuming challenge has been to choose a *good* pair of maximum length random sequences to generate bitstreams and maximize validation set accuracy. All maximum length random sequences utilized to obtain results in Fig. 5.22 have been created by randomly permuting ordered natural sequences.

Notice the bipolar model does not provide reasonable results for signal bit widths smaller than 8 bits, i.e. $N = 255$. Although the bipolar accuracy difference is not significative for 8-bit weights and 8-bit signals (8-bit/8-bit) case (0.04% below the fixed-point reference), the sign-magnitude model is much more convenient in general. Notice

[7]Recall that sign-magnitude bitstreams are two-wire bipolar representations and thus require half length bitstreams compared to the simple bipolar representation for the same signal bit width.

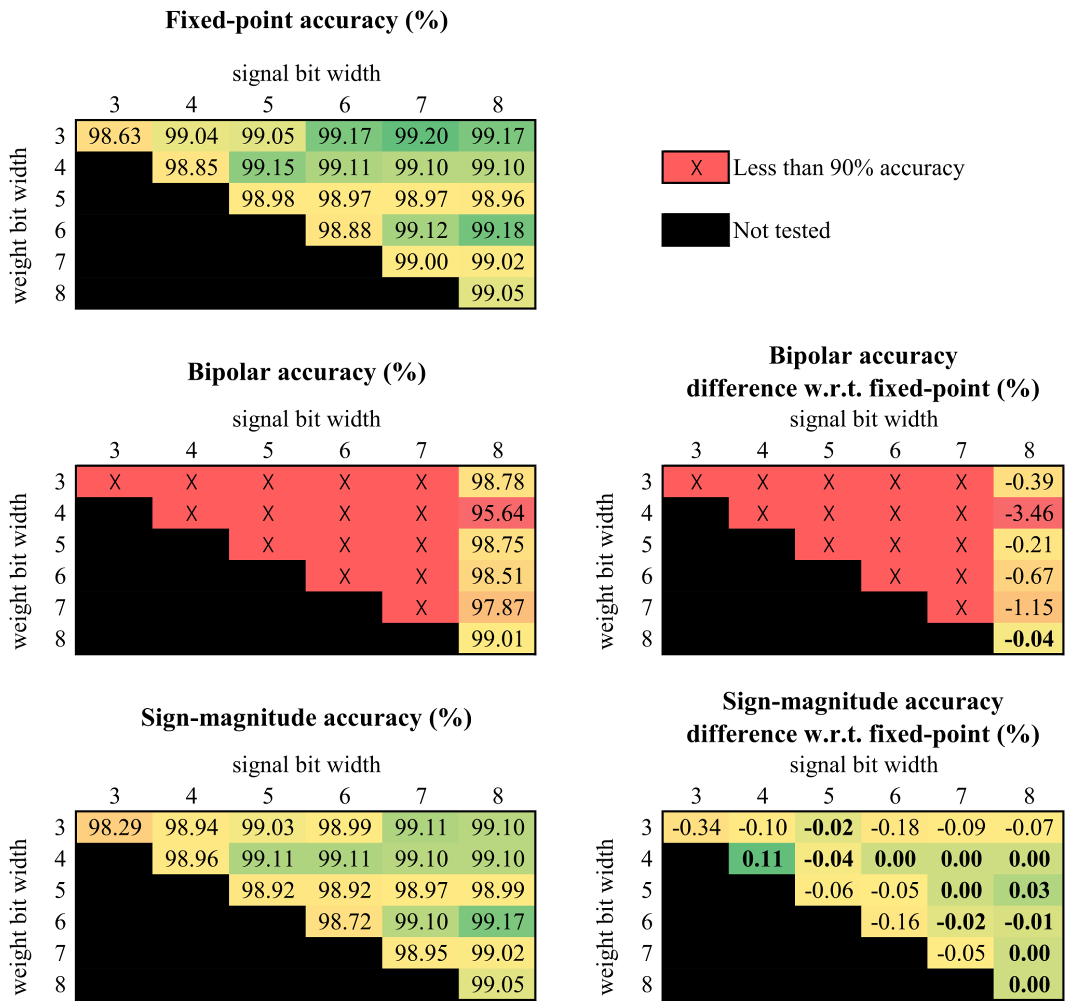
accuracy differences equal or better than the 8-bit/8-bit bipolar result are highlighted with bold numbers. In fact, these simulation results show that the proposed sign-magnitude model is almost equivalent to the fixed-point reference for most weight and signal bit width configurations. In the bipolar case, the main reason behind accuracy drops compared to fixed-point and sign-magnitude models are arithmetic errors in multiplication operations, which are propagated layer by layer. As explained in Section 2.2.4, the error in a single bipolar multiplication is doubled compared to the unipolar representation utilized in sign-magnitude operations. Since the bipolar zero does not exist in practice, activations that are supposed to represent a zero propagate additional errors to the next layer. Such error sources are the main bipolar model issue, especially for low signal bit widths. As an example, Fig. 5.23 represents intermediate APC array measurements compared to the fixed-point ones for 100 different (random) MNIST samples. The 8-bit/8-bit case represented in this figure presents less variance for the sign-magnitude model with respect to fixed-point data. In this 8-bit/8-bit case both SC models successfully propagate meaningful information through the whole network since data points are perfectly correlated to fixed-point data. However, the lower is the signal bit width, the higher is the variance compared to the ideal values. Also, in the 4-bit/4-bit case, the sign-magnitude model presents much larger errors but despite this discrepancy compared to the fixed-point reference, signals are still meaningful at a glance. The accuracy reported in Fig. 5.22 for this specific configuration (98.96) is not significantly affected. The same is not true for the bipolar case, in which large errors do not enable the signal to propagate properly even up to the second convolution, i.e. the second APC array in Fig. 5.23.

Nevertheless, the 8-bit/8-bit fixed-point model is the most common type of quantized forward propagation scheme in DL programming frameworks [103], [104] and the equivalent SC bipolar alternative provides competitive results with this specific configuration. In fact, recent experiments suggest that a pair of carefully designed random sequences might improve the bipolar model accuracy. Table 5.4 lists the obtained results using different random sequences for the 8-bit/8-bit bipolar model. The best results are for uncorrelated sobol sequences, which is in line with S. Liu and J. Han work [56]. In fact, recent works implementing other SC LeNet-5 variants utilize Sobol sequences to generate bitstreams [228].

Table 5.4: 8-bit/8-bit accuracy results for both fixed-point and different SNG random sequence types.

	8-bit/8-bit bipolar accuracy (%)
Random permutation	99.01
Best 8-bit LFSR pair	98.96
Sobol	99.06

Moreover, the 8-bit/8-bit SC bipolar model developed by C.F. Frasser et al. at the UIB Electronic Engineering Group, for which an FPGA implementation is already available and the VLSI synthesis results are known. Even though SC models are usually not optimal for FPGA implentations as it is the case for the RBF-NN due to the large amount of required registers to hold inputs and activations in parallel (Section 5.1). However, parallel SC CNN implementations benefit from input data sharing in convolution operations. According to the results reported in [216], the resulting



Fixed-point accuracy (%)

signal bit width

weight bit width	3	4	5	6	7	8
3	98.63	99.04	99.05	99.17	99.20	99.17
4	■	98.85	99.15	99.11	99.10	99.10
5	■	■	98.98	98.97	98.97	98.96
6	■	■	■	98.88	99.12	99.18
7	■	■	■	■	99.00	99.02
8	■	■	■	■	■	99.05

X Less than 90% accuracy

■ Not tested

Bipolar accuracy (%)

signal bit width

weight bit width	3	4	5	6	7	8
3	X	X	X	X	X	98.78
4	■	X	X	X	X	95.64
5	■	■	X	X	X	98.75
6	■	■	■	X	X	98.51
7	■	■	■	■	X	97.87
8	■	■	■	■	■	99.01

Bipolar accuracy difference w.r.t. fixed-point (%)

signal bit width

weight bit width	3	4	5	6	7	8
3	X	X	X	X	X	-0.39
4	■	X	X	X	X	-3.46
5	■	■	X	X	X	-0.21
6	■	■	■	X	X	-0.67
7	■	■	■	■	X	-1.15
8	■	■	■	■	■	**-0.04**

Sign-magnitude accuracy (%)

signal bit width

weight bit width	3	4	5	6	7	8
3	98.29	98.94	99.03	98.99	99.11	99.10
4	■	98.96	99.11	99.11	99.10	99.10
5	■	■	98.92	98.92	98.97	98.99
6	■	■	■	98.72	99.10	99.17
7	■	■	■	■	98.95	99.02
8	■	■	■	■	■	99.05

Sign-magnitude accuracy difference w.r.t. fixed-point (%)

signal bit width

weight bit width	3	4	5	6	7	8
3	-0.34	-0.10	**-0.02**	-0.18	-0.09	-0.07
4	■	**0.11**	**-0.04**	**0.00**	**0.00**	**0.00**
5	■	■	-0.06	-0.05	**0.00**	**0.03**
6	■	■	■	-0.16	**-0.02**	**-0.01**
7	■	■	■	■	-0.05	**0.00**
8	■	■	■	■	■	**0.00**

Figure 5.22: SC bipolar and sign-magnitude accuracies for different wight and signal bit widths compared to equivalent fixed-point models. All results are obtained from software simulations. SC results were obtained using maximum length sequences for bitstream generation (see text).

implementation requires more logic resources but no DSP blocks and the maximum throughput is much higher, which makes it more energy efficient than other fixed-point LeNet-5 implementations.

As an example, a throughput-optimized fixed-point FPGA implementation [229] requires about $233 \cdot 10^3$ ALM and 2907 DSP blocks to classify $10,617$ MNIST samples per second, i.e. 10.617 KIPS, resulting in an efficiency of about 0.421 KI/J. This efficiency can be improved using e.g. binary weights as in the FP-BNN FPGA implementation reported in [193], obtaining 11.22 KI/J (see Table 4.10 or 5.2 for additional details). On the other hand, the FPGA implementation of the bipolar SC LeNet-5 has a throughput of 294.118 KIPS and energy efficiency of 14.006 KI/J for the 8-bit/8-bit case. Therefore, this energy efficiency and the corresponding test set accuracy are much higher than those of the FP-BNN [193]. Moreover, the bipolar SC LeNet-5 authors showed that these numbers are significantly higher for the equivalent VLSI implementation using a TSMC 40 nm library. They obtained a throughput of 400.312 KIPS and energy efficiency of 614.920 KI/J for the 8-bit/8-bit model. Notice this energy efficiency is about 44 times higher than for the FPGA implementation.

These optimizations could also benefit from a bipolar SC implementation.

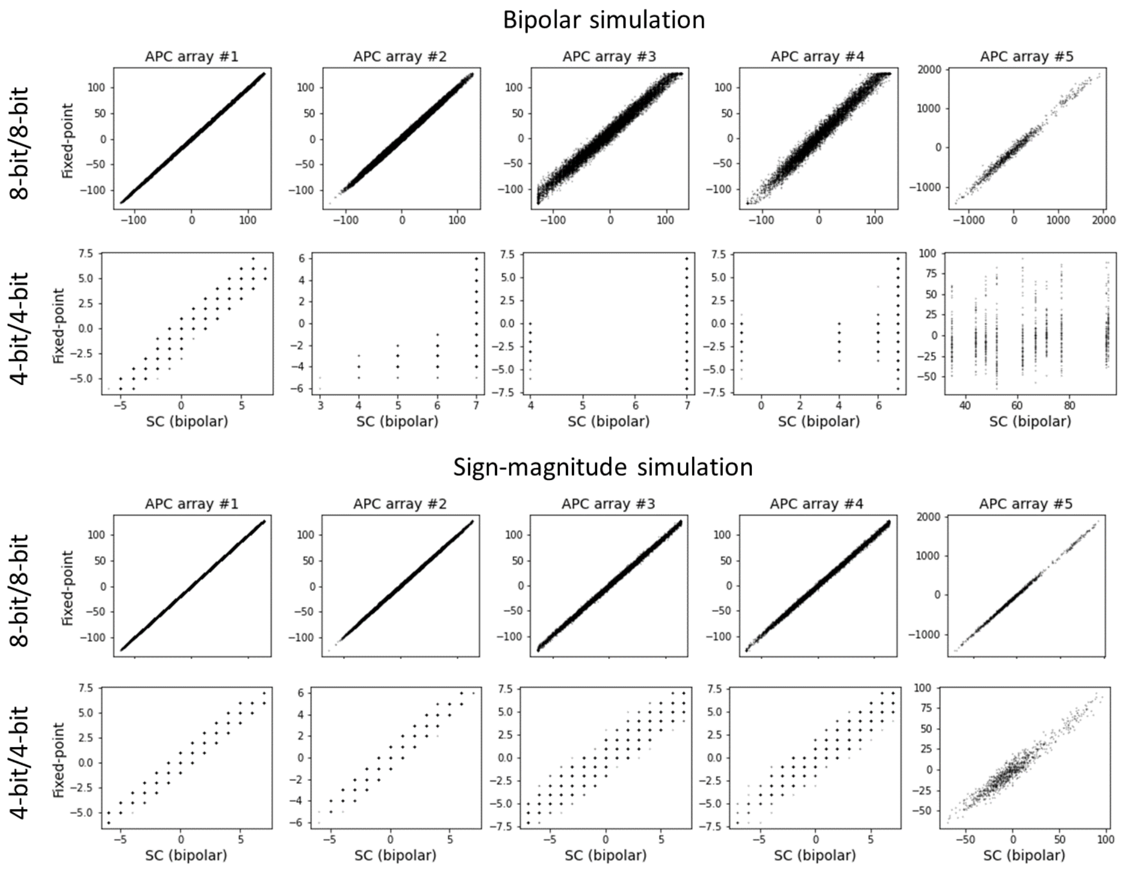


Figure 5.23: Fixed-point intermediate results as a function of simulated measurements at the corresponding intermediate APC arrays for both bipolar and sign-magnitude inference models. All data points are represented in the bipolar format after scaling, except the last one, which is not scaled in order to get more precision in the last layer. Here the notation n-bit/m-bit means that n bits are used to represent signals and the model parameters are 4-bit integers. Data points have been sampled from 100 different MNIST input samples.

5.2.6 Summary

A new quantization aware training approach targetting SC ANNs has been introduced to reduce the overall arithmetic error by modifying the resulting parameter distributions and iteratively quantizing parameter values, so that weights can be represented by integers with certain bit precision. Moreover, two different SC CNN layer models have been introduced for two different SC codings: bipolar and sign-magnitude. An SC version of the LeNet-5 has been simulated using both schemes, comparing the results with those obtained using an equivalent fixed-point forward propagation for different signal and weight bit widths. The simulations are based on a verified FPGA implementation for the bipolar case, together with corresponding VLSI synthesis. Moreover, according to the reported results, the quantization algorithm provides slightly better results for fixed-point models and improves those based on SC bipolar and sign-magnitude. The latter being almost equivalent to the fixed-point version. Further work in this line might explore e.g. binary or ternary weight quantization, which has not been included in this work, or sign-magnitude SC hardware implementations, which is as precise as fixed-point arithmetic and its latency is a half of the equivalent bipolar SC hardware.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In the present work, three main tasks have been developed according to the objectives initially established in Section 1.2. The first objective was related to the design and development of new hardware RC and ANN architectures to exploit the potential benefits of unconventional computing methodologies in terms of energy efficiency. The second objective was related to parallel hardware resource optimization strategies of ML algorithms. Additionally, the final system accuracy has been modeled and improved offline for fixed-point and SC implementations. The third and final objective was related to the FPGA implementation and evaluation of the proposed RC and ANN designs along this work. Since the main contributions are essentially structured in four sections, the relationship between each section and the three objectives is summarized below.

Ring topology Echo State Networks

(a) The proposed design includes architectural simplifications and low precision weights and activations, which contributes to make it more energy efficient than a floating-point ring topology reservoir with smooth activation functions.

(b) No fine tuning methods were required to match the overall accuracy obtained by the equivalent floating-point readout software implementation.

(c) The corresponding fixed-point FPGA implementation and evaluation was carried out successfully.

Reservoir Computing and Cellular Automata

(a) The proposed RC based on ECA feature expansion is more than 15 times more energy efficient than a binary weighted CNN implementation and requires about 8 times less FPGA logic resources, while maintaining a competitive accuracy (0.24% accuracy degradation w.r.t. the binary weighted CNN).

(b) An iterative first order gradient descent approach with direct fake 8-bit weight quantization has been proposed, achieving equivalent accuracy to that obtained by the limited memory BFGS algorithm with floating-point weights.

(c) The corresponding fixed-point FPGA implementation and evaluation has been carried out successfully.

Radial Basis Function Neural Networks

(a) The proposed RBF-NN SC hardware design obtained accuracies equivalent to those obtained in fixed-point for several datasets and there is still room for further optimizations in terms of weight and/or signal bit width, or even different RBF activations.

(b) The two RBF-NN layers were trained separately and the readout one is fine tuned using FPGA measurements from the hidden layer. Then the readout layer was trained with direct fake 8-bit weight quantization.

(c) The SC RBF-NN FPGA implementation and evaluation was successfully carried out for several datasets.

Convolutional Neural Networks

(a) The proposed parallel SC CNN designs are more energy efficient than other conventional approaches, mainly due to register sharing in CNN input feature maps feeding SC dot products.

(b) A specific backpropagation QAT approach was developed to improve SC LeNet-5 results. The simulations show the bipolar SC model reports competitive accuracy only for 8-bit signals, while the sign-magnitude SC model is almost equivalent to the fixed-point one for all weight and signal bit widths ranging from 3 to 8-bit.

(c) The corresponding bipolar SC FPGA implementation and VLSI synthesis was successfully carried out within the research group in collaboration with IMSE-CNM researchers[1].

Overall, the set of tasks carried out comply with the objectives set at the beginning of this thesis by far. It is worth highlighting the fact that some topics included here are probably consequence from the work done by previous PhD candidates who graduated within the research group. In chronological order, Dr. V. Canals (co-director of this thesis) already suggested the implementation of SC RBFNNs in 2012 as a possible extension to his PhD thesis [50], which has been successfully implemented in FPGA. Later, in 2017 M.L. Alomar proposed a multiplier-less ring topology RC architecture [230], which shares some similarities with the one introduced in Section 4.1, however, the original multiplier-less design was restricted to very low dimensional data and the training and evaluation approach was different from this work. Also in 2017, A. Oliver suggested the implementation of SC NNs to accelerate drug discovery in large data bases as a possible extension to his PhD thesis [231], which is not included in this thesis but hope the SC NN content described here will be helpful for any specific real life application such as drug discovery.

Additionally, during my PhD I participated in many interdisciplinary research activities. However, there are several activities which have not been included in this

[1] http://www.imse-cnm.csic.es/

thesis due to two main reasons. First, collaboration in works not directly related to the objectives set for this thesis. Second, work in progress with not enough conclusive results, mainly focused on exploring on-chip learning based on SC, including both supervised and unsupervised learning.

Finally, along the realization of the research work documented in this thesis, several publications have come to light and are listed in next section.

6.2 Dissemination of results

The aim of this section is to list our main contributions to the state of the art, including indexed international journals and international conference papers.

6.2.1 Contributions to indexed iternational journals

The following research papers are directly related to RC based on fixed-point implementations described in Chapter 4.

[138] A. Morán, C. F. Frasser, M. Roca, and J. L. Rosselló, "Energy-efficient pattern recognition hardware with elementary cellular automata," *IEEE Transactions on Computers*, vol. 69, no. 3, pp. 392–401, 2019

[137] A. Morán, V. Canals, F. Galan-Prado, C. F. Frasser, D. Radhakrishnan, S. Safavi, and J. L. Rosselló, "Hardware-optimized reservoir computing system for edge intelligence applications," *Cognitive Computation*, pp. 1–9, 2021

There are also two papers directly related to this thesis that are currently under review. The first one incorporates the SC RBFNN work described in Chapter 5.1 and the second one is partially related to the SC CNN work, respectively.

- A. Morán, V. Canals, L. Parrilla, C. F. Frasser, M. Roca, and J. L. Rosselló, "Inference based on stochastic computing radial basis functions," *IEEE transactions on neural networks*, 2021, under review

- C. F. Frasser, P. Linares-Serrano, A. Morán, J. Font-Rosselló, V. Canals, M. Roca, T. Serrano-Gotarredona, and J. L. Rosselló, "Fully-parallel stochastic computing design of convolutional neural networks for edge computing applications," *IEEE transactions on neural networks*, 2021, under review

In addition, I collaborated in the elaboration of a publication related to SC and SNNs, which has not been included as a separate hardware implementation since SNNs are out of the scope of this thesis. This was mainly developed by F. Galán-Prado as part of his thesis.

[234] F. Galán-Prado, A. Morán, J. Font, M. Roca, and J. L. Rosselló, "Compact hardware synthesis of stochastic spiking neural networks," *International journal of neural systems*, vol. 29, no. 08, p. 1 950 004, 2019

Finally, I published an additional paper that is not related to this thesis. This paper was the result of my MSc thesis, developed in collaboration with Dr. M.C. Soriano, but made during my PhD.

[235] A. Morán and M. C. Soriano, "Improving the quality of a collective signal in a consumer eeg headset," *Plos one*, vol. 13, no. 5, e0197597, 2018

6.2.2 Contributions to international conferences

The contributions directly related to ideas exposed throughout this document have been listed below.

[145] E. S. Skibinsky-Gitlin, M. L. Alomar, V. Canals, C. F. Frasser, E. Isern, F. Galán-Prado, A. Morán, M. Roca, and J. L. Rosselló, "Fpga-based echo-state networks," in *International Conference on Time Series and Forecasting*, Springer, 2018, pp. 135–146

[236] A. Morán, V. Canals, M. Roca, E. Isern, and J. L. Rosselló, "Fpga implementation of random vector functional link networks based on elementary cellular automata," in *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*, IEEE, 2020, pp. 1–6

Additionally, there is a recently submitted (unpublished) contribution:

- C. F. Frasser, P. Linares-Serrano, A. Morán, J. Font-Rosselló, V. Canals, M. Roca, T. Serrano-Gotarredona, and J. L. Rosselló, "Exploiting correlation in stochastic computing based deep neural networks," in *2021 XXXVI Conference on Design of Circuits and Integrated Systems (DCIS)*, IEEE, 2021, pp. 1–6, accepted for oral presentation

There is also a contribution based on the research paper from reference [234], however, it is not directly related to the topics discussed within this document. It exploits ideas from both SC and SNNs:

[238] F. Galán-Prado, A. Morán, J. Font, M. Roca, and J. L. Rosselló, "Stochastic radial basis neural networks," in *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, IEEE, 2019, pp. 145–149

There are two additional contributions in international conferences not related to the content exposed so far. These contributions reflect the first attempts to implement an unsupervised learning algorithm using SC digital hardware and are listed below.

[239] A. Morán, J. L. Rosselló, M. Roca, E. Isern, V. Martínez-Moll, and V. Canals, "Self-organizing maps hybrid implementation based on stochastic computing," in *2019 XXXIV Conference on Design of Circuits and Integrated Systems (DCIS)*, IEEE, 2019, pp. 1–6

[240] A. Morán, J. L. Rosselló, M. Roca, and V. Canals, "Soc kohonen maps based on stochastic computing," in *2020 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2020, pp. 1–7

Finally, very recent (unpublished) work also related to unsupervised on-chip learning describing how SC can be exploited to implement the evolving Autonomous Data Partitioning algorithm [241] has been submitted as a conference paper, see reference below.

- A. Morán, V. Canals, M. Roca, E. Isern, P. Angelov, and J. L. Rosselló, "Stochastic computing co-processing elements for evolving autonomous data partitioning," in *2021 XXXVI Conference on Design of Circuits and Integrated Systems (DCIS)*, IEEE, 2021, pp. 1–6, accepted for oral presentation

6.3 Future work

The knowledge acquired throughout the realization of this thesis encourage us to consider the following research lines may be of great interest for the future embedded ML/DL systems in the industry.

- The RC implementations described here are simple enough because the main goal was to reduce the overall energy consumption but these architectures might not be suitable for certain real life scenarios. Therefore, if the current architectures are not complex enough and increasing the reservoir size do not yield acceptable results, then it makes sense to implement higher complexity architectures, e.g. the ring topology ESN might be replaced by higher dimensional grid reservoirs with more than two nearest neighbors, and the ECA based reservoir or RVFL-like feature expansion might be substituted by unconstrained nearest neighbor rules for any number of spatial dimensions at the cost of having a much larger search space.

- Even though fully parallel SC implementations are very fast and in the case of the SC CNN more energy efficient than conventional instruction based computations, we have to admit that the required logic resources make the silicon implementation of large state-of-the-art CNNs not feasible in practice. Therefore, it would be interesting to implement an SC co-processor to accelerate the inference process in CNNs and ANNs in general, so that calculations would be sequenced and controlled by additional custom logic or a soft core rather than fully parallel.

- A relatively new research on on-chip learning motivated by a recent collaboration with the Lancaster Intelligent, Robotic and Autonomous systems (LIRA) Research Centre[2] is currently a work in progress and some preliminary are already documented. Current results are based on SC co-processing elements but in this case, we should not rule out the possibility of ending up manufacturing a more conventional design, which might be more suitable in terms of precision and flexibility.

- Finally, as already mentioned in the beginning (see Section 1.3) here we limited the scope of this thesis to digital designs and mainly FPGA prototyping but in the case of SC logic, it would benefit from a mixed-signal in-memory computing scheme, reducing the implementation cost by substituting expensive D-type flip-flops by simpler and cheaper SRAM memory cells and being able to introduce further optimizations by implementing APC logic with analog components. Although there is nothing done on this line, it is certainly a topic that would be worth exploring in the near future.

[2]https://www.lancaster.ac.uk/lira/

Appendix A

Gradient Descent Optimization

A.1 Single layer

A.2 Momentum

Taking as starting point Algorithm 1, gradient descent with momentum or Nesterov accelerated gradient borrows from physics the *momentum* term to *move* the parameter vector towards an optimal value with some inertia. If the momentum term is $\gamma \boldsymbol{M}$ and the parameters to be updated are $\boldsymbol{W}$, then computing $\boldsymbol{W} - \gamma \boldsymbol{M}$ gives an approximation of the next position of the parameters. This fact is reflected in Algorithm 4, which has the same purpose as Algorithm 1 iteration, but Adam converges faster [243]. Here γ is an additional hyperparameterl, however it is usually set to $\gamma = 0.9$ by default.

Algorithm 4: Nesterov accelerated gradient descent step.

```
**Input:** Hyperparameters $\alpha, \gamma$ and weight matrix $\boldsymbol{W}$
**Result:** Updated weight matrix
$\boldsymbol{M} := \gamma \boldsymbol{M} + \alpha \nabla_{\boldsymbol{W}} J(\boldsymbol{X}, \boldsymbol{W} - \gamma \boldsymbol{M}, \boldsymbol{Y})$;
$\boldsymbol{W} := \boldsymbol{W} - \gamma \boldsymbol{M}$;
```

A.3 Adam

Adaptive moment estimation (Adam) stores the exponential moving average (EMA) of the past gradients and squared gradients [77]. As regards the first momentum, the motivation is similar to Momentum, except that in this case $\boldsymbol{M}$ is the EMA of the Momentum, i.e. it is updated as

$$\boldsymbol{M} := \beta_1 \boldsymbol{M} + (1 - \beta_1) \nabla_{\boldsymbol{W}} J(\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{Y}) \tag{A.1}$$

where β_1 is the corresponding EMA coefficient. Similarly, for the second moment

$$\boldsymbol{V} := \beta_2 \boldsymbol{V} + (1 - \beta_2)(\nabla_{\boldsymbol{W}} J(\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{Y}))^2 \tag{A.2}$$

where $\boldsymbol{V}$ is the second momentum EMA and β_2 is the corresponding EMA coefficient.

Since EMA approximations need to be initialized before starting the training process, if $\boldsymbol{M}$ and $\boldsymbol{V}$ are initialized with zeros, then EMAs will be biased towards zero, at least during the initial iterations. This is a problem the authors of Adam solved using the bias-corrected first and second moment estimates $\hat{\boldsymbol{M}}$ and $\hat{\boldsymbol{V}}$, given by (A.3).

$$\hat{\boldsymbol{M}} = \frac{\boldsymbol{M}}{1-\beta_1^t} \qquad\qquad \hat{\boldsymbol{V}} = \frac{\boldsymbol{V}}{1-\beta_2^t} \tag{A.3}$$

where t is the time step or iteration index. Finally, the weights are updated according to (A.4).

$$\boldsymbol{W} := \boldsymbol{W} - \frac{\alpha}{\sqrt{\hat{\boldsymbol{V}}}+\epsilon} \odot \hat{\boldsymbol{M}} \tag{A.4}$$

All these steps have been summarized in Algorithm 5. The authors proposed $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ as default hyperparameters after comparing their simulations against other adaptive gradient descent methods.

Algorithm 5: Adam gradient descent step.

```
**Input:** Hyperparameters $\alpha, \beta_1, \beta_2, \epsilon$, time step $t$ and weight matrix $\boldsymbol{W}$
**Result:** Updated weight matrix
$\boldsymbol{G} := \nabla_{\boldsymbol{W}} J(\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{Y})$;
$\boldsymbol{M} := \beta_1 \boldsymbol{M} + (1-\beta_1)\boldsymbol{G}$;
$\boldsymbol{V} := \beta_2 \boldsymbol{V} + (1-\beta_2)\boldsymbol{G}^2$;
$\hat{\boldsymbol{M}} := \frac{\boldsymbol{M}}{1-\beta_1^t}$;
$\hat{\boldsymbol{V}} := \frac{\boldsymbol{V}}{1-\beta_2^t}$;
$\boldsymbol{W} := \boldsymbol{W} - \frac{\alpha}{\sqrt{\hat{\boldsymbol{V}}}+\epsilon} \odot \hat{\boldsymbol{M}}$;
```

Appendix B

Fixed-Point Arithmetic

This appendix contains common unsigned and signed fixed point arithmetic operations used throuout this document and listed in Table B.1 and B.2.

Unsigned Binary Weighted

Description	Operation	Condition	Output range
division by 2^n (shift right)	$z = \lfloor \frac{2}{2^n} \rfloor$	-	$\mathbb{Z} \cap \left[0, 2^{B_x - n} - 1\right]$
multiplication by 2^n (shift left)	$z = 2^n x$	$B_z \geq n + B_x$	$\mathbb{Z} \cap \left[0, 2^{B_x + n} - 1\right]$
multiplication by $1 - \frac{1}{2^n}$	$z = x - \lfloor \frac{x}{2^n} \rfloor$	-	$\mathbb{Z} \cap \left[0, 2^{B_x - n} - 1\right]$
multiplication by n	$z = nx$	$B_z \geq \log_2 \lceil n \rceil + B_x$	$\mathbb{Z} \cap \left[0, 2^{B_x + \lceil \log_2 n \rceil} - 1\right]$

Table B.1: Quantization behaviour for binary weighted coded unsigned numbers under the operations listed in this table. Shift operations are padded with zeros and B_x represents the bitwidth of x.

Two's Complement

Description	Operation	Condition	Output range
sign flip	$z = -x$	-	$\mathbb{Z} \cap \left[-2^{B_x - 1}, 2^{B_x - 1} - 1\right]$
division by 2^n (shift right[a])	$z = \lfloor \frac{2}{2^n} \rfloor$	$n \geq 0$	$\mathbb{Z} \cap \left[-2^{B_x - 1 - n}, 2^{B_x - 1 - n} - 1\right]$
multiplication by 2^n (shift left[b])	$z = 2^n x$	$B_z \geq n + B_x$	$\mathbb{Z} \cap \left[-2^{B_x - 1 + n}, 2^{B_x - 1 + n} - 1\right]$
multiplication by $1 - \frac{1}{2^n}$	$z = x - \lfloor \frac{x}{2^n} \rfloor$	-	$\mathbb{Z} \cap \left[-2^{B_x - 1 - n}, 2^{B_x - 1 - n} - 1\right]$
multiplication by n	$z = nx$	$B_z \geq \log_2 \lceil \mathrm{sgn}(n) n \rceil + B_x$	$\mathbb{Z} \cap \left[-2^{B_x - 1 + \lceil \log_2 n \rceil}, 2^{B_x - 1 + \lceil \log_2 n \rceil} - 1\right]$

Table B.2: Quantization behaviour for two's complement coded unsigned numbers under the operations listed in this table. Shift operations are padded with zeros and B_x represents the bitwidth of x. (a, b) Operations to perform division (multiplication) by a power of 2 are not exactly shift right (left) operations, see a detailed explanation in the text.

Appendix C

Random Number Generation

C.1 Linear Feedback Shift Register

An n-bit LFSR is composed by an n-bit shift register, i.e. an array of D-type flip-flops connected in series, and a feedback bit obtained by combining certain current state bit values, which is connected to the first flip-flop. Fig. C.1 shows an example LFSR block diagram for $n = 8$.

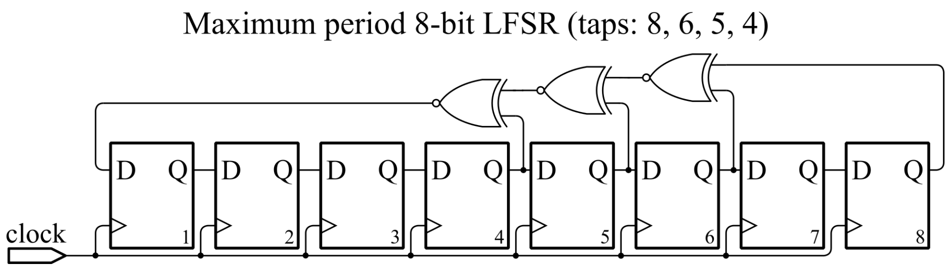


Figure C.1: Maximum length 8-bit LFSR block diagram.

In particular, we are interested in maximum length LFSR sequences (period $2^n - 1$, i.e. the zero is excluded since it results in all zeros next state) since in some cases it might be desirable to maximize SC number representation resolution. In order to obtain a maximum length sequence, the feedback bit must be generated from the XNOR combination of coprime taps, i.e. D-type flip-flop states from coprime positions, and the number of taps must be even, as in the 8-bit LFSR example shown in Fig. C.1. The interested reader can find a very complete list of taps corresponding to maximum length LFSR sequences for different n values in [244]. Due to its compact hardware implamentation compared to the rest of required SC logic resources, LFSRs are typically implemented on-chip.

C.2 rng_n1024_r32_t5_k32_s1c48

A general class of uniform random number generators based on LUT and shift register FPGA logic (LUT-SR) was introduced in [203] and the one utilized here is just a particular implementation within this more general framework. An example block diagram description of this type of design has been included in Fig. C.2.

Based on 5 different parameters (n, r, t, k and s), the PRNG structure becomes completely defined. Where n is the number of internal state bits in the RNG, so that the

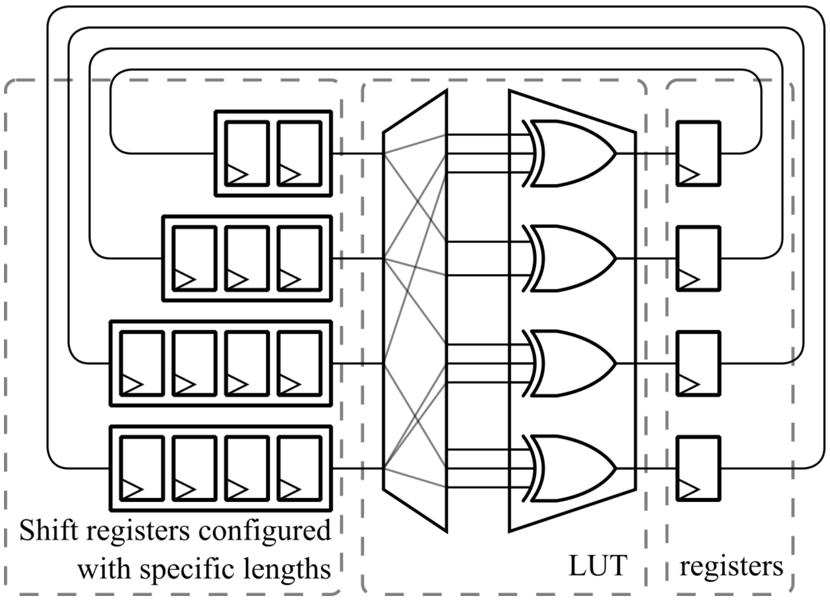


Figure C.2: Block diagram and connection characteristics of the LUT-SR class of PRNG. This figure has been adapted from [203].

period is $2^n - 1$, r is the output bit width, t is the number of single bit signals connected to XOR gate inputs, k is the maximum shift-register length and s is a parameter which needs to be chosen carefully for each parameter set (n, r, t, k). Fortunately, the paper provides a set of *valid* s values for some parameter combinations and C++ source code to generate hardwired HDL connections, defining a specific architecture. In our case we used a VHDL source code provided by Dr. L. Parrilla, which implements the PRNG for the following parameter set: $n = 1024$, $r = 32$, $t = 5$, $k = 32$ and $s = 0.1C48_{16}$, hence the name. This PRNG has been synthesized on-chip for the RBFNN bitstream generation. Even though it requires more logic resources than a simple LFSR, the required resources are negligible compared to the whole SC RBFNN design.

C.3 The ROM approach

Another possible and more flexible approach is to store pre-computed random sequences in a ROM and use them for bitstream generation, so that the stored numbers are not limited to any specific class of sequence generator. Although the hardware overhead is higher than in the previous on-chip implementations it allows the user to store any finite sequence and do not represent significative resource utilization when compared to a fully parallel SC design for relatively short sequences (e.g. 8-bit SC signal). In addition, the required logic is reduced by more than a half for a sign-magnitude implementation compared to the equivalent bipolar implementation because each number requires 1 bit less resolution and the sequence length required to achieve the same signal resolution is reduced by a half.

Bibliography

[1] M. P. Deisenroth, A. A. Faisal, and C. S. Ong, *Mathematics for machine learning*. Cambridge University Press, 2020.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[3] G. E. Moore *et al.*, *Cramming more components onto integrated circuits*, 1965.

[4] N. S. Kim, T. Austin, D. Baauw, *et al.*, "Leakage current: Moore's law meets static power," *computer*, vol. 36, no. 12, pp. 68–75, 2003.

[5] G. E. Moore *et al.*, "Moore's law at 40," *Understanding Moore's law: four decades of innovation*, pp. 67–84, 2006.

[6] C. Disco and B. van der Meulen, *Getting new technologies together: Studies in making sociotechnical order*. Walter de Gruyter, 1998, vol. 82.

[7] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[8] M. Bohr, "A 30 year retrospective on dennard's mosfet scaling paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[9] K. Mistry, "Tri-gate transistors: Enabling moore's law at 22nm and beyond," *Presentation at Semicon West*, 2014.

[10] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," *Turing Lecture*, 2018.

[11] D. Chen, J. Cong, S. Gurumani, W.-m. Hwu, K. Rupnow, and Z. Zhang, "Platform choices and design demands for iot platforms: Cost, power, and performance tradeoffs," *IET Cyber-Physical Systems: Theory & Applications*, vol. 1, no. 1, pp. 70–77, 2016.

[12] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[13] D. E. O'Leary, "Artificial intelligence and big data," *IEEE intelligent systems*, vol. 28, no. 2, pp. 96–99, 2013.

[14] Y. Morita, H. Fujiwara, H. Noguchi, *et al.*, "Area optimization in 6t and 8t sram cells considering v th variation in future processes," *IEICE transactions on electronics*, vol. 90, no. 10, pp. 1949–1956, 2007.

[15] J. Salamon, C. Jacoby, and J. P. Bello, “A dataset and taxonomy for urban sound research,” in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 1041–1044.

[16] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[17] E. Blem, J. Menon, and K. Sankaralingam, “A detailed analysis of contemporary arm and x86 architectures,” *UW-Madison Technical Report*, 2013.

[18] F. Gray, “Pulse code communication,” *United States Patent Number 2632058*, 1953.

[19] E. R. Associates, *High-speed Computing Devices by the Staff of Engineering Research Associates, Inc.* McGraw-Hill, 1950.

[20] W. Poppelbaum, A. Dollas, J. Glickman, and C O’Toole, “Unary processing,” in *Advances in computers*, vol. 26, Elsevier, 1987, pp. 47–92.

[21] J. J. Hopfield, “Pattern recognition computation using action potential timing for stimulus representation,” *Nature*, vol. 376, no. 6535, pp. 33–36, 1995.

[22] T. D. Sanger, “A probability interpretation of neural population coding for movement,” in *Advances in Psychology*, vol. 119, Elsevier, 1997, pp. 75–116.

[23] L. C. Gouveia, T. J. Koickal, and A. Hamilton, “An asynchronous spike event coding scheme for programmable analog arrays,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 4, pp. 791–799, 2010.

[24] C. R. Gallistel, “The coding question,” *Trends in Cognitive Sciences*, vol. 21, no. 7, pp. 498–508, 2017.

[25] V. T. Lee, A. Alaghi, J. P. Hayes, V. Sathe, and L. Ceze, “Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, IEEE, 2017, pp. 13–18.

[26] A Van der Ziel, “Thermal noise in field-effect transistors,” *Proceedings of the IRE*, vol. 50, no. 8, pp. 1808–1812, 1962.

[27] H. Tian, B. Fowler, and A. E. Gamal, “Analysis of temporal noise in cmos photodiode active pixel sensor,” *IEEE Journal of Solid-State Circuits*, vol. 36, no. 1, pp. 92–101, 2001.

[28] L. B. Kish, “End of moore’s law: Thermal (noise) death of integration in micro and nano electronics,” *Physics Letters A*, vol. 305, no. 3-4, pp. 144–149, 2002.

[29] K. H. Lundberg, “Noise sources in bulk cmos,” *Unpublished paper*, vol. 3, p. 28, 2002.

[30] G. Ghibaudo and T Boutchacha, “Electrical noise and rts fluctuations in advanced cmos devices,” *Microelectronics Reliability*, vol. 42, no. 4-5, pp. 573–582, 2002.

[31] B. R. Gaines, “Stochastic computing,” in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 149–156.

[32] J. Von Neumann, “Probabilistic logics and the synthesis of reliable organisms from unreliable components,” *Automata studies*, vol. 34, pp. 43–98, 1956.

[33] B. R. Gaines, "Origins of stochastic computing," in *Stochastic Computing: Techniques and Applications*, Springer, 2019, pp. 13–37.

[34] B. Gaines, "A stochastic analog computer," *Standard Telecommunication Laboratories Internal Memorandum*, pp. 1–10, 1965.

[35] W. Poppelbaum and C Afuso, "Noise-computer," *Univ. Illinois, Urbana, Dept. Computer Science, Quart. Tech. Progress Rep*, 1965.

[36] W. Poppelbaum, C Afuso, and J. Esch, "Stochastic computing elements and systems," in *Proceedings of the November 14-16, 1967, fall joint computer conference*, 1967, pp. 635–644.

[37] S. T. Ribeiro, "Comments on pulsed-data hybrid computers," *IEEE Transactions on Electronic Computers*, no. 5, pp. 640–642, 1964.

[38] ——, "Random-pulse machines," *IEEE Transactions on Electronic Computers*, no. 3, pp. 261–276, 1967.

[39] B. R. Gaines, "Stochastic computing systems," in *Advances in information systems science*, Springer, 1969, pp. 37–172.

[40] J. W. Esch, "-rascel-a programmable analog computer based on a regular array of stochastic computing element logic," 1969.

[41] P. Li and D. J. Lilja, "Using stochastic computing to implement digital image processing algorithms," in *2011 IEEE 29th International Conference on Computer Design (ICCD)*, IEEE, 2011, pp. 154–161.

[42] A. Alaghi, C. Li, and J. P. Hayes, "Stochastic circuits for real-time image-processing applications," in *Proceedings of the 50th Annual Design Automation Conference*, 2013, pp. 1–6.

[43] K. Kim, J. Kim, J. Yu, J. Seo, J. Lee, and K. Choi, "Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[44] A. Ardakani, F. Leduc-Primeau, N. Onizawa, T. Hanyu, and W. J. Gross, "Vlsi implementation of deep neural network using integral stochastic computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2688–2699, 2017.

[45] A. Ren, Z. Li, C. Ding, *et al.*, "Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 405–418, 2017.

[46] K. Kollmann, K.-R. Riemschneider, and H. C. Zeidler, "On-chip backpropagation training using parallel stochastic bit streams," in *Proceedings of Fifth International Conference on Microelectronics for Neural Networks*, IEEE, 1996, pp. 149–156.

[47] S. Liu, H. Jiang, L. Liu, and J. Han, "Gradient descent using stochastic circuits for efficient training of learning machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2530–2541, 2018.

[48] B. Yuan, Y. Wang, and Z. Wang, "Area-efficient scaling-free dft/fft design using stochastic computing," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 63, no. 12, pp. 1131–1135, 2016.

[49] V. Canals, A. Morro, A. Oliver, M. L. Alomar, and J. L. Rosselló, "A new stochastic computing methodology for efficient neural network implementation," *IEEE transactions on neural networks and learning systems*, vol. 27, no. 3, pp. 551–564, 2015.

[50] V. J. Canals Guinand *et al.*, "Implementación en hardware de sistemas de alta fiabilidad basados en metodologías estocásticas," Ph.D. dissertation, Universitat de les Illes Balears, 2017.

[51] B. D. Brown and H. C. Card, "Stochastic neural computation. i. computational elements," *IEEE Transactions on computers*, vol. 50, no. 9, pp. 891–905, 2001.

[52] P. Li, W. Qian, M. D. Riedel, K. Bazargan, and D. J. Lilja, "The synthesis of linear finite state machine-based stochastic computational elements," in *17th Asia and South Pacific Design Automation Conference*, IEEE, 2012, pp. 757–762.

[53] P. K. Gupta and R. Kumaresan, "Binary multiplication with pn sequences," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, no. 4, pp. 603–606, 1988.

[54] R. Eckhardt, S. Ulam, and J. Von Neumann, "The monte carlo method," *Los Alamos Science*, vol. 15, p. 131, 1987.

[55] R. E. Caflisch *et al.*, "Monte carlo and quasi-monte carlo methods," *Acta numerica*, vol. 1998, pp. 1–49, 1998.

[56] S. Liu and J. Han, "Energy efficient stochastic computing with sobol sequences," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, IEEE, 2017, pp. 650–653.

[57] J. Yu, K. Kim, J. Lee, and K. Choi, "Accurate and efficient stochastic computing hardware for convolutional neural networks," in *2017 IEEE International Conference on Computer Design (ICCD)*, IEEE, 2017, pp. 105–112.

[58] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolić, *Digital integrated circuits: a design perspective*. Pearson education Upper Saddle River, NJ, 2003, vol. 7.

[59] R. H. Johns and C. A. Doswell III, "Severe local storms forecasting," *Weather and Forecasting*, vol. 7, no. 4, pp. 588–612, 1992.

[60] L. Muda, M. Begam, and I. Elamvazuthi, "Voice recognition algorithms using mel frequency cepstral coefficient (mfcc) and dynamic time warping (dtw) techniques," *arXiv preprint arXiv:1003.4083*, 2010.

[61] Z.-H. Zhou, Y. Jiang, Y.-B. Yang, and S.-F. Chen, "Lung cancer cell identification based on artificial neural network ensembles," *Artificial Intelligence in Medicine*, vol. 24, no. 1, pp. 25–36, 2002.

[62] P. Baldi and Y. Chauvin, "Neural networks for fingerprint recognition," *neural computation*, vol. 5, no. 3, pp. 402–418, 1993.

[63] J. Zhang, Y. Yan, and M. Lades, "Face recognition: Eigenface, elastic matching, and neural nets," *Proceedings of the IEEE*, vol. 85, no. 9, pp. 1423–1435, 1997.

[64] Y. Sun, Y. Chen, X. Wang, and X. Tang, "Deep learning face representation by joint identification-verification," in *Advances in neural information processing systems*, 2014, pp. 1988–1996.

[65] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[66] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017.

[67] J. Mao, W. Xu, Y. Yang, J. Wang, Z. Huang, and A. Yuille, "Deep captioning with multimodal recurrent neural networks (m-rnn)," *arXiv preprint arXiv:1412.6632*, 2014.

[68] R. K. McConnell, *Method of and apparatus for pattern recognition*, US Patent 4,567,610, 1986.

[69] M. Xu, L.-Y. Duan, J. Cai, L.-T. Chia, C. Xu, and Q. Tian, "Hmm-based audio keyword generation," in *Pacific-Rim Conference on Multimedia*, Springer, 2004, pp. 566–574.

[70] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[71] J. Hestness, S. Narang, N. Ardalani, *et al.*, "Deep learning scaling is predictable, empirically," *arXiv preprint arXiv:1712.00409*, 2017.

[72] M. Meyer, L. Cavigelli, and L. Thiele, "Efficient convolutional neural network for audio event detection," *arXiv preprint arXiv:1709.09888*, 2017.

[73] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[74] R. Fletcher, *Practical methods of optimization*. John Wiley & Sons, 2013.

[75] J. Pearl, "Intelligent search strategies for computer problem solving," *Addision Wesley*, 1984.

[76] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[77] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[78] J. Nocedal, "Updating quasi-newton matrices with limited storage," *Mathematics of computation*, vol. 35, no. 151, pp. 773–782, 1980.

[79] S. Kullback and R. A. Leibler, "On information and sufficiency," *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[80] J. D. Rennie, "Regularized logistic regression is strictly convex," *Unpublished manuscript. URL people. csail. mit. edu/jrennie/writing/convexLR. pdf*, vol. 745, 2005.

[81] Y. Peng, J. Ke, S. Liu, J. Li, and T. Lei, "An improvement to linear regression classification for face recognition," *International Journal of Machine Learning and Cybernetics*, vol. 10, no. 9, pp. 2229–2243, 2019.

[82] X. Huang and W. Pan, "Linear regression and two-class classification with gene expression data," *Bioinformatics*, vol. 19, no. 16, pp. 2072–2078, 2003.

[83] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[84] T. Sasaki, N. Matsuki, and Y. Ikegaya, “Action-potential modulation during axonal conduction,” *Science*, vol. 331, no. 6017, pp. 599–601, 2011.

[85] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, *et al.*, “A million spiking-neuron integrated circuit with a scalable communication network and interface,” *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[86] M. Davies, N. Srinivasa, T.-H. Lin, *et al.*, “Loihi: A neuromorphic manycore processor with on-chip learning,” *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[87] A. L. Hodgkin and A. F. Huxley, “A quantitative description of membrane current and its application to conduction and excitation in nerve,” *The Journal of physiology*, vol. 117, no. 4, p. 500, 1952.

[88] C. Morris and H. Lecar, “Voltage oscillations in the barnacle giant muscle fiber,” *Biophysical journal*, vol. 35, no. 1, pp. 193–213, 1981.

[89] J. H. Lee, T. Delbruck, and M. Pfeiffer, “Training deep spiking neural networks using backpropagation,” *Frontiers in neuroscience*, vol. 10, p. 508, 2016.

[90] D. S. Broomhead and D. Lowe, “Radial basis functions, multi-variable functional interpolation and adaptive networks,” Royal Signals and Radar Establishment Malvern (United Kingdom), Tech. Rep., 1988.

[91] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, “Extreme learning machine: A new learning scheme of feedforward neural networks,” in *2004 IEEE international joint conference on neural networks (IEEE Cat. No. 04CH37541)*, Ieee, vol. 2, 2004, pp. 985–990.

[92] L. P. Wang and C. R. Wan, “Comments on” the extreme learning machine,” *IEEE Transactions on Neural Networks*, vol. 19, no. 8, pp. 1494–1495, 2008.

[93] G.-B. Huang, “Reply to “comments on “the extreme learning machine””,” *IEEE Transactions on Neural Networks*, vol. 19, no. 8, pp. 1495–1496, 2008.

[94] Y.-H. Pao, G.-H. Park, and D. J. Sobajic, “Learning and generalization characteristics of the random vector functional-link net,” *Neurocomputing*, vol. 6, no. 2, pp. 163–180, 1994.

[95] B. Igelnik and Y.-H. Pao, “Stochastic choice of basis functions in adaptive function approximation and the functional-link net,” *IEEE transactions on Neural Networks*, vol. 6, no. 6, pp. 1320–1329, 1995.

[96] H. Steinhaus, “Sur la division des corps matériels en parties,” *Bull. Acad. Polon. Sci*, vol. 1, no. 804, p. 801, 1956.

[97] J. MacQueen *et al.*, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Oakland, CA, USA, vol. 1, 1967, pp. 281–297.

[98] B. Scholkopf, K.-K. Sung, C. J. Burges, *et al.*, “Comparing support vector machines with gaussian kernels to radial basis function classifiers,” *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2758–2765, 1997.

[99] A. P. Dempster, N. M. Laird, and D. B. Rubin, “Maximum likelihood from incomplete data via the em algorithm,” *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 39, no. 1, pp. 1–22, 1977.

[100] P. P. Angelov, X. Gu, and J. C. Príncipe, "Autonomous learning multimodel systems from data streams," *IEEE Transactions on Fuzzy Systems*, vol. 26, no. 4, pp. 2213–2224, 2017.

[101] R. D. Neidinger, "Introduction to automatic differentiation and matlab object-oriented programming," *SIAM review*, vol. 52, no. 3, pp. 545–563, 2010.

[102] A. Paszke, S. Gross, S. Chintala, *et al.*, "Automatic differentiation in pytorch," 2017.

[103] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *arXiv preprint arXiv:1912.01703*, 2019.

[104] M. Abadi, P. Barham, J. Chen, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[105] H. Jaeger, "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note," *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, vol. 148, no. 34, p. 13, 2001.

[106] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: A new framework for neural computation based on perturbations," *Neural computation*, vol. 14, no. 11, pp. 2531–2560, 2002.

[107] A. Robinson and F. Fallside, *The utility driven dynamic error propagation network*. University of Cambridge Department of Engineering Cambridge, MA, 1987.

[108] P. J. Werbos, "Generalization of backpropagation with application to a recurrent gas market model," *Neural networks*, vol. 1, no. 4, pp. 339–356, 1988.

[109] M. C. Mozer, "A focused back-propagation algorithm for temporal pattern recognition," *Complex systems*, vol. 3, no. 4, pp. 349–381, 1989.

[110] Y. Jin and P. Li, "Ap-stdp: A novel self-organizing mechanism for efficient reservoir computing," in *2016 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2016, pp. 1158–1165.

[111] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[112] K. Cho, B. Van Merriënboer, C. Gulcehre, *et al.*, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[113] A. A. Rad, M. Jalili, and M. Hasler, "Reservoir optimization in recurrent neural networks using kronecker kernels," in *2008 IEEE International Symposium on Circuits and Systems*, IEEE, 2008, pp. 868–871.

[114] M. Lukoševičius, H. Jaeger, and B. Schrauwen, "Reservoir computing trends," *KI-Künstliche Intelligenz*, vol. 26, no. 4, pp. 365–371, 2012.

[115] B. Schrauwen, M. D'Haene, D. Verstraeten, and J. Van Campenhout, "Compact hardware liquid state machines on fpga for real-time speech recognition," *Neural networks*, vol. 21, no. 2-3, pp. 511–523, 2008.

[116] F. Schürmann, K. Meier, and J. Schemmel, "Edge of chaos computation in mixed-mode vlsi-a hard liquid," *Advances in neural information processing systems*, vol. 17, pp. 1201–1208, 2004.

[117] M. C. Soriano, S. Ortín, L. Keuninckx, *et al.*, "Delay-based reservoir computing: Noise effects in a combined analog and digital implementation," *IEEE transactions on neural networks and learning systems*, vol. 26, no. 2, pp. 388–393, 2014.

[118] L. Larger, M. C. Soriano, D. Brunner, *et al.*, "Photonic information processing beyond turing: An optoelectronic implementation of reservoir computing," *Optics express*, vol. 20, no. 3, pp. 3241–3249, 2012.

[119] Y. Paquot, F. Duport, A. Smerieri, *et al.*, "Optoelectronic reservoir computing," *Scientific reports*, vol. 2, no. 1, pp. 1–6, 2012.

[120] K. Vandoorne, W. Dierckx, B. Schrauwen, *et al.*, "Toward optical signal processing using photonic reservoir computing," *Optics express*, vol. 16, no. 15, pp. 11 182–11 192, 2008.

[121] K. Fujii and K. Nakajima, "Harnessing disordered-ensemble quantum dynamics for machine learning," *Physical Review Applied*, vol. 8, no. 2, p. 024 030, 2017.

[122] G. Tanaka, T. Yamane, J. B. Héroux, *et al.*, "Recent advances in physical reservoir computing: A review," *Neural Networks*, vol. 115, pp. 100–123, 2019.

[123] K. Nakajima, "Physical reservoir computing—an introductory perspective," *Japanese Journal of Applied Physics*, vol. 59, no. 6, p. 060 501, 2020.

[124] C. Fernando and S. Sojakka, "Pattern recognition in a bucket," in *European conference on artificial life*, Springer, 2003, pp. 588–597.

[125] M.-S. Lin, T.-C. Huang, C.-C. Tsai, *et al.*, "A 7-nm 4-ghz arm^1-core-based cowos^1 chiplet design for high-performance computing," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 4, pp. 956–966, 2020.

[126] A. Boutros, S. Yazdanshenas, and V. Betz, "You cannot improve what you do not measure: Fpga vs. asic efficiency gaps for convolutional neural network inference," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[127] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "Neurostream: Scalable and energy efficient deep learning with smart memory cubes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 420–434, 2017.

[128] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl™ deep learning accelerator on arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 55–64.

[129] S.-C. Luo, "Customization of a deep learning accelerator," in *2019 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, IEEE, 2019, pp. 1–2.

[130] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[131] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International conference on machine learning*, PMLR, 2015, pp. 1737–1746.

[132] A. Goel, C. Tung, Y.-H. Lu, and G. K. Thiruvathukal, "A survey of methods for low-power deep learning and computer vision," in *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, IEEE, 2020, pp. 1–6.

[133] D. Miyashita, S. Kousai, T. Suzuki, and J. Deguchi, "A neuromorphic chip optimized for deep learning and cmos technology with time-domain analog and digital mixed-signal processing," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 10, pp. 2679–2689, 2017.

[134] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High-performance fpga-based cnn accelerator with block-floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1874–1885, 2019.

[135] G. Feng, Z. Hu, S. Chen, and F. Wu, "Energy-efficient and high-throughput fpga-based accelerator for convolutional neural networks," in *2016 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, IEEE, 2016, pp. 624–626.

[136] E. Nurvitadhi, G. Venkatesh, J. Sim, *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 5–14.

[137] A. Morán, V. Canals, F. Galan-Prado, *et al.*, "Hardware-optimized reservoir computing system for edge intelligence applications," *Cognitive Computation*, pp. 1–9, 2021.

[138] A. Morán, C. F. Frasser, M. Roca, and J. L. Rosselló, "Energy-efficient pattern recognition hardware with elementary cellular automata," *IEEE Transactions on Computers*, vol. 69, no. 3, pp. 392–401, 2019.

[139] M. Alomar, E. S. Skibinsky-Gitlin, C. F. Frasser, *et al.*, "Efficient parallel implementation of reservoir computing systems," *Neural Computing and Applications*, vol. 32, no. 7, pp. 2299–2313, 2020.

[140] A. Rodan and P. Tino, "Minimum complexity echo state network," *IEEE transactions on neural networks*, vol. 22, no. 1, pp. 131–144, 2010.

[141] L. Appeltant, M. C. Soriano, G. Van der Sande, *et al.*, "Information processing using a single dynamical node as complex system," *Nature communications*, vol. 2, no. 1, pp. 1–6, 2011.

[142] M. L. Alomar, M. C. Soriano, M. Escalona-Morán, *et al.*, "Digital implementation of a single dynamical node reservoir computer," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 10, pp. 977–981, 2015.

[143] B. Penkovsky, L. Larger, and D. Brunner, "Efficient design of hardware-enabled reservoir computing in fpgas," *Journal of Applied Physics*, vol. 124, no. 16, p. 162 101, 2018.

[144] E. S. Skibinsky-Gitlin, M. L. Alomar, E. Isern, M. Roca, V. Canals, and J. L. Rossello, "Reservoir computing hardware for time series forecasting," in *2018 28th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, IEEE, 2018, pp. 133–139.

[145] E. S. Skibinsky-Gitlin, M. L. Alomar, V. Canals, *et al.*, "Fpga-based echo-state networks," in *International Conference on Time Series and Forecasting*, Springer, 2018, pp. 135–146.

[146] A.-L. Barabási, "Network science," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 371, no. 1987, p. 20 120 375, 2013.

[147] A. F. Atiya and A. G. Parlos, "New results on recurrent network training: Unifying the algorithms and accelerating convergence," *IEEE transactions on neural networks*, vol. 11, no. 3, pp. 697–709, 2000.

[148] H. Jaeger, M. Lukoševičius, D. Popovici, and U. Siewert, "Optimization and applications of echo state networks with leaky-integrator neurons," *Neural networks*, vol. 20, no. 3, pp. 335–352, 2007.

[149] H. Jaeger and H. Haas, "Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication," *science*, vol. 304, no. 5667, pp. 78–80, 2004.

[150] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave gaussian quantization," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 5918–5926.

[151] J. S. Walther, "The story of unified cordic," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 25, no. 2, pp. 107–112, 2000.

[152] M. C. Mackey and L. Glass, "Oscillation and chaos in physiological control systems," *Science*, vol. 197, no. 4300, pp. 287–289, 1977.

[153] S Ortín, M. C. Soriano, L Pesquera, *et al.*, "A unified framework for reservoir computing and extreme learning machines based on a single time-delayed neuron," *Scientific reports*, vol. 5, no. 1, pp. 1–11, 2015.

[154] S. S. Stevens and J. Volkmann, "The relation of pitch to frequency: A revised scale," *The American Journal of Psychology*, vol. 53, no. 3, pp. 329–353, 1940.

[155] W. Han, C.-F. Chan, C.-S. Choy, and K.-P. Pun, "An efficient mfcc extraction method in speech recognition," in *2006 IEEE international symposium on circuits and systems*, IEEE, 2006, 4–pp.

[156] J. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, and Y. Bengio, "Attention-based models for speech recognition," *arXiv preprint arXiv:1506.07503*, 2015.

[157] C.-C. Kao, W. Wang, M. Sun, and C. Wang, "R-crnn: Region-based convolutional recurrent neural network for audio event detection," *arXiv preprint arXiv:1808.06627*, 2018.

[158] J. Salamon and J. P. Bello, "Deep convolutional neural networks and data augmentation for environmental sound classification," *IEEE Signal Processing Letters*, vol. 24, no. 3, pp. 279–283, 2017.

[159] O. Kücüktopcu, E. Masazade, C. Ünsalan, and P. K. Varshney, "A real-time bird sound recognition system using a low-cost microcontroller," *Applied Acoustics*, vol. 148, pp. 194–201, 2019.

[160] B. Liu, Z. Wang, W. Zhu, *et al.*, "An ultra-low power always-on keyword spotting accelerator using quantized convolutional neural network and voltage-domain analog switching network-based approximate computing," *IEEE Access*, vol. 7, pp. 186 456–186 469, 2019.

[161] R. Saleh, S. Wilton, S. Mirabbasi, *et al.*, "System-on-chip: Reuse and integration," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1050–1069, 2006.

[162] M. A. Escalona-Morán, M. C. Soriano, I. Fischer, and C. R. Mirasso, "Electrocardiogram classification using reservoir computing with logistic regression," *IEEE Journal of Biomedical and health Informatics*, vol. 19, no. 3, pp. 892–898, 2014.

[163] J. C. Patra and R. N. Pal, "A functional link artificial neural network for adaptive channel equalization," *Signal Processing*, vol. 43, no. 2, pp. 181–195, 1995.

[164] N. Schaetti, M. Salomon, and R. Couturier, "Echo state networks-based reservoir computing for mnist handwritten digits recognition," in *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*, IEEE, 2016, pp. 484–491.

[165] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[166] C. R. Dyer and A. Rosenfeld, "Parallel image processing by memory-augmented cellular automata," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, no. 1, pp. 29–41, 1981.

[167] R. M. Haralick, S. R. Sternberg, and X. Zhuang, "Image analysis using mathematical morphology," *IEEE transactions on pattern analysis and machine intelligence*, no. 4, pp. 532–550, 1987.

[168] P. G. Tzionas, P. G. Tsalides, and A. Thanailakis, "A new, cellular automaton-based, nearest neighbor pattern classifier and its vlsi implementation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 3, pp. 343–353, 1994.

[169] W. Ge, R. T. Collins, and R. B. Ruback, "Vision-based analysis of small groups in pedestrian crowds," *IEEE transactions on pattern analysis and machine intelligence*, vol. 34, no. 5, pp. 1003–1016, 2012.

[170] Y. Qin, H. Lu, Y. Xu, and H. Wang, "Saliency detection via cellular automata," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 110–119.

[171] M. Chady and R. Poli, "Evolution of cellular-automaton-based associative memories," *COGNITIVE SCIENCE RESEARCH PAPERS-UNIVERSITY OF BIRMINGHAM CSRP*, 1997.

[172] N. Ganguly, P. Maji, S. Dhar, B. K. Sikdar, and P. P. Chaudhuri, "Evolving cellular automata as pattern classifier," in *International Conference on Cellular Automata*, Springer, 2002, pp. 56–68.

[173] P. Maji, C. Shaw, N. Ganguly, B. K. Sikdar, and P. P. Chaudhuri, "Theory and application of cellular automata for pattern classification," *Fundamenta Informaticae*, vol. 58, no. 3-4, pp. 321–354, 2003.

[174] O. Yilmaz, "Reservoir computing using cellular automata," *arXiv preprint arXiv:1410.0162*, 2014.

[175] ——, "Symbolic computation using cellular automata-based hyperdimensional computing," *Neural computation*, vol. 27, no. 12, pp. 2661–2692, 2015.

[176] ——, "Machine learning using cellular automata based feature expansion and reservoir computing.," *Journal of Cellular Automata*, vol. 10, 2015.

[177] S. Nichele and M. S. Gundersen, "Reservoir computing using non-uniform binary cellular automata," *arXiv preprint arXiv:1702.03812*, 2017.

[178] S. Nichele and A. Molund, "Deep learning with cellular automaton-based reservoir computing," 2017.

[179] N. McDonald, "Reservoir computing & extreme learning machines using pairs of cellular automata rules," in *2017 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2017, pp. 2429–2436.

[180] C. Oliveira and P. P. B. de Oliveira, "An approach to searching for two-dimensional cellular automata for recognition of handwritten digits," in *Mexican International Conference on Artificial Intelligence*, Springer, 2008, pp. 462–471.

[181] M. Halbach and R. Hoffmann, "Implementing cellular automata in fpga logic," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, IEEE, 2004, p. 258.

[182] Á. L. García-Arias, J. Yu, and M. Hashimoto, "Low-cost reservoir computing using cellular automata and random forests," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2020, pp. 1–5.

[183] E. Randazzo, A. Mordvintsev, E. Niklasson, M. Levin, and S. Greydanus, "Self-classifying mnist digits," *Distill*, vol. 5, no. 8, e00027–002, 2020.

[184] J. Von Neumann *et al.*, "The general and logical theory of automata," *1951*, pp. 1–41, 1951.

[185] S. Wolfram, *A new kind of science.* Wolfram media Champaign, IL, 2002, vol. 5.

[186] ——, "Theory and applications of cellular automata," *World Scientific*, 1986.

[187] W. Li and N. Packard, "The structure of the elementary cellular automata rule space," *Complex systems*, vol. 4, no. 3, pp. 281–297, 1990.

[188] S. Ninagawa, "Power spectral analysis of elementary cellular automata," *Complex Systems*, vol. 17, no. 4, p. 399, 2008.

[189] J. J. Weng, N. Ahuja, and T. S. Huang, *Learning recognition and segmentation of 3-d objects from 2-d images. in 1993 (4th) international conference on computer vision*, 1993.

[190] P. Y. Simard, D. Steinkraus, J. C. Platt, *et al.*, "Best practices for convolutional neural networks applied to visual document analysis.," in *Icdar*, Citeseer, vol. 3, 2003.

[191] J.-H. Lin, T. Xing, R. Zhao, *et al.*, "Binarized convolutional neural networks with separable filters for efficient hardware acceleration," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017, pp. 27–35.

[192] S. Ghaffari and S. Sharifian, "Fpga-based convolutional neural network accelerator design using high level synthesize," in *2016 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS)*, IEEE, 2016, pp. 1–6.

[193] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "Fp-bnn: Binarized neural network on fpga," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018.

[194] Y. Zhou and J. Jiang, "An fpga-based accelerator implementation for deep convolutional neural networks," in *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, IEEE, vol. 1, 2015, pp. 829–832.

[195] T.-H. Tsai, Y.-C. Ho, and M.-H. Sheu, "Implementation of fpga-based accelerator for deep neural networks," in *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, IEEE, 2019, pp. 1–4.

[196] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

[197] G. B. Moody and R. G. Mark, "The impact of the mit-bih arrhythmia database," *IEEE Engineering in Medicine and Biology Magazine*, vol. 20, no. 3, pp. 45–50, 2001.

[198] X. L. Zhang, H. Begleiter, B. Porjesz, W. Wang, and A. Litke, "Event related potentials during object recognition tasks," *Brain research bulletin*, vol. 38, no. 6, pp. 531–538, 1995.

[199] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*, Springer, 2016, pp. 525–542.

[200] Y. Ji, F. Ran, C. Ma, and D. J. Lilja, "A hardware implementation of a radial basis function neural network using stochastic logic," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2015, pp. 880–883.

[201] K. Wold and C. H. Tan, "Analysis and enhancement of random number generator in fpga based on oscillator rings," in *2008 International Conference on Reconfigurable Computing and FPGAs*, IEEE, 2008, pp. 385–390.

[202] L. Parrilla, E. Castillo, E Todorovich, A. García, D. P. Morales, and G. Botella, "Improvements for the applicability of power-watermarking to embedded ip cores protection: E-coreipp," *Digital Signal Processing*, vol. 44, pp. 110–122, 2015.

[203] D. B. Thomas and W. Luk, "Fpga-optimised uniform random number generators using luts and shift registers," in *2010 International Conference on Field Programmable Logic and Applications*, IEEE, 2010, pp. 77–82.

[204] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[205] *Banknote authentication dataset*, accessed on April 8, 2021. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/banknote+authentication.

[206] W. N. Street, W. H. Wolberg, and O. L. Mangasarian, "Nuclear feature extraction for breast tumor diagnosis," in *Biomedical image processing and biomedical visualization*, International Society for Optics and Photonics, vol. 1905, 1993, pp. 861–870.

[207] C Kaynak, "Methods of combining multiple classifiers and their applications to handwritten digit recognition," *Unpublished master's thesis, Bogazici University*, 1995.

[208] H. D. W. TN, "Receptive fields of single neurones in the cat's striate cortex," *Journal of Physiology*, vol. 148, p. 574 591, 1959.

[209] D. H. Hubel and T. N. Wiesel, "The period of susceptibility to the physiological effects of unilateral eye closure in kittens," *The Journal of physiology*, vol. 206, no. 2, pp. 419–436, 1970.

[210] K. Fukushima, "A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biol. Cybern.*, vol. 36, pp. 193–202, 1980.

[211] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang, "Phoneme recognition using time-delay neural networks," *IEEE transactions on acoustics, speech, and signal processing*, vol. 37, no. 3, pp. 328–339, 1989.

[212] Y. LeCun, B. Boser, J. S. Denker, *et al.*, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[213] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *2012 IEEE conference on computer vision and pattern recognition*, IEEE, 2012, pp. 3642–3649.

[214] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[215] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[216] C. F. Frasser, P. Linares-Serrano, V Canals, M. Roca, T Serrano-Gotarredona, and J. L. Rossello, "Fully-parallel convolutional neural network hardware," *arXiv preprint arXiv:2006.12439*, 2020.

[217] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv preprint arXiv:1806.08342*, 2018.

[218] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, and Z. Zhang, "Improving neural network quantization without retraining using outlier channel splitting," in *International conference on machine learning*, PMLR, 2019, pp. 7543–7552.

[219] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[220] A. Bulat and G. Tzimiropoulos, "Xnor-net++: Improved binary neural networks," *arXiv preprint arXiv:1909.13863*, 2019.

[221] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[222] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.

[223] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li, "Gxnor-net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework," *Neural Networks*, vol. 100, pp. 49–58, 2018.

[224] Z. Li, J. Li, A. Ren, *et al.*, "Heif: Highly efficient stochastic computing-based inference framework for deep neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 8, pp. 1543–1556, 2018.

[225] H. Sim and J. Lee, "Cost-effective stochastic mac circuits for deep neural networks," *Neural Networks*, vol. 117, pp. 152–162, 2019.

[226] R. M. Gray, "Toeplitz and circulant matrices: A review," 2006.

[227] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *arXiv preprint arXiv:2103.13630*, 2021.

[228] S. R. Faraji, M. H. Najafi, B. Li, D. J. Lilja, and K. Bazargan, "Energy-efficient convolutional neural networks with deterministic bit-stream processing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2019, pp. 1757–1762.

[229] Z. Liu, Y. Dou, J. Jiang, *et al.*, "Throughput-optimized fpga accelerator for deep convolutional neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 10, no. 3, pp. 1–23, 2017.

[230] M. L. A. Barceló, "Methodologies for hardware implementation of reservoir computing systems," Ph.D. dissertation, Universitat de les Illes Balears, 2017.

[231] A. O. Gelabert, "Desarrollo y aceleración hardware de metodologías de descripción y comparación de compuestos orgánicos," Ph.D. dissertation, Universitat de les Illes Balears, 2018.

[232] A. Morán, V. Canals, L. Parrilla, C. F. Frasser, M. Roca, and J. L. Rosselló, "Inference based on stochastic computing radial basis functions," *IEEE transactions on neural networks*, 2021, under review.

[233] C. F. Frasser, P. Linares-Serrano, A. Morán, *et al.*, "Fully-parallel stochastic computing design of convolutional neural networks for edge computing applications," *IEEE transactions on neural networks*, 2021, under review.

[234] F. Galán-Prado, A. Morán, J. Font, M. Roca, and J. L. Rosselló, "Compact hardware synthesis of stochastic spiking neural networks," *International journal of neural systems*, vol. 29, no. 08, p. 1 950 004, 2019.

[235] A. Morán and M. C. Soriano, "Improving the quality of a collective signal in a consumer eeg headset," *Plos one*, vol. 13, no. 5, e0197597, 2018.

[236] A. Morán, V. Canals, M. Roca, E. Isern, and J. L. Rosselló, “Fpga implementation of random vector functional link networks based on elementary cellular automata,” in *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*, IEEE, 2020, pp. 1–6.

[237] C. F. Frasser, P. Linares-Serrano, A. Morán, *et al.*, “Exploiting correlation in stochastic computing based deep neural networks,” in *2021 XXXVI Conference on Design of Circuits and Integrated Systems (DCIS)*, IEEE, 2021, pp. 1–6, accepted for oral presentation.

[238] F. Galán-Prado, A. Morán, J. Font, M. Roca, and J. L. Rosselló, “Stochastic radial basis neural networks,” in *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, IEEE, 2019, pp. 145–149.

[239] A. Morán, J. L. Rosselló, M. Roca, E. Isern, V. Martínez-Moll, and V. Canals, “Self-organizing maps hybrid implementation based on stochastic computing,” in *2019 XXXIV Conference on Design of Circuits and Integrated Systems (DCIS)*, IEEE, 2019, pp. 1–6.

[240] A. Morán, J. L. Rosselló, M. Roca, and V. Canals, “Soc kohonen maps based on stochastic computing,” in *2020 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2020, pp. 1–7.

[241] X. Gu, P. P. Angelov, and J. C. Príncipe, “A method for autonomous data partitioning,” *Information Sciences*, vol. 460, pp. 65–82, 2018.

[242] A. Morán, V. Canals, M. Roca, E. Isern, P. Angelov, and J. L. Rosselló, “Stochastic computing co-processing elements for evolving autonomous data partitioning,” in *2021 XXXVI Conference on Design of Circuits and Integrated Systems (DCIS)*, IEEE, 2021, pp. 1–6, accepted for oral presentation.

[243] Y. E. Nesterov, “A method for solving the convex programming problem with convergence rate o (1/kˆ 2),” in *Dokl. akad. nauk Sssr*, vol. 269, 1983, pp. 543–547.

[244] M. George and P. Alfke, “Linear feedback shift registers in virtex devices,” *Xilinx apprication note XAPP210*, 2007.