Universitat
de les Illes Balears

DOCTORAL THESIS
2017

METHODOLOGIES FOR HARDWARE IMPLEMENTATION OF RESERVOIR COMPUTING SYSTEMS

Miquel Lleó Alomar Barceló

Physics Department
Electronics Engineering Group

Universitat
de les Illes Balears

DOCTORAL THESIS
2017

Doctoral Programme of Physics

METHODOLOGIES FOR HARDWARE IMPLEMENTATION OF RESERVOIR COMPUTING SYSTEMS

Miquel Lleó Alomar Barceló

Thesis Supervisor: Josep Lluís Rosselló Sanz

Thesis Co-Supervisor: Vicente José Canals Guinand

Doctor by the Universitat de les Illes Balears

To Arian and Arsham

Contents

Abstract

Inspired by the way the brain processes information, artificial neural networks (ANNs) were created with the aim of reproducing human capabilities in tasks that are hard to solve using the classical algorithmic programming. The ANN paradigm has been applied to numerous fields of science and engineering thanks to its ability to learn from examples, adaptation, parallelism and fault-tolerance. Reservoir computing (RC), based on the use of a random recurrent neural network (RNN) as processing core, is a powerful model that is highly suited to time-series processing.

Hardware realizations of ANNs are crucial to exploit the parallel properties of these models, which favor higher speed and reliability. On the other hand, hardware neural networks (HNNs) may offer appreciable advantages in terms of power consumption and cost. Low-cost compact devices implementing HNNs are useful to support or replace software in real-time applications, such as control, medical monitoring, robotics and sensor networks. However, the hardware realization of ANNs with large neuron counts, such as in RC, is a challenging task due to the large resource requirement of the involved operations. Despite the potential benefits of hardware digital circuits to perform RC-based neural processing, most implementations are realized in software using sequential processors.

In this thesis, I propose and analyze several methodologies for the digital implementation of RC systems using limited hardware resources. The neural network design is described in detail for both a conventional implementation and the diverse alternative approaches. The advantages and shortcomings of the various techniques regarding the accuracy, computation speed and required silicon area are discussed. Finally, the proposed approaches are applied to solve different real-life engineering problems.

Resum

Inspirades en la forma en què el cervell processa la informació, les xarxes neuronals artificials (XNA) es crearen amb l'objectiu de reproduir habilitats humanes en tasques que són difícils de resoldre mitjançant la programació algorítmica clàssica. El paradigma de les XNA s'ha aplicat a nombrosos camps de la ciència i enginyeria gràcies a la seva capacitat d'aprendre dels exemples, l'adaptació, el paral · lelisme i la tolerància a fallades. El *reservoir computing* (RC), basat en l'ús d'una xarxa neuronal recurrent (XNR) aleatòria com a nucli de processament, és un model de gran abast molt adequat per processar sèries temporals.

Les realitzacions en maquinari de les XNA són crucials per aprofitar les propietats paral · leles d'aquests models, les quals afavoreixen una major velocitat i fiabilitat. D'altra banda, les xarxes neuronals en maquinari (XNM) poden oferir avantatges apreciables en termes de consum energètic i cost. Els dispositius compactes de baix cost implementant XNM són útils per donar suport o reemplaçar el programari en aplicacions en temps real, com ara de control, supervisió mèdica, robòtica i xarxes de sensors. No obstant això, la realització en maquinari de XNA amb un nombre elevat de neurones, com al cas de l'RC, és una tasca difícil a causa de la gran quantitat de recursos exigits per les operacions involucrades. Tot i els possibles beneficis dels circuits digitals en maquinari per realitzar un processament neuronal basat en RC, la majoria d'implementacions es realitzen en programari usant processadors convencionals.

En aquesta tesi, proposo i analitzo diverses metodologies per a la implementació digital de sistemes RC fent ús d'un nombre limitat de recursos de maquinari. Els dissenys de la xarxa neuronal es descriuen en detall tant per a una implementació convencional com per als distints mètodes alternatius. Es discuteixen els avantatges i inconvenients de les diferents tècniques pel que fa a l'exactitud, velocitat de càlcul i àrea requerida. Finalment, les implementacions proposades s'apliquen a resoldre diferents problemes pràctics d'enginyeria.

Resumen

Inspiradas en la forma en que el cerebro procesa la información, las redes neuronales artificiales (RNA) se crearon con el objetivo de reproducir habilidades humanas en tareas que son difíciles de resolver utilizando la programación algorítmica clásica. El paradigma de las RNA se ha aplicado a numerosos campos de la ciencia y la ingeniería gracias a su capacidad de aprender de ejemplos, la adaptación, el paralelismo y la tolerancia a fallas. El *reservoir computing* (RC), basado en el uso de una red neuronal recurrente (RNR) aleatoria como núcleo de procesamiento, es un modelo de gran alcance muy adecuado para procesar series temporales.

Las realizaciones en hardware de las RNA son cruciales para aprovechar las propiedades paralelas de estos modelos, las cuales favorecen una mayor velocidad y fiabilidad. Por otro lado, las redes neuronales en hardware (RNH) pueden ofrecer ventajas apreciables en términos de consumo energético y coste. Los dispositivos compactos de bajo coste implementando RNH son útiles para apoyar o reemplazar al software en aplicaciones en tiempo real, como el control, monitorización médica, robótica y redes de sensores. Sin embargo, la realización en hardware de RNA con un número elevado de neuronas, como en el caso del RC, es una tarea difícil debido a la gran cantidad de recursos exigidos por las operaciones involucradas. A pesar de los posibles beneficios de los circuitos digitales en hardware para realizar un procesamiento neuronal basado en RC, la mayoría de las implementaciones se realizan en software mediante procesadores convencionales.

En esta tesis, propongo y analizo varias metodologías para la implementación digital de sistemas RC utilizando un número limitado de recursos hardware. Los diseños de la red neuronal se describen en detalle tanto para una implementación convencional como para los distintos métodos alternativos. Se discuten las ventajas e inconvenientes de las diversas técnicas con respecto a la precisión, velocidad de cálculo y área requerida. Finalmente, las implementaciones propuestas se aplican a resolver diferentes problemas prácticos de ingeniería.

List of publications

Journal articles

- M. L. Alomar, M. C. Soriano, M. Escalona-Moran, V. Canals, I. Fischer, C. R. Mirasso, and J. L. Rossello. Digital implementation of a single dynamical node reservoir computer. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(10): 977–981, Oct 2015.
- M. L. Alomar, V. Canals, N. Perez-Mora, V. Martinez-Moll, and J. L. Rossello. FPGA-based stochastic echo state networks for time-series forecasting. *Computational Intelligence and Neuroscience*, 2016.
- J. L. Rossello, M. L. Alomar, A. Morro, A. Oliver, and V. Canals. High-density liquid-state machine circuitry for time-series forecasting. *International Journal of Neural Systems*, 26(5), Feb 2016.
- M. L. Alomar, V. Canals, E. Isern, M. Roca, and J. L. Rossello. Efficient parallel implementation of reservoir computing systems. *Neural Computing and Applications*, 2017. Submitted on Jun. 2017 (under review).

These articles represent the basis of chapter 8, chapter 4, chapter 5, and chapter 6, respectively. Throughout my PhD, I have also contributed to the development and writing of some journal publications that have not been directly included in my thesis. These are listed below:

- A. Morro, V. Canals, A. Oliver, M. L. Alomar, and J. L. Rossello. Ultra-fast data-mining hardware architecture based on stochastic computing. *PLoS ONE*, 10(5), May 2015.
- V. Canals, A. Morro, A. Oliver, M. L. Alomar, and J. L. Rossello. A new stochastic computing methodology for efficient neural network implementation. *IEEE Transactions on Neural Networks and Learning Systems*, 27(3): 551–564, Mar 2016.
- A. Morro, V. Canals, A. Oliver, M. L. Alomar, F. Galan-Prado, P. J. Ballester, and J. L. Rossello. A stochastic spiking neural network for virtual screening. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–5, 2017.
- V. Canals, C. F. Frasser, M. L. Alomar, A. Morro, A. Oliver, M. Roca, E. Isern, V. Martinez-Moll, E. Garcia-Moreno, and J. L. Rossello. Noise tolerant

probabilistic logic for statistical pattern recognition applications. *Integrated Computer-Aided Engineering,* 2017. Submitted on Feb. 2016 (accepted with minor changes).

Conference papers

- M. L. Alomar, V. Canals, V. Martinez-Moll, and J. L. Rossello. Low-cost hardware implementation of reservoir computers. In *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2014 24th International Workshop on*, pages 1–5, Sep 2014.
- M. L. Alomar, V. Canals, V. Martinez-Moll, and J. L. Rossello. *Advances in Computational Intelligence: 13th International WorkConference on Artificial Neural Networks, IWANN 2015. Proceedings, Part II*, chapter Stochastic-Based Implementation of Reservoir Computers, pages 185–196. Springer International Publishing, Cham, Jun 2015.
- M. L. Alomar, V. Canals, A. Morro, A. Oliver, and J. L. Rossello. Stochastic hardware implementation of liquid state machines. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 1128–1133, Jul 2016.
- M. L. Alomar, V. Canals, E. Isern, M. Roca, and J. L. Rossello. Fault-tolerant Echo State Networks for temporal signal classification. In *Design of Circuits and Integrated Circuits (DCIS), 2017 Conference on*, pages 1–5, Nov 2017. Submitted on May 2017 (under review).

These papers cover the topic of the works described in chapter 4, chapter 5 and chapter 9. Other conference publications where I have contributed but which have not been directly included in my thesis are:

- J. L. Rossello, V. Canals, A. Oliver, M. Alomar, and A. Morro. Spiking neural networks signal processing. In *Design of Circuits and Integrated Circuits (DCIS), 2014 Conference on*, pages 1–6, Nov 2014.
- V. Canals, M. L. Alomar, A. Morro, A. Oliver, and J. L. Rossello. Noise-robust hardware implementation of neural networks. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–8, Jul 2015.
- V. Canals, A. Morro, A. Oliver, M. L. Alomar, and J. L. Rossello. An unconventional computing technique for ultra-fast and ultra-low power data mining. In *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2015 25th International Workshop on*, pages 40–46, Sep 2015.
- M. L. Alomar, V. Canals, J. L. Rossello, and V. Martinez-Moll. Estimation of global solar radiation from air temperature using artificial neural networks based on reservoir computing. In *11th ISES EuroSun Conference, International Conference on Solar Energy for Buildings and Industry*, pages 1–6, Oct 2016.

- V. Canals, A. Oliver, M. L. Alomar, N. Perez-Mora, A. Moia, V. Martinez-Moll, M. Roca, E. Isern, E. Garcia-Moreno, and J. L. Rossello. Mid-term photovoltaic plant power generation forecast model using a time delayed artificial neural networks: application for a grid-connected PV plant at Balearic Islands, Spain. In *11th ISES EuroSun Conference, International Conference on Solar Energy for Buildings and Industry*, Oct 2016.
- V. Canals, A. Oliver, M. L. Alomar, M. Roca, E. Isern, E. Garcia-Moreno, A. Morro, F. Galan, J. Font-Rossello, and J. L. Rossello. Robust stochastic logic for pattern recognition. In *Design of Circuits and Integrated Circuits (DCIS), 2016 Conference on*, pages 1–6, Nov 2016.

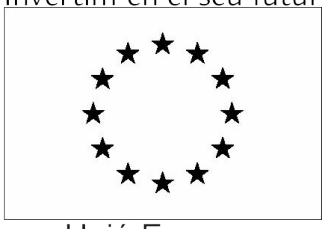
Patents

- V. Canals, A. Morro, J. L. Rosselló, M. L. Alomar, A. Oliver. "Sistemas digitales probabilísticos inmunes al ruido electromagnético". Patent P201331656, May 2015.

Acknowledgments

This work has been supported by a fellowship (FPI/1513/2012) financed by the European Social Fund (ESF) and the "Govern de les Illes Balears (Conselleria d'Educació, Cultura i Universitats)", and by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant contracts TEC2011-23113 and TEC2014-56244-R.

Invertim en el seu futur



Unió Europea
Fons Social Europeu

Part I.

Introduction and objectives

1. Motivation and objectives

1.1. Motivation

Numerous real-world applications require the execution of tasks that humans can accomplish with apparent ease, but are difficult to perform through standard programmed algorithms. Emblematic examples are speech processing and image recognition. Today's society demands the development of smart methodologies for extracting useful information from the increasingly large amounts of data it generates. The field of "machine learning" investigates systems that are not limited to repeating an explicit set of instructions defined by a programmer, but present the ability of learning by generalization from data. Artificial neural networks (ANNs) are one of such machine learning techniques. They mimic some general features of the brain on the hope to reproduce its capabilities.

ANNs make it possible to automate tasks that are complex to be programmed with sequential computers and represent a useful engineering tool for a variety of technical applications, such as pattern recognition and function approximation. In particular, recurrent neural networks (RNNs), a type of ANN characterized by the presence of closed loops, present the capacity of extracting temporal information from sequential data, which is crucial to perform tasks such as speech recognition or time-series forecasting.

Nevertheless, despite the potential capacities of RNNs for solving complex temporal machine learning tasks, the application of this approach to real-world problems is limited by its complex training procedure, which is relatively time consuming and requires substantial skill to be successfully applied. Reservoir computing (RC) offers a practical alternative to such hard training. With an strategic design of the network, RC reduces the complex training of recurrent networks to a simple linear regression problem facilitating their practical application. The plausibility of RC systems and, in general, of ANNs for modeling biological networks and simulating computation processes in the brain is also an important reason that motivates their development and study. In this thesis, however, I am mainly interested in RC networks as a tool for solving engineering problems. RC has been shown to be a successful approach to perform time-series prediction and classification tasks in many different application fields, such as digital signal processing, robotics, computer vision, medicine and finances. The major advantages of RC over "traditional" RNNs are fast learning speed, ease of implementation and minimal human intervention, which, all in all, allow for a more widespread use of RNNs.

Machine learning techniques such as ANNs and RC systems are usually implemented in software by means of general-purpose sequential processors. However, this is unsatisfactory for certain applications, which require specific hardware to realize such algorithms. More specifically, the development of specialized hardware is crucial to achieve fast, efficient and reliable neural networks. Contrary to software solutions, hardware-based neural networks (HNNs) can exploit the inherent parallelism of ANNs offering a significant speedup in applications that demand a high-volume and real-time processing, such as computer vision tasks and data mining. On the other hand, HNNs allow to reduce the power consumption, which is interesting for applications requiring autonomy and/or mobility, such as robot control, wireless sensor networks and wearable medical devices. In addition, specialized hardware implementing neural networks may offer advantages in terms of cost and reliability (fault-tolerance), which is important for massively produced electronics devices and safety-critical applications, respectively.

RC has received a lot of attention from the research community since its appearance in the last decade. Nevertheless, the development of methods for its hardware implementation still deserves to be extensively explored. Despite the potential benefits of hardware digital circuits to perform RC-based neural processing (e.g., speed gains and power savings), very few implementations of this type have been proposed. It must be noted that the efficient hardware implementation of RC networks using compact devices, such as low-cost field-programmable gate arrays (FPGAs), is not straightforward due to the usually large number of nodes employed in RC networks (typically on the order of 50 to 1000) and the high chip area required by the operations involved with each processing node. Consequently, unconventional techniques must be investigated to obtain smart implementations using reduced hardware resources. The development of efficient dedicated hardware realizations of RC may extend the utility of the approach to a wider range of applications, especially those demanding real-time processing capabilities and/or with constraints on power supply.

The development of specific hardware realizing machine learning algorithms such as ANN models is usually a long process that requires specialized knowledge on digital hardware design. Unlike personal computers or low-cost microcontrollers, hardware devices such as FPGAs cannot be configured using standard sequential programming languages, but require the use of hardware description languages, such as VHDL, which is normally time consuming. It must be considered that each implementation must be tailored to the specific application in order to satisfy the particular requirements and optimally employ the hardware resources. As a result, building such hardware may be expensive. The availability of a tool allowing to easily generate the VHDL code associated with a particular network design for any desired configuration parameters would facilitate the implementation of HNNs. Such a tool for the automatic generation of RC hardware could accelerate the hardware development cycle of these implementations, and even enable their use to non-experts in the field of digital hardware design.

1.2. Objectives

These are the general objectives of the present thesis:

- Develop and provide a detailed description of different methodologies for implementing RC systems in digital hardware. The proposed approaches include a binary logic conventional implementation, stochastic computing, a multiplier-less design, and delay-based schemes.
- Implement the proposed designs using programmable hardware devices (FPGAs) and evaluate their performance for benchmark tasks. Analyze the advantages and shortcomings of each implementation scheme and assess their adequacy for particular applications.
- Develop software programs capable of emulating the different hardware implementations, which is useful to adequately train the systems and also to evaluate them prior to hardware implementation.
- Provide structural hardware description (VHDL) codes of the presented realizations, which may help potential users to develop designs adapted to their particular needs. Further facilitate and accelerate the implementation process of FPGA-based RC systems by developing software programs to automatically generate the VHDL codes of the different proposed designs.
- Demonstrate the usefulness of the developed implementations applying them to solve several real-life engineering problems. Describe additional potential application areas for the proposed hardware-based RC systems.

1.3. Structure

The thesis is organized as follows:

Chapter 2 presents an overview of the research field of artificial neural networks, in particular of the reservoir computing technique. The motivation and principles of neural network operation are introduced along with different types of network architectures and applications. Three fundamental versions of reservoir computing (echo state networks, liquid state machines and single dynamical node RC) are described in detail. Special attention is given to the motivations behind the hardware implementation of artificial "intelligent" systems such as those based on the reservoir computing methodology, which is the subject matter of the following chapters.

Chapter 3 presents a "conventional" digital hardware design for a fully-parallel echo state network. Such implementation serves as benchmark for the subsequent alternative designs and also to introduce several issues related to the implementation of neural networks in digital circuitry.

Chapter 4 proposes the use of stochastic computing, a low-cost alternative to conventional binary computing that reduces the circuitry devoted to the arithmetic

operations in echo state networks. The principles and various applications of the stochastic computing approach are presented and applied to build echo state networks. The resulting "stochastic ESN" implementation is benchmarked against the conventional design of previous chapter evaluating the advantages and shortcomings. The design and some results presented in this chapter have been published in the *Computational Intelligence and Neuroscience* journal ([ACPM+16]). Preliminar results were also presented in the conference papers [ACMMR14] and [ACMMR15].

Chapter 5 also provides an implementation scheme for reservoir networks based on stochastic computing. However, it offers a different view on stochastic ESNs, which are interpreted as an approximation to liquid state machines given the capability of "stochastic neurons" (i.e., sigmoid-like neuron models implemented with the stochastic computing approach) to simplistically emulate the spiking behavior. This chapter is based on the article [RAM+16], published in the *International Journal of Neural Systems*, and on the conference paper [ACM+16].

Chapter 6 presents a compact digital hardware design for echo state networks based on constraining the resolution of the synaptic weight values, which allows replacing multipliers by simpler operations consuming minimal resource. The proposed implementation is termed "hardware ESN without multipliers". This approach has been presented in the article [ACI+17a], submitted to the *Neural Computing and Applications* journal.

Chapter 7 introduces a design based on the idea of sequentializing the operation of the echo state network so that a single neuron evaluated at different temporal positions emulates the whole network. That is, the hardware requirements are relaxed at the cost of a longer computation time. The approach is referred to as a "delay-based ESN" since the output of the single implemented neuron at a given time is given in terms of its value at a previous time step.

Chapter 8 presents a digital realization of a single dynamical node reservoir computer, which emulates the nodes of a recurrent network by sampling the solutions of a delay differential equation. As in the delay-based ESN, a serial computation scheme is followed so that the output of each "virtual" node of the network is computed as a function of its delayed values. However, in this case, a differential equation is solved to determine the output of each node instead of a simpler discrete-time recursive formula. The information presented in this chapter is partly based on the article [ASEM+15], published in the journal *IEEE Transactions on Circuits and Systems II*.

In chapter 9, the value of the proposed reservoir computing implementations is highlighted applying a selected design to perform different tasks of practical relevance, such as handwriting recognition and the equalization of a communication channel. Furthermore, numerous potential application areas of the systems developed throughout the thesis are described. Some of this chapter's results have been presented in the conference paper [ACI+17b].

Finally, chapter 10 compares the proposed designs in terms of area requirements,

accuracy and computation speed. This allows to assess the suitability of the different approaches for certain application fields. The conclusions of the thesis are drawn and some lines for future research are proposed.

2. Introduction

2.1. Overview

This chapter introduces the research field of artificial neural networks (ANNs) and, in particular, the reservoir computing (RC) technique. Among the recent ANN approaches, RC stands out for its capability to deal with temporal signals while presenting a fast and easy training procedure. Different reasons motivate the study of RC systems and, in general, of ANNs, such as their biological plausibility or their analogy with complex dynamical systems. My main interest in RC networks, however, is their practical utility as a tool for solving engineering problems, such as modeling nonlinear dynamical systems, performing time-series forecasting, temporal pattern recognition or nonlinear regression. More specifically, this thesis is aimed at presenting methodologies for the efficient digital hardware implementation of RC systems, which is crucial for their widespread use in real-time and real-world applications.

Next section presents the general concept of ANNs, the different types of neuron models, network architectures, and their main applications. Then, sec. 2.3 describes the RC approach with its different versions and the applications it is particularly well suited for. Finally, sec. 2.4 describes the motivations behind the realization of ANNs in hardware and presents a brief survey of previous works on this topic.

2.2. Artificial neural networks

Artificial neural networks (ANNs) are a computational approach inspired (to a greater or lesser extend) by the way our brain processes information. From the engineering point of view, the reason that motivates the development of ANNs is the necessity to create more "intelligent" systems. Even though there are many tasks that are ideally suited to be solved by conventional sequential computers (e.g., to perform precise and fast arithmetic operations), there are some problems that are unsolvable or difficult to solve in a deterministic way through the use of sequential algorithms.

Let us consider, for example, a problem of visual pattern recognition such as the recognition of handwritten alphanumeric characters or of a particular object in a picture. It is difficult to program a sequential computer system to perform this

type of inherently parallel tasks. A number of sophisticated techniques must be used, perhaps even failing to find an acceptable solution. On the other hand, people and animals are fast and efficient recognizing images. This is more surprising if we take into account that modern electronic devices (with a switching frequency on the order of a few *GHz*) are more than a million times faster than our neurons (with a response time typically on the order of 1-10 *ms*). The reason for this is partly answered by the fact that the brain's architecture is significantly different from that of a conventional computer. The massive parallelism and inter-connectivity observed in the biological systems is the key of the many different processing tasks that can be efficiently performed by the brain.

The socially demanded applications that we expect our computers to perform (e.g., speech processing or image recognition) can be easily accomplished by humans, but are too complex to be executed using the classical deterministic algorithms. Given the increasing amount of available information and its importance in today's society, it is necessary to look for alternative methods capable to automatically process such amounts of complex information. The traditional field of computer science deals with the automation of information processing while the areas of computational intelligence and machine learning (ML) are focused on creating methods for solving complex tasks that require some kind of intelligence. That is, systems that can learn from experience instead of repeating an explicit set of instructions so that they are able to give non-trivial responses when they are faced with stimuli they have not seen before.

ANNs are one of the different ML techniques, which present the capability of learning by generalization from training examples (instead of being explicitly programmed). Among other ML algorithms, we can mention support vector machines (SVMs), decision trees, hidden Markov models (HMMs), genetic algorithms, deep learning or cluster analysis ([Mit97], [Kec01]). ANNs borrow features from the brain's behavior to reproduce its flexibility and power by artificial means. A number of different names are used in the literature to refer to the ANN approach ([RM86], [FS91], [Zur92], [Bis95], [MMR97], [Hay01], [Dre05], [Gal07], [MOM12]), such as artificial neural systems (ANSs), neurocomputing, network computation, connectionism, parallel distributed processing, layered adaptive systems, or neuromorphic systems.

A neural-network structure is a collection of parallel processors connected together in the form of a directed graph. Fig. 2.1 schematically represents a typical network diagram, where the basic processing elements are called *artificial neurons*, or simply *neurons*. Often we also refer to them as *nodes*. In some cases, they can be considered as threshold units that fire when their input stimulus exceeds a certain value. The connections are indicated by arcs and the information flow is indicated through the use of arrowheads. Neurons are often organized in *layers*, and feedback connections both within the layer and toward adjacent layers are, in principle, allowed. Each connection strength is expressed by a numerical value called a *weight*, which is modified during the training process.

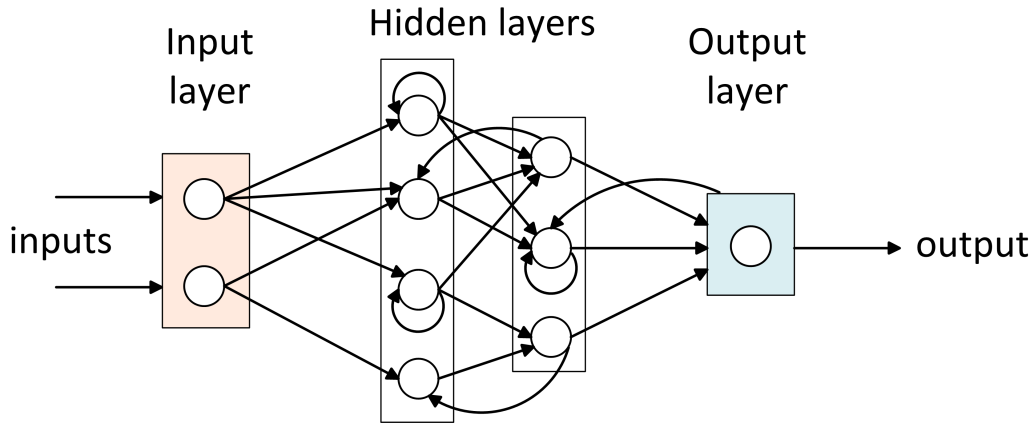


Figure 2.1.: Schematic representation of an artificial neural network (ANN).

The input and output layer contain the problem's input and desired output variables. For example, in a handwritten character recognition task the inputs may be the pixel intensity values of the image or the trajectories of a pen tip along different directions while the output units represent each one of the characters to be identified (the network must activate the output that most resembles the given input pattern).

In contrast to conventional computers, which are programmed to perform specific tasks, ANNs must be taught, or trained. They have the capability to learn the desired functional dependencies from example data pairs (supervised learning). In this case, the process of training the network corresponds to changing the connection weights systematically to reproduce the desired input-output relationships. Once the network is trained, it is able to reproduce the desired outputs even when stimuli it has never seen before are processed. Moreover, there are different techniques that enable the training of the ANN in an unsupervised way so that the network is able to differentiate stimuli without the presence of a "teacher" indicating which kind of stimulus is sequentially charged in the inputs during the training phase. For this kind of learning, the network is able to produce the same response every time the stimulus is repeating a previously observed pattern. This ability to generalize examples in an "intelligent" way to new inputs is difficult to be reproduced by traditional algorithmic computers.

The field of neural networks is broad and interdisciplinary. The present thesis focuses on their use as learning machines to solve complex engineering problems, but there are more reasons that motivate research in ANNs. From the perspective of computational neuroscience, neural-network models are constructed and simulated to understand better how the brain works and the way it performs computation ([Mac87]). Similarly, psychologists look at artificial neural networks as possible prototype structures of human-like information processing. On the other hand, physicists and mathematicians are interested in studying the properties of neural networks as complex nonlinear dynamical adaptive systems ([Mac02]).

The beginning of neurocomputing goes back to the 1940s when McCulloch and Pitts introduced the first neural network computing model using simple binary threshold functions for the neurons ([MP43]). They noted that many arithmetic and logical operations could be implemented using such models. In 1949, Hebb wrote *The Organization of Behavior* ([Heb49]), an influential book that pointed out the fact that neural connections are strengthened each time they are used. That is, he proposed a learning rule for synaptic modification. In the late 1950s, Rosenblatt invented a device called the *perceptron*, a two-layer network (i.e., with no hidden layers) employing the McCulloch-Pitts neuron model, which was capable of learning certain classification tasks by adjusting connection weights ([Ros58]). Slightly later, Widrow developed a different type of neural-network processing element called ADALINE ([Wid60]), which was trained by a gradient descent rule to minimize the mean square error. Such powerful learning method is still nowadays widely used. In 1969, the book *Perceptrons* by Minsky and Papert ([MP69]) showed the limited capability of simple perceptrons. More especifically, it was demonstrated that the perceptron was not computationally universal as it was unable to solve the classic XOR (exclusive or) problem. This publication is often considered to have caused the decline of the field of neural networks for almost two decades. In the 1980s, the works of Hopfield ([Hop88]) and Rumelhart ([RM86], [RHW86]) with multiple-layered neural networks motivated many researchers to feel renewed interest in neural computation, after which the field exploded again. Nowadays, ANNs still represent an active field of research with relevant applications in various disciplines, such as computer vision ([CMGS10], [JWW15]) and automatic speech recognition ([VSS06], [YD15], [SSR+15]). Recent ANN approaches include long short-term memory (LSTM, [HS97]), reservoir computing (RC, [MNM02], [LJ09]) and deep neural networks (DNNs, [Sch15]) among others. The present thesis deals with the RC technique.

Even though neuroscience has inspired the development of ANSs, this thesis is not directly concerned with biological networks. Rather, my primary interest in ANN models is due to their capability of learning desired input-output relationships from representative data, which makes them a useful engineering tool for a variety of technical applications ([JM99], [Dre05]). ANSs allow us to automate tasks that are complex to be programmed with sequential computers, such as complex pattern recognition. In addition, they provide an arbitrary function approximation mechanism that "learns" from observed data ([Mit16]). That is, they can be used as black-box models to estimate or approximate functions that depend on a number of inputs without the need of deriving an explicit model equation. This is particularly useful in applications where the complexity of the data or task make unfeasible to obtain an analytical model. For example, such black-box models can be used to simulate, predict, filter, classify or control nonlinear dynamical systems, which abound in science and engineering ([JH04]).

There is no doubt that the massively parallel computational networks open a range of opportunities in the areas of artificial intelligence, computational theory, sig-

nal processing, modeling and simulation, and others. However, the development of hardware is crucial for the future of neurocomputing since the achievement of fast, efficient and reliable neural networks depends on hardware being specified for its eventual use ([MS10]). This motivates the particular interest of this thesis in the implementation of ANNs (following the RC approach) in digital hardware for building circuit-based intelligent machines.

Next subsections describe the most commonly employed neuron models and how the neural units can be arranged to form different network structures. Finally, the potential application areas of ANNs are described in more detail.

2.2.1. The spiking neuron model

Spiking neural networks represent the last generation of ANNs in an attempt to emphasize the neuro-biological aspects of artificial neural computation. Real biological neurons communicate with each other by means of sequences of electrical pulses, spikes (Fig. 2.2). This is the most characteristic feature of spiking neural models. In addition, the spiking neuron model presents the following general properties ([PK11]):

- The neuron processes the information that comes from the inputs to produce a single spiking signal at the output.
- Its inputs can be excitatory or inhibitory depending on whether they increase or decrease the probability of generating a spike.
- Its dynamics is characterized by at least one state variable. The model produces a spike when the internal variables reach a certain state.

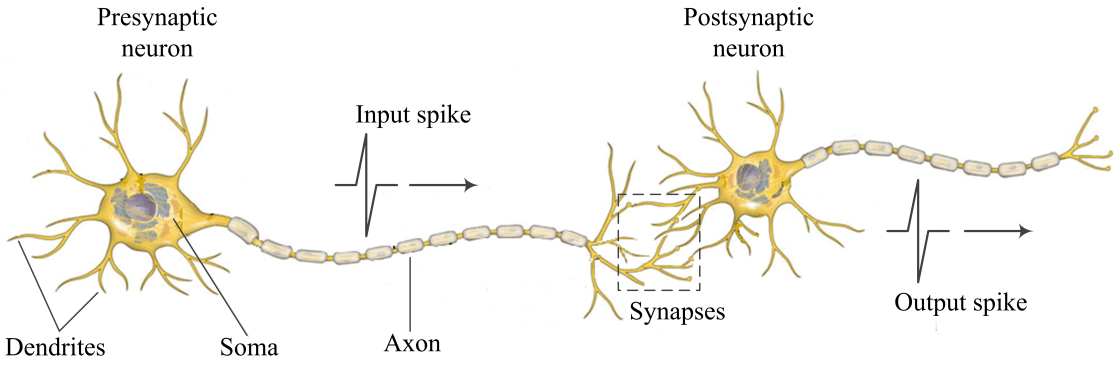


Figure 2.2.: Drawing of two connected biological neurons, which communicate through sequences of spikes. The main parts of the neuron are the dendrites, the axon and the soma (cell body). The presynaptic neuron connects with the postsynaptic one through the synapses. Image adapted from [web16c].

The basic mechanism describing how spiking neurons work is illustrated in Fig. 2.3: the neuron sends out an electrical pulse when it has received a sufficient number of

pulses from other neurons. The membrane potential of a spiking neuron is modeled by a dynamic variable and works as a leaky integrator of the incoming spikes so that newer spikes contribute more to the potential than older ones. If this sum is higher than a predefined threshold, the neuron fires a spike at its output.

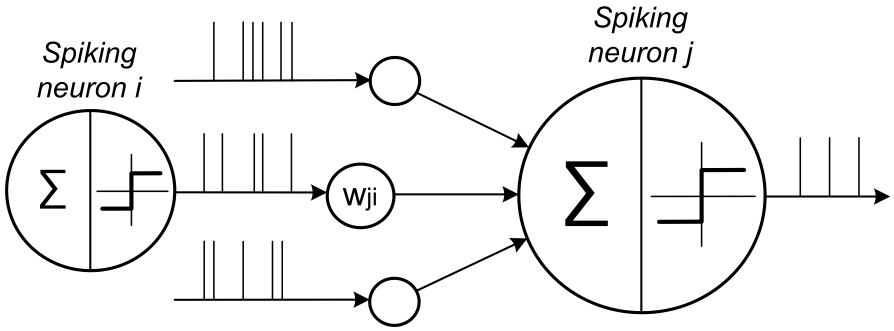


Figure 2.3.: Basic operation mechanism of the spiking neuron: the incoming pulses (spike trains, represented by vertical lines) increase the membrane potential until it reaches a threshold value and an output spike is fired. Figure adapted from [Boo04].

More specifically, the biological process of spike transmission can be described as follows ([GB14]). The action potentials (spikes) travel along the axons and activate synapses (Fig. 2.2). These synapses release a neurotransmitter that quickly diffuses to the postsynaptic neuron. There, the neurotransmitters affect the neuron's membrane potential. Excitatory Postsynaptic Potentials (EPSPs) increase the membrane potential (depolarize), and in the absence of new stimuli, this excitation leaks away with a typical time constant. On the other hand, Inhibitory Postsynaptic Potentials (IPSPs) decrease the membrane potential (hyperpolarization). When sufficient EPSPs arrive at a neuron, the membrane potential may depolarize enough to reach a certain threshold, and the neuron generates a spike itself while the membrane potential is reset. The generated spike function is the stimulation of other neurons. After the spike emission, the neuron enters a resting state (the refractory period) in which it cannot send out a spike again. Fig. 2.4 illustrates the response of the membrane potential to incoming spikes (only EPSPs are considered).

The neuron's mathematical model is a dynamical system (describing how the input spike train is transformed into an output spike train) that can be given on different levels of abstraction. While some models are very detailed, usually aimed at performing accurate simulations of the biological processes occurring in a single neuron, others are more abstract and generally intended to build networks of neurons and to make them "learn" something. The most famous example of detailed model is the Hodgkin-Huxley model ([HH52]). Other models representing different trade-offs between neuroscientific realism and computational complexity are the Spike Response Model ([MB99], [Boo04]), the Integrate-and-Fire (IF) model ([Ste67], [MB99]), the Leaky-Integrate-and-Fire (LIF) model ([GK02]), the Quadratic-Integrate-and-Fire

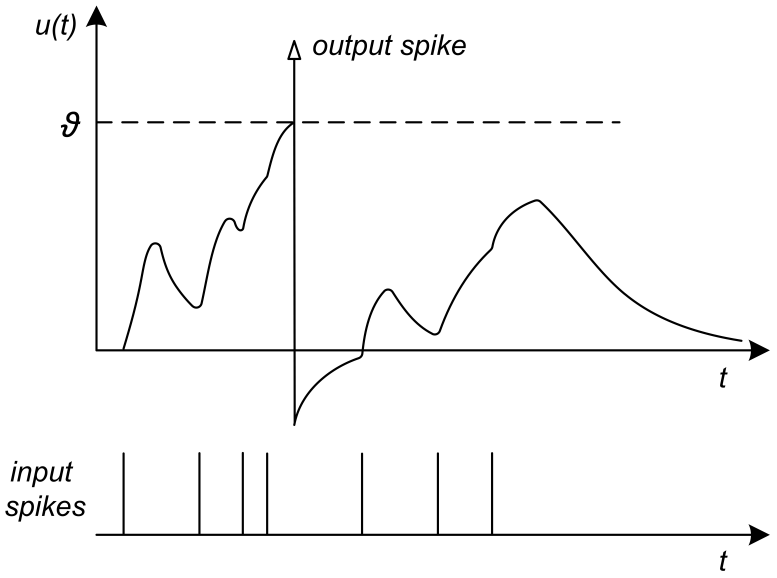


Figure 2.4.: Response of the membrane potential $u(t)$ to incoming spikes (only excitatory pulses are considered): $u(t)$ increases when spikes arrive to the neuron and decays without new inputs. Whenever $u(t)$ crosses the threshold value ϑ, an output spike is generated and $u(t)$ is reset to a low value.

model ([BL03]), the FitzHugh-Nagumo model ([Fit61]), the Morris-Lecar model ([Che93]), the Hindmarsh-Rose model ([HR82]) and the Izhikevich model ([Izh03], [Izh04]).

The most widely employed spiking neuron models are the IF and LIF models as they are relatively simple to implement while capturing generic properties of neural activity ([Vre02]). These consider biological neurons as point dynamical systems, which implies that the properties of biological neurons related to their spatial structure are neglected.

The input and output of a spiking neuron is described by a series of firing times (a spike train), which give the time at which the neuron produces a pulse. The shape of the pulse is neglected since it is generally accepted that all the neural information is carried by the timing of the spikes. Therefore, the output of a spiking neuron can be expressed as in 2.1:

$$S(t) = \sum_f \delta(t - t^f) \tag{2.1}$$

Where $f = 1, 2, ...$ is the label of the spike and $\delta(.)$ is the Dirac delta function, which satisfies $\delta(t) \neq 0$ for $t = 0$ and $\int_{-\infty}^{\infty} \delta(t) = 1$.

The LIF model is based on, and most easily explained by, principles of electronics.

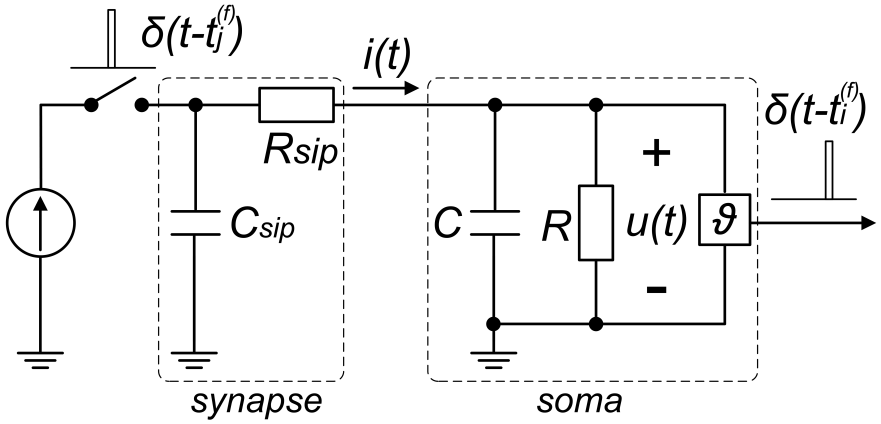


Figure 2.5.: Leaky-integrate-and-fire neuron model.

The schematic circuit representing the LIF unit is illustrated in Fig. 2.5. The incoming spike (presynaptic action potential coming from another neuron) is transformed by a low-pass filter (using the capacitor C_{sip} and the resistor R_{sip}) into a current pulse *i(t)* that flows into the postsynaptic neuron. This low-pass filter represents the synapse, junction where the presynaptic and postsynaptic neurons communicate with one another. The resulting current pulse charges the leaky integrate-and-fire circuit (composed of a capacitor C in parallel with a resistor R) increasing the membrane potential (voltage over the capacitor) *u(t)*. The neuron fires a spike at time t^f whenever the membrane potential *u(t)* reaches a certain threshold value ϑ. Immediately after a spike the neuron state is reset to a new value $u_{res} < \vartheta$ and maintained at that low level for the time interval representing the absolute refractory period Δ^{abs}. Therefore, the dynamics of the LIF neuron can be described by equations 2.2, 2.3 and 2.4:

$$C\frac{du(t)}{dt} = -\frac{1}{R}u(t) + (i_o(t) + \sum_j w_j \cdot i_j(t)) \tag{2.2}$$

where C represents the membrane capacitance, R is the soma's resistance, $i_j(t)$ is the input current from the j-th synaptic input and w_j represents the strength of the j-th synapse. The formula 2.2 also includes an external current $i_0(t)$ that may drive the neural state. For the particular case in which $R \to \infty$, equation 2.2 describes the simpler IF model.

The firing time $t^{(f)}$ is defined by the threshold criterion :

$$t^{(f)} : \; u(t^{(f)}) = \vartheta \tag{2.3}$$

An additional equation is necessary to impose the reset condition immediately after $t^{(f)}$:

$$\lim_{t \to t^{(f)}, t > t^{(f)}} u(t) = u_{res} \tag{2.4}$$

The combination of leaky integration (2.2) and reset (2.4) defines the basic integrate-and-fire model ([Ste67]). In addition, to incorporate an absolute refractory period, we proceed as follows. If *u(t)* reaches the threshold at time $t = t^{(f)}$, we interrupt the dynamics (2.2) for an absolute refractory time Δ^{abs} and restart the integration at time $t^{(f)} + \Delta^{abs}$ with the new initial condition u_{res}.

An exemplary simulation of a LIF neuron driven by an external input current $i_0(t)$ is shown in Fig. 2.6.

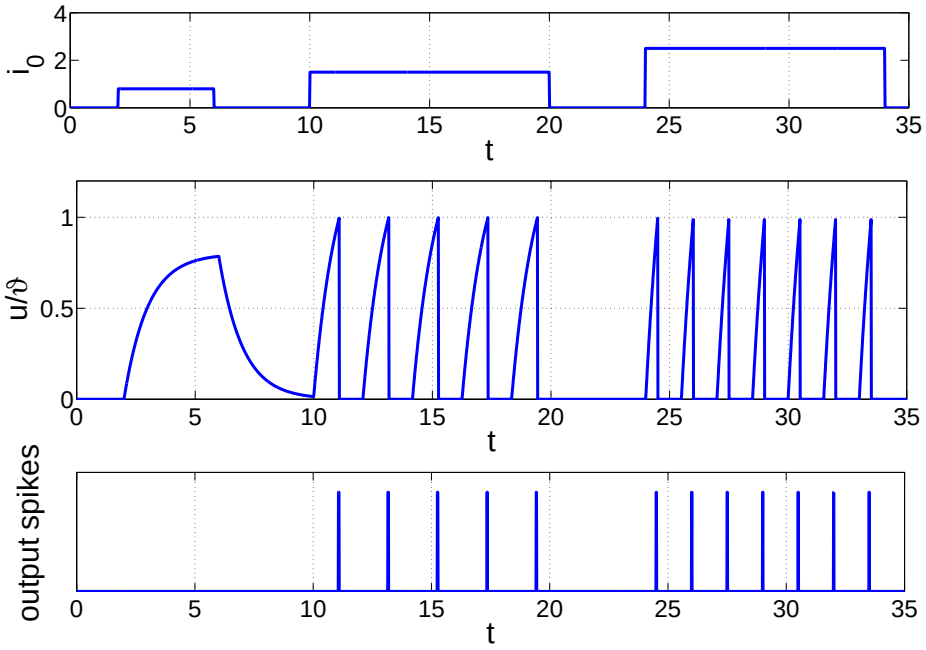


Figure 2.6.: Exemplary simulation of a LIF neuron driven by an external input current $i_0(t)$. The top panel displays the input current, the middle panel depicts the membrane potential, and the bottom panel shows the firing times of the output spikes. The variables are represented in a dimensionless form as proposed in [LR03]. The current $i_0(t)$ increases the membrane potential $u(t)$ towards the firing threshold ϑ. The neuron emits a spike when *u(t)* reaches the threshold value, and then *u(t)* is reset to u_{res} (here assumed to be $u_{res} = 0$). After firing, *u(t)* is hold at u_{res} for a refractory period. It can be observed that higher values of the input current generate output spikes with a higher rate.

2.2.1.1. Spike coding

For spiking neural networks to perform computations, it is necessary to give some meaning to neural spiking. The way how information is represented in biological neural systems is an open question. A number of different neural codes has been proposed based on observations of the nervous system in animals.

Rate coding has been the dominant paradigm of neural information encoding in neuroscience for many years. It considers the rate of spikes in a certain interval as the only measure for the information conveyed. This encoding is motivated by the observation that physiological neurons tend to fire more often for stronger stimuli and in an unpredictable way ([AZ26]). Rate coding is the notion behind standard artificial sigmoidal neurons (sec. 2.2.2). However, recent findings suggest that, at least for some neural systems, information is more likely to be encoded in the precise timing of the spikes ([WRK15]). In particular, findings in the field of neurology show that some neurons perform computations too quickly for the underlying sensory process to rely on the estimation of the neural firing rate over extended time windows (for example, human neurons in the cortex performing facial recognition, [TDVR01]). This does not mean that rate coding is not used, but other pulse encoding schemes are favored when speed is an issue ([Vre02]).

A different neural coding strategy is the "time-to-first-spike" coding, which enables ultra-fast information processing ([PK11]). In this approach, information is carried in the latency time between the beginning of the stimulus and the time of the first spike in the neural response. Other neural coding schemes include (among others) phase coding, population activity coding, rank order coding and neural codes based on correlation, which encode the information by defined correlations between the spike timing of selected neurons. The reader is referred to [PK11], [GB14] and [WRK15] for more detailed information on the different methods to encode analog information in spike trains.

2.2.2. The discrete-time (sigmoidal) neuron

As introduced in sec. 2.2.1.1, a spiking neuron can be modeled by means of the standard artificial sigmoidal neuron when the firing rate is considered to carry the neural information. The operation of the sigmoidal unit is illustrated in Fig. 2.7. In such a discrete-time model, the output of the neuron (activation) is an analog quantity (usually lying between 0 and 1) instead of a train of spikes. The synapse between two neurons is modeled by a weight variable that describes the strength of the impact on the postsynaptic neuron. The weights can be positive or negative to model excitatory or inhibitory synapses, respectively. The sigmoidal neuron sums up all the weighted firing-rates of its presynaptic neurons to get its potential. The neuron's output is calculated from this potential using the activation function, which typically has a sigmoid shape (hence the name sigmoidal neuron). The activation

variable in this model can be seen as the firing rate of the neuron (number of spikes in a certain period of time).

It is worth noting that a single spiking neuron is a dynamical system (where the incoming spikes are integrated over time to produce the output). Its output not only depends on the current input received from other neurons, but also on an internal state which evolves over time. On the other hand, the sigmoidal neuron is static (an output is produced for a given input at each discrete time step).

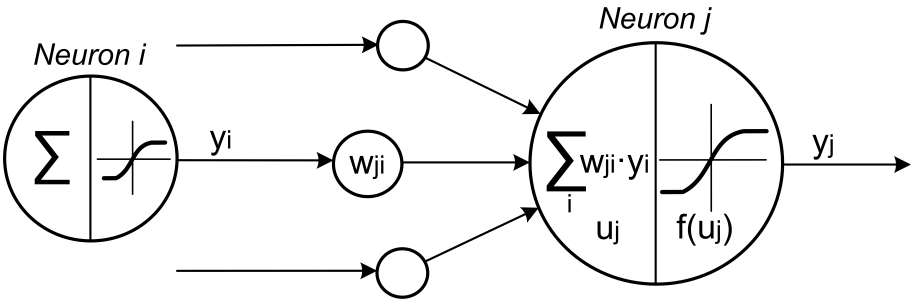


Figure 2.7.: Basic operation of the sigmoidal neuron. The activation potential (u_j) is first calculated as the weighted sum of the neuron inputs ($\sum_i w_{ji} y_i$) , and then passed to the non-linear activation function ($f(.)$, typically with sigmoid shape) to get the neuron output (y_j). The weight value w_{ji} represents the strength of the synapse between the presynaptic unit (neuron i) and the postsynaptic one (neuron j). Figure adapted from [Boo04].

Therefore, the behavior of the standard sigmoidal neuron can be formally described as follows:

$$y_j = f(\sum_i w_{ji}\, y_i + b_j) \tag{2.5}$$

where w_{ji} is the connection weight between neuron i and neuron j, y_i is the output (activation level) of the i-th neuron, b_j is the neuron bias and f is the transfer or activation function.

2.2.2.1. The activation function

The transfer function f determines the behavior of the analog (discrete-time) neuron. It usually has a sigmoid shape, that is, an "S" shaped curve. However, it may also take the form of other nonlinear functions, such as the step function or the piece-wise linear function. Even the identity function is sometimes used to build ANNs (in this case, the neuron is called linear). Some typical examples of transfer functions are depicted in Fig. 2.8.

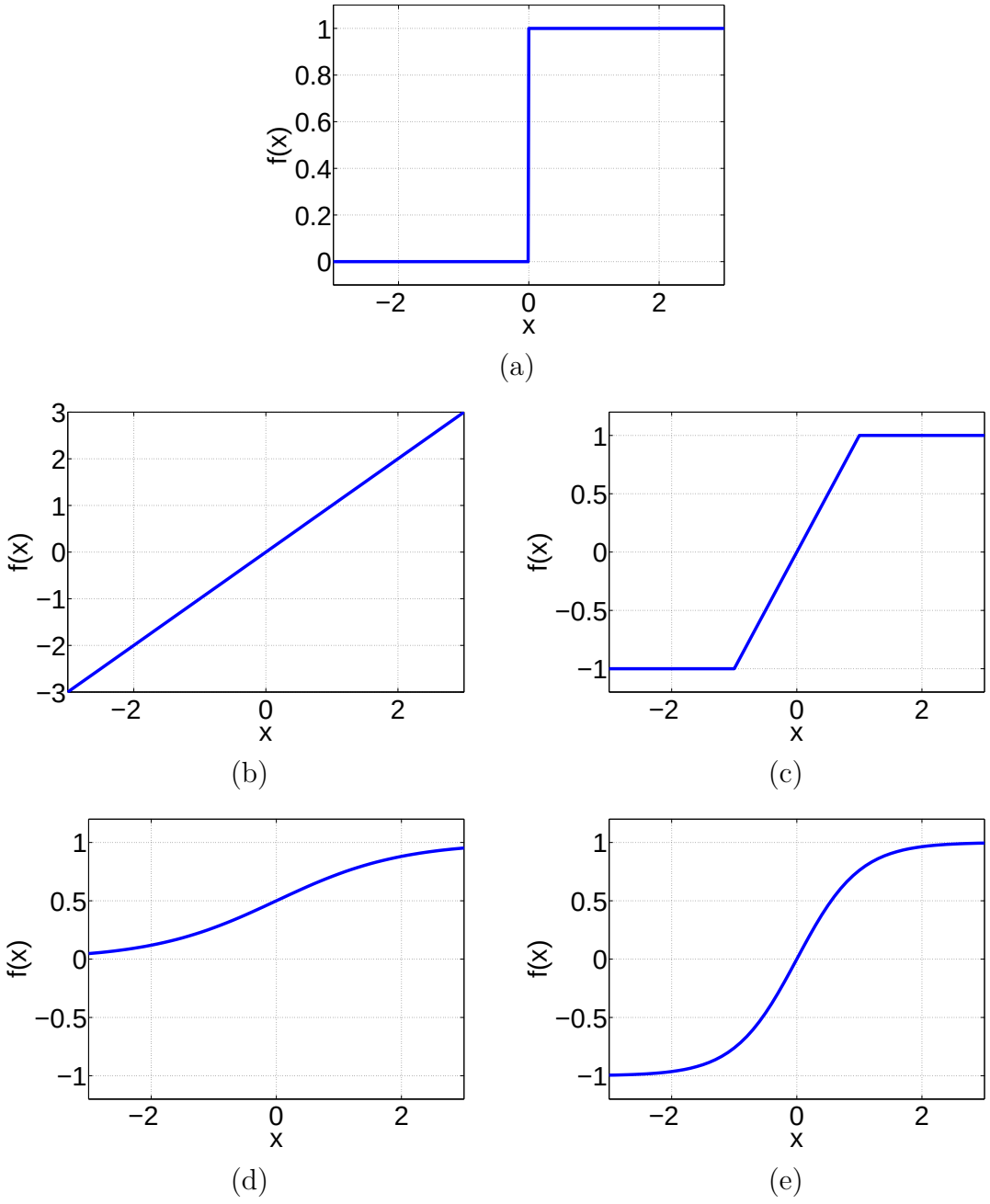


Figure 2.8.: Most common activation functions: threshold (a), linear (b), piecewise linear (c), fermi (d) and hyperbolic tangent (e).

The most commonly used sigmoid functions are the hyperbolic tangent (being $[-1, 1]$ its output range) and the fermi (or logistic) function ($[0, 1]$ range). The fermi function is defined by the formula 2.6 and is related to the hyperbolic tangent function $[tanh(.)]$ according to 2.7:

$$fermi(x) = \frac{1}{1 + exp(-x)} \tag{2.6}$$

$$tanh(x) = 2\, fermi(2x) - 1 \tag{2.7}$$

The Heaviside (threshold, or step) function represents the binary neuron, which corresponds to the first generation of neurons proposed by McCulloch and Pitts ([MP43]). In this case, the neuron can only give a digital output: it sends a binary high value ("1") if the sum of the weighted inputs surpasses the threshold level, and a low value ("0") otherwise.

On the other hand, the neurons using a continuous function (instead of the threshold one) belong to the second generation of neurons, which allows analog outputs. Networks of neurons of this type are more powerful than the ones based on first-generation units (they can perform the same functions using fewer nodes). The neurons of the second generation are also more biologically realistic and similar to the spiking neurons (representing the third and last generation) than the first generation ones since they can model the spiking frequency (firing rate) and not only a high or low value.

The second-generation activation functions are often required to be continuous, derivable and bounded. The necessity for being derivable comes from the fact that the most common learning algorithms for training an ANN to perform a certain function need to compute the derivative of the transfer function ([RHW86]).

2.2.3. Network architectures

Given the models of neurons we can define a network of artificial neurons. A network is defined by a set of neurons and the connections between them, determined by a weight matrix ($\mathbf{W}$). Network topologies can be classified into two general categories: feed-forward and recurrent.

2.2.3.1. Feed-forward neural networks

A feed-forward neural network (FFNN) has its neurons organized in layers with no feedback or lateral connections. The inputs to the network are fed in through the

input layer and the outputs are read out at the output layer. Intermediate neurons are grouped in the hidden layers. The input signal propagates through the network only in a forward direction, on a layer-by-layer basis as illustrated in Fig. 2.9. Such type of network is often referred to as a multi-layer perceptron (MLP, [MAL88]) although the neurons do not necessarily have to use a Heaviside step function (as in the original perceptron model proposed by Rosenblatt, [Ros58]), but can take on any arbitrary activation function.

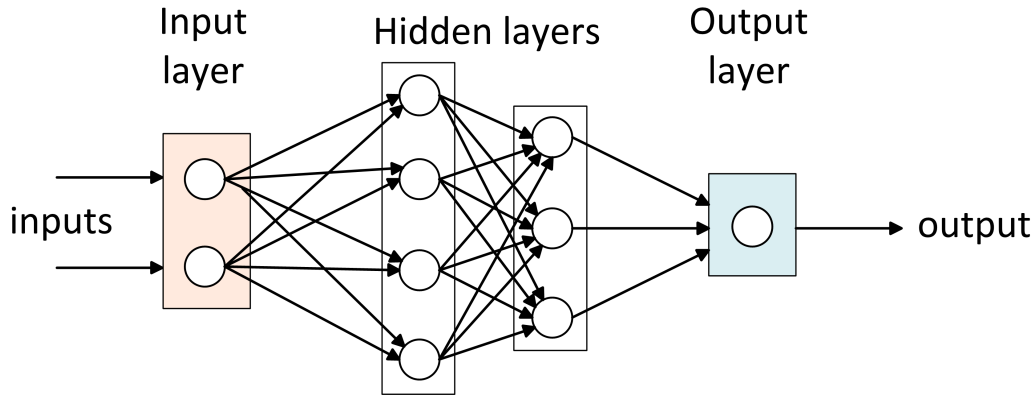


Figure 2.9.: Schematic representation of a feed-forward neural network (FFNN).

Adjusting the connection weights alters the information flow through the network. That is, if the strengths of the incoming signals of a neuron are modified, the output signal also changes in strength. The process of training the network corresponds to changing the connection weights systematically to encode the desired input-output relationships. Error backpropagation ([RHW86]) is the most commonly used supervised learning algorithm for FFNNs. Similarly to the least mean squares algorithm, it iteratively adapts the network weights based on corrections that minimize the mean square error between the target values and those produced by the network. Since the change in each weight is calculated through the derivative of the error (gradient-descent approach), a continuous activation function is required for the neurons.

MLPs are universal function approximators ([Cyb89]), so they can be used for mathematical regression and as classifiers. They have been widely studied and employed due to their learning and generalization capabilities. However, a drawback of MLPs is their inability to process temporal information, which is crucial to perform tasks such as speech recognition or time-series forecasting where the order of the input sequences is relevant. A solution to this problem is to take a window of delayed values of the input stream as inputs to the network instead of only one ([WHH$^+$89]). Another approach is to add recurrent connections to the network, which leads to a recurrent neural network.

2.2.3.2. Recurrent neural networks

Recurrent neural networks (RNNs) present feedback connections between neurons so that information can propagate forward and backwards through the layers. An example of this topology is illustrated in Fig. 2.1. The layered structure, however, can be substituted by an equivalent architecture with a single hidden layer containing internal recurrent connections as depicted in Fig. 2.10.

The recurrent connections endow the network with "memory" as the activation values of the neurons (states) do not only depend on the current value of the input signal but also on the previous network states, and therefore (recursively) on the entire input history. In an analogy with digital electronics, FFNNs (whose neuron states are fully determined by the values of the inputs) form a combinational circuit while RNNs (depending on previous network states) constitute a sequential one. RNNs are dynamical systems whose states may evolve in time even in the absence of external inputs whereas FFNNs provide passive and reactive functions to the input stimuli.

The major advantage of RNNs over FFNNs is that they can implicitly learn temporal tasks. The states of a RNN are nonlinear transformations of the input history that can be effectively used for processing temporal context information. As a matter of fact, RNNs are universal approximators of dynamical systems ([FN93], [SZ07]), which makes them a promising tool for applications requiring nonlinear time-series processing.

In recent years, RNNs have been extensively used to successfully solve computationally hard problems such as speech recognition, machine control or dynamical system modeling and prediction ([ZB01], [LZPH14], [BM15], [MYWW15], [AGAS+15], [MM15]). RNNs based on different "Deep Learning" schemes ([YD15], [Sch15], [LBH15]) and long short-term memory (LSTM, [HS97], [LW15]) approaches are specially outstanding for their high accuracy.

Nevertheless, RNNs present the shortcoming of being difficult to train. The training procedure for RNNs is complex and very time consuming. As in FFNNs, the training of RNNs is based on gradient descent, a method of gradually adapting all the network weights according to the output error gradients (derivatives with respect to the weights) so that such output training error is minimized. "Backpropagation through time" (BTT, [Wer90]) is one of the most prominent algorithms for RNN training. A review of the numerous existing approaches can be found in [AP00] and [Jae02]. In general, these algorithms suffer from slow convergence and may eventually end in local minima. In some cases, such methods might even not converge due to bifurcations during the training process (i.e., infinitesimally small changes to the network weights lead to drastic discontinuous changes in its behavior, [Doy92]). The calculation of each iteration for updating the weight parameters is computationally intensive and many update cycles are often required. As a result, considerable processing resources need to be dedicated to train large recurrent networks.

The long training time of RNNs can be compensated through parallel computing. In particular, the use of general purpose graphical processing units (GPUs) instead of ordinary CPUs makes possible to reduce the training times for some networks from months to days ([Edw15]). The recent success of recurrent deep neural networks can be partly attributed to the availability of powerful computing architectures such as GPUs ([SSC16]). However, the complex training of RNNs needs very specialized knowledge, substantial skill and experience to be successfully applied.

To sum up, RNNs are a very powerful tool for solving complex temporal machine learning tasks, but its application to real-world problems involves high computational training costs and is reserved for experts in the field. Reservoir computing offers a practical alternative to the hard traditional training of RNNs.

2.2.4. Applications of ANNs

ANNs allow to infer a function from observations. That is, they can be used to approximate a target function in a specific task by only using measured data. Prior knowledge about the input-output relationship is not necessary. What is more, when the networks present recurrences (RNNs), they can be employed for building mathematical models of any dynamical system from the experimental data, which is usually referred to as "system identification" or "black-box" modeling. This is of great utility when it is unfeasible to obtain an analytical model due to the complexity of the process. Complex dynamical systems are present in a variety of fields: physics, biology, engineering, economics, medicine, etc. Therefore, ANNs can be applied in many different disciplines ([JM99], [Dre05], [JIM16]).

The general tasks ANNs can be applied to include function approximation, time-series prediction, classification (both pattern and sequence recognition) and data processing such as filtering or clustering. Application areas are (among many others) the control of machines and industrial processes ([BL91], [CR95], [ZL08]), robotics ([Bur05b], [ASD+07]), computer vision (object recognition, face identification, event detection in video sequences, etc., [PDS09], [CMS12], [OZW+16]), radar systems ([WGM+03]), speech and bio-metric feature recognition (e.g., gesture, gait, handwriting, fingerprint, etc., [ASB13], [LW15], [ERG15], [YAT16]), thermal engineering and renewable energy systems ([MJM12], [FFD12]), fault detection ([JM15]), sensor networks ([CSC13], [AMN14]), medical diagnosis (e.g., classification of heart beat and brain activity signals, [AKS05], [SE05], [GTK+14]), finances (e.g., investment support systems based on the prediction of stock markets, [MM15]) and data mining ([LSL96]).

2.3. Reservoir computing

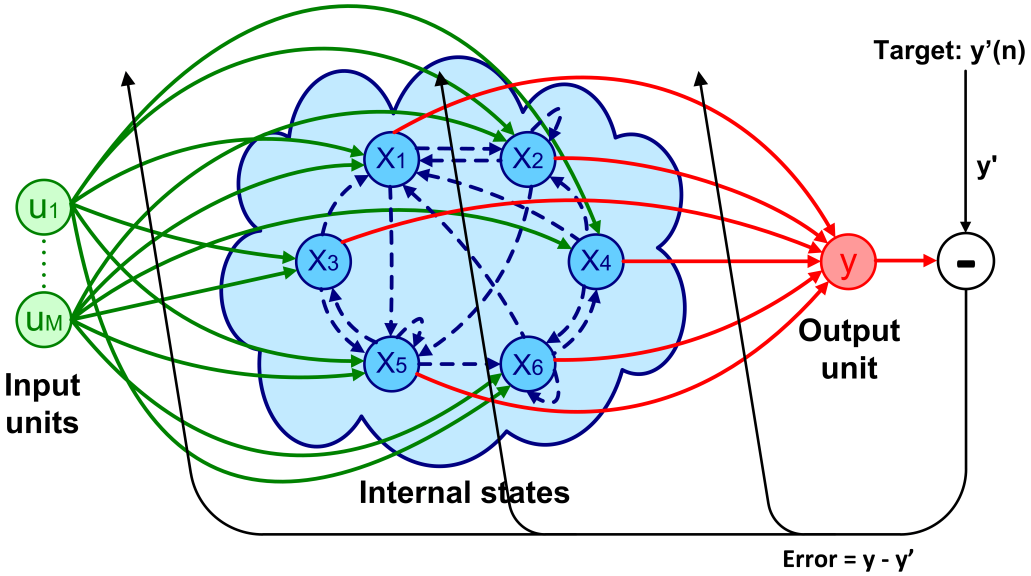
Reservoir computing (RC) is a relatively recent technique for implementing and training RNNs. Contrary to conventional RNNs, where all connections need to be adapted to minimize the training error, most connection weights in an RC system are kept fixed and only an output layer is configurable as illustrated in Fig. 2.10. This strategic design reduces the complex and time-consuming training procedure of classical fully-trained RNNs to a simple linear regression problem, which enormously facilitates the practical application of RNNs. In other words, RC takes advantage of the memory properties of recurrent networks while avoiding the difficulties associated with their training ([PK11]).

RC avoids the shortcomings of RNN training mentioned in sec. 2.2.3.2 by separating the whole recurrent network in two different parts (Fig. 2.10(b)):

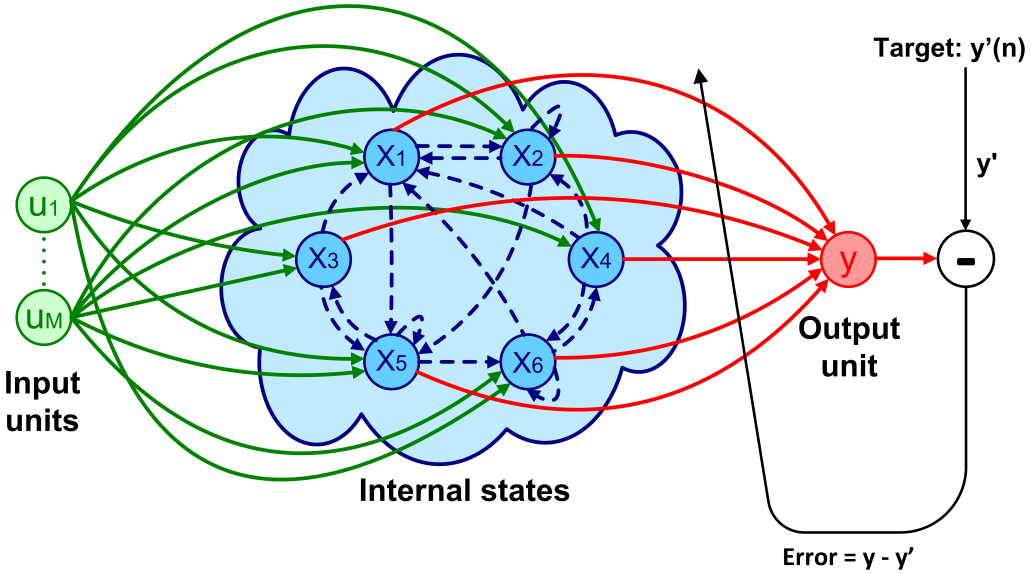
- The reservoir, which consists of a RNN that is randomly generated and remains unchanged throughout the training process. Under the influence of input signals, the neurons in the reservoir exhibit transient responses (nonlinear transformations of the input history).
- The output layer, a linear weighted sum of the input-excited reservoir states. The weights of this linear combination are obtained by linear regression, using the teacher signal as target.

The strategy of treating the recurrent part of the network as a generic device and concentrating the learning efforts on the training of linear readouts from the recurrent circuit was already suggested in the 1960s by Rosenblatt ([Ros62]). In addition, some research works in the field of computational neuroscience also proposed the use of network architectures similar to RC in the 1990s ([Dom95], [BM95]) describing a randomly constructed recurrent network that is left unchanged and connected to an easy-to-train output layer. However, the RC concept was more rigorously investigated and became popular over the last decade, after the publications of Jaeger ([Jae01]) and Maass ([MNM02]). The report of Jaeger ([Jae01]) suggested the concept of RC in the context of non-spiking ANNs under the name of echo state networks (ESNs) while the work of Maass ([MNM02]) adopted RC for spiking networks under the name of liquid state machines (LSMs). The term reservoir computing was proposed ([VSDS07]) to jointly refer to the fundamental idea behind these two independently developed approaches. The ease of use and good performance of the RC methodology led to a fast growth of the field (see, for example, [VSDS07], [JMP07], [LJ09] and [LJS12] for an overview of the related research). Nowadays, RC is regarded as one of the basic paradigms of RNN modeling ([Jae07b]).

As a particular type of RNN, RC represents a promising tool for nonlinear time series applications, such as temporal pattern classification and time-series prediction. For example, RC has been shown to present outstanding performance in predicting chaotic dynamics ([JH04], [RT11], [NR15], [GSS15]) and in speech recognition ([VSSVC05], [JLPS07]). Beside its high modeling accuracy, RC systems present the



(a)



(b)

Figure 2.10.: (a) General RNN architecture: all connection weights are adapted in order to minimize the training error. (b) RC system: only the connections coupling the RNN to the output unit (in red color) are adapted, the rest remain unchanged allowing a simple training procedure.

advantage that their easy training algorithm, contrary to conventional RNNs, does not need substantial skill and experience to be successfully applied.

The biological plausibility of RC-based systems has been another reason motivating their development. The basic idea of RC to make use of the transient states generated in an intricate network of neurons (the reservoir) when stimulated by an external input is indeed based on the way our brain seems to process the information arriving from sensory inputs. This was contemplated in the works [BM95] and [MNM02]. Following studies further discussed the neurophysiological reality of the RC scheme. For example, a LSM was proposed in [YT07] for the modeling of the cerebellum. Other works supporting the idea of the brain behaving similarly to a reservoir computer are [RHL08], [NUSM09] and [SvHS+13].

Another benefit of RC is that the reservoir does not need to be customized for a specific task. That is to say, the same reservoir network can be used as a generic computational tool to perform multiple tasks concerning the same input. Obviously, a different readout must be applied for each task. For example, in a time-series forecasting task, the same reservoir network can be used to predict the one-step and several-step ahead values simultaneously by training a different output layer for each one of the desired prediction horizons. In [Ver04], it was shown that it is possible to do speech and speaker recognition simultaneously with the same reservoir.

RC is currently a productive research area under continuous development. Numerous modifications extending the original approach have been proposed (see, for example, [XYH07], [HH10], [RT11], [BR13] and [GSS15]). More specifically, it has been shown that some adaptation within the reservoir may lead to better results than a random and fixed structure ([HMM03]). On the other hand, new ways of reading out from the reservoirs ([DSVC+09]), including combining them into larger structures ([DZ07], [NR15]), have been devised and analyzed. Therefore, the initial idea of having a fixed randomly created reservoir and training only the readout has been replaced by a current paradigm of RC (differentiated from conventional RNN techniques) where the reservoir (recurrent part of the network) is trained separately from the output layer. A review of different methods ("recipes") aimed at improving either the reservoir part or the linear readout is presented in [LJ09].

Apart from the two pioneering methods of RC (ESNs and LSMs) and their subsequent ramifications ([LJ09]), there is a more recent trend that perceives the RC principle as an strategy to implement useful computations on generic dynamical systems, treating them as reservoirs ([Ver09], [LJS12]). That is, the idea of RC is brought beyond the field of ANNs so that any high-dimensional, input-driven dynamical system, operated in the correct dynamic regime, can be used as a reservoir that combined with a linear processing method allows solving complex tasks. Such dynamical systems aimed at producing a nonlinear transformation of the temporal input signal can be implemented not only through numerical simulations but also in a physical realization. The possibility to perform useful computations using unconventional hardware platforms (physical dynamical systems) is, perhaps, the most

appealing aspect of this modification of the RC concept. Illustrative examples of this idea include physical implementations of RC-based computations by means of analog electronics ([ASVDS+11]), optoelectronic ([LSB+12], [PDS+12]) and optical systems ([VDS+08], [DSS+12]). Even the use of a bucket of water has been reported as a possible medium for RC-like processing of the input signals ([FS03]). In the context of this so-called Physical RC approach ([HFN14]), it is worth mentioning the works [SHP11], [HSBD14] and [NHLP15], which further analyze the use of physical bodies as computational resources. Such idea is related to the field of morphological computation ([HIF+11], [HIF+12], [HFN14]), which is particularly appealing for robotic applications ([PVCL06], [ZNS+13], [NHK+13]). In general, the novel Physical RC methodology can be used to complement information processing with non-conventional devices, even if they are impractical to implement basic electronic logic gates or memory cells. The output layer of such reservoirs, however, is typically implemented using conventional digital electronic computers.

Within the dynamical system-based RC approach, a particular architecture stands out for minimizing the implementation design to a simple nonlinear dynamical system subject to a self-feedback loop with delay ([ASVDS+11], [STP13], [OSP+15], [SBEM+15]). This method, along with the more traditional two main streams of RC (ESNs and LSMs) are described in more detail below.

2.3.1. Echo state networks

Echo state networks (ESNs) is one of the two initial reservoir computing methods ([Jae01], [JH04]). It was developed in the frame of machine learning applications based on the observation that often it is not necessary to adapt all the connection weights in a RNN, but it is sufficient to train a linear readout from it to obtain good performance results in a number of tasks. The fixed part of the RNN is the "reservoir" network, which presents a dynamical behavior driven by an input stream. The reservoir states, $\mathbf{x}(t)$, can be intuitively viewed as "echoes" of the input signal, $\mathbf{u}(t)$, which motivates the term "echo state network" for the reservoir and, in general, for the whole RC system consisting of an input layer, the recurrent reservoir network, and the memory-less (non-recurrent) output layer. Such scheme is illustrated in Fig. 2.11.

Before describing more formally ESNs, let me introduce the similarities of RC with the so-called kernel methods in machine learning. The objective of RC (and of any approach in machine learning) is to implement a specific nonlinear transformation $\mathbf{y}_{target}(t)$ of an input signal $\mathbf{u}(t)$ (where $t = 1, 2, 3, \ldots$ indicates the different data points in the data set). In non-temporal tasks, the data points are independent of each other while in a temporal task $\mathbf{y}_{target}(t)$ and $\mathbf{u}(t)$ represent signals in a discrete time domain and the desired output function may have memory of previous input values. Mathematically, the goal in a non-temporal task is to learn (from data) a function $\mathbf{y}(t) = \mathbf{y}(\mathbf{u}((t))$ that minimizes the error ($E(\mathbf{y}, \mathbf{y}_{target})$) between $\mathbf{y}(t)$ and

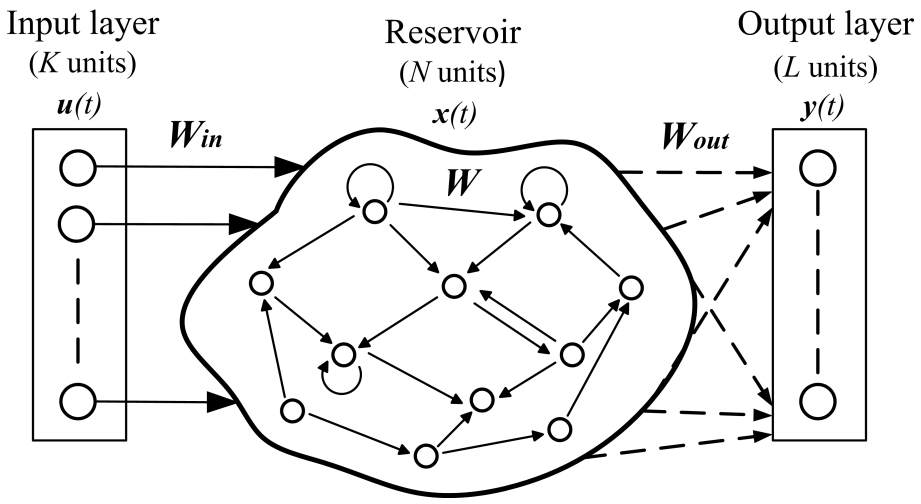


Figure 2.11.: General architecture of a reservoir computer. All connections in the system are randomly chosen and kept fixed except for the ones that couple the reservoir on the output layer (dashed arrows).

$\mathbf{y}_{target}(t)$ whereas in a temporal task the function to be learned depends on the input history: $\mathbf{y}(t) = \mathbf{y}(\mathbf{u}(t), \mathbf{u}(t-1), \mathbf{u}(t-2), \ldots)$. Different functions can be used for measuring the error, such as the mean square error (MSE, 2.8), the normalized mean-square error (NMSE, 2.9) or the normalized root-mean-square error (NRMSE, 2.10) among others:

$$MSE(\mathbf{y}, \mathbf{y}_{target}) = < \|\mathbf{y}(t) - \mathbf{y}_{target}(t)\|^2 > \tag{2.8}$$

$$NMSE(\mathbf{y}, \mathbf{y}_{target}) = \frac{< \|\mathbf{y}(t) - \mathbf{y}_{target}(t)\|^2 >}{< \|\mathbf{y}_{target}(t) - < \mathbf{y}_{target}(t) > \|^2 >} \tag{2.9}$$

$$NRMSE(\mathbf{y}, \mathbf{y}_{target}) = \sqrt{\frac{< \|\mathbf{y}(t) - \mathbf{y}_{target}(t)\|^2 >}{< \|\mathbf{y}_{target}(t) - < \mathbf{y}_{target}(t) > \|^2 >}} \tag{2.10}$$

where $< \cdot >$ stands for the mean and $\| \cdot \|$ denotes the Euclidean distance (remind that the Euclidean distance of two vectors $\mathbf{p} = (p_1, p_2, \ldots)$ and $\mathbf{q} = (q_1, q_2, \ldots)$ is defined as $\|\mathbf{q} - \mathbf{p}\| = \sqrt{\sum_i (q_i - p_i)^2}$).

To achieve the learning of the functional relation between $\mathbf{u}(t)$ and $\mathbf{y}_{target}(t)$, the reservoir structure (Fig. 2.11) performs a mapping of the (low-dimensional) input into a high-dimensional state space (corresponding to the vector of the reservoir's

neuron outputs, $\mathbf{x}(t) = (x_1(t), x_2(t), \ldots, x_N(t))^T$, being N the number of reservoir units). The expanded state vector $\mathbf{x}$ is then utilized with linear methods (linear regression) to get an estimation ($\mathbf{y}$) of the desired output function $\mathbf{y}_{\text{target}}$. That is to say, the estimated output function can be expressed as follows:

$$\mathbf{y}(t) = \mathbf{W}_{out}\,\mathbf{x}(t) \tag{2.11}$$

where $\mathbf{W}_{out}$ is the matrix that contains the trained output layer weights. A constant bias term may be added to 2.11. However, it can be considered to be implicitly implemented in the equation assuming that $\mathbf{x}(t)$ contains an additional constant element and that $\mathbf{W}_{out}$ contains a new column of weights. On the other hand, the input $\mathbf{u}(t)$ can also be included as an extra feature in $\mathbf{x}(t)$. This corresponds to a direct connection from the input to the output layer (not represented in Fig. 2.11 for the sake of simplicity) and can be explicitly expressed as follows:

$$\mathbf{y}(t) = \mathbf{W}_{out}\,[\mathbf{x}(t) \mid \mathbf{u}(t)] \tag{2.12}$$

where the symbol | denotes the concatenation of vectors. Equation 2.11 can be further extended to include a nonlinear function ($\mathbf{f}_{out}$) of the linear combination of the "echo" states:

$$\mathbf{y}(t) = \mathbf{f_{out}}(\mathbf{W}_{out}\,[\mathbf{x}(t) \mid \mathbf{u}(t)])$$

However, such nonlinearity is not usually employed in RC and $\mathbf{f}_{out}$ is simply chosen as the identity (equation 2.12) .

The idea of nonlinearly expanding the input to a higher dimension space that allows more easily extracting the desired characteristics to build an estimate of the target output is the basis of a number of widely used approaches to nonlinear modeling. Such "expansion" methods include support vector machines (SVMs, [Bur98]), feed-forward neural networks (FFNNs, sec. 2.2.3.1), radial basis function approximators (RBFs, [AZIPY93]) and extreme learning machines (ELMs, [HWL11]) among others. The function $\mathbf{x}(\mathbf{u}(t))$ that transforms the input $[\mathbf{u}(t)]$ into the higher-dimensional vector $[\mathbf{x}(t)]$ is usually referred to as "kernel" in this context. Therefore, RC can be viewed as an "expansion" method that uses the nonlinear dynamic reservoir as kernel. However, it is worth noticing that the reservoir not only allows for learning a nonlinear relation between the input ($\mathbf{u}$) and the desired output ($\mathbf{y}_{\text{target}}$), but also a temporal function of the input signal. This is thanks to the feedback loops present in the reservoir (recurrences), which endow the kernel function with memory of the input history as the reservoir states (neuron outputs) depend on the current value

of the input signal but also on its previous values. That is, the expansion function can be written as

$$\mathbf{x}(t) = \mathbf{x}(\mathbf{u}(t),\, \mathbf{u}(t-1),\, \mathbf{u}(t-2),\, \ldots) \tag{2.13}$$

or, alternatively, in a recursive form:

$$\mathbf{x}(t) = \mathbf{x}(\mathbf{x}(t-1),\, \mathbf{u}(t)) \tag{2.14}$$

More specifically, the nonlinear expansion performed by the reservoir has the form

$$\mathbf{x}(t) = \mathbf{f}(\mathbf{W}_{in}\,\mathbf{u}(t) + \mathbf{W}\,\mathbf{x}(t-1)) \tag{2.15}$$

where $\mathbf{x}(t)$ represents the vector of reservoir neuron activities at the discrete time step t, $\mathbf{f}$ is the neuron activation function (applied element-wise: $\mathbf{f} = (f_1,\, f_2,\, \ldots\, f_N)^T$) and $\mathbf{W}_{in}$ and $\mathbf{W}$ are the matrices of input and internal network connections, respectively, as indicated in Fig. 2.11. Bias terms are considered implicitly in the formula. A more general model can also include the possibility of feedback connections ($\mathbf{W}_{back}$) from the output layer units to the reservoir:

$$\mathbf{x}(t) = \mathbf{f}(\mathbf{W}_{in}\,\mathbf{u}(t) + \mathbf{W}\,\mathbf{x}(t-1) + \mathbf{W}_{back}\,\mathbf{y}(t-1)) \tag{2.16}$$

To sum up, an ESN is a recurrent discrete-time neural network with K input units, N reservoir neurons and L outputs (general setup depicted in Fig. 2.11), which is governed by the state equations 2.16 and 2.11. In such equations, the values of the input, reservoir, and output units at time step t are denoted by $\mathbf{u}(t) = (u_1(t),\, u_2(t),\, \ldots, u_K(t))^T$, $\mathbf{x}(t) = (x_1(t),\, x_2(t),\, \ldots, x_N(t))^T$, and $\mathbf{y}(t) = (y_1(t),\, y_2(t),\, \ldots, y_L(t))^T$, respectively. Accordingly, the dimensions of the weight matrices are $N \times K$ for $\mathbf{W}_{in}$, $N \times N$ for $\mathbf{W}$, $N \times L$ for $\mathbf{W}_{back}$, and $L \times N$ for $\mathbf{W}_{out}$. The weight matrices of equation 2.16 ($\mathbf{W}_{in}$, $\mathbf{W}$ and $\mathbf{W}_{back}$) are initialized at random while those of the output layer (equation 2.11, $\mathbf{W}_{out}$) need to be trained.

Regarding the reservoir activation function ($\mathbf{f}$), it is typically the hyperbolic tangent ($\tanh(\,\cdot\,)$) or any other sigmoid function although a linear reservoir (using the identity activation function) may also be considered for ESNs. The use of spiking neuron models instead of discrete-time analog neurons leads to a different "brand" of RC (LSMs) that is discussed in sec. 2.3.2.

In an ESN, the reservoir ($\mathbf{x}(\,\cdot\,)$) and the linear readout ($\mathbf{y}(\,\cdot\,)$) serve different purposes: the reservoir expands the input history ($\mathbf{u}(t)$, $\mathbf{u}(t-1)$, $\mathbf{u}(t-2)$, $\ldots$) into a

high-dimensional state space ($\mathbf{x}(t)$) whereas the output layer combines those resulting neuron signals into the desired output. As the readout performs a non-temporal function, training it is relatively simple (linear regression). Nevertheless, the reservoir needs to be built so that the echo states ($\mathbf{x}(t)$) represent a rich enough expansion of the input.

2.3.1.1. General rules for building a good reservoir

The generic guidelines or "recipes" to produce a rich reservoir are presented in [Jae01] and [Jae02]. Essentially, for the reservoir providing many different and loosely coupled states, it must fulfill the conditions of being

1. big, with a sufficiently large number of neurons (N ranging from tens to thousands);
2. sparsely connected, the weight matrix $\mathbf{W}$ must contain relatively few nonzero values (the connectivity is usually set between 1 and 20 per cent);
3. randomly connected, the connection weights (those that are different from zero) are randomly generated from a uniform distribution.

The input weights ($\mathbf{W}_{in}$) are also usually generated from a uniform distribution over an interval $[-a, a]$ (although the matrix, in this case, is normally dense instead of sparse). The value of the scaling factor a is a free parameter that can be adapted to optimize the ESN performance for a specific task. Intuitively, the choice of this value is related to the degree of nonlinearity required by the particular task since input signals close to 0 result in an operation of the sigmoid neurons that is approximately linear while inputs with a higher amplitude tend to drive the neurons towards the saturated nonlinear behavior.

On the other hand, a good reservoir must satisfy the "echo state property (ESP)", which states that the reservoir states must be influenced by inputs from the recent past, but independent of the inputs from the far past. In other words, the reservoir dynamics must present a short-term memory that gradually vanishes with time. In practice, the echo state property is almost always ensured if the reservoir weight matrix $\mathbf{W}$ satisfies $\rho(\mathbf{W}) < 1$, where $\rho(\mathbf{W})$ denotes the matrix's spectral radius (that is, the largest absolute eigenvalue $\rho(\mathbf{W}) = |\lambda_{max}|$). Consequently, to account for the ESP, the reservoir connection matrix $\mathbf{W}$ is typically scaled as $\mathbf{W} \leftarrow \alpha \frac{\mathbf{W}}{|\lambda_{max}|}$, where α is a scaling parameter in the (0, 1) range.

The optimal value for α depends on the characteristics of the given task. More specifically, on the required amount of memory and nonlinearity. The matrix $\mathbf{W}$ sets the recurrent connections in the reservoir, and therefore it is related to the network's memory. As a result, α should be set to values close to 1 if the task requires long memory, and close to 0 otherwise. However, it must also be considered that (just as in the case of the scaling of the input weight matrix $\mathbf{W}_{in}$) larger values

of α tend to drive the reservoir states towards the nonlinear behavior of the sigmoid units.

To sum up, the reservoir depends on a few free parameters (connectivity, input scaling and spectral radius) that must be adjusted for the given specific task. In practice, the values of these parameters can be chosen by means of numerical simulations that scan the system's performance for a number of different configurations. The determination of the "best" configuration parameters is usually done employing a set of data (the "validation" set) different from the training set (used to train the ESN's readout with a particular parameter configuration) and from the test set (employed to calculate the system's performance when the reservoir is set with the optimal parameters). This way, the generic random reservoirs are adapted (optimized) for a particular task (input and target output data) through the selection of the free parameters.

2.3.1.2. Different reservoir topologies

Some works have explored the possibility of using even simpler topologies of the reservoir than the classical ESN ([CM05], [FE05], [VSS07], [RT11]). In [RT11], a systematic investigation of the reservoir construction is conducted with the purpose of finding an ESN structure with minimum complexity. Several easily structured topology templates are analyzed on a number of widely used time series benchmarks of different origin and characteristics and compared with the classical ESN. The connection weights in the classical ESN reservoir, as well as the input weights, are randomly generated, but the proposed simplified models are deterministically constructed.

Three alternative structures are considered in [RT11] (illustrated in Fig. 2.12): the delay line reservoir (DLR) composed of units organized in a line with a common weight r for the feed-forward connections; the DLR with feedback connections (DLRB), which presents the same structure as the DLR but each reservoir unit is also connected to the preceding neuron with a different feedback weight, b; and the simple cycle reservoir (SCR) where units are organized in a cycle with identical connection weight r. Regarding the input layer, it is fully connected to the reservoir in all cases, all input connections have the same absolute weight value $v > 0$, and the sign of each input weight (v_i, $i = 1 \dots N$) is determined randomly by a random draw from a Bernoulli distribution of mean 1/2 (unbiased coin).

The investigation results showed that the simple deterministically constructed cycle reservoir (SCR) is comparable to the standard echo state network methodology presenting practically the same performance in all tasks. Such topology will be used throughout this thesis given its simplicity (all nodes are equal, with only two connections, one with the input unit and another one with the preceding neuron), which is advantageous for hardware implementation over the conventional ESN random structure.

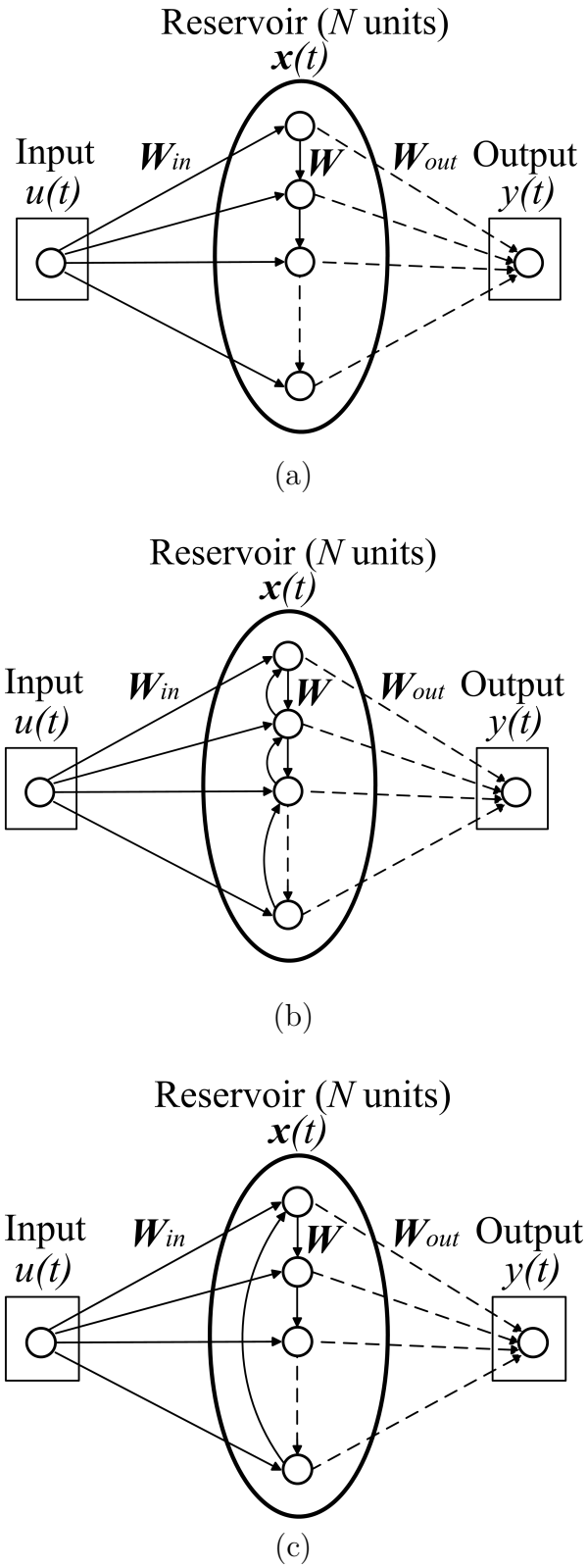


Figure 2.12.: Alternative ESN topologies: delay line reservoir, DLR (a); DLR with feedback connections, DLRB (b); and simple cycle reservoir, SCR (c).

The SCR construction leaves the user with two free parameters to be set: r and v. The optimum values for them can be obtained scanning the system's performance through numerical simulations on the validation set.

Some variations of the SCR construction have been proposed and analyzed, such as the cycle reservoir with jumps (CRJ, [RT12]) and the adjacent-feedback loop reservoir (ALR, [SCL$^+$12b], [SCL$^+$12a]). Both approaches present superior performance to standard ESNs on a variety of temporal tasks.

On the other hand, the idea of building ESNs with multiple reservoirs has been proposed in several works ([XYH07], [DZ07], [YM12], [NR15]). It was motivated by the fact that a single reservoir is only able to support a limited number of "time scales". For example, it is not possible for an ESN to be trained to work as a multiple superimposed oscillator, even with a simple function consisting of two sine waves, such as $sin(0.2\,t) + sin(0.311\,t)$ ([XYH07]). It has been theorized that this is due to the fact that the neurons in the same reservoir are still coupled so strongly that the ESN is poor in dealing with different time scales simultaneously ([WGS05]), which can be alleviated by using a structured or clustered reservoir (composed of several reservoirs sparsely interconnected) allowing multiple decoupled internal states. This way the different timescales can "live" in their own sub-reservoir. The different proposed approaches using ESNs with clustered reservoirs ([XYH07], [DZ07], [NR15]) have been shown to outperform the conventional ESN with a single reservoir in terms of accuracy of approximating highly complex dynamical systems (e.g., for prediction of chaotic time-series).

Similarly, following the idea that shallow architectures (i.e., nonhierarchical structures based on a single reservoir) are incapable of learning really complex intelligent tasks, hierarchical architectures of ESNs with an arbitrary number of layers have been studied ([Jae07a]). The structure enables discovering features on different time scales. Such approach making use of multiple layers of reservoirs has been termed "deep reservoir computing network" ([JWW15]) and has been successfully applied to various tasks, such as sequence classification ([PDW14]) and handwritten digit recognition ([JWW15], [SSC16]) presenting competitive results compared to state-of-the-art methods.

2.3.1.3. Training of the readout

Once the reservoir is constructed (either following the classical random approach, sec. 2.3.1.1, or any other simplified architecture, sec. 2.3.1.2), the network can be simulated and trained. I assume here the use of a simple linear readout (equation 2.11), as originally proposed for ESNs ([Jae01]), which has been shown sufficient for many relevant tasks and allows an efficient training. The data set (comprising the input $\mathbf{u}(t)$ and the desired output $\mathbf{y}_{target}(t)$ sequences with $t = 1, 2, \ldots T_{data}$, being T_{data} the number of time steps in the data set) is split in a subset of training samples ($t = 1, 2, \ldots T_{train}$) and another one of testing points ($t = T_{train}+1, \ldots T_{data}$).

The reservoir states $\mathbf{x}(t)$ corresponding to the given input sequence are obtained recursively, for each time step, according to the state update equation 2.15 (assuming the case of no feedback connections from the output layer to the reservoir). The network states are usually initialized to zero values (i.e., $\mathbf{x}(0) = \mathbf{0}$) to start the simulation of the network.

The reservoir states $\mathbf{x}(t)$ produced by presenting the reservoir with $\mathbf{u}(t)$ are collected over the training period (i.e., $t = 1, 2, \ldots T_{train}$) in a large matrix ($\mathbf{X}$) of dimension $N \times T_{train}$:

$$\mathbf{X} = \begin{pmatrix} x_1(1) & x_1(2) & \cdots & x_1(T_{train}) \\ x_2(1) & x_2(2) & \cdots & x_2(T_{train}) \\ \vdots & \vdots & \ddots & \vdots \\ x_N(1) & x_N(2) & \cdots & x_N(T_{train}) \end{pmatrix} \tag{2.17}$$

where $x_i(t)$ denotes the state of the i-th neuron at time step t. The desired outputs ($y_{target,j}(t)$, for $j = 1 \ldots L$) are also concatenated in a matrix ($\mathbf{Y}_{target}$) of dimension $L \times T_{train}$:

$$\mathbf{Y}_{target} = \begin{pmatrix} y_{target,1}(1) & y_{target,1}(2) & \cdots & y_{target,1}(T_{train}) \\ y_{target,2}(1) & y_{target,2}(2) & \cdots & y_{target,2}(T_{train}) \\ \vdots & \vdots & \ddots & \vdots \\ y_{target,L}(1) & y_{target,L}(2) & \cdots & y_{target,L}(T_{train}) \end{pmatrix} \tag{2.18}$$

The training of the output layer consists in computing the matrix $\mathbf{W}_{out}$ that minimizes the error between $\mathbf{y}(t)$ and $\mathbf{y}_{target}(t)$ ($E(\mathbf{y}, \mathbf{y}_{target})$; e.g., given by equation 2.8), which can be expressed in terms of matrices $\mathbf{X}$ and $\mathbf{Y}_{target}$ as follows ([Ver09]):

$$\mathbf{W}_{out} = \min_{\mathbf{W}'} \|\mathbf{W}'\,\mathbf{X} - \mathbf{Y}_{target}\|^2 \tag{2.19}$$

This problem corresponds to a least squares regression of the desired responses $\mathbf{Y}_{target}$ on the predictors in $\mathbf{X}$, which can be formulated according to the least squares normal equation ([Bro09]):

$$\mathbf{W}_{out}\,\mathbf{X}\,\mathbf{X}^{\mathrm{T}} = \mathbf{Y}_{target}\,\mathbf{X}^{\mathrm{T}} \tag{2.20}$$

In practice, this can be solved performing the next matrix algebra operations:

$$\mathbf{W}_{out} = \mathbf{Y}_{target}\,\mathbf{X}^{\mathrm{T}}(\mathbf{X}\,\mathbf{X}^{\mathrm{T}})^{-1} \tag{2.21}$$

An initial washout period of the training run containing transitory state values is usually discarded from matrix $\mathbf{X}$ before calculating the output weights. Once the output layer has been trained, the network can be simulated for the test set in the same way as for the training set and the final readouts ($\mathbf{y}(t)$) can be computed according to equation 2.11. At this point, the performance of the ESN on the test set can be evaluated by means of any desired error measure, such as the MSE, NMSE or NRMSE (equations 2.8, 2.9 and 2.10, respectively).

Usually, a validation set is considered apart from the training and testing sets. This is employed to select the optimal configuration for some parameters of the network (e.g., the input scaling and the spectral radius) using data not presented in the training process. After configuring the network with the optimal parameters, it is finally evaluated on the test set as described above.

The method provided by equation 2.21 can be extended to introduce a regularization term aimed at reducing the magnitudes of the weights in $\mathbf{W}_{out}$, which mitigates sensitivity to noise and overfitting (i.e., the model "memorizing" training data rather than "learning" to generalize from trend). The resulting method, known as ridge regression ([WSS08a]), reads

$$\mathbf{W}_{out} = \mathbf{Y}_{target}\,\mathbf{X}^{\mathrm{T}}(\mathbf{X}\,\mathbf{X}^{\mathrm{T}} + \alpha^2\,\mathbf{I})^{-1} \tag{2.22}$$

where $\mathbf{I}$ is the identity matrix of dimension $N \times N$ and α is a regularization parameter whose optimum value (the one that yields the best performance on unseen data) needs to be determined.

An equivalent way to perform regularization is by adding noise to the training data as noted in [WSS08a]. The use of regularization or other more sophisticated regression methods, such as the Wiener-Hopf approach ([OXP07]), may improve the system's performance compared to the standard least squares solution of equation 2.21. On the other hand, online training methods, such as least mean squares (LMS) and recursive least squares (RLS), can be of particular interest in applications requiring online model adaptation ([JH04], [HLM14]). For example, in [Jae03], the joined use of ESNs and the RLS algorithm has been applied to track a non-linear system whose parameters (and therefore the targeted input-output relation) change over time.

In this thesis, I have limited to the standard least-squares regression method (neither regularization nor online training have been employed). More specifically, I have used the "regress" (multiple linear regression) MATLAB function ([web17a]), which is based on matrix commands to determine the solution to the normal equation 2.20.

The described training procedure can also be used for pattern classification tasks where the desired output is not a real value but a category indicator. In this case, a real-valued readout is employed for each class and the strengths of such outputs are interpreted as "votes" for the corresponding classes. For example, a constant value $y_{target,j}(t) = 1$ can be assigned to the desired output when the input signal (at time step t) corresponds to that of the right class j and a value equal to -1 otherwise. To perform the classification, the network readout signals ($y_j(t)$) are cumulatively added throughout a certain number of time steps (usually corresponding to the duration of the temporal pattern) after which sufficient information has been reached to make the decision. At this point, the output category matching the input signal ($j = k$) is assumed to be the one that provides the greatest value of the sum $\sum_t y_j(t)$.

In the case of an ESN presenting feedback connections from the output layer to the reservoir ($\mathbf{W}_{back}$, equation 2.16), the training can be carried out through "teacher forcing", an strategy to uncouple the recurrent relationship between the reservoir and the readout. It consists in feeding the reservoir with the desired output $\mathbf{y}_{target}(t)$ instead of the real output $\mathbf{y}(t)$ during the training process. Including the recurrence between the reservoir and the readout is required for some tasks, such as pattern generation ([Jae01]). Although the models with feedback weights $\mathbf{W}_{back}$ work properly if the output is learned very precisely ($\mathbf{y}(t) \simeq \mathbf{y}_{target}(t)$), they often suffer from instability caused by the amplification of small errors that can eventually make $\mathbf{y}(t)$ deviate from the expected $\mathbf{y}_{target}(t)$.

2.3.2. Liquid state machines

Liquid state machines (LSMs) are the other early approach of RC ([MNM02]). Essentially, a LSM consists in a fixed reservoir network made up of spiking neurons (sec. 2.2.1) along with an adaptable output layer. LSMs were developed in the context of neuroscience aimed at modeling computation in biological neural systems, which motivated the use of realistic spiking neuron models and biologically plausible connectivity structures and weight settings ([NMM03], [MNM04], [MJS05], [HM07]). In this approach, the reservoir network forming a dynamical system is conceived as a "liquid" surface that exhibits perturbations caused by the external inputs (usually presented as spike trains). The readout neurons can extract from the current state of such recurrent neural circuit (the reservoir) information about current and past inputs that may be needed for diverse tasks. The readout can be sigmoid or linear as in ESNs, which requires a mechanism for averaging the spike trains of the reservoir neurons to obtain real-valued outputs that can be processed using, for example, linear regression. In some cases, spiking neurons (or even feed-forward networks of spiking neurons) are employed in the output layer and more complex training procedures are required ([CR07], [SVVC07]).

Apart from their original interest for biological modeling, LSMs can be employed as a powerful machine learning tool. Although ESNs are more widely used in engineering applications as they are easier to implement and require less computational

resources, LSMs may outperform ESNs in certain tasks given that spiking neurons (present in LSMs) are able to perform more complicated information processing than the analog neurons (employed in ESNs), which only emulate mean firing rates of biological neurons as introduced in sec. 2.2.2. For example, LSMs have been shown to present outstanding performance in real-world problems such as the noisy temporal pattern classification of textures ([Vre04]), speech recognition ([VSSVC05], [SDVC08]) and robotics applications ([JM05]).

2.3.3. Single dynamical node reservoir computing

A more recent trend of RC is based on exploiting the computational capacities of certain dynamical systems, which can be used as reservoirs to solve complex tasks combined with a simple linear processing method. This concept opens the door to physical realizations that can be built using dedicated hardware. See, for instance, the works [VDS+08], [ASVDS+11], [LSB+12], [PDS+12], [DSS+12], [BSMF13], [SOK+14], [NVDVDS14], [FVVW+14] and [HSR+15]. In the context of this specific type of RC systems, it stands out the use of a dynamical system that comprises a single nonlinear node delay-coupled to itself, which allows the easy realization in optical and electronic hardware. Many of the physical realizations of the RC concept are based on this particular setup.

While classical reservoirs such as ESNs and LSMs present an explicit spatial structure of multiple connected nodes, the reservoir in delay-based systems is implicit in the sampled solutions of a single delay differential equation (DDE). That is, the points obtained from different time positions throughout the delay period represent "virtual nodes" that play the same role as the nodes in a traditional reservoir. Such reservoir based on a single dynamical node is usually called delay-coupled reservoir (DCR, [STP13]) or time-delay reservoir (TDR, [GHLO16]). The general concept of this approach is depicted in Fig. 2.13, which illustrates how the recurrent reservoir network is emulated by the single nonlinear dynamical node (NL) subject to delayed feedback. The virtual nodes in the reservoir are defined as N equidistant temporal points throughout the delay interval τ. That is to say, they are separated in time by a period $\theta = \tau/N$. The value of the delayed variable ($x(t)$) at each one of these points is analogous to the output of the neurons in a classical reservoir. These states characterize the transient response of the nonlinear node (NL) to a certain input at a given time, and therefore represent a nonlinear transformation of the input signal and of its history (taken into account through the delay term in the DDE).

More specifically, the TDR approach is formalized by the following DDE (equation 2.23, the sampled solutions of which are used as reservoir states):

$$\frac{dx(t)}{dt} = [-x(t) + f(x(t-\tau),\, J(t))]\frac{1}{T} \tag{2.23}$$

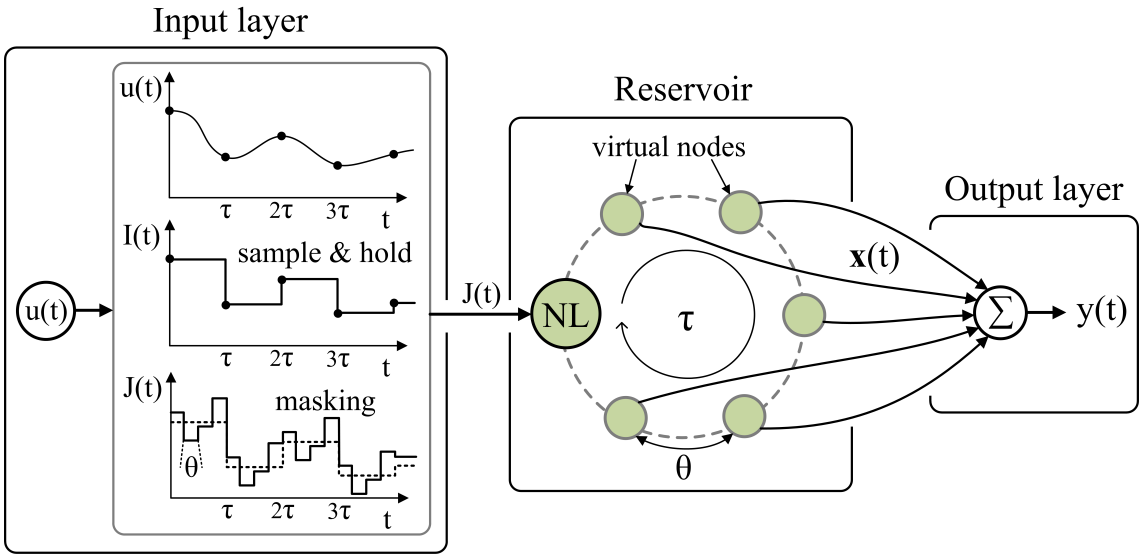


Figure 2.13.: Schematic view of a reservoir computer based on a single nonlinear node (NL) with delay (τ). Virtual nodes are defined as temporal positions in the delay line.

where x denotes the dynamical variable, τ is the delay time, $J(t)$ is the signal driving the system (derived from the input stream $u(t)$ through temporal multiplexing as I describe next), f is a nonlinear function (often referred to as kernel), and T is the system's time constant, which is usually omitted as it can be normalized to $T = 1$ using a dimensionless time in the equation.

Time-continuous delay systems are infinite dimensional and exhibit a short-term memory. This properties motivate their use as reservoirs. In addition, the idea of replacing the large networks employed in traditional RC by a single-node architecture requiring minimal resources to be implemented seems very attractive.

The injection of the input signal into the reservoir is achieved by multiplexing it in time as illustrated in Fig. 2.13. First, the input stream $u(t)$ undergoes a sample and hold operation for time periods of length τ. That is, the input $u(t)$ is discretized using a time step equal to τ resulting in the signal $I(t) = u(\lfloor \frac{t}{\tau} \rfloor)$, where $\lfloor \cdot \rfloor$ denotes the integer part (truncation) of the number. Then, $I(t)$ is multiplied by a mask ($\mathbf{M} = (m_1, \ldots, m_N)^T$, representing a vector of weight values where N is the number of virtual nodes) that is piece-wise constant for short periods of length θ (corresponding to the separation between the virtual nodes). The resulting signal, $J(t) = m_i\, I(t)$ (with $i = 1, \ldots, N$), is a continuous-time scalar function constant over θ periods that is employed to feed the dynamical node. The masking procedure prevents the dynamics of the system from saturating.

The evolution of $x(t)$ is found by numerically solving the DDE. For example, if we consider the Euler time-discretization of equation 2.23 with an integration step Δt,

$x(t)$ can be recursively found by the next expression:

$$\frac{x(t) - x(t - \Delta t)}{\Delta t} = -x(t) + f(x(t - \tau),\, J(t)) \tag{2.24}$$

The state variable $x(t)$ is readout (taken as the output of a virtual node) every time period θ (which corresponds to a certain number of integration steps). It has been shown ([GHLO15]) that any virtual neuron value (noted as $x(t + i\theta)$, i representing the neuron index) can be expressed as a linear combination of the immediately previous neuron value ($x(t + (i - 1)\theta)$) and a nonlinear function of both the same neuron value in the previous layer (that is, the state value a delay period before, $x(t+i\theta-\tau)$) and the input at the current time ($J(t+i\theta)$). Such resulting processing scheme for the "delay-based" dynamical system is illustrated in Fig. 2.14.

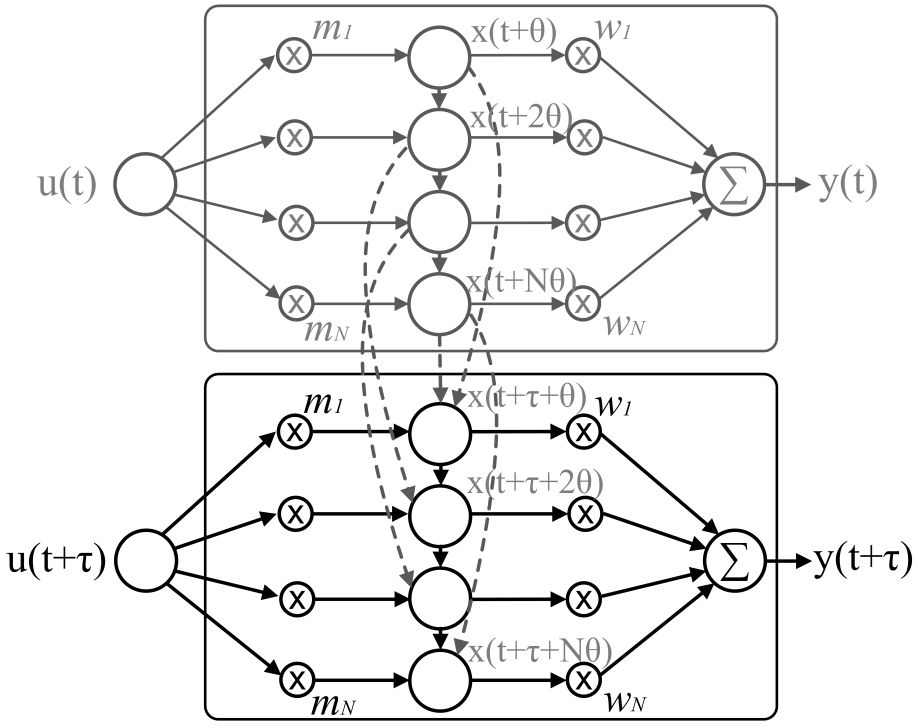


Figure 2.14.: Computing process scheme of the delay-based dynamical node RC system. The output of a virtual node (i) at a certain time t ($x_i(t) \equiv x(t + i\theta)$) depends on the external input $u(t)$ (multiplied by the corresponding mask, m_i), on the previous node state ($x_{i-1}(t) \equiv x(t+(i-1)\theta)$), and on the state of the same node a delay period before ($x_i(t - 1) \equiv x(t + i\theta - \tau)$). The final output $[y(t)]$ is computed every time period τ as a linear combination of the virtual node states obtained within that delay period $[t + \theta,\, t + \tau]$.

Usually, the notation of the classical reservoir is employed to denote the virtual

nodes in the TDR approach so that $x_i(t)$ refers to $x(t + i\theta)$, $x_i(t - 1)$ refers to $x(t + i\theta - \tau)$, and $J_i(t)$ corresponds to $J(t + i\theta)$.

The separation among virtual nodes (θ) has an important role and can be used to optimize the reservoir performance ([ASVDS$^+$11]). When $\theta < T$ the states of the virtual nodes are mainly influenced by the states of the neighboring nodes, while large values of the distance between neurons give predominance to the node states at the previous delay period and to the input signal.

After processing the input signal, a training algorithm assigns an output weight to each virtual node (w_i), such that the weighted sum of the states ($y(t) = \sum_{i=1}^{N} w_i\, x_i(t)$) approximates the desired target value as closely as possible. The training follows the standard procedure for RC (linear regression) described in sec. 2.3.1.

As regards the choice of the nonlinear function in 2.23, it depends on the particular physical implementation that is envisioned. The most explored nonlinear functions in the literature are the Mackey-Glass ([MG77]) and the Ikeda ([Ike79]) kernels. The Mackey-Glass oscillator, extended from the original dynamical system to include the external input $J(t)$, presents the following form:

$$f(x, J) = \frac{\eta\,(x + \gamma\, J)}{1 + (x + \gamma\, J)^p} \tag{2.25}$$

where η, γ and p are parameters of the kernel. Such non-linearity can be implemented through an analogue electronic system ([NPT95]) serving as physical platform for the TDR-based computations as shown in [ASVDS$^+$11]. On the other hand, the Ikeda kernel is employed in optical RC implementations ([LSB$^+$12]). It reads

$$f(x, J) = \eta\, \sin^2(x + \gamma\, J + \phi) \tag{2.26}$$

where η, γ and ϕ are parameters. For both models, η and γ represent the feedback strength and the input gain, respectively. Different nonlinear functions, such as the one proposed in [GSMOP12], may also be used.

The performance of a TDR system for a given task depends on the kernel parameters (η, γ, ...). Therefore, apart from the output layer weights, the TDR parameters also need to be tuned in order to find the configuration that performs better. Usually, such optimal configuration is found through numerical scannings.

The network topology that results from the TDR approach where each neuron is connected to the preceding one in a chain and it is time-delay connected to the same neuron in the previous time (as observed in Fig. 2.14) resembles the SCR structure of neurons organized in a ring (Fig. 2.12). The similarity of both schemes have been noted in [GSMOP12]. Indeed, the TDR methodology can be viewed as

an innovative way to serially implement a network of nonlinear nodes with cyclic connectivity. This, however, is achieved through the solutions of a continuous-time dynamical system (equation 2.23) instead of using a discrete-time one as in ESNs (equation 2.16).

The reservoir map constructed by sampling the solutions of the differential equation 2.23 has been shown ([GHLO15]) to be equivalent to a description of the form

$$\mathbf{x}(t) = \mathbf{F}(\mathbf{x}(t-1), \mathbf{J}(t)) \tag{2.27}$$

that corresponds to a non-autonomous discrete-time dynamical system such as that used for ESNs (equation 2.14).

A sequential implementation of the ESN with chain topology (SCR network) is presented in chapter 7. There, I highlight the similarities between such sequential design (Fig. 7.3) and the delay-based dynamical node RC (Fig. 2.14).

Finally, it is worth noticing that although the idea of reducing a complex network to a single node paves the way to minimal hardware implementations, it presents the major shortcoming of requiring a serial feeding of the nodes, which implies a lower processing speed compared to the classical parallel designs of RC (ESNs and LSMs).

2.3.4. Applications of RC

Reservoir computing, as a RNN approach, can be used as a tool for black-box modeling of nonlinear dynamical systems, which embraces tasks such as simulation, control, approximation of temporal functions, generation and detection of temporal patterns, time-series prediction, memorization, filtering and classification. Application fields include digital signal processing (e.g., speech recognition, [JTDM15], and audio processing, [HH10]), robotics (e.g., robot control and localization, [SP05], [ASD+07]), computer vision (e.g., event detection in video sequences, [JWW15]), language processing (e.g., prediction of words in a sentence, [TBCC07]), music generation ([Eck06]), medicine (e.g., monitoring of physiological signals, [BVN+13]) and economy (e.g., financial forecasting, [IJK+07]) among others. Below, I briefly describe temporal pattern recognition and time-series prediction, which are present in the majority of RC applications. Note that a more detailed description of the application areas of RC (particularly focused on the potential applications of hardware-based RC systems) is given in chapter 9.

2.3.4.1. Temporal pattern recognition

The task of temporal pattern recognition consists in the classification of consecutive sections of the input signal (a time-series) into one of several categories. An illus-

trative example of time-series classification is the online handwriting recognition, which employs the trajectories of the pen tip as input sequence. Typical handwriting temporal patterns (given by the differentiated writing trajectories, v_x and v_y) corresponding to three different characters ("a", "b" and "c") are represented in Fig. 2.15. Two samples (produced by the same writer) are shown for each character. Subtle variations can be observed from one to another sample of the same class. The solution of the classification problem of handwriting trajectories using the ESN approach is presented in sec. 9.2.

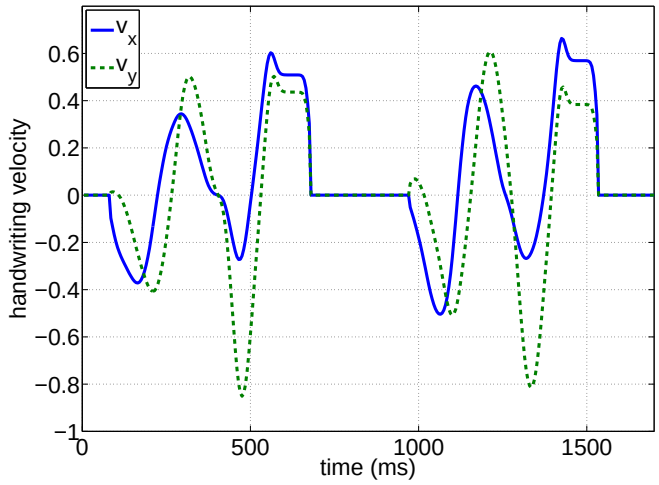
Other emblematic examples of temporal sequence recognition, which may be dealt with RC techniques, are speech recognition ([ZYCS15]), classification of the activities performed by persons in video sequences ([YM12]), classification of faults in industrial processes and plants ([JM15]), classification of heartbeat and brain activity signals for medical diagnosis ([EMSFM15], [BVvM$^+$11]), gait recognition (identifying humans through the style in which they walk, [YAT16]) and voice verification ([DN16]).

2.3.4.2. Time-series prediction

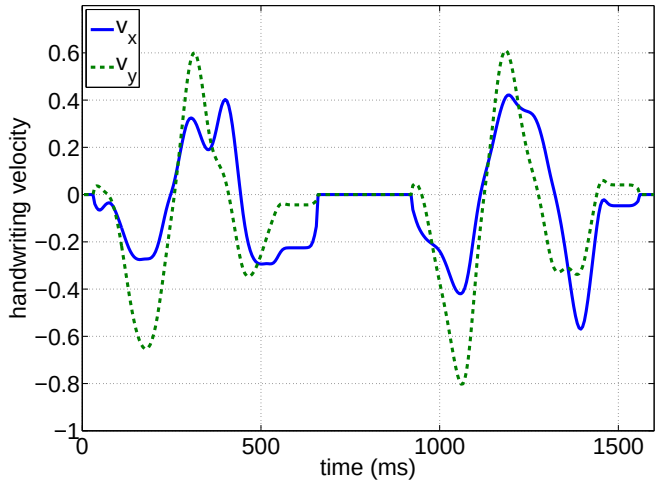
In time-series prediction or forecasting, the objective is to predict future values based on previously observed ones. Thus, the input sequences are mapped onto a real-valued output sequence that represents one-step or several-step ahead predictions of the desired variable. A representative example is the Santa Fe laser time-series prediction task, a widely used benchmark in the RC literature ([RT11]). The task consists in forecasting an experimental recording of the output power of a far-infrared laser operating in a chaotic regime. It is usually evaluated for one-step ahead predictions. That is, the value of the series at the current time is introduced each time step as input to the system and the time-series value corresponding to the next time step must be predicted. Data are available at [WG15]. A fragment of such time-series can be observed in Fig. 3.17. This task is used throughout this thesis for the evaluation of the proposed RC hardware implementations.

Other time-series often employed to evaluate the prediction accuracy of RC systems are the IPIX radar backscatter data from an ocean surface (analyzed in sec. 5.4.2 and available at [BC01a]) and other synthetic data sets, such as the Mackey-Glass ([JH04]), the Henon map ([Hen76]), and the NARMA (nonlinear autoregressive moving average time-series, see, e.g., [AP00]).

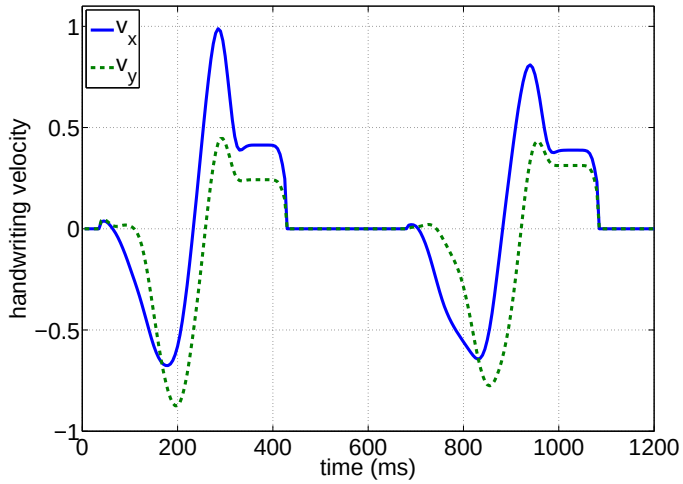
Apart from these more or less academic examples, time-series forecasting can be applied to real-world engineering problems such as predictive control of nonlinear plants ([SP05]), motion prediction of robots ([Bur05a]), damage detection in structural health monitoring systems ([WDH15]) and financial forecasting ([MM15]).



(a)



(b)



(c)

Figure 2.15.: Temporal patterns of the handwriting trajectories (v_x and v_y) for characters "a" (a), "b" (b) and "c" (c).

2.4. Hardware implementation of neural networks

Although the majority of ANNs are implemented in software using conventional processors, some applications require the use of specific hardware. In some cases, for example, a personal computer (PC) cannot be employed to perform the desired task, but it must be carried out through compact and energy-efficient objects allowing for autonomy and mobility. On the other hand, some real-time applications demand a processing speed that is too high for conventional PCs or digital signal processors (DSPs).

Unlike general-purpose sequential processors, hardware devices specifically tailored to implement ANNs (often called hardware neural networks, HNNs) can benefit from the inherent parallelism in the neural processing. In general, HNNs can offer advantages in terms of speed, energy consumption, cost and fault-tolerance.

Specialized hardware implementing massively parallel network architectures can be used to speed up the neural processing in applications that demand high-volume real-time processing, such as computer vision tasks (see, for example, the implementations presented in [LGMARB+05], [AKK+16] and [UHS16] for image/video processing), image search ([KSH12]) and data search and mining ([LRS+12], [MCO+17]). Apart from a timely data processing of large data sets, HNNs make possible reducing the power consumption. This is particularly interesting for autonomous/mobile applications constrained in terms of power supply. Examples of this include, among others, the control of machines (e.g., mobile robots) and industrial processes ([CR95], [LBH02], [LZF06], [ZL08]), distributed sensory networks ([MP08], [CSC13]), portable medical applications ([PCFB09], [RGW+09], [SAC+10]), handwriting ([SJM04], [MPR09], [IBA+10]) and speech recognition ([WSLW95], [SDVC08], [LHP+16]) systems.

Hardware implementations can also reduce the total cost, which is extremely important for ubiquitous consumer products, such as mobile electronics devices. For example, low-cost HNNs performing real-time image and speech processing can be interesting for smartphones and tablets ([KKJ+15], [MKM15]). Another potential advantage of HNNs over sequential processor-based implementations is related to the capability of distributed architectures to correctly operate (though with slightly reduced performance) in the presence of faults in some components by virtue of the inherently redundant network structure. Such fault-tolerance is crucial in applications that require complete availability or are safety critical (i.e., those systems whose failure could result in loss of life, significant property damage or damage to the environment, such as medical devices, aircraft flight control, weapons and nuclear systems; see, e.g., [LH94] and [Kni02]).

Specialized ANN hardware can be used either to support or replace software. An example of the first situation is the use of HNNs as accelerators for mobile application processors ([KKJ+15]). Many applications on mobile devices (smartphones, tablets, etc.) demand large amounts of computation that must be efficiently ex-

ecuted to achieve an extended battery lifetime. Application-specific accelerators, along with general-purpose CPU cores, are commonly employed on mobile application processors to deliver both high performance and energy efficiency. Neural network accelerators are used as function approximators and allow significant performance and efficiency gain especially in applications that tolerate a certain degree of error, such as video and audio processing ([ESCB13]).

HNN-based co-processing (supporting conventional PCs) can also be of interest in applications such as stock market prediction, image classification and data mining ([MCO+17]) where high volumes of data must be analyzed and/or the task requires a very high number of neurons. For example, [IJK+07] and [JWW15] report the use of massive networks, both with approximately 50000 nodes, for financial forecasting and image recognition tasks, respectively.

On the other hand, software is usually completely replaced by hardware realizations in those applications involving energy consumption restrictions that necessitate embedded systems or low-power devices. This is the case of wearable ([SAC+10]) and implantable ([RWRI09]) medical devices, wireless sensor networks ([MP08]), control units ([LZF06]) or any other smart systems that need to operate autonomously without power grid access. In any case, it must be noted that, in practice, a HNN realizing an ANN model alone is usually not sufficient by itself and a fully operational system may demand many other components for sensor acquisition, pre and post processing of the input and output signals, etc.

Some studies implementing neural network models in hardware have included the training phase of the algorithm in addition to its execution. For instance, [SMA07] and [OZJM+16] provide hardware implementations of the back-propagation learning algorithm, which can significantly reduce the required training time compared to a training performed externally in a PC, especially when large data sets must be learned.

Scalable general purpose parallel computers and graphical processing units (GPUs) have often been used to accelerate the execution and training process of neural network models ([SS98], [LDL+16]). Similarly, microcontrollers are widely used in applications requiring "intelligent" low-cost and low-power devices (e.g., for control, [LSM05], and wireless sensor networks, [MP08]) as they are economical and easy to program (standard programming languages can be used) although limited in terms of memory size and computing speed. Nevertheless, neuromorphic engineering proposes a different solution constructing specific chips with circuits designed to implement neural networks from the ground.

Neuromorphic hardware can be classified in two main categories: analog ([Mea89], [Hir93]) and digital ([Kun93], [Ien95]). On the one hand, analog circuits benefit from consuming very little silicon area and presenting high processing speed, although this comes at the price of a limited accuracy of the network components. In addition, these systems are complex to reconfigure; this is the reason why analog implementations have primarily focused on trying to emulate the characteristics and behavior

of real biological neurons ([MB99]). On the other hand, digital implementations of ANNs offer higher flexibility and accuracy. In addition, the digital design enables a straightforward integration of the hardware neural network module into more complex digital designs. Their major shortcoming is the high number of required transistors when compared with analog implementations, which implies higher cost and more power consumption ([LB12]). Apart from the two main streams for HNN realization (analog and digital), there are also hybrid (mixed analog/digital, [AE94]) and non-electronic implementations, such as the optical ones ([MFS96]).

There exist many HNNs for real-world applications, such as optical character recognition, speech recognition, control, robotics, etc. Some examples include the "tensor processing unit" developed by Google ([Jou16]), the "TrueNorth" chip produced by IBM ([MAAI+14]) and the neuromorphic hardware implementations developed under the "SpiNNaker" ([FLP+13]), "BrainScaleS" ([SBG+10]) and "Silicon Neurons" ([ILBH+11]) projects. The reader is referred to [MS10] for a comprehensive review of the HNN research over the last two decades including numerous academic and commercial examples.

The present thesis deals with the digital hardware implementation of reservoir computing systems. Even though a number of physical realizations of the RC approach have been proposed (e.g., [FS03], [VDS+08], [ASVDS+11], [LSB+12], [PDS+12], [DSS+12], [BSMF13], [NVDVDS14] and [FVVW+14]), most of them are implemented using unconventional hardware platforms, usually optical systems based on the single dynamical node setup described in sec. 2.3.3. To the best of my knowledge, only two RC digital hardware implementations can be found in the literature ([SDVC08] and [WLL16]), which are based on the liquid state machine (LSM) approach (sec. 2.3.2). I will mainly focus on echo state networks (ESNs, sec. 2.3.1) providing different digital implementation designs seeking a reduced usage of hardware resources. I will also present a digital implementation of the RC approach that employs a single nonlinear oscillator with delayed feedback as dynamical node.

The designs proposed throughout this thesis will be implemented and evaluated using field-programmable gate arrays (FPGAs). FPGAs are a reconfigurable alternative to ASICs (application-specific integrated circuits). FPGAs can be configured to implement different combinational and sequential logic designs. Although they were originally created with the aim of prototyping digital circuits for ASICs, recent technological advances have made possible the construction of FPGAs with high processing power and memory storage enabling their direct use in a wide range of applications (e.g., robotics, industrial control, telecommunications, aerospace and defense, and medical electronics, [MIC+11], [RAVPM15]). FPGAs are a suitable hardware resource for neural network implementations since they are intrinsically parallel devices (as is the processing of information in neural network models) and can be reconfigured by the user. Despite being slower and consuming more power than dedicated ASICs, FPGAs allow a much faster implementation cycle and if a relatively small number of chips is required, they are cheaper than ASICs. To sum up, FPGAs represent a compromise between the programmability of classic proces-

sors and the parallel nature and high speed of ASICs. Some examples of FPGA implementations of neural networks can be found in [OR06], [NDMM07], [MHS08], [DGLZ12] and [CMA+16].

FPGA boards are programmed using hardware description languages (HDL), such as VHDL (VHSIC hardware description language) or Verilog. Their configuration is usually a time consuming task if compared to microprocessor programming, but HDLs are independent of devices and manufacturers, so they are compatible with many different platforms and can be synthesized into any FPGA without major changes. Indeed, the developed designs can also be exported to build specific chips if desired.

Despite recent advances on the computational power of FPGA boards, their circuit density is still quite limited compared to ASICs. This makes challenging the task of implementing large ANN models, as is the case of reservoir networks, often with hundreds or thousands of neurons, especially in low-cost devices. The fundamental problems ("bottlenecks") limiting the size of FPGA-based ANNs are the nonlinear activation function and the multiplication operation ([ASG91], [HAM07]), whose implementation requires large resource. As each synaptic connection in an ANN requires a single multiplier, the fully-parallel execution of a massive network requires a high number of multipliers. Throughout this thesis, I propose and analyze the use of different methods aimed at efficiently implementing reservoir networks. More specifically, chapter 4 and 5 make use of bit-serial stochastic computing ([BH94]) to reduce the circuitry devoted to the arithmetic operations, chapter 6 proposes to constrain the resolution of the weights, which allows replacing multipliers by simpler shift-and-add operations; finally, chapter 7 and 8 are based on the sequential execution of some parts of the network model.

The hardware designs proposed in this thesis have the purpose to extend the utility of the RC approach to those applications where software solutions are not satisfactory, such as real-time systems requiring very large computational power, energy-efficient solutions for mobile/autonomous devices, massively produced price-sensitive consumer electronic products, and fault-tolerant systems for the processing of safety-critical tasks. Each presented design will be described in detail and provided, in most cases, with exemplary VHDL codes. Such codes can be easily generated through an automatic tool translating the desired network configuration onto hardware, which facilitates and accelerates the process of neural hardware design.

Part II.

Methodologies and results

3. Conventional implementation of echo state networks

3.1. Overview

The benefits of the hardware realization of neural network models for certain applications have been described in sec. 2.4. In this chapter, I present the conventional parallel implementation of echo state networks (ESNs) using digital hardware, that is to say, a direct physical realization of the ESN model introduced in the previous chapter using conventional Boolean logic. On the one hand, this description is intended to highlight the challenges of such implementation regarding the high number of required hardware resources. On the other hand, the proposed design serves as a reference that will allow to examine the hardware resource saving of alternative approaches. In addition, general issues regarding the implementation of neural networks in digital circuitry, such as the numerical representation and the nonlinear activation function, are described. The functionality of the present implementation is demonstrated through a chaotic time-series prediction task and the consumed hardware resources are presented.

3.2. Circuit design of the artificial neuron

As introduced in sec. 2.2, the basic neural network processing unit is the artificial neuron. In the case of the commonly used discrete-time artificial neuron (sec. 2.2.2), an activation potential is first calculated as the weighted sum of the neuron inputs, and then passed to the non-linear activation function (typically with sigmoid shape) to get the neuron output. This processing, which involves multiplications, additions and the computation of a non-linear function, is illustrated for a two-input neuron in Fig. 3.1.

The circuit design of such a two-input neuron is depicted in Fig. 3.2. The inputs, here named $in_1(t)$ and $in_2(t)$, are weighted by their corresponding factors v and r. The resulting products p_1 and p_2 are added to provide the activation potential. This is finally passed to a non-linear function block that calculates the final output. The output value is stored in a register so that it can be used by another neuron in the next time step (in case the neurons are arranged in a recurrent network). A number

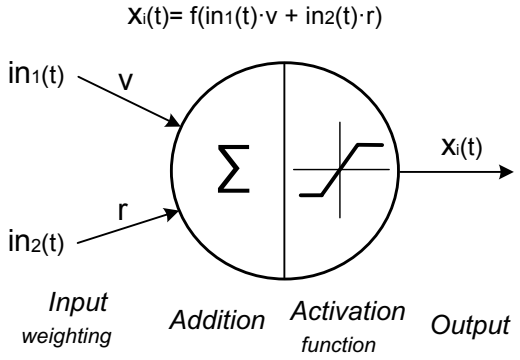


Figure 3.1.: Schematic diagram of a two-input sigmoid neuron.

of n bits is used to represent the inputs, weights and neuron outputs. This section describes in detail how the required operations are implemented in digital circuitry.

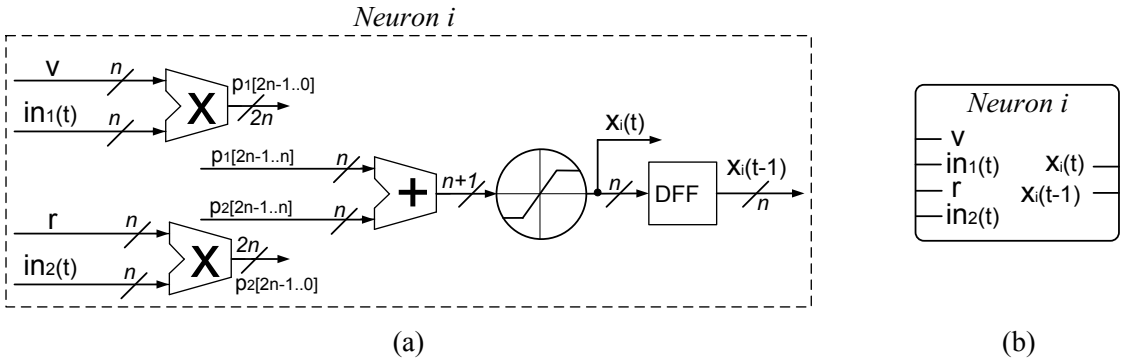


Figure 3.2.: Conventional circuit design of a discrete-time two-input neuron (a) and schematic representation (b).

3.2.1. The format of numerical representation

Throughout this thesis, only fixed-point notation is used. The area requirements of the complex circuits needed to implement floating-point operations are extensive whereas fixed-point arithmetic makes use of simpler integer adders and multipliers.

The fixed-point format is defined as follows:

$$[s]a.\mathrm{b} \tag{3.1}$$

where the s is an optional variable denoting the sign bit that is 0 for positive and 1 for negative values, a is the number of integer bits and b is the number of fractional bits.

If the sign bit is used, the two's complement notation is employed, otherwise the unsigned notation is used. Fig. 3.3 exemplifies the meaning of both numerical representations.

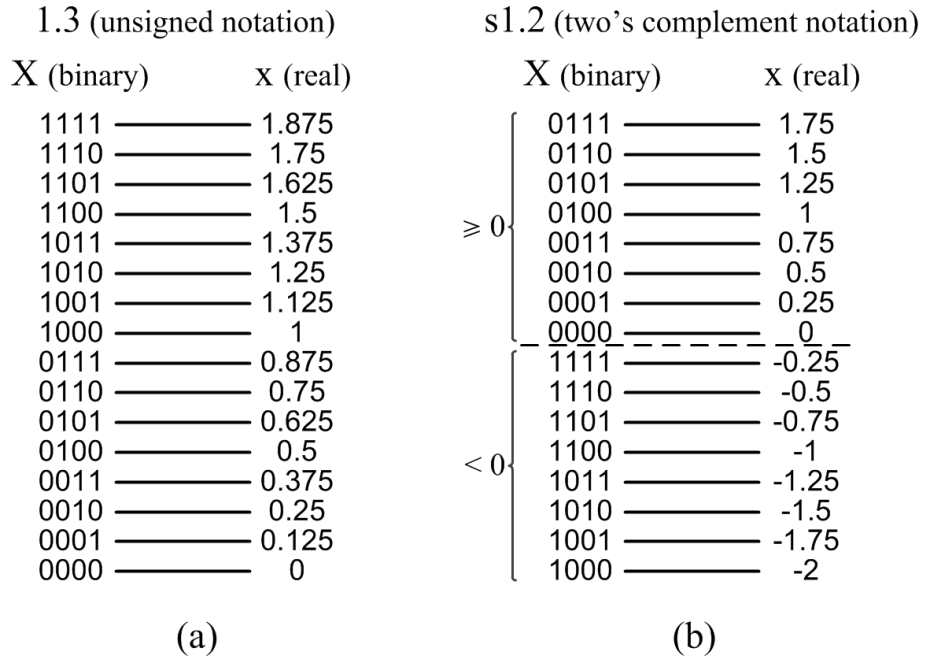


Figure 3.3.: Examples of binary representation (X) of real quantities (x) according to the unsigned (a) and signed (two's complement) notation (b).

For the unsigned notation $a.b$, the minimum x_{min} and maximum x_{max} values that can be represented are

$$\begin{cases} x_{min} = & 0 \\ x_{max} = & 2^a - 2^{-b} \end{cases} \tag{3.2}$$

while for the signed two's complement notation $s\ a.b$ the representation range is given by

$$\begin{cases} x_{min} = & -2^a \\ x_{max} = & 2^a - 2^{-b} \end{cases} \tag{3.3}$$

The maximum truncation error, defined as the absolute difference between a real

number and its corresponding binary representation, for both notations is

$$E_{max} = 2^{-(b+1)} \tag{3.4}$$

The conversion of a binary number X to its corresponding real value x in the case of using the unsigned notation $a.b$ is given by

$$x = \frac{X}{2^b} \tag{3.5}$$

For example, the binary magnitude X = "1101" corresponds to the real value $x = \frac{13}{2^3} = 1.625$ when the unsigned notation 1.3 is employed.

In the two's complement notation, the opposite of a number is obtained by negating all the bits of that number and adding a "1" to the result. For instance, the result of applying this operation to the number "0011" (representing the real value +0.75 using the two's complement notation $s1.2$) is NOT("0011") + "1" = "1100" + "1" = "1101", which represents the value -0.75.

To convert a binary number X to its corresponding real value x in the case of using the signed notation $sa.b$, equation 3.5 can be used if that number is positive (the first bit is a 0). For negative values, first the opposite of the number must be obtained, and then apply the equation to the result. For example, the real value corresponding to "1011" is the opposite of NOT("1011") + "1" = "0101", which corresponds to 1.25 in the $s1.2$ notation, so that the magnitude "1011" corresponds to -1.25.

A practical procedure to convert any binary magnitude to its corresponding real value for the case of the signed notation $s\,a.b$ consists in flipping the first bit of the binary number (changing it from 1 to 0 and vice versa) to obtain the number X', and then the real quantity can be calculated according to the next expression:

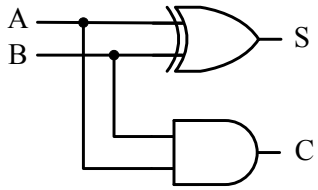
$$x = \frac{X'}{2^b} - 2^a \tag{3.6}$$

3.2.2. Arithmetic operations

3.2.2.1. The half adder and the full adder

The single-bit half adder and the full adder are the basic building blocks used to implement arithmetic circuits such as multiple-bit binary adders and multipliers. The most basic form of the half adder circuit is illustrated in Fig. 3.4. It consists of a XOR gate producing the sum bit S of the inputs (A and B) and an AND gate

producing the carry output bit C. The carry signal represents an overflow into the next digit of a multiple-bit addition. The half adder is useful for adding two single-bit numbers, but for the addition of binary numbers containing several bits it is necessary to account for a carry in bit.

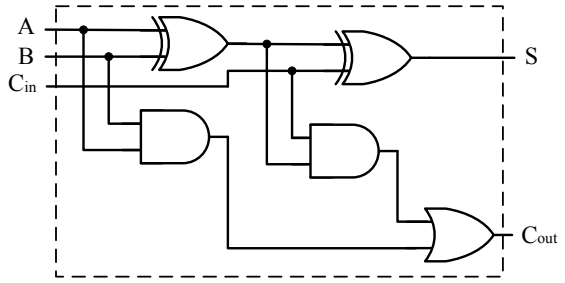


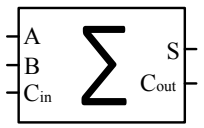
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(a) (b)

Figure 3.4.: Half adder circuit (a) and its true table (b).

The full adder adds three single-bit numbers (A, B and C_{in}) producing the sum S and carry out C_{out} bits. The full adder is built combining two half adders and an OR gate. The circuit of the full adder along with its block diagram symbol and the true table is shown in Fig. 3.5.





A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a) (b) (c)

Figure 3.5.: Full adder circuit (a), block diagram symbol (b) and true table (c).

3.2.2.2. The parallel adder

Multiple full adders can be combined to form a circuit that adds multiple-bit numbers. Each full adder inputs a carry in bit that is the carry out bit of the previous adder. This is called a ripple-carry adder or parallel adder. The parallel adder is illustrated in Fig. 3.6 for the case of two 4-bit input numbers. Note that the parallel adder shown performs the sum of two unsigned integers a[3..0] and b[3..0]. In case of using two's complement notation to represent negative numbers, some additional logic is required around this basic adder.

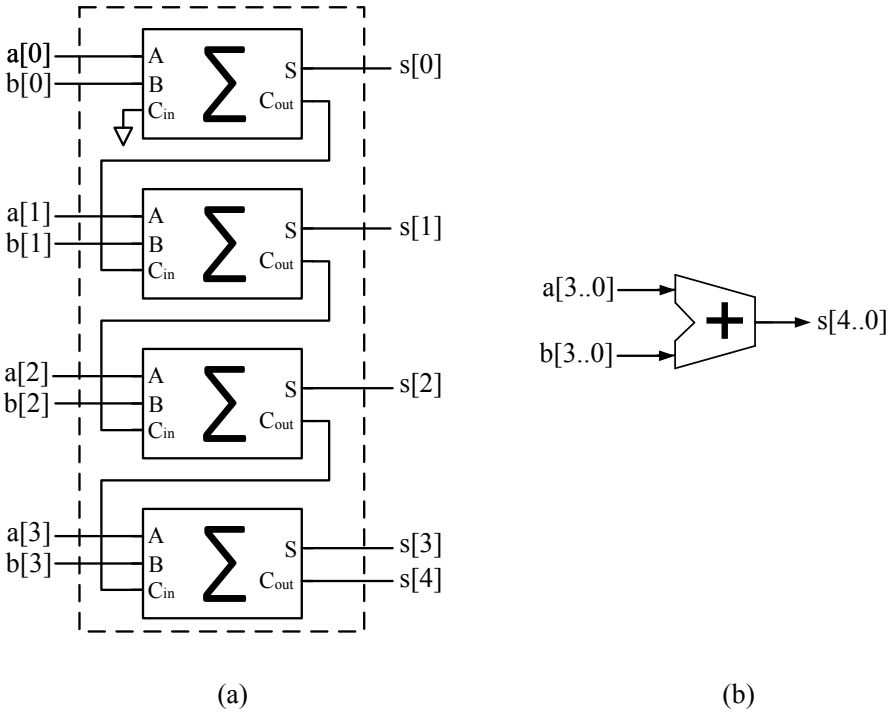


Figure 3.6.: 4-bit unsigned parallel adder: circuit (a) and compact symbol representation (b).

3.2.2.3. The parallel multiplier

The technique used to implement a digital multiplier involves computing a set of partial products, and then summing them together. Therefore, multipliers are built using binary adders. The multiplication process is shown in Fig. 3.7 for two unsigned 4-bit integers a[3..0] anb b[3..0]. As it can be appreciated, the multiplication of two binary numbers comes down to calculating partial products, shifting them left, and finally adding them together. The resulting final unsigned product p[7..0] contains a number of bits equal to the number of bits of the multiplicand plus the number of bits of the multiplier. Fig. 3.8 shows the diagram of an unsigned 4-bit parallel multiplier (all bits are presented simultaneously to the system). This multiplier needs to include some modifications in order to support two's complement notation signed numbers. For more details, see [BW73] and [Beh00].

The hardware description of the parallel adder and the parallel multiplier using VHDL can be expressed using a very simple piece of code by means of the predefined "+" (addition) and "*" (multiplication) operators ([Ped04]). The VHDL code for the addition and multiplication operations is shown in Algorithm 3.1.

Note that either unsigned or signed notation can be employed for the variables just by declaring the proper package of the *ieee* library (*std_logic_unsigned* or

					a[3]	a[2]	a[1]	a[0]	multiplicand
				X	b[3]	b[2]	b[1]	b[0]	multiplier
					b[0]·a[3]	b[0]·a[2]	b[0]·a[1]	b[0]·a[0]	
				b[1]·a[3]	b[1]·a[2]	b[1]·a[1]	b[1]·a[0]	0	
			b[2]·a[3]	b[2]·a[2]	b[2]·a[1]	b[2]·a[0]	0	0	
	+	b[3]·a[3]	b[3]·a[2]	b[3]·a[1]	b[3]·a[0]	0	0	0	
p[7]	p[6]	p[5]	p[4]	p[3]	p[2]	p[1]	p[0]	product	

Figure 3.7.: Multiplication process of two unsigned four-bit binary numbers (a[i] · b[j] denotes the Boolean product of the two bits: a[i] AND b[j]).

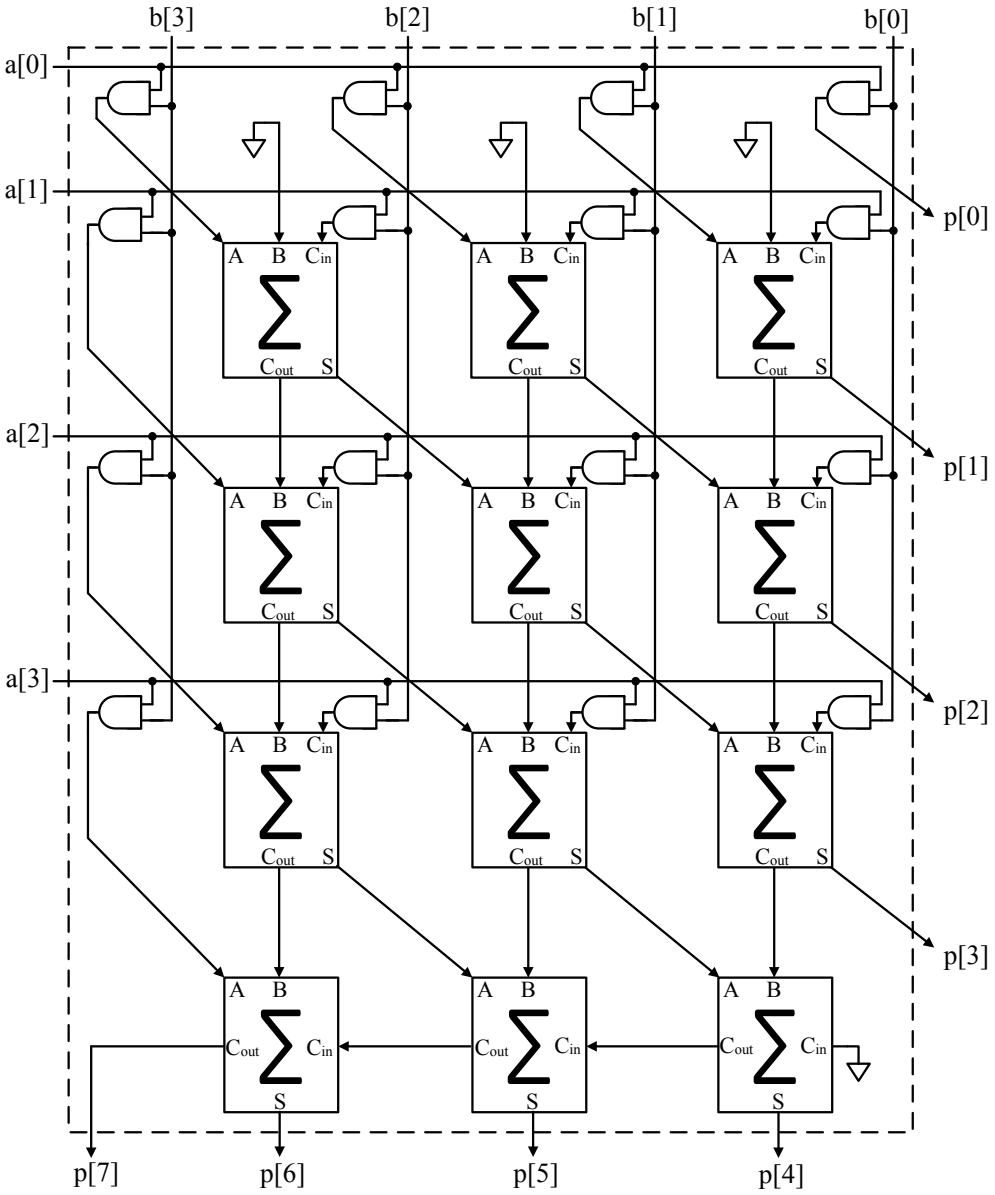
Algorithm 3.1 VHDL code for the addition and multiplication of two 4-bit inputs.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;

ENTITY arithmetic_operations IS
  PORT (a, b : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
    sum: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
    prod: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END ENTITY arithmetic_operations;

ARCHITECTURE behavior OF arithmetic_operations IS
BEGIN
  sum <= a + b;
  prod <= a * b;
END behavior;
```

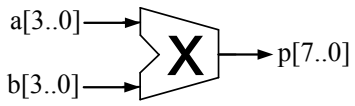
std_logic_signed for using the unsigned and signed data type, respectively) at the beginning of the code.

3.2.3. The activation function

The computation of the nonlinear activation function is a crucial issue for the neural implementation. The hyperbolic tangent sigmoid function introduced in sec. 2.2.2.1 is one of the most commonly used. An exact implementation of the sigmoid function in digital hardware requires many resources due to the involved exponentiation and division operations. Therefore, the use of approximations is necessary. A great deal of research effort has been directed to the development of efficient designs of the sigmoid function and a range of digital hardware approximations with different accuracy can be found in the literature ([ASG91], [Kwa92], [ZVDF96], [BTdC02], [Tom03], [DCFEB13]). The selection of an approximation method with high accuracy increases the hardware while too low accuracy implies poor performance. I



(a)



(b)

Figure 3.8.: 4-bit unsigned parallel multiplier: circuit (a) and block symbol representation (b).

present here some approaches that are used throughout this thesis.

3.2.3.1. Piece-wise linear approximations

The piece-wise linear approach consists of approximately reproducing the desired nonlinear function by means of a number of linear segments. The most simple of these approximations is built using only three segments as described in 3.7 and depicted in Fig. 3.9.

$$f(x) = \begin{cases} -1, & x < -1 \\ x, & -1 \leq x < 1 \\ 1, & x \geq 1 \end{cases} \tag{3.7}$$

The VHDL code implementing this function is shown in Algorithm 3.2. Note that the signed notation is necessary since both the domain and range of the hyperbolic tangent function include negative values. $s1.15$ notation is used for the input of the function assumed to be in the [-2, 2] range while $s0.15$ is employed for the output, which is bound in the [-1, 1] interval.

Algorithm 3.2 VHDL code for the simple 3-segment linear approximation to the hyperbolic tangent.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;

ENTITY f_tanh_approx_3_segments IS
  PORT ( x: IN STD_LOGIC_VECTOR (16 DOWNTO 0); -- s1.15
         f: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)); -- s0.15
END ENTITY f_tanh_approx_3_segments;

ARCHITECTURE function OF f_tanh_approx_3_segments IS
begin
  f <= x"7FFF" when (x > '0' & x"7FFF") else
       x"8000" when (x < '1' & x"8000") else
       x(16) & x(14 DOWNTO 0);
END function;
```

A more accurate approximation was proposed by Alippi and Storti-Gajani in [ASG91]. It is based on the selection of semi-integer breakpoints (that is, $\pm 1/2, \pm 1, \pm 3/2, \pm 2$, ...) and on setting the gradient of the linear segments as power of two numbers ($1/2, 1/4, 1/8, 1/16$, ...) as expressed in 3.8. The resulting approximation along with

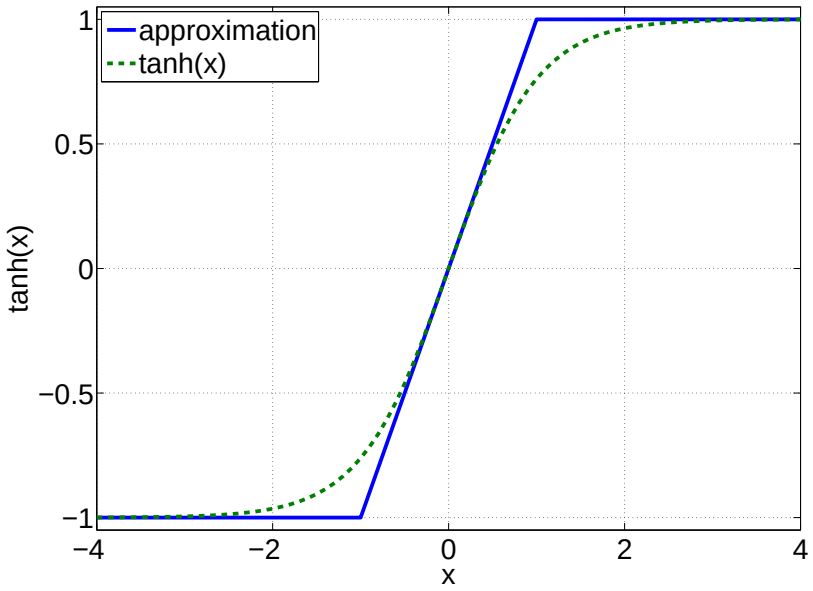


Figure 3.9.: Simple piece-wise linear approximation to the hyperbolic tangent with three segments.

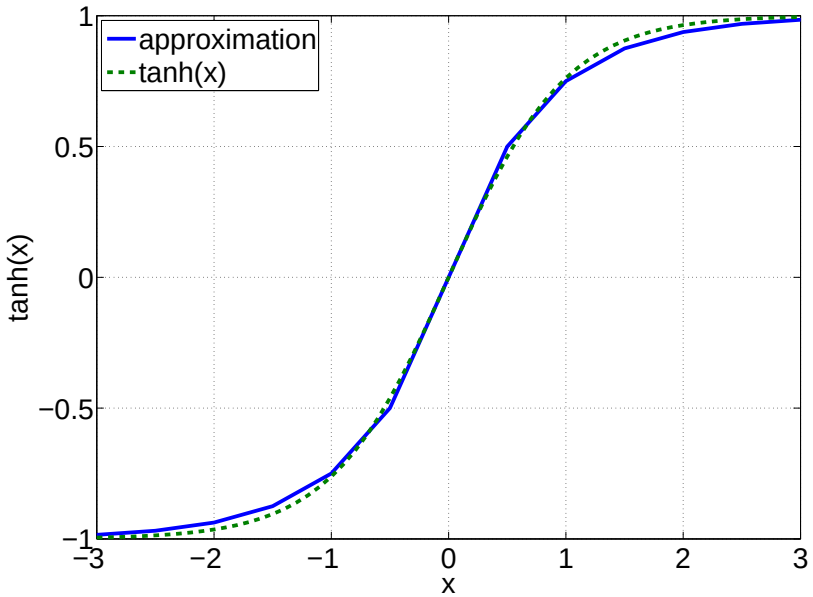


Figure 3.10.: Piece-wise linear approximation to the hyperbolic tangent function proposed by Alippi and Storti-Gajani ([ASG91]).

the exact function is illustrated in Fig. 3.10. It can be observed that this approach allows obtaining low error values.

$$f(x) = \begin{cases} \frac{1}{2}x, & |x| < 1/2 \\ \frac{1}{4}x, & 1/2 \leq |x| < 1 \\ \frac{1}{8}x, & 1 \leq |x| < 3/2 \\ \frac{1}{16}x & 3/2 \leq |x| < 2 \\ \ldots & \ldots \end{cases} \tag{3.8}$$

This approximation can be expressed according to 3.9 for negative values of x.

$$f(x) = \frac{1 + \frac{\widehat{2x}}{2}}{2^{|\lceil 2x \rceil|}} - 1, \; x < 0 \tag{3.9}$$

Where $\lceil 2x \rceil$ stands for the integer part of $2x$ and $\widehat{2x}$ denotes the decimal part defined as follows:

$$\widehat{2x} = 2x + |\lceil 2x \rceil| \tag{3.10}$$

The function value for positive values of x can be calculated using the symmetry property of the hyperbolic tangent function ($\tanh(-x) = -\tanh(x)$):

$$f(x) = -f(-x), \; x > 0 \tag{3.11}$$

This piece-wise linear technique involves low computational complexity since the formula 3.9 does not require multipliers but only bit shifts and additions. Furthermore, it offers high computation speed as only one clock cycle is necessary for the output assessment. Other piece-wise linear approximations, such as the one presented in [BTdC02] based on recursive calculations, allow to improve the accuracy at the cost of a slower computation speed (several clock cycles are required).

3.2.3.2. Combinational approximations

An efficient alternative for implementing a function when its input and output contain few bits is the direct bit-level mapping by means of a purely combinational system. It consists in expressing each output bit in canonical form as a sum of products of the input bits. That is to say, the input bits are firstly multiplied using

AND gates, and then the resulting products are summed by a multiple-input OR gate. Tommiska analyzed such an implementation, named the SIG-sigmoid approximation ([Tom03]), for different resolutions of the input and output quantities. He followed a procedure to minimize the sum-of-products representation for the output bits of the hyperbolic tangent function. Since no arithmetic operations are needed, the area requirements remain low. The SIG-sigmoid implementation concept using a direct bit-level mapping is illustrated in Fig. 3.11.

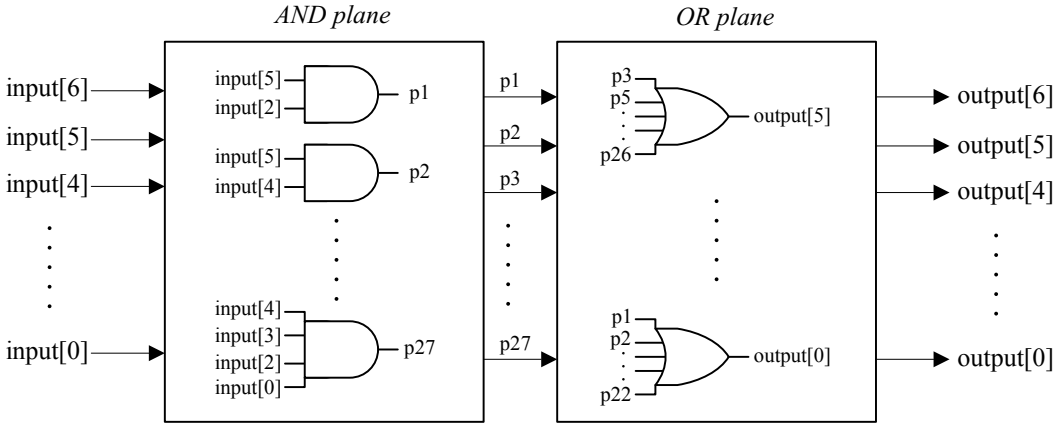
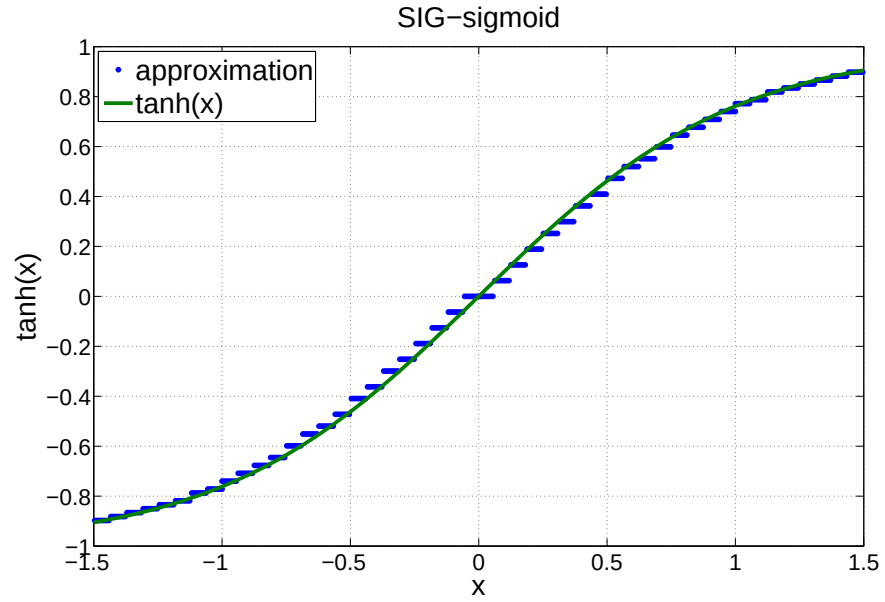


Figure 3.11.: Implementation scheme of the SIG-sigmoid function using direct bit-level mapping proposed by Tommiska ([Tom03]).

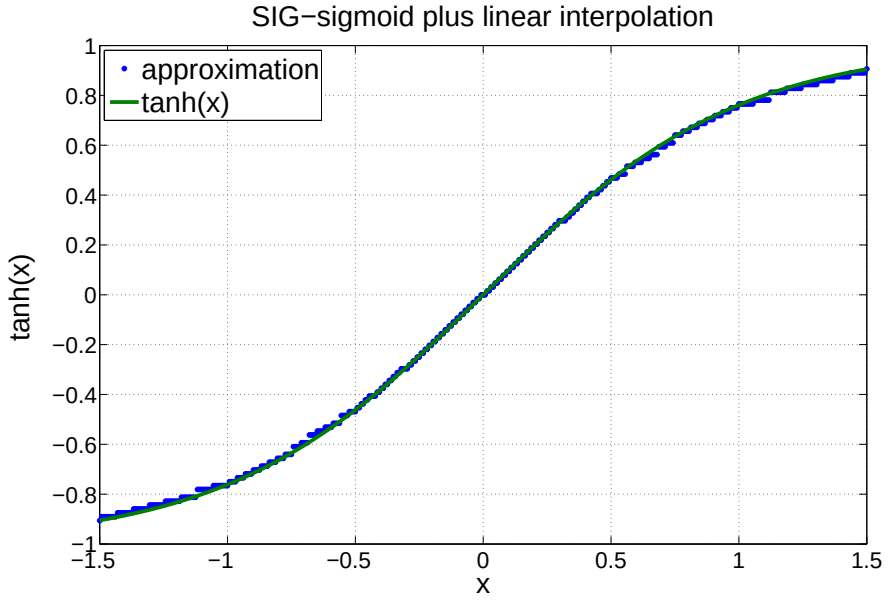
The results reported in [Tom03] show that, compared to the previous approach of Alippi and Storti-Gajani, the SIG-sigmoid approximation presents a significant improvement in terms of accuracy with a small increase in the use of hardware resources (36 logic elements for the former approach and 45 for the latter). The computation time of the sigmoid function for the combinational approach is also a single clock cycle. Experimental measurements of the SIG-sigmoid approximation with a resolution of 7 bits for both input and output values are presented in Fig. 3.12(a). The fixed-point formats used for this design are $s.3.3$ for the input and $s0.6$ for the output. It can be noticed the approximated function exhibits discontinuities due to the limited precision of the implementation. Interpolation can be used to overcome this limitation. In particular, I propose to employ linear interpolation using powers of two for the gradient of the interpolation segments as in the Alippi and Storti-Gajani technique so that the area requirements are kept low. The function with improved precision using interpolation is displayed in Fig. 3.12(b). In this case, the input signal uses two additional fractional bits so that it is represented by the format $s3.5$.

3.2.3.3. Other approaches

Other techniques include the implementation of the sigmoid function as a piece-wise second order approximation ([ZVDF96]) according to 3.12. The main disadvantage



(a)



(b)

Figure 3.12.: Experimental measurements of the SIG-sigmoid function (a) and of the improved approximation with linear interpolation (b).

of this approach is the requirement of multipliers, which considerably increases the hardware resource utilization.

$$f(x) = c_0 + c_1 x + c_2 x^2 \tag{3.12}$$

The sigmoid function can also be implemented as a lookup table. In this case, the accuracy of the function depends on the number of approximated values stored in the memory. Nevertheless, although the available internal memory of FPGA devices has increased with the enhancement of FPGA technology, self-contained neurons are more desirable since the limited memory of the devices is often necessary for other purposes.

3.3. Implementation of the ESN architecture

The implementation of the complete set of individual neurons forming the echo state network is carried out according to the simple cycle reservoir (SCR) topology introduced in sec. 2.3.1.2. The SCR simplified topology is illustrated in Fig. 3.13. The greatest benefit of this structure is the reduced number of connections (that is, synapse multiplications) compared to the classical ESN architecture with random connections, which makes it more appropriate for a hardware implementation. In addition, the digital design of such an SCR network can be easily automated for any number of units since all neurons have the same structure (one connection input from a neighboring neuron and a second one from the input layer) and it is independent on the size of the system.

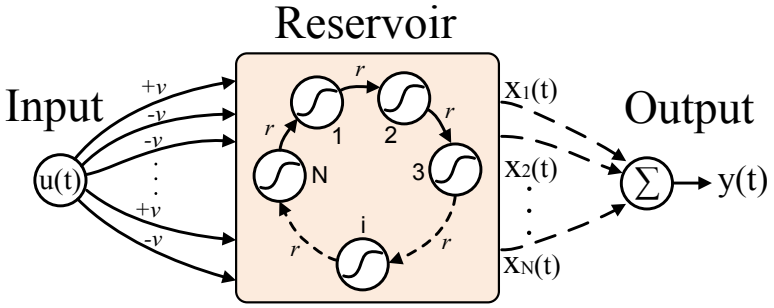


Figure 3.13.: Simple cycle reservoir (SCR) topology. Units are organized in a cycle.

Fig. 3.14 shows how the individual neuron blocks are organized to form the ESN with cyclic architecture of Fig. 3.13. The proposed conventional implementation of the reservoir network has been realized in a medium-sized FPGA (Altera Cyclone IV EP4CE115F297C7N) using 16-bit precision ($n = 16$) for the signals. The simple

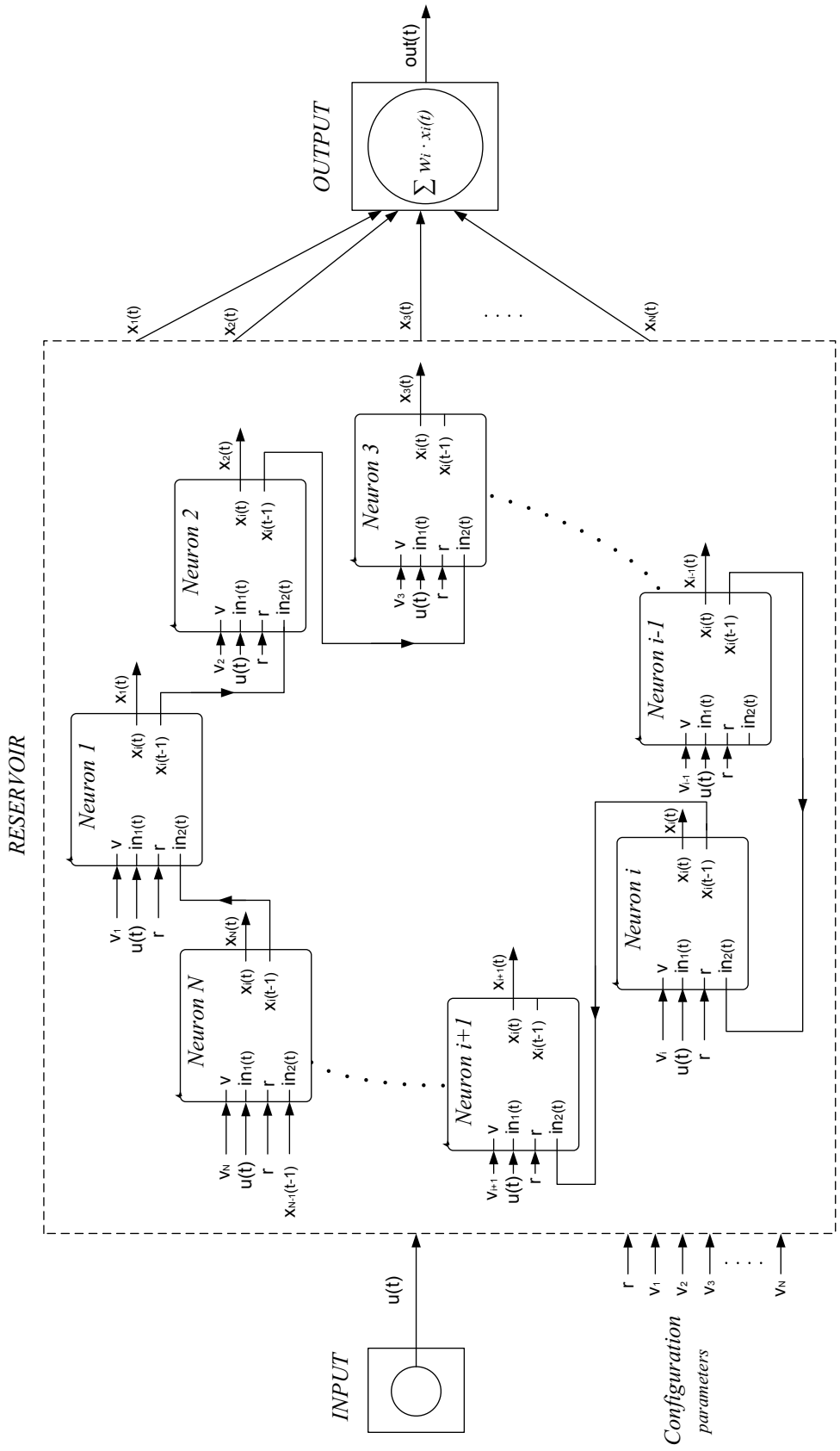


Figure 3.14.: Conventional implementation of the ESN with cyclic topology composed of N two-input neurons.

piece-wise linear approximation with three segments has been used for the sigmoid activation function. The fixed-point format $s0.15$ has been used for the neuron inputs, weights and outputs since their values are limited to the $[-1, 1]$ range.

Note that the output of each of the reservoir neurons is computed as a nonlinear transformation of the weighted addition of its inputs (according to Fig. 3.1 and Fig. 3.2). An additional constant bias term could be included although it has been observed to have little impact on the network's performance. Similarly, a bias term might also be added in the linear output neuron. On the other hand, a direct connection from the input unit to the output layer could be included as expressed in equation 2.12, which may increase the system's accuracy in certain tasks.

A software program (using the MATLAB language) has been developed which allows the SCR network to be automatically exported to a VHDL hardware description. The program generates the VHDL code (as a text file) for the reservoir with any desired number of neurons and weight configuration. Finally, the resulting VHDL code can be synthesized to an actual hardware implementation. An example of the VHDL code describing the SCR hardware realization for 50 neurons is shown in sec. A.1 of Appendix A.

The process followed to implement the proposed reservoir network design in the FPGA is illustrated in Fig. 3.15. Once the VHDL code of the ESN with the desired configuration has been generated (as a text file) through the MATLAB script, the Quartus II software ([web17c]) is employed to compile and synthesize the specified circuit in the FPGA. Finally, the FPGA-based network may be tested capturing the experimental results through the Quartus' logic analyzer.

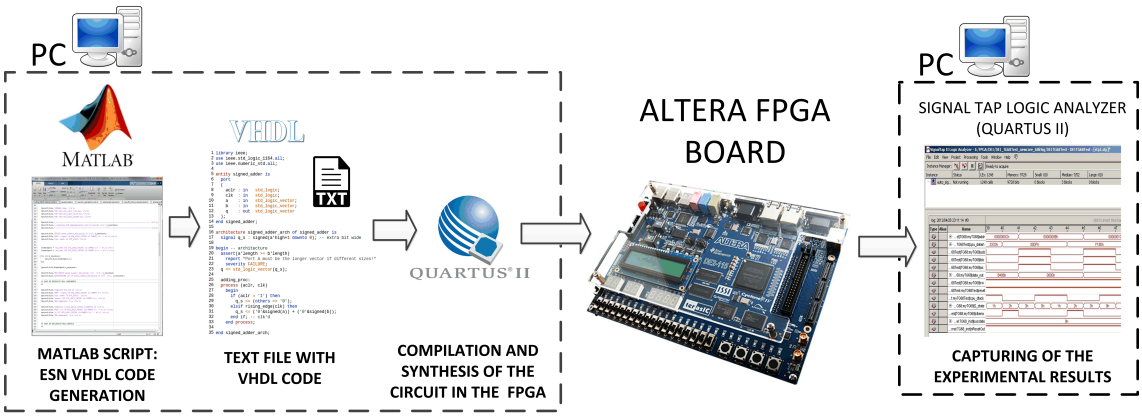


Figure 3.15.: Implementation process of the ESN design into the FPGA and eventual measuring of the experimental results.

The consumed hardware resources of the implementation for different values of the number of neurons N are presented in Tab. 3.1. The unit usually employed to compare the area requirements of FPGA-based designs is the logic element (LE), although it may vary depending on the manufacturer. Contrary to ASICs, where

the entire architectural structure is defined in the design process, FPGAs consist of a regular structure of reprogrammable basic functional elements (the logic elements) that are connected by hierarchical routing. A description of the internal structure of a LE in a modern Altera FPGA device can be found in [Tom03].

N (neurons)	50	100	150	200
Logic elements	19147 (16.7%)	37631 (32.9%)	56073 (49%)	74544 (65.1%)
Registers	800 (0.7%)	1600 (1.4%)	2400 (2.1%)	3200 (2.8%)

Table 3.1.: Hardware resource utilization of the Altera Cyclone IV FPGA for the proposed ESN conventional implementation.

3.4. Experimental results

The performance of the implementation has been tested for the Santa Fe time-series prediction task, a widely used benchmark in the RC literature ([RT11]). As introduced in sec. 2.3.4.2, it consists in the one-step ahead prediction of an experimental recording of the output power of a far-infrared laser operating in a chaotic regime. A total of 4000 samples of the original laser data set ([WG15]) are employed; the first 2000 for training, the next 1000 for validation, and the remaining 1000 for testing.

To perform the measurements of the proposed conventional ESN circuitry, an internal RAM memory supplies the input signal to the reservoir network every time step (a single clock cycle). The resulting network outputs (individual neuron states) are monitored using a logic analyzer.

A numerical model of the reservoir hardware (also programmed using MATLAB), which truncates the resolution of the variables according to the digital design, is employed for training the system (following the standard procedure for ESNs, sec. 2.3.1.3). Such numerical model has been observed to exactly reproduce the FPGA results. That is, a perfect agreement (with zero error) was found between the experimental and the simulated neuron states. This software is also used to determine the configuration parameters providing the best performance of the system for the validation set. Finally, the hardware realization is set up with the optimum weights and analyzed with the test set. Fig. 3.16 shows the network performance (prediction error) for the validation set as a function of the configuration parameters r and v.

The network's output layer is computed by software as a linear combination of the experimental neuron states, which are read with a 12-bit precision. The experimental prediction when using 200 neurons is shown for a fragment of the test set in Fig. 3.17. The test performance for several sizes of the reservoir is displayed in Fig. 3.18. These results cannot be directly compared to those of previous conventional software implementations (e.g., [RT11]) since the present hardware realization

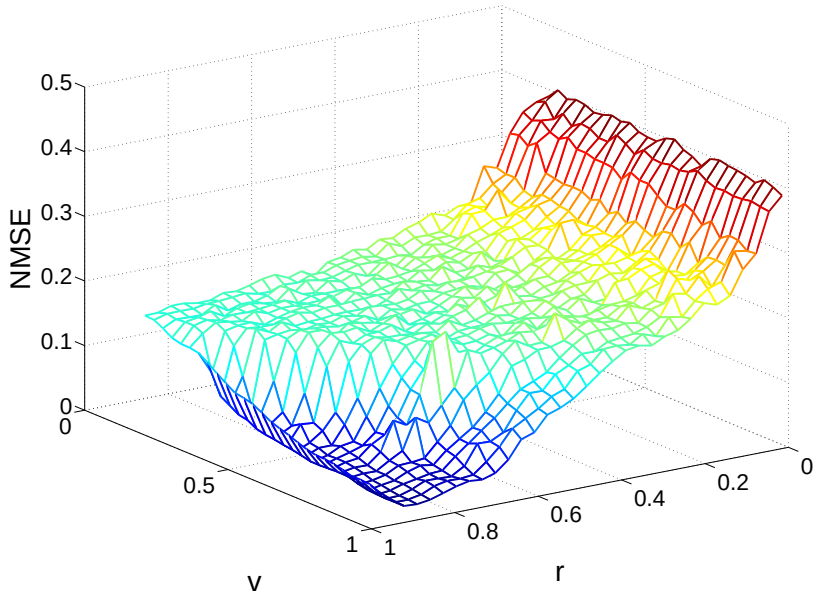


Figure 3.16.: Performance (NMSE) of the conventional ESN implementation with 200 neurons in the Santa Fe time-series prediction task as a function of the weight parameters r and v.

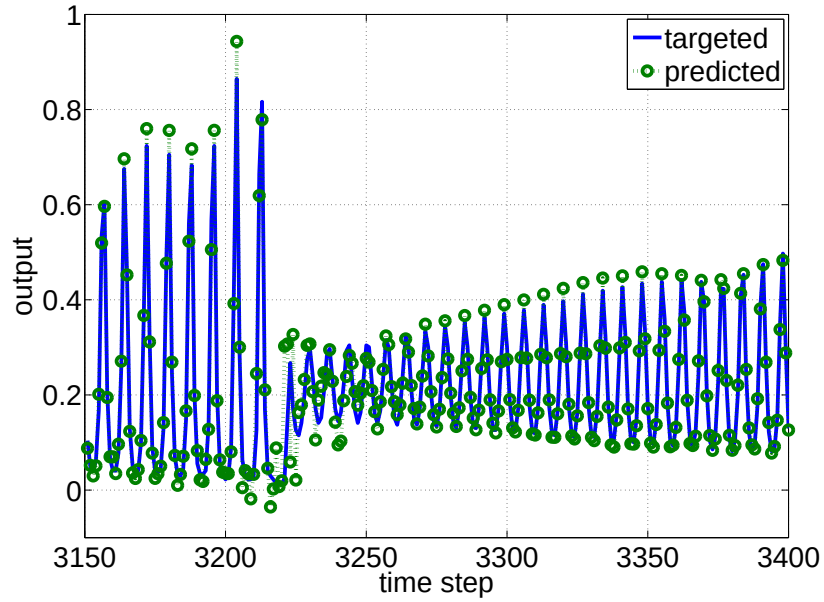


Figure 3.17.: Fragment of the Santa Fe laser time-series: original values and one-step ahead predictions performed by the proposed conventional ESN implementation with 200 neurons.

employs a simplified activation function, which implies a lower system's performance than the hyperbolic tangent. Nevertheless, it has been observed that an equivalent software implementation using the same three-segment sigmoid approximation employed in hardware (although with full-resolution) provides practically the same results (with only a maximum difference of 10^{-4} in the NMSE values due to the limited resolution of the hardware).

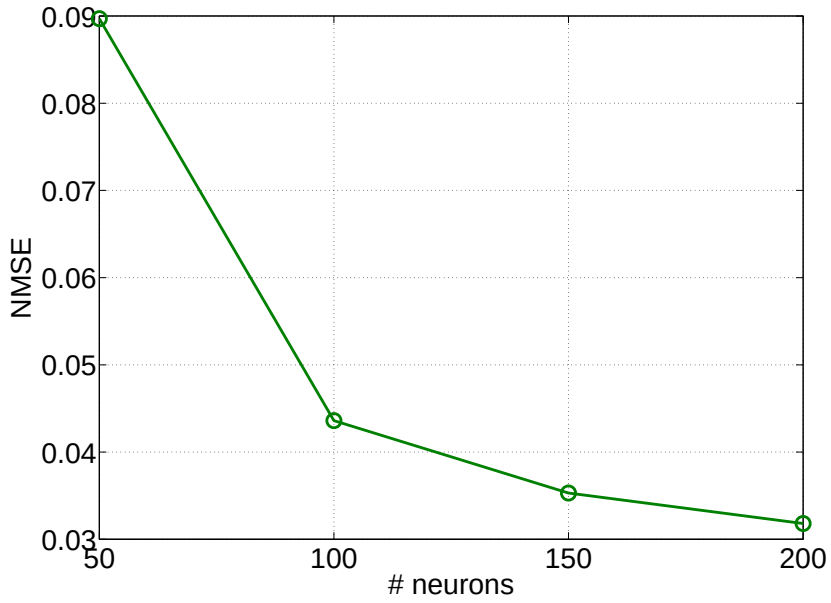


Figure 3.18.: Performance (NMSE) in the time-series prediction task as a function of the reservoir size for the proposed conventional hardware ESN.

3.5. Discussion

A conventional implementation of an echo state network (ESN) in digital hardware has been described and analyzed. The presented design allows the parallel computation of all neuron outputs in a single clock cycle (20ns for a typical clock frequency of 50MHz) and provides results with the expected accuracy (NMSE values). More specifically, I have focused on the implementation of the ESN with a simple cyclic topology (SCR), which represents a considerable improvement regarding the number of connections compared to a standard random ESN. Nevertheless, despite being completely functional, the conventional parallel design consumes many hardware resources, even in the case of using a rough approximation for the sigmoid activation function, mainly due to the large area requirement of the binary products. For instance, the ESN realization with 200 units employs about 74000 logic elements, which represents most of the area in a medium-cost FPGA. Therefore,

this implementation is not appropriate for low-cost and low-power devices. Alternative implementation approaches are necessary to reduce the network's chip area requirements, especially those related to the multiplication operations. This would enable fitting a greater number of concurrent neurons in a limited-size device, which is crucial to better exploit the parallelism of neural networks.

4. Stochastic echo state networks

4.1. Overview

Hardware implementations of artificial neural networks (ANNs) allow to exploit the inherent parallelism of these systems. Nevertheless, they require a large amount of resources in terms of area and power dissipation. The results presented in chapter 3 exemplify the necessity of investigating alternative ways of realizing reservoir computing (RC) models (in particular, echo state networks, ESNs) in digital hardware using few resources. Here, we show a new approach to implement ESNs with digital gates. The proposed method is based on the use of probabilistic computing concepts to reduce the hardware required to perform the arithmetic operations. The result is the development of a highly functional system with low hardware resources. The presented methodology is used to implement massive reservoir networks and applied to chaotic time-series forecasting. The main aim of this chapter is to analyze the practicality of the stochastic computing technique (SC) to build ESNs. I discuss the advantages and limitations of the proposed approach compared to the binary logic conventional implementation of chapter 3 examining the hardware resource saving.

4.2. Stochastic computing

4.2.1. Basic concept

Stochastic computing (SC) was introduced in the 1960s ([PAE67], [Gai67], [Rib67]) as a low-cost alternative to conventional binary computing. It enables very low-cost implementations of arithmetic operations using standard logic elements. The basic concept of SC is that information is represented by sequences of random bits. More specifically, the information is carried by the frequency of ones in a sequence as illustrated in Fig. 4.1. In this example, the bit-stream (stochastic signal) contains 25% of ones and 75% of zeros, which expresses a represented number $p = 0.25$. Both the random bit-streams $p(t)$ and the probability they represent p are usually denoted with the same letter and referred to as stochastic numbers. It is worth noting that the representation is not unique; for example, different sequences such as {01000100} and {00100001}, represent the same information (a value related to the probability of the pulsed signal being in the high level, in this case $p = 1/4$).

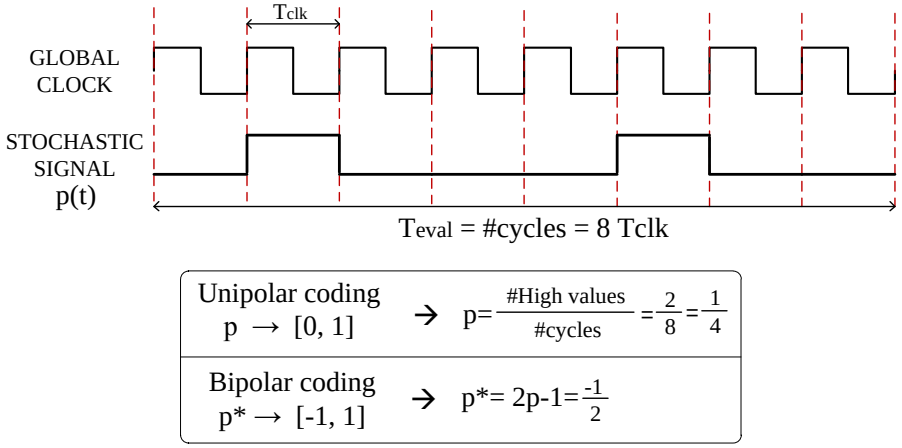


Figure 4.1.: Basic concept of the stochastic codification. Information is coded as the probability p of the pulsed signal being in the high level.

The random bit sequences (stochastic bit-streams) follow probabilistic laws when they are evaluated through logic gates. The probability-based coding of SC provides a natural way of operating with analog quantities (since probabilities are defined between 0 and 1) using digital circuitry. For instance, the AND gate provides an output signal with a switching probability equal to the product of its inputs (that is, the collision probability between signals). The product operation of two independent stochastic signals using an AND gate is illustrated in Fig. 4.2. As can be seen, the inputs to the AND gate represent the numbers $2/8$ and $4/8$ and we correctly get an output corresponding to $2/8 \cdot 4/8 = 1/8$. However, a different possible SC representation of the input numbers may not lead to the same exact result but to an approximation to the product. In SC, a probabilistic error is always present and long bit sequences need to be evaluated to obtain accurate results. In addition, the stochastic bit-streams must be suitably uncorrelated or independent so that the operations are performed correctly.

The major benefit of the SC technique is that it considerably reduces the hardware area compared to the traditional digital implementations. In the case of the multiplication operation, for example, a high number of logic elements is required for a conventional parallel multiplication (illustrated in Fig. 3.8 for input variables with a precision of 4 bits) while a single logic gate is necessary when using the SC approach. The resource saving of SC comes at the cost of the mentioned lower accuracy and longer evaluation time. That is to say, SC needs a high number of clock cycles to assess the result of an operation that can be performed in a single clock cycle using conventional binary computing.

Another appealing feature of SC implementations is a high degree of error tolerance. Stochastic circuits tolerate environmental errors that seriously affect the behavior of conventional circuits. A single bit flip (especially of a high significance bit) causes

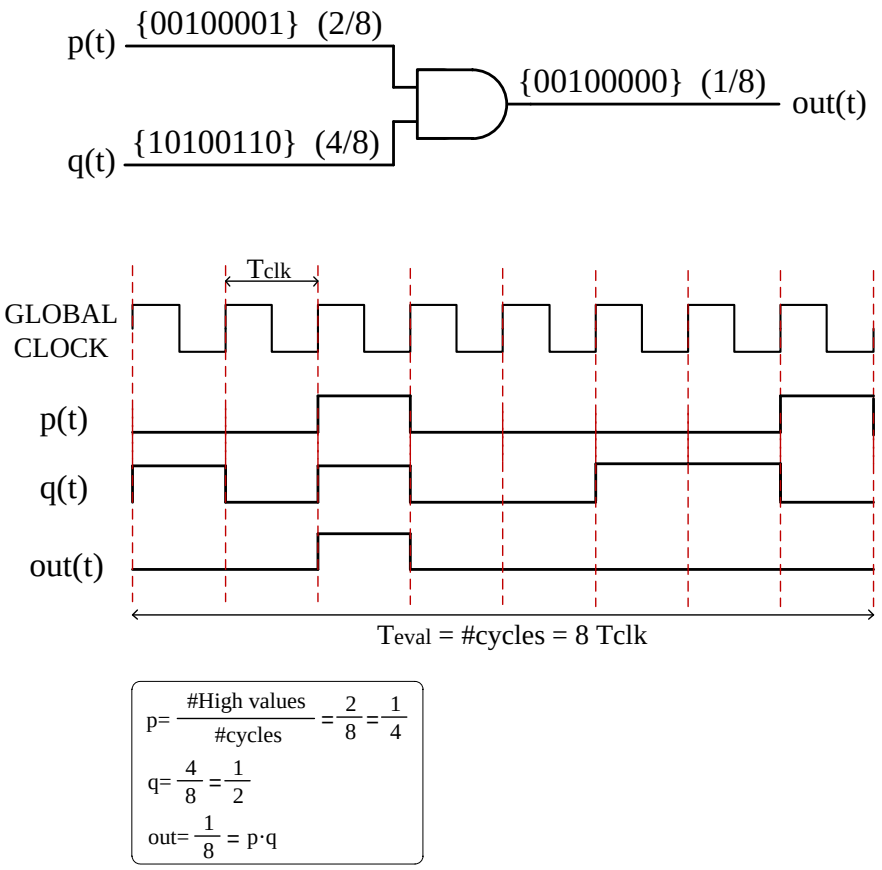


Figure 4.2.: Product operation of two stochastic signals with switching activities $p = 0.25$ and $q = 0.5$ performed by means of an AND gate.

a huge error on a binary circuit, but flipping a few bits in a long bit-stream has little effect in the value of the stochastic number represented. Therefore, SC can be interesting for applications like spacecraft electronics which operate under radiation-induced error conditions.

Other benefits of SC over conventional computing are the capability to trade off the computation time and accuracy without hardware changes and the simplicity of communications, which only require one wire per signal.

Despite the minimal hardware resources required to implement mathematical operations using stochastic bit-streams, the computations in the SC framework require additional circuitry to convert the binary numbers into stochastic signals and vice versa. In particular, large numbers of stochastic number generators (SNGs) are usually necessary in order to operate with uncorrelated bit-streams. Since SNGs can account for a significant portion of the circuit ([QLR+11]), the reduction of the SNG count is an important challenge for SC.

Two different codifications are mainly used in SC to relate the frequency of ones in a sequence of bits and the corresponding represented information (indicated in

Fig. 4.1): the unipolar coding and the bipolar one. The unipolar mapping directly assigns the probability of observing a '1' in a sequence of bits $p(t_i)$ (denoted as $P_p = Probability(p(t_i) = \text{"1"})$) to the represented value p, so that $p = P_p$, $(0 \leq p \leq 1)$. The inconvenience of this codification is that negative numbers cannot be represented directly since probabilities are positively defined. In bipolar coding, this issue is overcome by defining the bipolar value p^* as $p^* = (2 \cdot P_p - 1)$, $(-1 \leq p^* \leq 1)$. For the SC-based neural network implementation presented in this chapter, the bipolar format is used.

SC can also deal with values out of the $[-1, 1]$ range using the Extended Stochastic Logic (ESL) codification ([CMO+16]). This encoding extends the representation range to any real number using the ratio of two bipolar-encoded pulsed signals and presents a particularly high tolerance to faults ([CAM+15]). Its major shortcoming is a high conversion error when using high values of the represented data.

4.2.2. Basic circuitry

The SC approach represents variables by statistical averages of random pulse streams. The data to be processed (binary values) need to be converted to pulsed signals before entering the probabilistic computing system, and the resulting pulsed signals from the stochastic computations must be converted again to their equivalent binary values. Therefore, any stochastic computing system consists of three basic stages as illustrated in Fig. 4.3: binary to pulse conversion, SC-based operations and pulse to binary conversion.

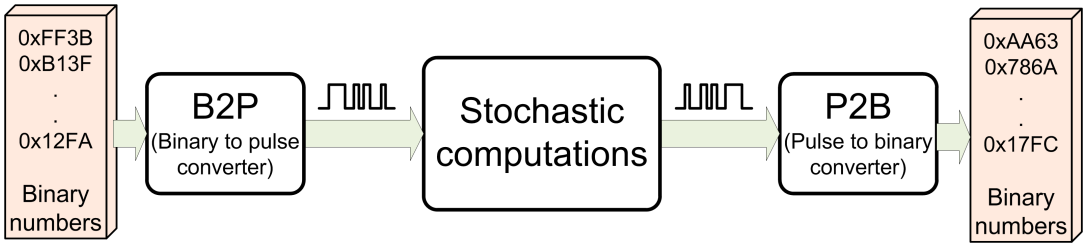


Figure 4.3.: Basic stages of a stochastic computing system.

The structure of the stochastic circuit to perform a particular mathematical operation depends on the codification used (unipolar or bipolar). Conversely, the blocks to convert data from the binary to the stochastic domain and vice versa are common to both codifications.

4.2.2.1. Stochastic computing operations

It has already been introduced that the product is performed by an AND gate when the unipolar coding is used. This is due to the fact that the pulsed output of the

AND gate (*out(t)*) will only have a high level when both input pulse streams *p(t)* and *q(t)* have a high level, and therefore the output probability equals the product of the input probabilities ($out = p \cdot q$). The addition operation is less intuitive. It cannot be directly implemented with an OR gate since this implements the operation $p + q - p \cdot q$, which is only a good approximation to the addition for low probability values. The scaled addition or mean value of two switching signals ($out = 1/2(p+q)$) is implemented using a multiplexer and a binary counter. The counter supplies the selection signal to the multiplexer so that the output signal changes alternately between *p(t)* and *q(t)*. A NOT gate converts the probability p at the input to the complementary $1 - p$ at the output.

The stochastic circuits may implement very different operations when the bipolar codification is used. For instance, when two switching signals representing the bipolar values $p^*$and q^*operated by an AND gate result in a pulsed signal with an encoded value $out^* = 1/2(p^* + q^* + p^* \cdot q^* - 1)$. This is shown in 4.3, where the variable change from the bipolar to the unipolar coding (4.2) is employed.

$$out(t) = p(t)\ AND\ q(t) \rightarrow out = p \cdot q \tag{4.1}$$

$$\begin{cases} p^* &= 2p - 1 \\ q^* &= 2q - 1 \\ out^* &= 2out - 1 \end{cases} \longrightarrow \begin{cases} p &= \frac{p^*+1}{2} \\ q &= \frac{q^*+1}{2} \\ out &= \frac{out^*+1}{2} \end{cases} \tag{4.2}$$

$$out = p{\cdot}q = \frac{p^* + 1}{2} \cdot \frac{q^* + 1}{2} = 1/4(p^*{\cdot}q^*{+}p^*{+}q^*{+}1) \Longrightarrow out^* = 1/2(p^*{+}q^*{+}p^*{\cdot}q^*{-}1) \tag{4.3}$$

Similarly, an OR gate performs the function $out^* = 1/2(p^* + q^* - p^* \cdot q^* + 1)$, and the product can be simply implemented with a XNOR gate in the bipolar codification ($out^* = p^* \cdot q^*$). A NOT gate provides the negation of the encoded input value so that $out^* = -p^*$, and the scaled summation is implemented with a multiplexer just as in the unipolar case. The basic arithmetic circuits for both codifications are depicted in Fig. 4.4.

It is worth noting that the described circuits perform the desired operations provided that the input bit-stream signals are uncorrelated. The use of correlated signals, however, extends the range of possible operations ([AH13a], [MCO+15]). To cite an example, the function performing the absolute value of the input subtraction ($out = |p - q|$) can be implemented for the unipolar coding with a single XOR gate where the inputs are correlated bit-streams as described in [MCO+15].

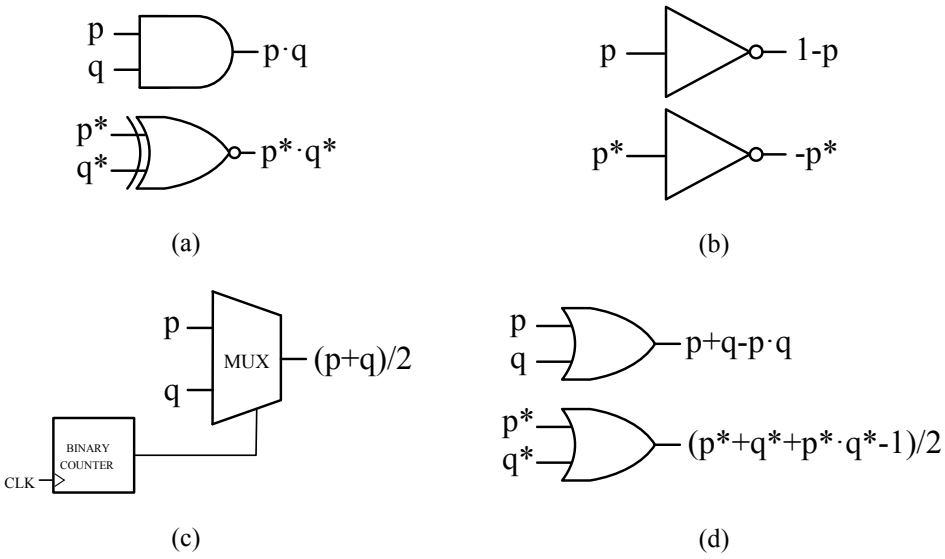


Figure 4.4.: Stochastic arithmetic circuits. (a) Unipolar and bipolar multipliers. (b) Unipolar complementary operation and bipolar negation. (c) Adder used for both unipolar and bipolar notation. (d) Function performed by the OR gate according to the unipolar and bipolar format.

More complex functions, such as the division, square root, exponentiation or the hyperbolic tangent function can also be implemented in the SC framework ([BC01b]). An extensive description and deduction of SC-based operations using the different codifications (including the extended ones) is given in ([CG12]).

The VHDL hardware description of a SC system performing the multiplication operation of two 16-bit input numbers (*in1* and *in2*) is presented in Algorithm 4.1. The bipolar codification is used, and therefore the stochastic circuit consists of a single XNOR gate. The representation format of numerical quantities as binary numbers according to the desired codification is presented in sec. 4.2.2.2 along with a description of the conversion blocks employed to convert binary magnitudes into stochastic bit-streams and vice versa. Note that different seed values are used for each binary-to-stochastic conversion to produce uncorrelated stochastic signals. An evaluation time equal to $T_{eval} = 2^{16} - 1 = 65535$ clock cycles is employed so that the resulting binary output (*out_multiplication*) is updated every T_{eval} period.

4.2.2.2. Data conversion

A stochastic computing system requires converting any real number (either in the unipolar [0, 1] or in the bipolar [−1, 1] range) represented by a binary magnitude P to its equivalent stochastic bit-stream with probability p before the probabilistic computations. Similarly, the resulting pulsed signals must be finally converted

Algorithm 4.1 VHDL code for a SC system performing a multiplication using the bipolar codification.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY SC_system IS
  PORT (clk, rst, en: IN STD_LOGIC;
    in1: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    in2: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    out_multiplication: OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
END SC_system;

ARCHITECTURE arch OF SC_system IS
SIGNAL seed1, seed2: STD_LOGIC_VECTOR (25 DOWNTO 0);
SIGNAL eval_period, out_multip_bin: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL out_multip_stoch: STD_LOGIC;

COMPONENT b2p_16bits
  PORT (binary_in: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    clk,rst,en: IN STD_LOGIC;
    seed: IN STD_LOGIC_VECTOR (25 DOWNTO 0);
    stoch_out: OUT STD_LOGIC);
END COMPONENT;

COMPONENT p2b_16bits
  PORT (evaluation_period: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    stoch_in : IN STD_LOGIC;
    clk, rst, en: IN STD_LOGIC;
    binary_out: OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
END COMPONENT;

BEGIN
seed1(25 DOWNTO 0)<="00000100000000100101010010";
seed2(25 DOWNTO 0)<="00110101000001010110110110";
eval_period (15 DOWNTO 0) <= x"FFFF";

-- binary to stochastic conversions
b2p_1: b2p_16bits
  PORT MAP(in1, clk, rst, en, seed1, in1_stoch);
b2p_2: b2p_16bits
  PORT MAP(in2, clk, rst, en, seed2, in2_stoch);

-- stochastic computation
out_multip_stoch <= (in1_stoch XNOR in2_stoch);

-- stochastic to binary conversion
p2b_0: p2b_16bits
  PORT MAP(eval_period, out_multip_stoch, clk, rst, en, out_multip_bin);

out_multiplication <= out_multip_bin;

END arch;
```

into their equivalent binary values. The binary magnitude P representing the real number p_{real} is obtained by the simple formula 4.4:

$$P = \lfloor p_{real} \cdot (2^n - 1) \rceil \tag{4.4}$$

where n is the number of bits employed to represent the real value and $\lfloor \cdot \rceil$ denotes the nearest integer value. For example, the real value $p_{real} = 0.5$ is represented by the binary magnitude $P = 32767$ when using a 16-bit resolution ($n = 16$). In the case of working with negative values (bipolar codification range), the numbers must be first normalized to the unipolar range. For instance, the real number $p^*_{real} = -0.5$ is represented by the value $p_{real} = (p^*_{real} + 1)/2 = 0.25$ in the unipolar range, which corresponds to the 16-bit binary magnitude $P = 16383$.

Note that this notation to represent real values with binary numbers is slightly different from the conventional binary one. Remind equation 3.5 ($p_{real} = P/2^n$) for converting a binary number to its corresponding real value in the unsigned notation (assuming a number of n bits to represent the fractional part of the value and no integer bits), which employs the term 2^n instead of $(2^n - 1)$ as used in equation 4.4. The representation range for the unipolar binary notation in the SC framework is $[0, 1]$ whereas for the conventional unsigned notation , the range does not include the value 1: $[0, 1 - 2^{-n}] = [0, 1)$. Accordingly, the range for the bipolar binary notation used in SC is $[-1, 1]$ while for the conventional signed notation it is $[-1, 1 - 2^{-n}] = [-1, 1)$. Examples of the binary representation of real quantities in SC compared to the conventional binary notation are shown in Fig. 4.5 and Fig. 4.6 for the unipolar and bipolar codification respectively.

Although the stochastic and conventional binary codifications are not strictly equivalent, in practice, a binary value resulting from stochastic computations can be directly used for further processing using conventional binary circuitry if it is necessary. In the case of the bipolar codification, the first bit of the binary magnitude resulting from stochastic computations must be flipped so that it is converted to the two's complement notation (see Fig. 4.6). The error made in this approximate conversion (between the stochastic and conventional codifications) is negligible compared to the probabilistic error of stochastic computations and considering the usually high number of bits employed to represent a quantity in SC.

Stochastic to binary conversion. Since the probability value p carried by the stochastic pulsed signal is related to the number of 1s contained in the bit-stream, it suffices to count these 1s in order to extract p. The pulsed to binary conversion block (P2B) is illustrated in Fig. 4.7(a). It consists of two n-bit counters and an n-bit register. The first counter evaluates the number of high values (1s) provided by the stochastic signal throughout N_C clock cycles. The second counter is employed to reset the first counter and to load the register with a new value every N_C clock

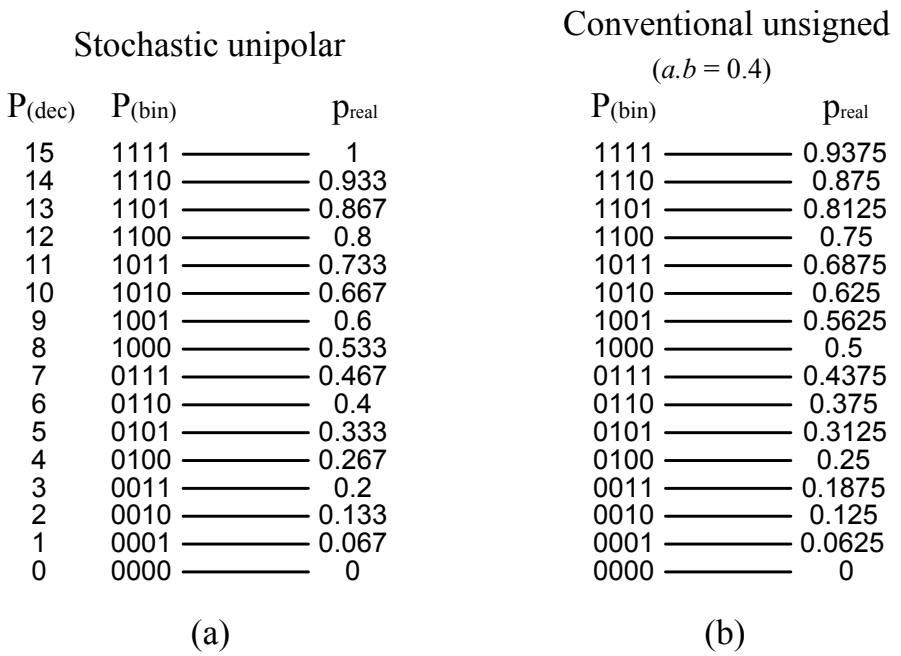


Figure 4.5.: Binary representation of 4-bit numbers according to the stochastic unipolar codification (a) and to the conventional unsigned notation (b).

cycles. Therefore, the output of the P2B block is an n-bit number that changes every N_C cycles. Usually, the number of cycles to evaluate the stochastic signal is chosen to be the maximum allowed by the counter size, that is, $N_C = 2^n - 1$, so that the P2B output (P) directly provides an approximation to the probability p in binary format. The evaluation time of the stochastic computing system is $T_{eval} = N_C \cdot T_{clk}$, where T_{clk} denotes the system's clock period.

Note that in SC, it is necessary to evaluate a bit-stream of length $2^n - 1$ to represent a binary number with a precision of n bits. Therefore, increasing the precision in one bit requires doubling the bit-stream length, which implies an exponential dependence of the evaluation time with precision.

The low accuracy issue in SC due to the fluctuations inherent in random numbers has already been mentioned. More specifically, the P2B conversion involves a statistical error. The probability of obtaining an output corresponding to X high-level values (1s) in a sequence of the random variable $p(t)$ evaluated throughout N_C clock cycles is given by the binomial distribution:

$$Prob(X) = \binom{N_C}{X} p^X \, (1-p)^{Nc-X} \tag{4.5}$$

The mean value of the P2B output X is the expected exact conversion ($\bar{X} = p\, N_C =$

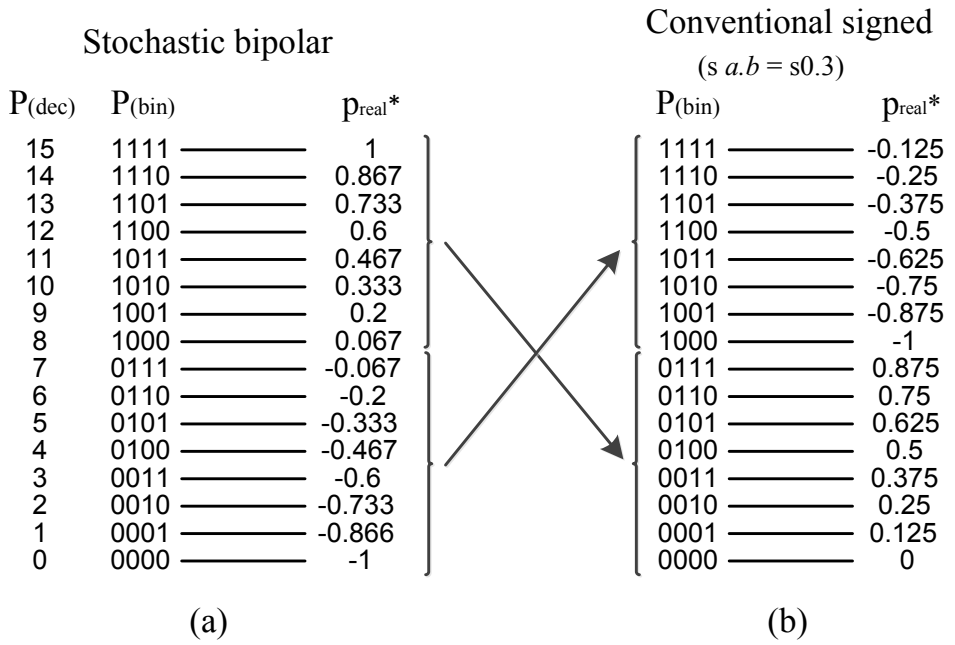


Figure 4.6.: Binary representation of 4-bit numbers according to the stochastic bipolar codification (a) and to the conventional signed two's complement notation (b). An approximate conversion from one notation to the other can be performed by just flipping the first bit.

P) while the standard deviation is

$$\sigma = \left[N_C\, p\, (1-p)\right]^{1/2} \tag{4.6}$$

This formula for the standard deviation implies that the accuracy of conversions depends on the encoded probability values. Values of p near to 0 or 1 imply a low conversion error whereas $p = 0.5$ presents the maximum error.

The relative error in a conversion is proportional to the standard deviation $Error \propto \sigma/N_C \propto N_C^{-1/2}$, and therefore it can be reduced increasing the evaluation time according to 4.7, which shows the trade-off in SC between accuracy and evaluation time:

$$Error \propto \frac{1}{T_{eval}^{1/2}} \tag{4.7}$$

Apart from the statistical conversion errors, stochastic computations may present additional inaccuracies as stochastic numbers go through a sequence of operations due to correlations among the bit-streams. These correlations may arise, for ex-

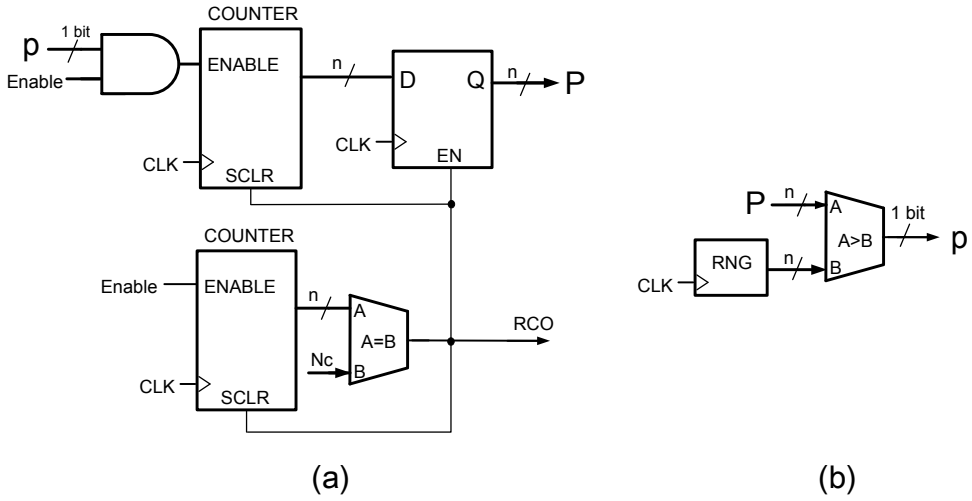


Figure 4.7.: Pulse to binary converter P2B(N_C) (a) and binary to pulse converter B2P.

ample, when feedback is present or when operating signals derived from a common input. In these cases might be convenient to regenerate the stochastic bit-streams by converting them to binary values and again to pulsed signals.

Binary to stochastic conversion. The conversion from binary magnitudes (P) to pulsed stochastic signals ($p(t)$) is carried out by the binary-to-pulsed (B2P) block illustrated in Fig. 4.7(b). The conversion involves generating an m-bit random or pseudo-random binary number in each clock cycle by means of a random number generator (RNG), and comparing it to the n-bit input binary number P. The comparator provides a "1" if P is greater than the random magnitude and a "0" otherwise. Therefore, the output of the comparator provides a bit-stream with probability $p = P/2^n$ (of getting a "1") provided that the random numbers are uniformly distributed.

Random number generation. The B2P circuit must produce random and uncorrelated bit-streams to avoid computation inaccuracies that may result from similarities among the different signals. Linear feedback shift registers (LFSRs) are commonly used as pseudo-random number generators in SC designs ([AH13b]). An LFSR is an array of n interconnected flip-flops with feedback to its input from a combination of the outputs of its various stages gated together in XNOR gates (an example 8-bit LFSR circuit is shown in Fig. 4.8). This linear feedback structure provides deterministic sequences of n-bit binary numbers that are uniformly distributed and behave as random numbers. These pseudo-random numbers have a finite period of repetition that has an exponential dependence with the number of bits. The feedback configurations enabling maximal-length generators are given in [Kor66]. These

optimum LFSRs generate number sequences with a repetition period equal to $2^n - 1$ (the zero value is excluded). The VHDL code describing the configuration applied in the present work for a 26-bit shift register, is shown in Algorithm 4.2 (note that the most significant 16 output bits of the 26-bit LFSR are used as random number).

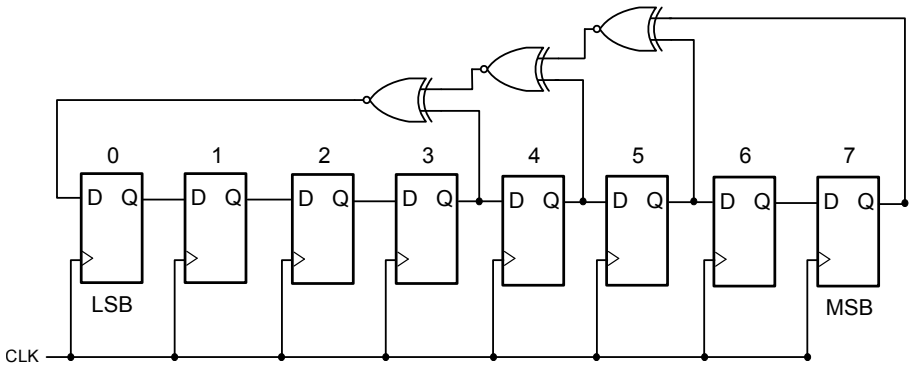


Figure 4.8.: Example of 8-bit linear feedback shift registers (LFSR) circuit that can be used to generate pseudo-random sequences cycling through 255 values.

The initial values of the registers contained in the LFSR determine the first number (known as seed) of the pseudo-random generated sequence. Changing the seed value allows obtaining different number sequences. For operations that need uncorrelated signals, it is necessary to define different seeds for each LFSR block. On the other hand, for operations requiring correlated signals, the same LFSR output must be employed for all stochastic variables.

4.2.3. Applications

SC allows to implement arithmetic operations and complex functions with relative simplicity and high robustness. The implementation of massive parallel computing systems using SC is of particular interest since it enables the possibility of having many small-size processing elements working in parallel, which may compensate for the long computation time of the SC approach.

Image processing applications are an example where SC can be used efficiently. They usually require the transformation of large numbers of pixels, which can be performed using low-cost SC-based circuits operating in parallel ([OKM+15], [HNBO03], [NS16]). In addition, the high tolerance to bit-flip errors in noisy environments is another feature of SC that can be exploited in image processing as shown in [QLR+11], [LL11], [MZD12] and [NS16].

Digital signal processing is another application for SC. For instance, SC can be used to implement finite impulse response (FIR) filters ([WHCE15]) or the decoding of low-density parity check (LDPC) codes ([AH13b]). LDPC codes consist of error-correcting codes that enable sending data over noisy channels at very high rates and

Algorithm 4.2 VHDL code describing the 26-bit LFSR employed to generate pseudo-random sequences of 16-bit numbers.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY lfsr_26bits IS
  PORT (clk, reset, enable: IN STD_LOGIC;
    seed: IN STD_LOGIC_VECTOR (25 DOWNTO 0);
    pseudorandom_out: OUT STD_LOGIC_VECTOR (15 DOWNTO 0) );
END lfsr_26bits;

ARCHITECTURE lfsr_26 OF lfsr_26bits IS
SIGNAL tmp: STD_LOGIC_VECTOR(25 DOWNTO 0);
BEGIN
PROCESS (clk, reset)
  BEGIN
  IF reset ='1' THEN
    tmp<=seed;
  ELSIF (enable='1') THEN
  IF (clk'EVENT AND clk='1') THEN
    for i in 0 to 24 loop
      tmp(i+1) <= tmp(i);
    end loop;
    tmp(0) <= tmp(25) XNOR tmp(5) XNOR tmp(1) XNOR tmp(0);
  END IF;
  END IF;
END PROCESS;

pseudorandom_out <= tmp(25 DOWNTO 10);

END lfsr_26;
```

are nowadays employed in communication standards such as WiFi. SC is appropriate for this application due to its probabilistic nature. In addition, SC allows efficient parallel processing of the involved operations, which results in fast implementations ([NMSG11]).

Recently, SC has also been used for data mining. The work presented in [MCO+15] proposes an SC-based parallel implementation of a similarity search algorithm that speeds up the screening process of huge databases by a factor of 7 when compared to a conventional digital implementation using the same hardware area. The higher speed of the SC-approach is achieved at the cost of a certain loss of accuracy that is not critical for this particular application.

In general, SC has been recognized as potentially useful in specialized systems where small size, low power, or soft error tolerance is required and limited precision or speed is acceptable ([AH13b]).

ANNs are also massively parallel systems that can benefit from the SC technique

allowing the implementation of large numbers of simple computational elements on a single integrated circuit. Some examples of HNNs using stochastic computing are found in [BH94], [BC01c], [BC01b], [PZD+03] and [SGMA15]. Even though SC-based ANNs seem unlikely to achieve speed-up compared to the conventional binary logic ones, they can be an interesting solution for those electronic systems implementing computational intelligence techniques and requiring low power dissipation but not demanding very high computational speed such as wireless sensor networks ([KFV11]), predictive controllers, or medical monitoring applications. In the latter case, a software implementation of RC was found to achieve state-of-the-art performance in the classification of electrocardiographic (ECG) signals ([EMSFM15]). Since this medical application requires a sampling time of about 1 ms, in general ECG classification would be compatible with an FPGA-based stochastic implementation of reservoir computing in real-time. An illustrative example of the use of a SC-based ANN for the control unit of an induction motor is found in [ZL08], where the stochastic implementation is shown to exhibit lower hardware cost than conventional microprocessor-based designs for the same application. Another stochastic HNN is proposed in [LZF06] for the control of a small wind turbine system. Recently, a new encoding scheme for SC with extended noise-tolerance (the ESL codification) has been used to implement neural networks performing regression and pattern recognition tasks ([CMO+16]).

Reviewing the scientific literature, we can state that although there were some proposals and simulations of the RC technique using stochastic bit-stream neurons [VSS05], the work presented in this chapter (previously published in [ACPM+16]) represents the first comprehensive design, implementation and examination of the ESN structure based on SC.

4.3. Stochastic implementation of neural networks

4.3.1. Stochastic design of the neuron

The SC-based implementation of a two-input sigmoid neuron (Fig. 3.1) is illustrated in Fig. 4.9. The bipolar codification is employed to perform the stochastic computations. In a first stage, the input and weight binary values are transformed to pulsed signals (through B2P blocks) so that they can be processed by the stochastic circuit. The first input signal (I_1) is assumed to be externally supplied to the system as a binary magnitude whereas the second one (i_2) comes directly from another neuron as a stochastic bit-stream. The multiplication and addition operations are implemented in the stochastic computing framework by means of an XNOR gate and a multiplexer, respectively, as described in sec. 4.2.2.1. The bit-stream resulting from the input weighting and addition is transformed to a binary number (P2B block) and evaluated classically (using conventional computing) by means of a sigmoid function. Finally, the binary outcome is converted again into a stochastic bit-stream so

that it can be further processed by another neuron.

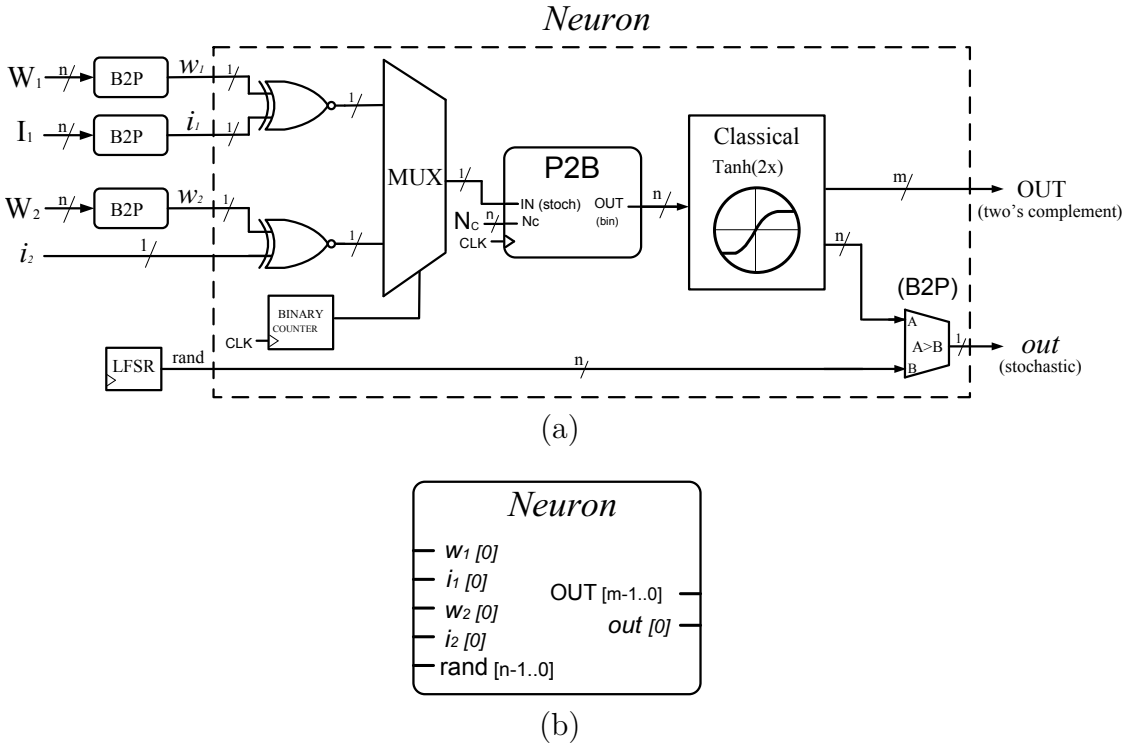


Figure 4.9.: SC-based two-input sigmoid neuron (a) and block diagram (b). The linear part of the neuron uses probabilistic logic whereas the nonlinear activation function is implemented using conventional computing.

The VHDL code of the stochastic neuron design is presented in sec. A.2. The precision is set to $n = 16$ bits and the evaluation time to $N_C = 2^{16} - 1 = 65535$ clock cycles. It is worth noting that the block performing the nonlinear function first needs to convert the binary input signal from the bipolar stochastic representation to the conventional two's complement notation. This is done by just flipping the most significant bit of the binary magnitude. In addition, the output of the neuron's linear part must be multiplied by 2 to compensate the scaled sum performed by the multiplexer. This multiplication is internally performed by means of a simple shift of the binary number one position to left. The output that results from applying the hyperbolic tangent function is provided both in the bipolar stochastic format (with n bits) and as a two's complement magnitude (with a precision of m bits; more specifically, I use $m = 8$). If the approximation used to calculate the output of the sigmoid function employs a lower precision, only the most significant bits of the input signal are utilized. For the present implementation (as described below), the approximation used considers 9 bits for the input signal and produces the output with a precision of 7 bits.

4.3.1.1. The activation function

Although there are different pure stochastic approaches to reproduce the sigmoid function ([BH94], [BC01b], [CMO+16]), for the present research I have adopted an implementation of the hyperbolic tangent function based on conventional binary logic. In particular, I use the SIG-sigmoid approximation presented in sec. 3.2.3.2. As illustrated in Fig. 4.9, the stochastic signal resulting from the weighting and addition of the inputs needs to be converted (P2B block) to its corresponding binary value before it can be processed by means of the hyperbolic tangent classical approach. Therefore, in this section I propose a design combining both stochastic and conventional computing. Stochastic arithmetic allows reducing the computation hardware area required to implement the arithmetic operations present in neural networks while conventional binary logic is used to increase the accuracy of the nonlinear activation function.

In a previous design ([CMO+16]), the hyperbolic tangent function is implemented only using SC (see Fig. 4.10). This implementation makes use of the P2B output (governed by a binomial distribution) evaluated throughout K clock cycles. The resulting signal is compared (every clock cycle) to the binary value $K/2$ producing a switching signal with a probability that approximates the sigmoid function. This pulsed signal needs to be regenerated (by means of a pair of P2B and B2P blocks) to result in a bit-stream with random distribution. The last B2P converter can be omitted in case a stochastic output signal is not necessary but only the corresponding binary magnitude. The gradient G of the resulting function $\tanh(Gx)$ can be modified with the parameter K as tabulated in [CMO+16].

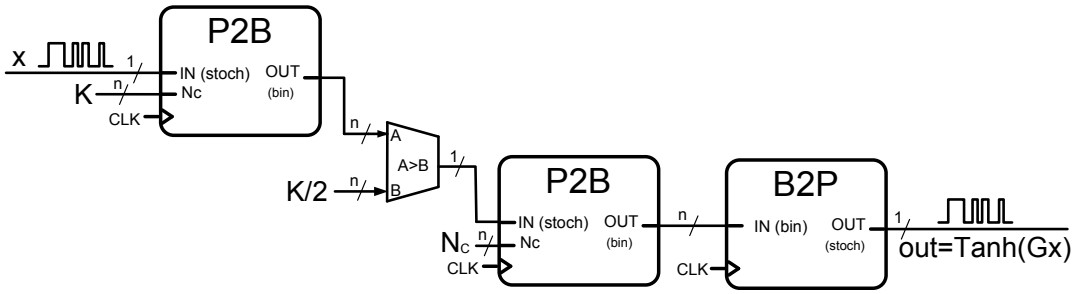


Figure 4.10.: Stochastic implementation of the hyperbolic tangent function proposed in [CMO+16].

The proposed circuit of Fig. 4.9 for the two-input sigmoid neuron (utilizing the classical SIG-sigmoid approximation) has been tested for a set of values of the input signals. The results are compared with those of an equivalent circuit using the design of Fig. 4.10 for the sigmoid function. In both cases the experimental results were obtained employing an evaluation time corresponding to $N_C = 2^{16} - 1 = 65535$ clock cycles. The two input signals I_1 and I_2 have been given 32 different values in the $[-1, 1]$ range while the weights have been fixed to $W_1 = W_2 = 1$. The

parameter K has been assigned the value $K = 11$, which corresponds to the function $\tanh(2.09 \cdot x)$. The results for both approaches are presented in Fig. 4.11. It can be observed that the mixed stochastic-traditional method is more accurate than the purely probabilistic design of [CMO+16]. The logic resources are similar in both cases, and therefore the mixed approach is more convenient due to its higher precision.

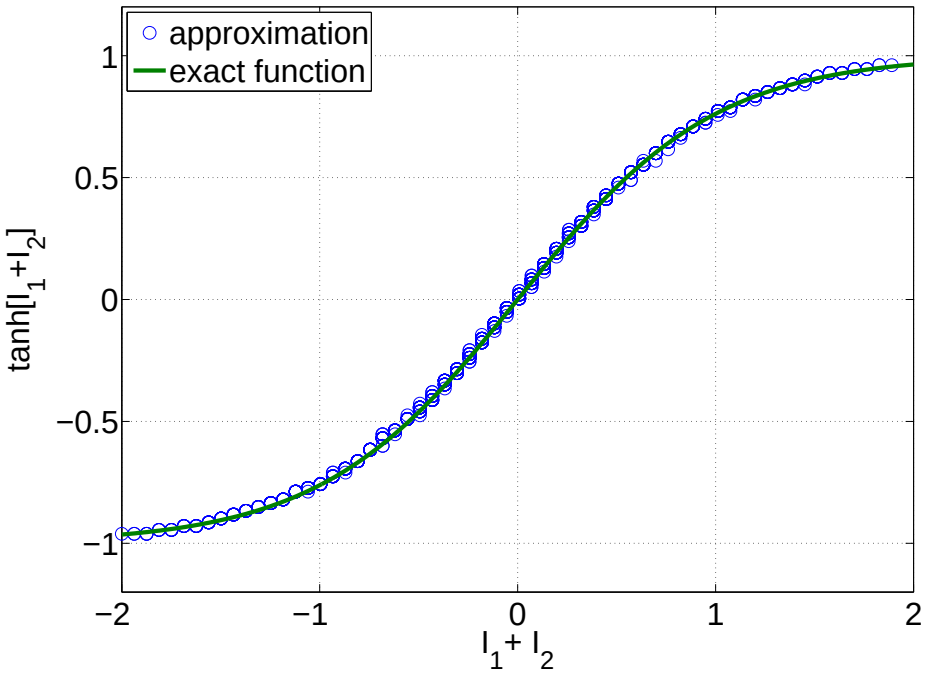
It can be appreciated in the results of both approaches that, as expected from the probabilistic error associated to the P2B conversions (4.6), the output values near to 0 (that correspond to a probability 1/2 of the stochastic bit-stream being in the high state) present a higher dispersion than those values near to -1 and 1 (corresponding to probabilities 0 and 1, respectively).

4.3.2. Stochastic ESNs

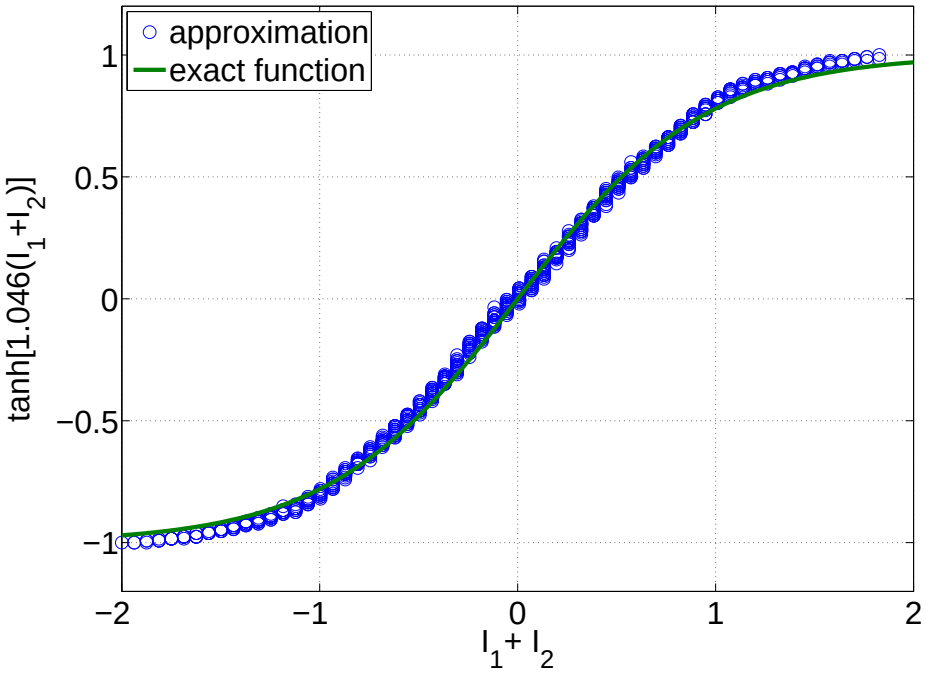
The simple cycle reservoir (SCR) topology introduced in sec. 2.3.1.2 is employed to implement the SC-based echo state network. Remind the particular benefits of this structure for hardware implementation regarding the reduced number of connections and the simplicity to automate the network's design compared to the classical ESN architecture with random connections. The neuron design of Fig. 4.9 can be used as a block to build a SCR network of neurons where the first (external) input is common to all neurons and the second input is utilized for internal connections between neurons. Such a modular network design is illustrated in Fig. 4.12.

It is worth highlighting that the four sequences of pseudo-random numbers required to convert the input and weight values to stochastic bit-streams do not need to be different for each neuron but can be shared by all neurons. In the SCR structure, each neuron presents only one connection from another neuron, and therefore the correlation between the output bit-streams of different neurons does not affect the result of the computations. However, the general ESN architecture with random connections between neurons would require generating different random number sequences to produce uncorrelated outputs for each neuron. In addition, the use of only three different weight values for the network connections (r, v and $-v$) also favors a resource-efficient implementation since only two B2P blocks are necessary (the negative signal is obtained from the positive one with a NOT gate) whereas a configuration with many different weight parameters would require a B2P converter for each one of them. Since B2P conversions account for a significant portion of the hardware resources in SC-based designs, the use of common random number generators shared by all neuron units allows an important reduction of the number of required logic elements per neuron.

The output layer (linear combination of the neuron outputs) is implemented using conventional binary logic. More specifically, the final readout is computed sequentially by means of a multiply-accumulate (MAC) circuit that performs the multiplication of the neuron state x_i by the corresponding weight w_i, and then adds the



(a)



(b)

Figure 4.11.: Experimental measurements of the SC-based 2-input neuron using the classical SIG-sigmoid approach for the hyperbolic tangent function (a) and using the probabilistic approximation of [CMO+16] (b).

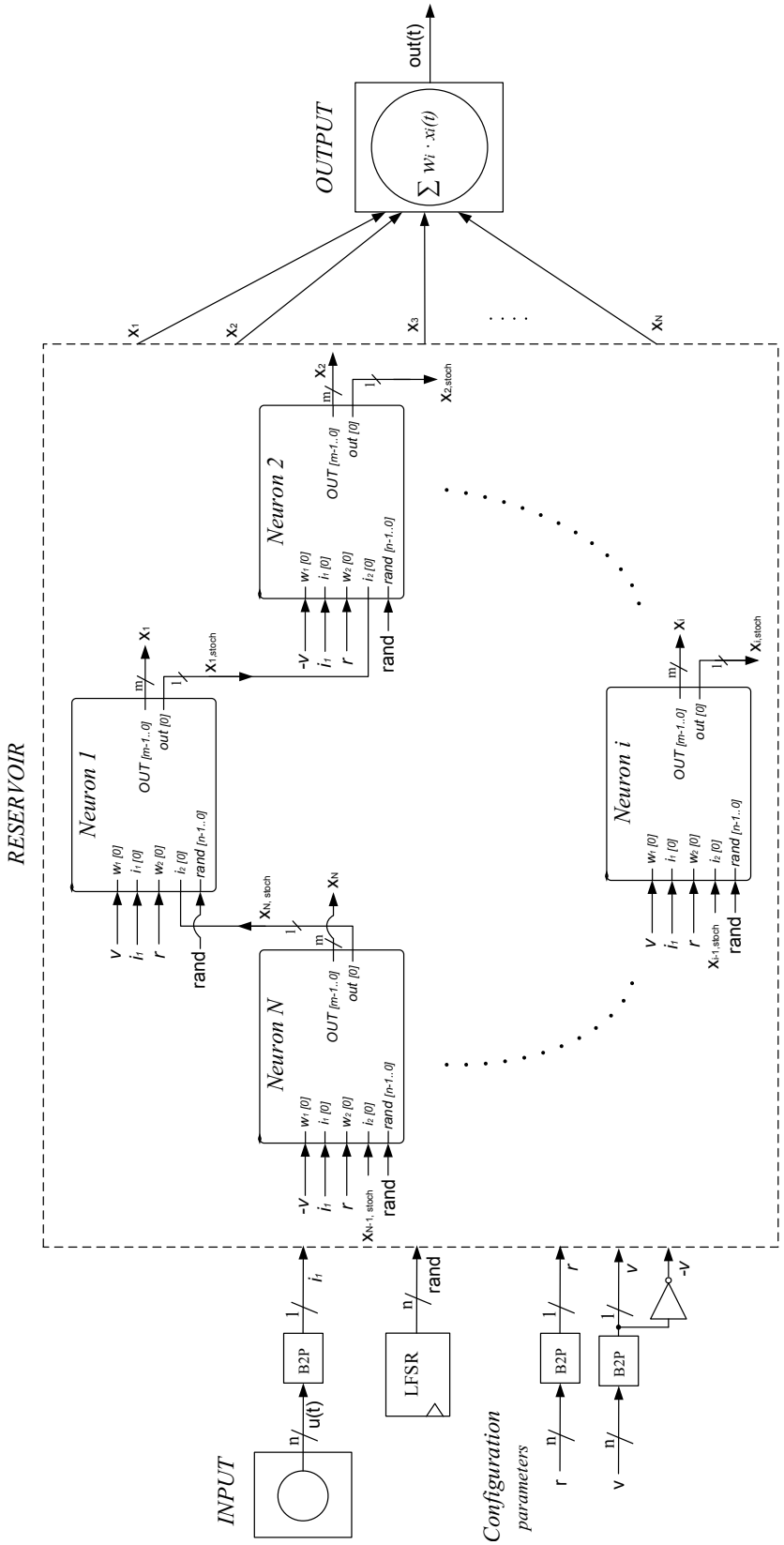


Figure 4.12.: SC-based implementation of an ESN with cyclic architecture. A few pseudo-random number generators are shared by all neurons.

result ($prod_i$) to the previously accumulated value. The operation is iterated for all neurons ($i = 1 \ldots N$). The schematic representation of this circuit is shown in Fig. 4.13. As can be observed, a precision of 8 bits is used for the neuron outputs and weight values while 16 bits are employed for the final readout.

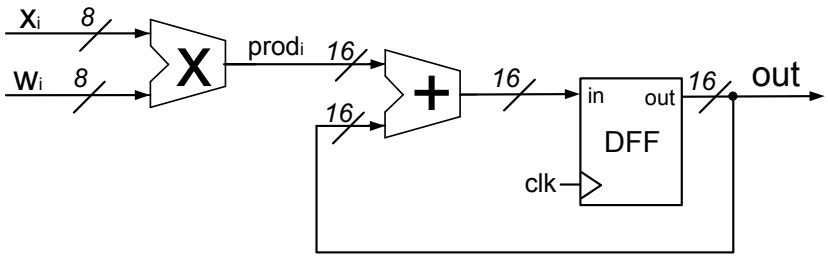


Figure 4.13.: Schematic representation of the multiply-and-accumulate circuit employed to implement the output layer.

The VHDL code of the complete SC-based ESN is presented in sec. A.2 for the case of a network with $N = 20$ neurons. A software program has been developed which allows exporting automatically the proposed stochastic ESN design to a VHDL hardware description. The program generates the VHDL code for the reservoir with any desired number of neurons and weight configuration. This VHDL code can finally be synthesized to an actual hardware implementation.

The hardware resource consumed by the SC-based implementation of ESNs is presented in next section for different network sizes. A breakdown indicating the logic elements used by each component of the neuron is also included.

4.4. Experimental results

4.4.1. Proof-of-concept example

As an example of functionality of the presented methodology, the proposed stochastic ESN design with a small number of neurons ($N = 20$) has been synthesized on an Altera Cyclone IV medium cost FPGA (EP4CE115F297C7N) and trained to perform a simple nonlinear transformation task. More specifically, the function that the network must learn is the next:

$$y(t)_{teach} = 3/4 \cdot u(t)^3 \tag{4.8}$$

where a sinusoidal input $u(t) = \sin(2\pi \cdot t/T)$ with 20 points per period ($T = 20$) is used to drive the system.

To perform the measurements of the proposed SC-based ESN circuitry, an internal RAM memory supplies the input signal to the reservoir network every time step corresponding to a certain number of clock cycles given by T_{eval}. The resulting network outputs (final readout and individual neuron states) are measured at the end of each evaluation period (by that time, the result of the stochastic computations is already available). Their values are stored in memory, which can be finally displayed using the signal logic analyzer.

A numerical model of the hardware SC-based ESN has been developed with MATLAB for a more efficient training and debugging. This model emulates in software the generation of the stochastic bit-streams and the operations among them. The resolution of the binary variables is limited according to the hardware. The present nonlinear transformation task is used to validate the MATLAB simulation model by comparing the experimental results of the FPGA implementation with the numerical ones. Fig. 4.14 shows the comparison of the experimental results and numerical simulations for the evolution of two selected neuron states in the 20-unit ESN. As can be appreciated, a good agreement is obtained. Slight differences are mainly due to the probabilistic nature of the approach.

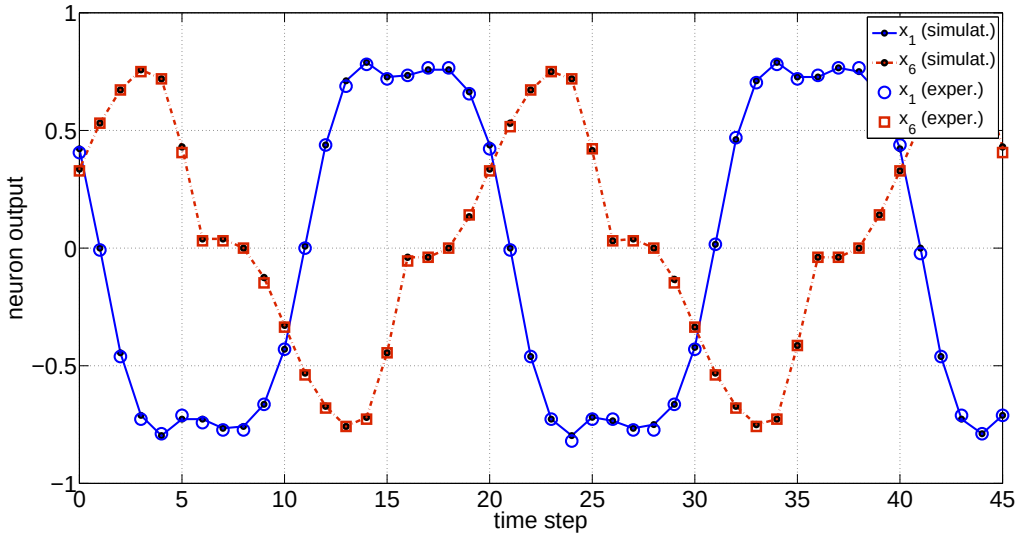


Figure 4.14.: Traces of two arbitrarily selected neurons from the reservoir when driven by a sinusoidal input. Experimental values (symbols) are plotted along with the numerical results (lines).

The MATLAB model of the stochastic hardware allows us to perform a comprehensive evaluation to find the configuration of the stochastic reservoir network providing optimum results. Fig. 4.15 shows how the network performance (measured through the mean square error, MSE) is examined as a function of the configuration parameters r and v. Fig. 4.15(a) shows the performance results assessed according to a conventional computing approach while Fig. 4.15(b) illustrates the results that

correspond to the stochastic methodology. The approach based on conventional computing is referred to as *deterministic* in Fig. 4.15 to indicate that it does not use the probabilistic methodology. This deterministic approach corresponds to the system described in chapter 3, but it employs a more accurate approximation to the sigmoid function just as in the proposed stochastic ESN implementation. In addition, the resolution of the variables has been limited according to that of the stochastic ESN design (for example, the neuron outputs and output weights are given with 8 bits) so that both systems (conventional and stochastic) employ an equivalent precision and their results can be fairly compared. Fig. 4.15 shows that the MSE values can be quite different in both scenarios, and therefore the conventional ESN model cannot be used to determine the optimum configuration for the stochastic hardware implementation (but the simulations of the stochastic hardware are necessary).

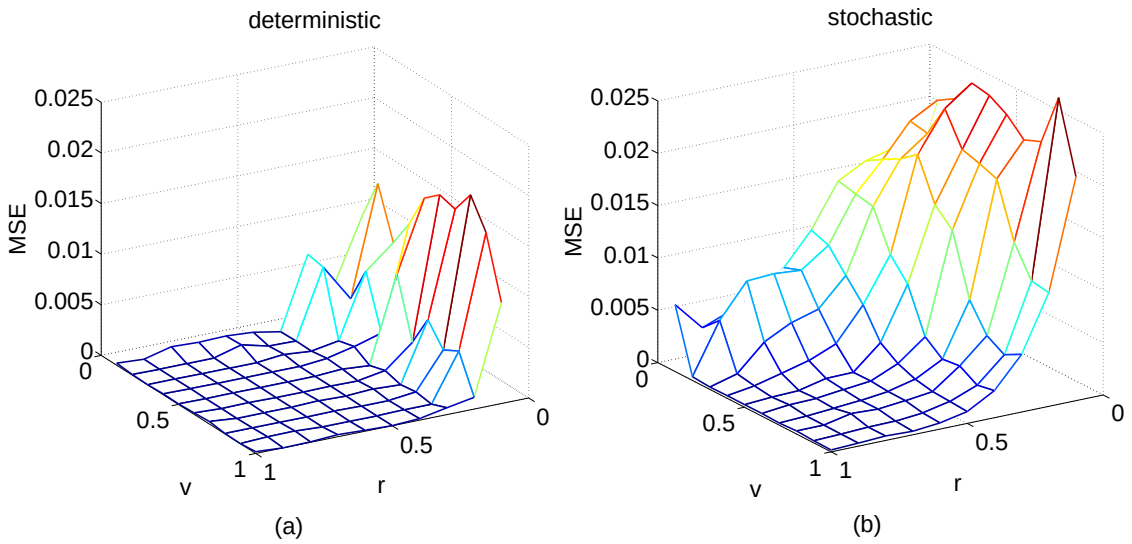


Figure 4.15.: Simulation results for the mean square error (MSE) in the nonlinear transformation task according to the conventional (deterministic) (a) and stochastic (b) approaches. The two scanned parameters are the weight values r and v. The number of neurons is fixed to $N = 20$. A randomly generated distribution of the input weights is used. The evaluation time is fixed to $T_{eval} = 2^{16} \cdot T_{clk}$ for the stochastic approach.

Once the optimum parameters were determined, the hardware was configured, trained, and experimentally tested. The training (assessment of the output layer optimal weights) consists of a linear regression of the teacher output (y_{teach}) on the reservoir states (x_i, $i = 1 \ldots N$) as described in sec. 2.3.1. It was carried out using the experimental outputs of the individual neurons. Although the software implementation quite faithfully reproduces the hardware results, it was not used to train the system since smaller error results were obtained when directly using the experimental outputs.

In the time-series experiment, I performed a total of 250 time steps. The first 20 time steps (the transient) were neglected, results from $t = 21$ to $t = 125$ were used to train the network's readout, and with data from $t = 126$ to $t_{max} = 250$ I tested the network.

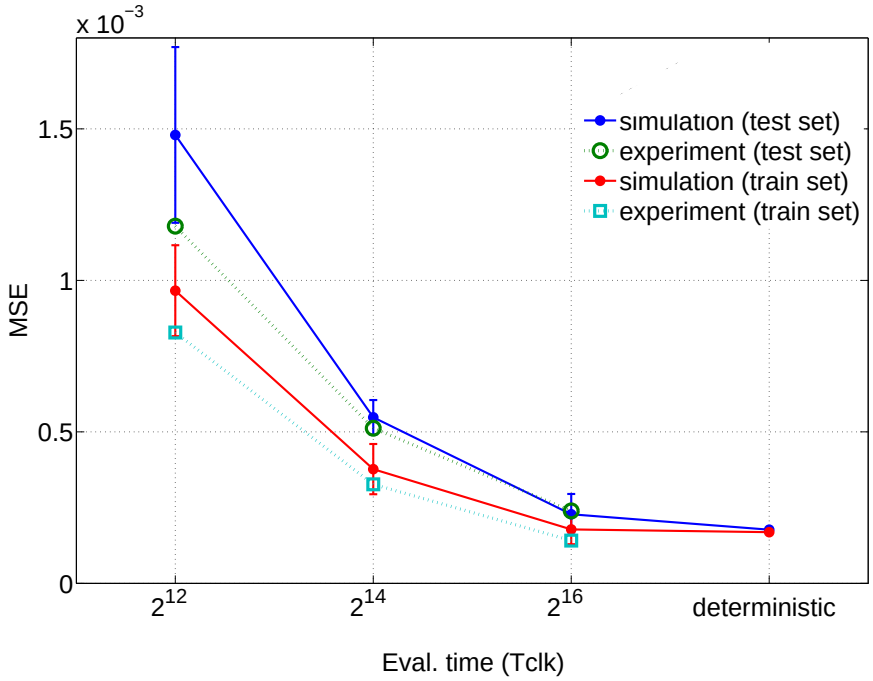


Figure 4.16.: Experimental (symbols) and simulation (lines) performance results (MSE) of the stochastic ESN implementation configured with the optimum weight parameters. The simulation results correspond to the average value of ten trials using different sequences of random numbers. Error bars indicate the standard deviation of the multiple runs. The performance is represented for different evaluation periods and for both the train and test sets. The performance corresponding to a conventional (deterministic) implementation is also given as a reference.

An experimental training error $MSE_{train} = 1.41 \cdot 10^{-4}$ and a test error of $MSE_{test} = 2.39 \cdot 10^{-4}$ were obtained when using a 16-bit precision (that is to say, an evaluation time for the stochastic computations of $T_{eval} = 2^{16} \cdot T_{clk}$). The performance results for several evaluation periods are represented in Fig. 4.16 along with the performance corresponding to a conventional (deterministic) implementation configured with the same parameters than the stochastic reservoir. The results of the experimental measurements are compared to those obtained from simulations. The minor differences are most likely due to the inherent probabilistic fluctuations of the stochastic approach. It must be considered that stochastic computations provide the results with a certain dispersion from the expected value according to 4.7. Therefore, the neuron states (used to calculate the network's readout) for the training set may differ

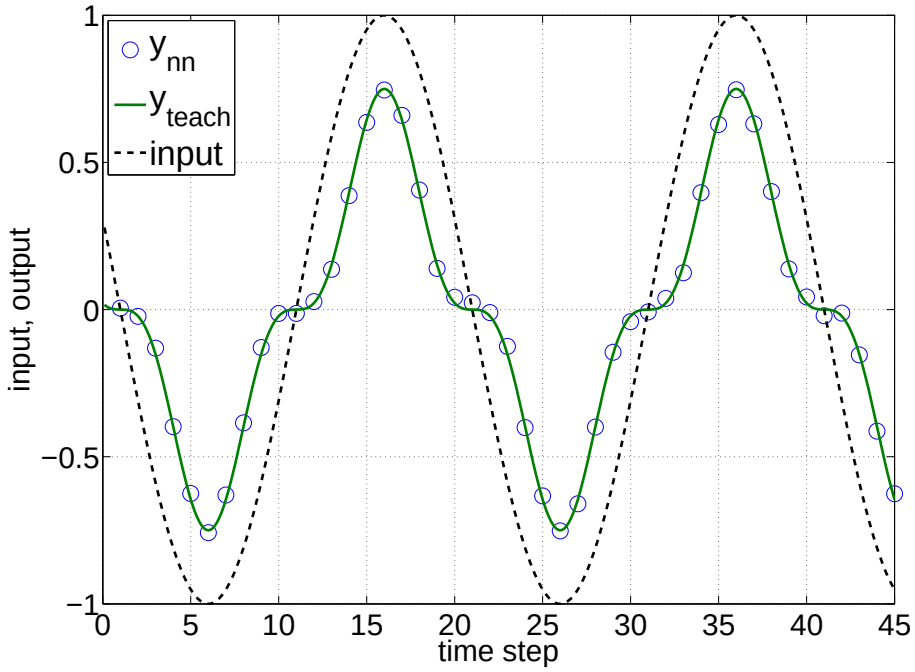


Figure 4.17.: Experimental results of the FPGA-based stochastic ESN (symbols) for the case of $T_{eval} = 2^{16}\, T_{clk}$ along with the input (dashed line) and desired output signal (solid line).

slightly from those of the test set even when using exactly the same input to drive the system. Since the physical implementation and the simulation do not use the same random numbers to generate the stochastic signals, the differences between the neuron states in the test and training set can be (by chance) smaller or greater in one of the systems (real hardware implementation or simulation) than in the other, which results in a slightly higher or lower performance. This explains why the discrepancies between the hardware and the numerical model tend to increase with a lower evaluation time given that the dispersion of the measurements grows with the inverse of the evaluation time (4.7). In addition, a part of the slight differences between the simulation and the experiments may be due to the fact that the sigmoid function employed in the model does not reproduce exactly the approximation implemented in hardware.

The simulations of this experiment were carried out a number of ten times (using a different sequence of random numbers for each run) in order to assess the dispersion of the final results. The values presented in Fig. 4.16 correspond to the mean of the multiple trials, which are shown along with the standard deviation as an indicator of the probabilistic error. The agreement between simulations and experiments (within the error range) validates the MATLAB model to estimate the hardware performance.

On the other hand, Fig. 4.16 shows that both stochastic reservoirs (MATLAB model and experimental measurements) gradually approach the expected deterministic behavior when increasing the evaluation time.

Finally, Fig. 4.17 shows the measurements at the readout of the stochastic reservoir along with the desired behavior (y_{teach}). The evaluation time of the network is of the order of 1.3 ms when using an evaluation time of 2^{16} clock cycles and a 50 MHz clock signal. This computation time is reduced when using smaller evaluation periods at the cost of a lower accuracy (as depicted in Fig. 4.16). Note that, for larger reservoirs, the processing time is kept fixed since it only depends on the number of clock cycles used in the P2B conversions.

4.4.2. Time-series prediction

A more complex task is implemented for a proper validation of the proposed methodology. This task consists in the one-step ahead prediction of the Santa Fe data set ([WG15]), which represents a benchmark in the RC literature. As in chapter 3, 4000 samples from the original data set are used; the first 2000 for training, the next 1000 for validation, and the remaining 1000 for testing.

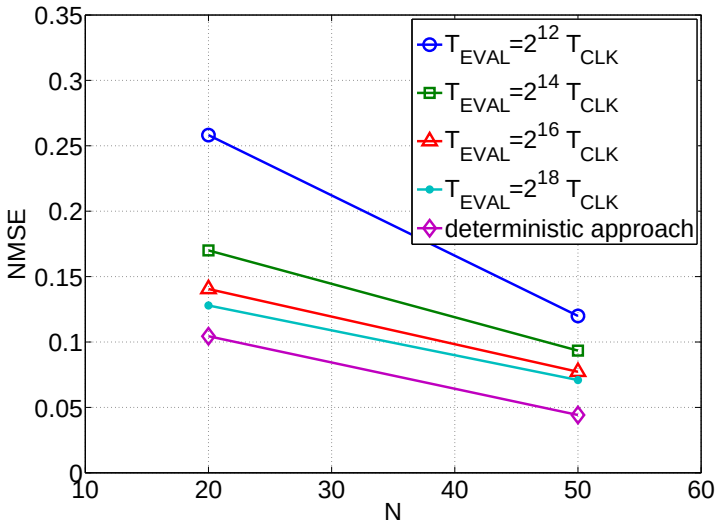


Figure 4.18.: Performance results (NMSE) of the stochastic ESN in the time-series prediction task. The values are displayed for a 20-unit and for a 50-unit reservoir using different evaluation periods. The corresponding results obtained with the conventional (deterministic) approach are also represented for reference.

The analysis of the proposed stochastic ESN implementation for this task is conducted using the MATLAB model of the stochastic hardware for two different reservoirs (with $N = 20$ and $N = 50$ neurons) and for different evaluation time periods.

A procedure similar to the one used in the previous section is followed, first determining the optimum configuration and subsequently testing the network. The configuration parameters allowing the best performance error for the validation set were applied to the network when processing the test set. The final optimum performance results (NMSE values) are depicted in Fig. 4.18 as a function of the number of neurons in the reservoir (N) for different evaluation time periods (corresponding to 12, 14, 16 and 18 precision bits). The performance corresponding to a conventional (deterministic) implementation (configured with the weight parameters r and v that yield an optimum result) is also represented for reference. It can be observed that the stochastic results gradually approach the deterministic ones when increasing the evaluation time.

Fig. 4.19 shows a fragment of the predictions performed by the SC-based ESN when using $N = 50$ and an evaluation time of 2^{16} clock cycles along with the targeted values in the Santa Fe prediction task.

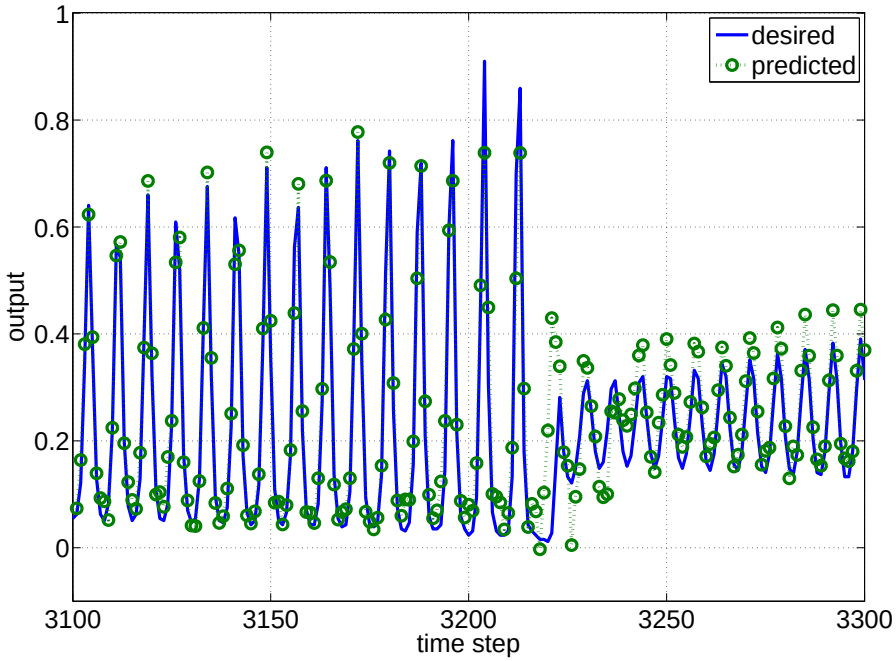


Figure 4.19.: Segment of the Santa Fe laser time-series (predicted and targeted values). Predictions performed using the stochastic methodology with $N = 50$ and $T_{eval} = 2^{16}\, T_{clk}$.

4.4.3. Hardware resource usage

The hardware resources required to implement the proposed SC-based reservoir networks are presented in Tab. 4.1 and in Fig. 4.20. The results are compared to

those that correspond to the conventional implementation of chapter 3. A slight difference in the number of logic elements presented here and those of chapter 3 is due to the fact that the present measurements of the spent hardware resources have been taken for a circuit that includes all the neuron states as output signals (the same has been considered for the implementation based on the stochastic approach) whereas the results presented in chapter 3 were obtained considering a circuit that only has an output signal corresponding to one of the neuron states (the rest of the states are assigned to intermediate signals, which involve less circuitry than output signals). It can be observed that the stochastic architecture requires about four times less area than the conventional hardware implementation. It is worth noticing that the probabilistic methodology allows the massive implementation of reservoir networks in medium and even low cost FPGAs. However, the conventional implementation of the 50-neuron reservoir does not fit in low cost devices such as, for example, the Cyclone III EP3C16 containing about 15000 logic elements ([web16d]).

Approach	***Stochastic***		***Conventional***	
N (neurons)	20	50	20	50
Logic elements	2186 (1.9%)	5306 (4.6%)	9013 (7.9%)	19975 (17.4%)
Registers	858 (0.7%)	2054 (1.8%)	320 (0.3%)	800 (0.7%)

Table 4.1.: Hardware resource utilization of the Altera Cyclone IV FPGA for the ESN implementation according to the stochastic and conventional approaches.

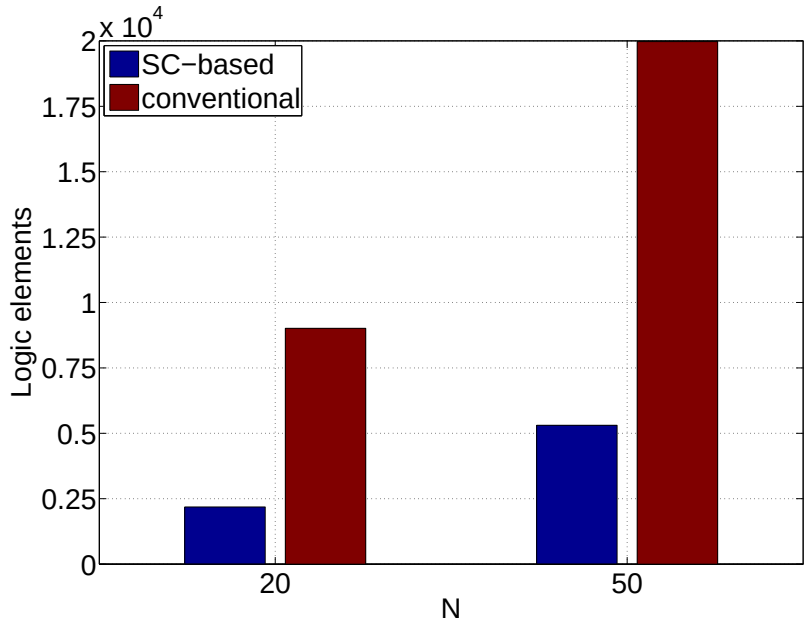


Figure 4.20.: Comparison of the logic elements spent by the stochastic implementation of the ESN and the conventional one.

The implemented reservoir did not require any memory bits except the ones used to store the input data values. The values shown in Tab. 4.1 are referred only to the reservoir network (the requirements to implement the final readout are not included).

A breakdown of the hardware requirements of each component of the SC-based neuron is illustrated in Fig. 4.21. A typical stochastic neuron consumes 103 logic elements (LEs) of which only 1 is required by the stochastic circuit performing the additions and multiplications. The area cost of the architecture is dominated by the P2B converter (46 LEs) and by the hyperbolic function (38 LEs). Finally, the comparator uses 18 LEs.

The B2P converters, which are used as common elements by all the neurons, use 46 LEs each whereas the random number generator (based on a 26-bit LFSR) consumes 26 LEs. Therefore, significant resource saving is achieved by sharing these components.

The proposed SC-based neuron design seems to be optimum in terms of hardware resources. Further reduction of the area requirements is only possible at the cost of a loss in accuracy using, for example, a rougher approximation of the sigmoid function or lower order P2B converters (the presented results are for neurons using pulsed signal conversions to 16-bit binary magnitudes).

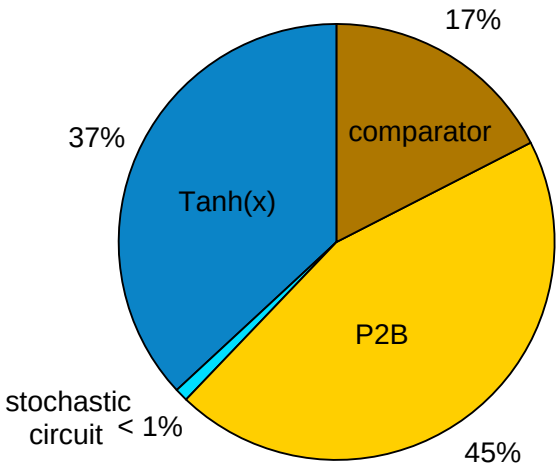


Figure 4.21.: Breakdown of the hardware requirements of each component of the SC-based neuron.

4.5. Discussion

In this chapter, I have proposed and analyzed an alternative architecture that exploits stochastic computing for making time-series prediction with echo state networks. It has been found that the stochastic architecture requires less area than a conventional hardware implementation. This characteristic makes possible the ESN implementation using low cost FPGA devices. Moreover, it has the advantage of being much more tolerant to soft errors (bit flips) than the deterministic implementation (see sec. 9.1.1), which makes it particularly useful for applications that need to operate in harsh environments such as space.

However, it should be noted that the stochastic implementation requires relatively many clock cycles to achieve a given precision compared to a binary logic conventional implementation. For instance, to get a 16-bit resolution, a computation time of 2^{16} clock cycles is needed.

Therefore, in general, potential applications of the stochastic implementations are specialized systems where small size, low cost, low power, or soft-error tolerance is required, and limited speed is acceptable. The presented SC-based ESN approach can be an interesting solution, by way of example, for electronic systems implementing computational intelligence techniques and requiring low power dissipation such as wireless sensor networks, predictive controllers, or medical monitoring applications.

For the ESN, a ring topology has been selected since hardware resources are minimized with this configuration while the precision of the network is not decreased with respect to a classical random one. In particular, the cyclic network structure allows reducing the number of stochastic number generators (SNGs), which are expensive in terms of hardware resources, by sharing a few of these blocks among all neurons. In addition, I have proposed an area-efficient design that employs probabilistic logic for the arithmetic operations and conventional binary logic for the nonlinear activation function (a mixed architecture). It has been observed that the area cost of the proposed implementation is dominated by the P2B converters and the sigmoid function.

A software program has also been developed that makes possible to automatically generate the hardware description code of the proposed implementation for any desired reservoir size and weight configuration. This reduces the long hardware design process for a specific application.

The proposed methodology has been used to implement a massive reservoir network and has exhibited considerable performance in a chaotic time-series prediction task. Time-series prediction usually requires high precision results. This makes necessary to employ long evaluation periods for the stochastic computations to achieve the desired accuracy. It would be interesting to analyze the use of the present SC-based implementation for a task where a lower accuracy is acceptable. For example, pattern recognition applications using noisy input signals are likely to allow satisfactory results with lower precision signals, which imply a shorter computation time. It is

worth noting that a lower evaluation time also involves some additional area reduction (simpler counters can be used in the P2P block and smaller LFSR circuits can be used to generate the random numbers). The application of the stochastic ESNs to temporal pattern recognition tasks is presented in chapter 9.

Reservoir networks present some advantages compared to conventional recurrent neural networks that enable a more efficient hardware implementation. A major benefit of RC networks is their sparse connectivity. This characteristic allows a simple wiring that matches the FPGA capabilities. Additionally, a simple training process can be performed offline.

The use of the stochastic logic implies certain constraints. The shortcomings are the evaluation time and the precision. Nevertheless, these drawbacks are compensated by the lower hardware and by the stochastic logic's inherent noise immunity which, all in all, allow a massive, parallel, and reliable implementation.

5. Stochastic echo state networks as liquid state machines

5.1. Overview

In the previous chapter, I have presented the implementation of a discrete-time (sigmoidal) neuron design using a mixed architecture with stochastic computing (SC) and traditional binary logic. The proposed neural design has been used to realize echo state networks (ESNs) in digital hardware. SC is based on the use of random bit-streams that loosely resemble the neural spikes of biological neurons ([Hay15]). As illustrated in Fig. 5.1, neural signals consist of sequences of noisy spikes that can be represented as bit-streams. The spike trains in real neural systems appear to have a stochastic or random nature ([SMK+00]). This has been one of the main reasons to assume that a significant amount of the information they convey is in the firing rate ([CGRS09]). The apparent lack of reproducible spike patterns is partly due to the probabilistic nature of the synaptic transmission mechanism (each synaptic vesicle releases its "quantum" of transmitter from the neuron presynaptic terminal with a given probability, which can be understood as a measurement of the connection weight). The general similarities between signals in neural processing and those in SC (in view of their probabilistic essence) suggest that SC can be conveniently used to codify neural information.

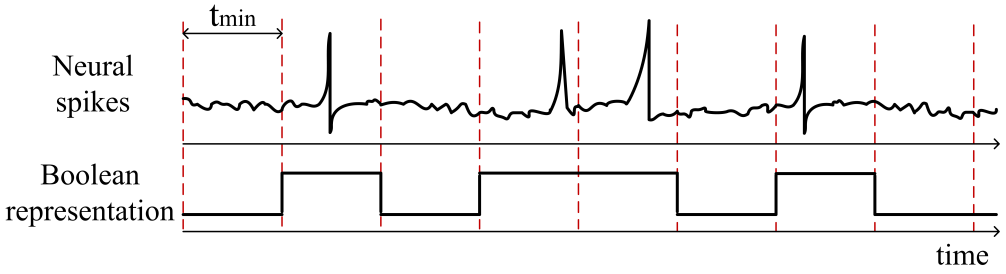


Figure 5.1.: Neural spike train and its Boolean representation as the bit-stream {01011010}. The probabilistic pulses generated in SC-based systems loosely resemble the spikes emitted by biological neurons.

In this chapter, I show how a variant of the stochastic neuron of chapter 4 can be interpreted as an approximation to the spiking neuron. More specifically, I illustrate

the equivalence of the proposed SC-based design and the stochastic spiking neuron (SSN) model developed by Rosselló et al. ([RCOM14]). The SSN model roughly retains the basic features of the spiking neuron using probabilistic rules, but its major contribution is the possibility to correlate/decorrelate the spike emission of different units, which allows to increase the network's functionality.

The proposed stochastic neuron design is employed to implement a reservoir network (that is, a liquid state machine, LSM, sec. 2.3.2, since the neurons can be regarded as being spiking) in a FPGA. The implemented network is tested for several benchmark time-series prediction tasks and the hardware resource requirements are analyzed.

5.2. The stochastic spiking neuron model

5.2.1. Introduction

Stochastic spiking neural networks (SSNN) are a recently proposed hardware solution ([RCMO12], [RCOM14]) based on a simple spiking neuron model capable to reproduce the probabilistic nature of synaptic transmissions, thus replicating the intrinsic stochastic behavior of real biological neurons. Instead of assuming a neural coding based on the exact timing of action potentials at each neuron, the SSN model assumes a probabilistic codification based on the apparent lack of reproducible spike patterns in real neural signals. Apart from the frequency of spikes over a time window (considered in the conventional firing-rate coding), a probabilistic codification of the neural spikes also takes into account the correlation between neural signals. The combination of both parameters (firing-rate and signal correlation) has been shown to make possible the implementation of high-speed pattern recognition systems unable to be reproduced if only considering a firing-rate coding ([RCOM14]).

The greatest benefit of such a probabilistic approach (combining firing rate and correlation) is its simplicity compared to other timing codes such as rank order coding ([SK10]) or spike-time coding ([BPK02]). In this approach, the functionality realized by a neuron depends on the type of correlation between the different input signals ([RCOM14]). Since the exact timing of neural spikes does not need to be taken into account, the classical spike profile can be substituted by a binary sequence with a characteristic time (t_{min}) that represents the maximum time interval in the spike time-series in which a neuron can provide a maximum of one spike. The parameter t_{min} can be understood as being the minimum time response of the fastest neuron in the network and corresponds to the time base of the digital signal (that is, $t_{min} = T_{clk}$). At time step t_i, the Boolean value associated to the k-th neuron output signal can be high ($x_k(t_i) = 1$) if a spike is present within the time interval $[t_i,\, t_i + t_{min}]$, or low ($x_k(t_i) = 0$) if there is no spike (see Fig. 5.1). In this approach, both time-series (spike train and bit-stream) provide the same information and the temporal mean value of x_k represents the switching activity (firing rate) of the k-th neuron.

5.2.2. The operation mechanism

More specifically, the SSN model is based on a variation of the binding neuron (BN) model ([Vid07]). The BN model describes the neural functionality in terms of discrete events (digital pulses emulating the action potentials of real neurons). Each input impulse is stored for a fixed period of time after which it is completely disregarded. As usual, the binding neuron fires if the number of stored impulses (representing the membrane potential) reaches a definite threshold value v_{th}. This is a useful model for neural description that can be easily implemented using digital circuitry. The digital circuit for the SSN is represented in Fig. 5.2. It is made up of a few simple digital blocks: Boolean gates, a counter, a random number generator (RNG), a register and a comparator. The simple mechanism of this neural model can be described as follows. The incoming pulses are summed up (every clock cycle T_{clk}) in the counter throughout an evaluation time period T_{eval} (as in SC, the evaluation period consists of a definite number of clock cycles: $T_{eval} = N_C \cdot T_{clk}$). The counter's output provides an estimate of the mean membrane over-voltage s_k (defined as the difference between the membrane potential and the reference voltage: $s_k = v_s - v_{rest}$) at the end of each evaluation period. This value is stored in the register for a whole evaluation period (while the new incoming spikes are being integrated by the counter). Every clock cycle, the over-voltage s_k (resulting from the previous evaluation period) is compared with the threshold level $v_k = v_{th} - v_{rest}$ (that varies chaotically from a clock cycle to another). The value of the variable threshold $v_{th} - v_{rest}$ is generated by means of a random number generator. The neuron emits a spike ($x_k = 1$) if s_k exceeds $v_{th} - v_{rest}$ and is kept at a low level ($x_k = 0$) otherwise.

The incoming excitatory signals (e_j) contribute to increase the value of the digital counter (mean membrane over-voltage s_k) while the inhibitory ones (i_j) constrain the action of the excitatory pulses. Signals p_{kj} (and q_{kj}) represent the probability of excitatory (and inhibitory) signal transmission from the j-th to the k-th neuron, consisting of binary bits oscillating with a specific switching activity that is proportional to the probability of synaptic transmission. That is, p_{kj} and q_{kj} measure the strength of connection between the two neurons and its biophysical meaning is related to the probability of vesicle release in the synaptic connection. The signals e_j and i_j carry the neural activity (firing rate) in their mean (probabilistic) value, and therefore can be operated with the probabilistic weights just as the variables in the SC approach. AND gates are employed to obtain a pulsed signal with switching activity equal to the collision (product) of the incoming probabilities ($e_j \cdot p_{kj}$) and OR gates can be used to approximately perform the addition of different probabilistic signals. This is how the stochastic transmissions (and thus the weights of neural connections) are emulated in the digital circuitry used to reproduce the neural model. Note that (before entering the counter) the result of joining the excitatory contributions is multiplied by that of the inhibitory signals (through an AND gate) so that inhibitory spikes may cancel the effect of excitatory ones (shunting

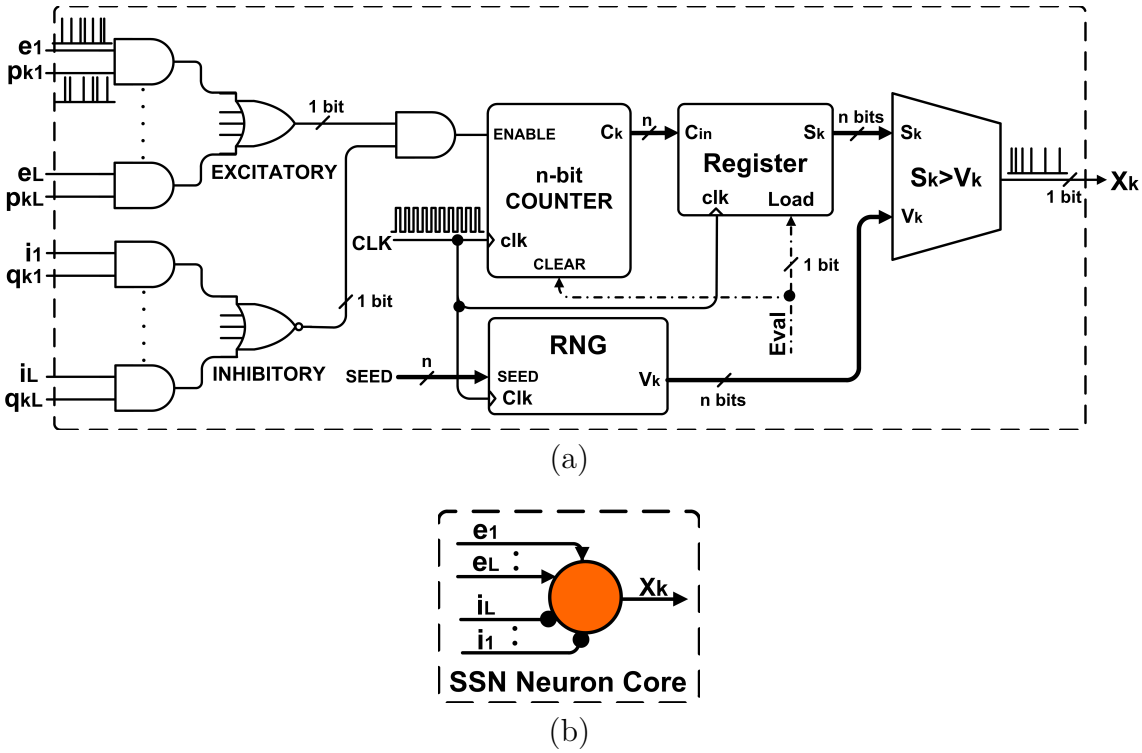


Figure 5.2.: Digital implementation of the Stochastic Spiking Neuron (SSN) with its characteristic blocks: the logic inputs, a digital counter, a register, a comparator and a random number generator (RNG) to provide the variable threshold v_k (a). Schematic block diagram (b).

inhibition).

The probabilistic bit-streams corresponding to the synaptic weights p_{kj} and q_{kj} can be generated using binary-to-pulsed (B2P) blocks (sec. 4.2.2.2). On the other hand, the digital neuron design of Fig. 5.2 requires an external signal (*Eval*) in charge of resetting the counter and enabling to load a new value in the register at the end of each evaluation period. This signal can be provided by a counter and a comparator as shown in Fig. 4.7(a).

The behavior of the SSN is illustrated in Fig. 5.3. It can be observed that the input spikes entering the counter (*ENABLE* signal) contribute to increase the membrane over-voltage s_k (output of the binary counter referred to as c_k in Fig. 5.2 and Fig. 5.3, which is captured by the register at the end of the evaluation period). The value of the estimated over-voltage corresponding to the previous evaluation period is compared (each time step, T_{clk}) with the reference voltage $v_{th} - v_{rest}$ (named v_k in Fig. 5.2 and Fig. 5.3). When s_k is over v_k, an action potential (high value of the neuron output, x_k) is generated. The reference signal v_k oscillates randomly to emulate the variation of v_{th} with respect to v_{rest} in real biological systems. Such random oscillation is assumed to be much faster than the typical spiking activity of neurons so that v_k is given a new random value for each time step T_{clk}. A

linear feedback shift register (LFSR) block (described in sec. 4.2.2.2) can be used to digitally implement a random signal generator emulating such oscillations.

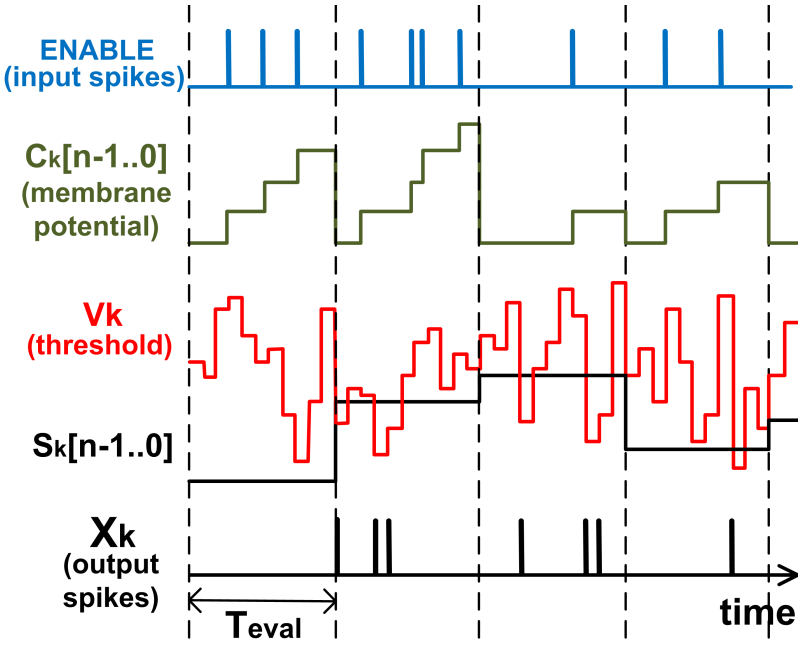


Figure 5.3.: Temporal evolution of the different signals in the SSN model.

Formally, the SSN output can be expressed according to 5.1:

$$x_k(t_i) = \theta(s_k(t_i) - v_k(t_i)) \tag{5.1}$$

where θ is the Heaviside function, $x_k(t_i)$ is the neuron output (at the current time step t_i), $v_k(t_i)$ is the variable threshold and $s_k(t_i)$ is the neuron's internal state, which takes a constant value throughout the whole evaluation period ($[t_i, t_{i+N_c}]$) and is proportional to the activity (rate of high values) of the input signal (entering the digital counter) evaluated during the previous step.

The SSN model essentially keeps the basic operation mechanism of the integrate-and-fire model (Fig. 2.3) and retains the general bio-motivated properties of a spiking neuron model (sec. 2.2.1). Namely, the neuron is able to convert the information of many incoming spiking signals into a single spiking output, both excitatory and inhibitory connections are allowed and an internal variable (representing the membrane potential) controls the spike emission (the output spike is fired in case that potential surpasses a certain threshold value). However, it must be noticed that in the SSN model, the output spikes that correspond to a sequence of incoming spikes are not generated at the current time step (as in real neurons), but on the next evaluation period (once the input sequence has been evaluated). In addition, it must

be noticed that no leakage effect is considered in this model. That is to say, the effect of the input spikes does not decay with time (contrary to real neurons where newer spikes contribute more to the potential than older ones), but it is remembered throughout the whole evaluation period after which the potential is reset. On the other hand, the refractory period is not explicitly modeled, but it is assumed to be equal to the minimum temporal step ($t_{min} = T_{clk}$).

5.2.3. Neural synchronization

A key point of the stochastic neural model is that the varying difference between the threshold and the resting potential ($v_k = v_{th} - v_{rest}$) can be used to synchronize neurons. Synchronicity implies a correlated emission of spikes. Detailed computational simulations of the default-mode brain network model have shown that synchronized oscillation may be present even in distant brain regions ([YLN12]). The SSN model allows to synchronize different neurons by sharing the same sequence of random values for v_k (that is, using the same seed value to initialize the random number generator of the different neurons, Fig. 5.2). When the neurons use different sequences for v_k they are de-synchronized (or chaotically related). Formally, the correlation/de-correlation of two neurons (k and j) can be expressed as follows:

$$\begin{cases} x_k \perp x_j & if\ v_k \neq v_j \\ x_k \parallel x_j & if\ v_k = v_j \end{cases} \tag{5.2}$$

where $\perp$ indicates that the signals are uncorrelated and $\parallel$ denotes correlation.

It has already been introduced in sec. 4.2.2.1 that stochastic circuits may perform different operations depending on whether the input bit-stream signals are correlated or not. Therefore, the SSN design of Fig. 5.2 can implement a range of different functions of the input signals ($e_1(t) \dots e_L(t)$ and $i_1(t) \dots i_L(t)$) by synchronizing/de-synchronizing some of these signals as observed in [RCOM14]. For example, according to the SC approach (and assuming the use of the unipolar codification), the AND gate placed just before the binary counter (Fig. 5.2) performs a multiplication of the incoming bit-streams (named $p(t)$ and $q(t)$, for example) when these are uncorrelated ($out = p \cdot q$), but a drastically different function results if correlated signals are used ($out = min(p, q)$). Similarly, the OR gates joining the different weighted input signals perform the operation $out = p + q - p \cdot q$ (considering a two-input case) if $p(t)$ and $q(t)$ are not correlated whereas they implement the function $out = max(p, q)$ otherwise.

The functions resulting from the use of correlated signals (synchronous case) present abrupt changes (related to the *min* and *max* functions, which select one of the inputs) while a smoother behavior takes place in the case of uncorrelated signals

(de-synchronization). The work presented in [RCOM14] suggests that the transformations related to the chaotic case (uncorrelated signals) minimize the information loss and are suited for tasks such as image convolution. On the other hand, the functions resulting from the combination of both synchronized and de-synchronized groups of neurons can be associated to more complex nonlinear processing tasks like pattern discrimination, where the signal information is drastically reduced to the useful one.

5.3. The proposed stochastic neuron design

The proposed variant of the SSN is illustrated in Fig. 5.4. This circuit is functionally equivalent to that of Fig. 5.2, but the operations are implemented assuming the bipolar codification ($[-1, 1]$ range for the probabilistic variables) instead of the unipolar considered in Fig. 5.2 ($[0, 1]$ range). Accordingly, the synaptic weighting of the inputs is realized through XNOR gates. The weighted signals are then nonlinearly transformed by means of a simple OR gate. Just as in Fig. 5.2, a counter integrates the resulting pulses (high binary values) throughout the evaluation period (defined as N_c clock cycles), after which the result is stored in a n-bit register and the counter is restarted (remind that the P2B block consists of a counter and a storage register, Fig. 4.7). The P2B's output magnitude (representing the mean membrane potential) is compared (at each T_{clk}) with a variable threshold value generated by a LFSR to produce the pulsed neuron's output. Note that, in this scheme, the inhibitions present in the SSN are represented as negative weight values.

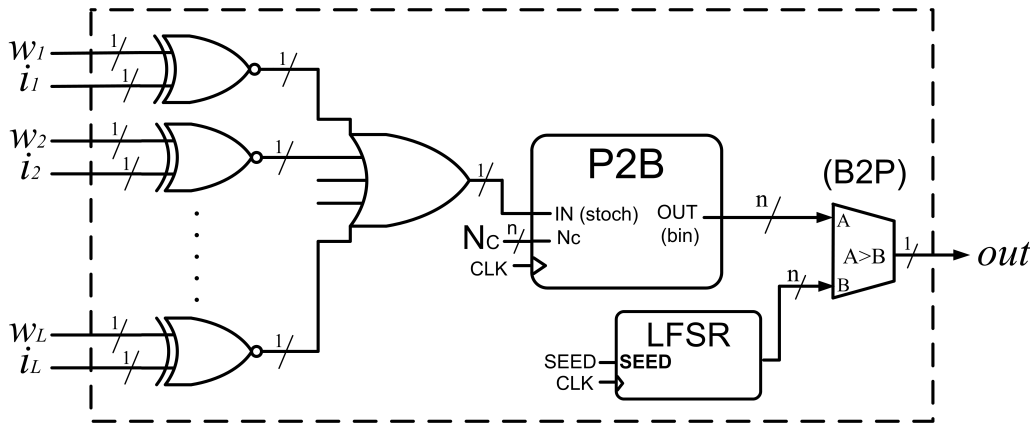


Figure 5.4.: Proposed variation of the SSN.

The same as in the SSN model, the proposed design can easily include correlations between different neurons. Two neurons produce uncorrelated outputs if they employ different seed values for the LFSR. On the other hand, two neurons are synchronized (they present correlated signals) when their LFSRs are initialized with the same value (in fact, the whole LFSR block can be shared by all synchronized units to

reduce the resource utilization). For the work presented here, however, I will limit to the case of functions resulting from uncorrelated signals.

As in previous chapters, I focus on the implementation of the reservoir network with cyclic topology (SCR) introduced in sec. 2.3.1.2. Due to the particular configuration of the connections in this structure, the neurons only require two input signals (one for the external forcing function driving the system and another one for the internal connection with other neurons). The reduced circuit for such a two-input unit is shown in Fig. 5.5. A number of these neurons can be arranged in a cyclic topology exactly as shown in Fig. 4.12. The B2P blocks used for the external input and input weights, as well as the LFSR, can be shared by all neurons. For this particular topology, the use of a common random number generator does not alter the neuron's function since each neuron only receives a single input from another one. In other words, all the neurons are synchronized, but it does not affect the result as their signals are never mixed together.

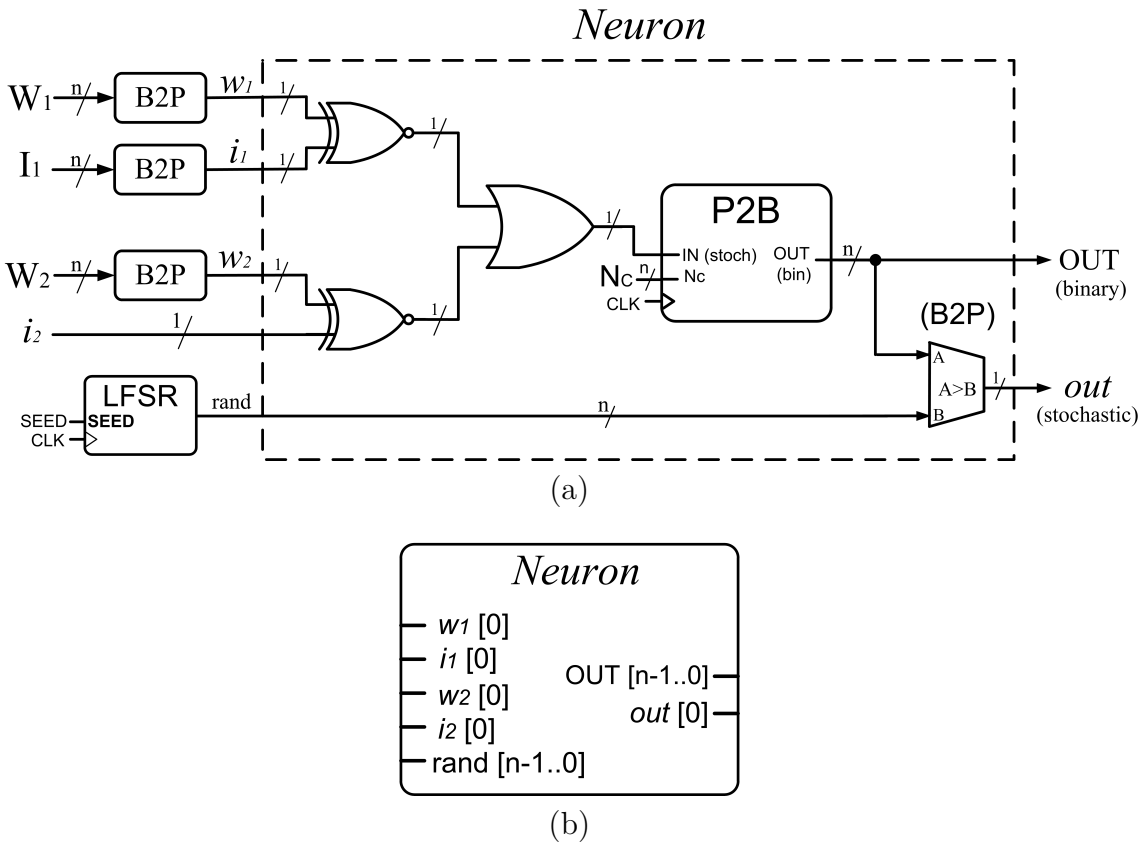


Figure 5.5.: Two-input stochastic "spiking" neuron design (a). Schematic block diagram (b).

The neuron design of Fig. 5.5 performs exactly the same functionality as the ensemble of seven bioinspired stochastic neurons (design of Fig. 5.2) with the configuration shown in Fig. 5.6 used in our previous work [RAM+16] to implement each of the

reservoir nodes.

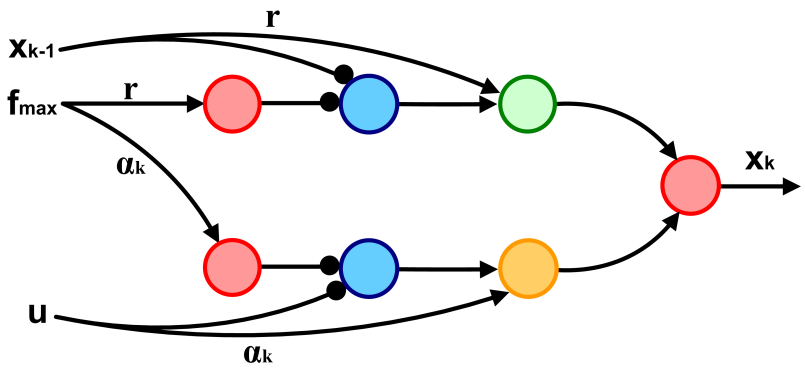


Figure 5.6.: Ensemble of stochastic spiking neurons (SSNs) employed in [RAM+16] to implement each one of the nodes of a cyclic reservoir. x_{k-1} corresponds to the signal coming from the neighboring node and u to the external input; r and α_k are their corresponding probabilities of synapse transmission (the connection weights are only indicated when they are different to 1). Such design can be replaced (assuming $f_{max} = 1$) by the equivalent circuit of Fig. 5.5 (where the external input and that coming from another neuron are noted as i_1 and i_2 while w_1 and w_2 represent their respective weights).

It can be noticed that the proposed implementation of Fig. 5.5 is actually a version of the SC-based sigmoid neuron design presented in the previous chapter (Fig. 4.9). Here, an OR gate of the weighted inputs is used to perform the nonlinear activation function instead of the specific circuitry implementing the addition and sigmoid function. The VHDL code describing the neuron implementation is presented in sec. A.3. For this design, a P2B block of 20 bits has been used, which allows a higher precision of the stochastic computations. In particular, the evaluation period has been set to $T_{eval} = (2^{19} - 1) \cdot T_{clk}$ (that is, $N_c = 2^{19} - 1$). The neuron's binary output is given (in the two's complement format) with a precision of 16 bits (the less significant bits are disregarded).

As introduced in sec. 4.2.2.1, the function implemented by an OR gate of two switching input signals representing the bipolar values i_1 and i_2 is

$$f(i_1, i_2) = OR(i_1, i_2) = \frac{i_1 + i_2 + i_1 \cdot i_2 - 1}{2} \tag{5.3}$$

while the function implemented by the sigmoid neuron design of Fig. 4.9 (assuming an encoded value of the weight signals given by $w_1 = w_2 = 1$) can be expressed as

$$f(i_1, i_2) = tanh(i_1 + i_2) \tag{5.4}$$

Both functions are represented in Fig. 5.7. The function implemented by the OR gate (equation 5.3) is bound to the $[-1, 1]$ range given that the inputs (i_1 and i_2) are limited to the same range. Unlike other ANN learning algorithms (such as classical gradient descent training methods) requiring the activation function to satisfy some conditions (such as derivability), the RC training approach is not restrictive with the function employed to non-linearly transform the inputs. Indeed, the RC principle has been used as a strategy to implement useful computations by means of various dynamical systems employed as nonlinear reservoirs that transform the input signal into a high-dimensional state ([LJS12]).

It is worth noting that the simplified nonlinear function implemented through the OR gate allows a significant saving of the hardware resources (see next section for the network's consumed logic elements).

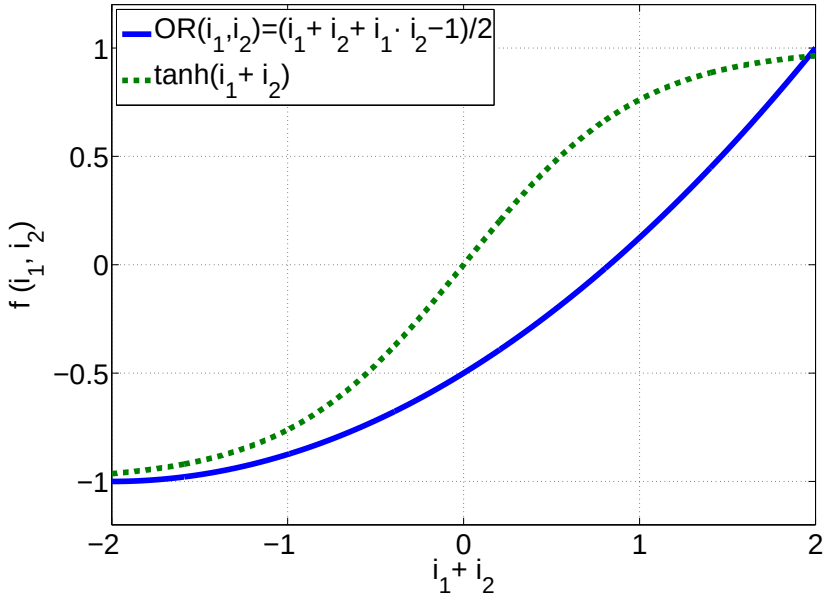


Figure 5.7.: Probabilistic function implemented by the circuit of Fig. 5.5 (continuous line) compared to that of Fig. 4.9 (classical sigmoid, dashed line) assuming $w_1 = w_2 = 1$.

Since the proposed SC-based discrete-time continuous-value neuron can be perceived as a simplistic way to emulate a spiking node, a reservoir network of such stochastic units can be regarded as a LSM.

5.4. Experimental results

A network of the proposed SC-based neurons (Fig. 5.5 and sec. A.3) has been implemented in a medium cost FPGA (Altera Cyclone IV) following the reservoir cyclic

architecture as illustrated in Fig. 4.12. The size of the network has been fixed to 20 units ($N = 20$) and the evaluation period to $T_{eval} = (2^{19} - 1) \cdot T_{clk}$ (that is, $N_c = 2^{19} - 1$) with a clock frequency of 50 MHz ($T_{clk} = 20\,ns$). The network's performance has been evaluated for three forecasting tasks with different degrees of complexity.

For each of the studied examples, a simulation has been carried out to determine the optimum configuration of the reservoir's weight parameters r and v. The numerical model (developed with MATLAB) that has been used to emulate the hardware implementation consists of an ESN that employs the function of equation 5.3 instead of the conventional sigmoid function. The results of the SC-based hardware implementation have been assumed to tend to those of this software realization given the long evaluation period selected for the stochastic computations. The resolution of the variables in the simulation model have also been limited according to the hardware. The numerical simulations employ the training data set to train the network's output layer and the test set to analyze the network's performance for a number of different configurations (values of r and v).

After finding the optimum parameters by software, the hardware was configured, trained, and experimentally tested. To perform the measurements of the proposed SC-based LSM circuitry, an internal RAM memory supplies the input signal to the reservoir network every evaluation period corresponding to N_c clock cycles. The resulting reservoir states (mean firing rate of the individual spiking neurons, x_k, $k = 1 \ldots N$) are measured at the end of each evaluation period (by that time, the result of the integrated stochastic computations is already available). Their values are stored in memory, which can be finally displayed using the signal logic analyzer.

The training (assessment of the output layer optimal weights) consists of a linear regression of the desired teacher output (to be predicted, y_{teach}) on the reservoir states (x_k, $k = 1 \ldots N$) as described in sec. 2.3.1.3. It has been carried out using directly the experimental data provided by the FPGA for the training set. Finally, the output weights resulting from the training process were employed to calculate the network's final readout as a linear combination of the experimental neuron states, which were read with a precision of 16 bits. The network's output layer was computed by software. The readout values obtained for the test set are used to evaluate the system's performance for each particular task as an error between the desired and the network's predicted output.

The required hardware resources to implement the proposed LSM employed for the three examples are presented in Tab. 5.1. The measurements of the spent hardware resources have been taken, as in chapter 4, for the reservoir network circuit that includes all the neuron states as output signals. It can be observed that the present implementation requires a 17% less silicon area than that of chapter 4. Therefore, the simplified nonlinear function implemented through the OR gate allows a significant hardware resource saving. It must be noticed that the P2B blocks in the present implementation require more logic elements since they include 20-bit counters and

registers ($n = 20$) while those of previous chapter used $n = 16$. In addition, some additional hardware is required for the circuit presented here due to the fact that the network outputs have been given a precision of 16 bits but they were limited to 8 bits in the design of chapter 4.

Logic elements	1812 (1.6%)
Registers	824 (0.7%)

Table 5.1.: Hardware resource utilization of the Altera Cyclone IV FPGA for a 20-unit LSM implementation using the proposed stochastic neuron design.

It is worth highlighting that the proposed methodology can be used to implement massive reservoir networks in medium and even low cost FPGA devices.

5.4.1. A simple forecasting example

As an example of functionality of the proposed methodology, a first simple forecasting task was selected, which consists in the one-step ahead prediction of a periodic function:

$$u(t) = sin(\frac{2\pi t}{100})\, sin(\frac{8\pi t}{100}) \tag{5.5}$$

where t represents the time step of the system so that every T_{eval} a new value of $u(t)$ is provided to the neural reservoir. This input signal is represented in Fig. 5.8 along with the associated dynamics of some of the nodes in the reservoir. The neuron states, which oscillate according to the input stimuli, are used to calculate the final network's prediction output. Results are shown in Fig. 5.9, corresponding to the one-step ahead values of the input signal. A good match can be observed.

For this time-series experiment, 500 time steps were employed. The first 50 steps were neglected (initial transient response), the next 350 values were used for training and the last 100 for testing. The experimental error for the test set was found to be $NMSE = 0.001$.

5.4.2. Prediction of real-life sea clutter radar returns

For a further validation of the proposed methodology, I study the prediction of a noisy real-life signal presenting an irregular behavior. The target signal is the sea clutter data; that is, the radar back-scatter from an ocean surface, collected by the McMaster University IPIX radar ([BC01a]). The IPIX radar was originally developed for iceberg detection and its measured data can be used to test algorithms

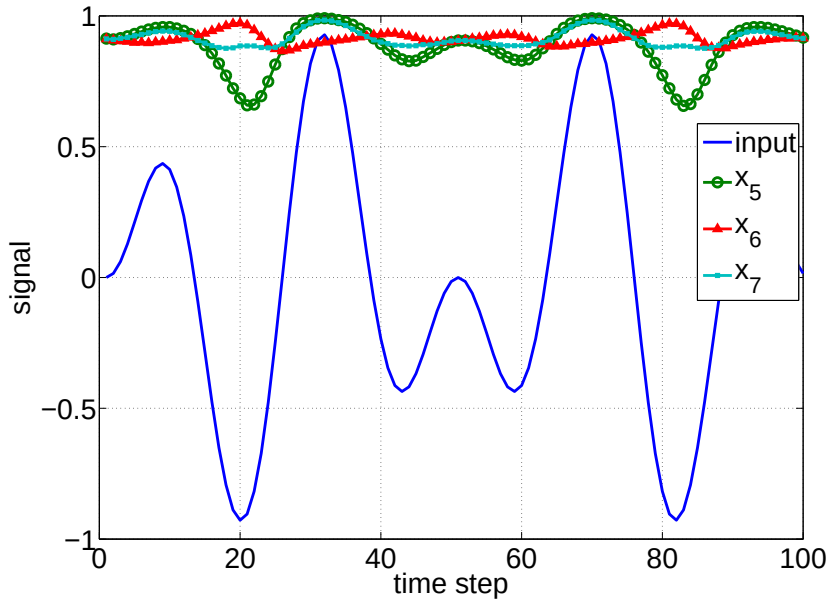


Figure 5.8.: Traces of three selected neurons from the reservoir (symbols) along with the input signal driving the system (continuous line).

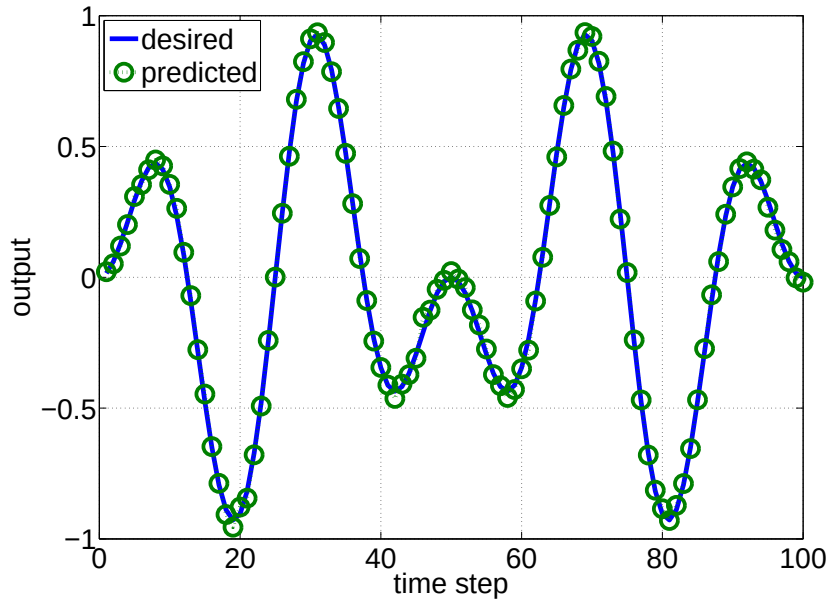


Figure 5.9.: Desired signal to forecast (continuous line) and reservoir's predicted values (circles).

aimed at the intelligent detection of small objects in sea clutter. A precise prediction of the sea clutter data can be used for this object-detection purpose. Therefore, the goal of the present task is to predict this signal one (or several) time steps in the future.

The same data set used in other studies ([XYH07], [RT11], [DSS+12]), has been employed here. It is termed the low sea state and corresponds to an average wave height of 0.8 meters. The output of the radar demodulator is two-dimensional (out_I and out_Q, which correspond to the in-phase and in-quadrature components). However, as in previous studies using this data ([XYH07], [RT11]), the radar signal has been reduced to one dimension ($out = \sqrt{out_I^2 + out_Q^2}$). The first 1000 samples of the sea clutter data were used to train our 20-node reservoir (with a washout time of 150 points), and the next 1000 samples as test set.

The prediction task has been performed for one and for five steps ahead. The reservoir predictions are shown in Fig. 5.10 for both cases. The error was found to be $NMSE = 0.005$ for the one-step experiment and $NMSE = 0.075$ for the five-step case.

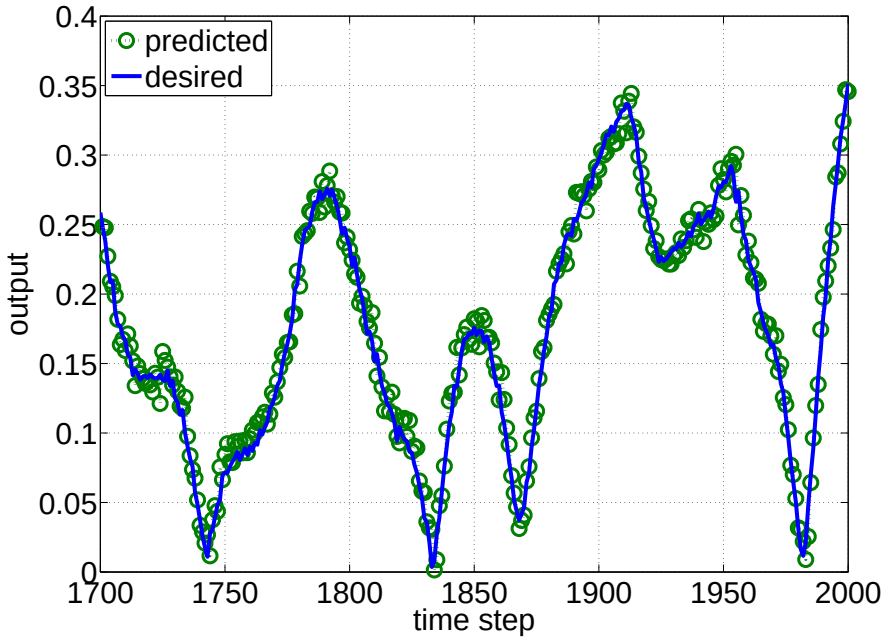
As a reference for comparison, a conventional software implementation of an ESN with cyclic topology using conventional sigmoid neurons ($tanh$ function) provided the next results for the same data set: $NMSE = 0.002$ for the one-step experiment and $NMSE = 0.033$ for the five-step case. A very simple approach that performs the one-step ahead predictions directly using the current step values (i.e., $u(t+1) = u(t)$), provides results with an error $NMSE = 0.007$.

5.4.3. Chaotic time-series prediction task

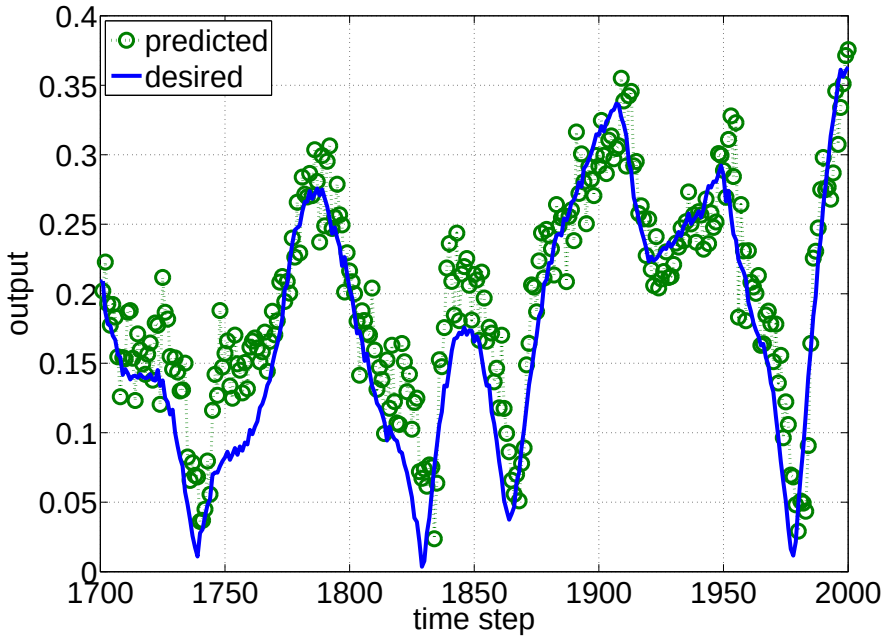
Finally, the proposed SC-based reservoir network has been tested through the more complex Santa Fe laser time-series prediction task (already used in chapter 3 and chapter 4). For the experiment, 3000 samples from the original data set ([WG15]) have been used, the first 2000 for training (disregarding an initial washout of 150 samples), and the remaining 1000 for testing.

Fig. 5.11 shows how the network performance (measured through the NMSE) is examined as a function of the configuration parameters r and v through numerical simulations (that approximately emulate the hardware implementation) to find the configuration of the stochastic reservoir network providing optimum results prior to configuring and experimentally testing the network in hardware.

The error of the hardware-based predictions was found to be $NMSE = 0.116$. Note that this result is similar to that estimated for the SC-based ESN using sigmoid neurons presented in chapter 4 ($NMSE = 0.128$ when using an evaluation period $T_{eval} = 2^{18}\, T_{clk}$).



(a)



(b)

Figure 5.10.: Targeted sea clutter data (continuous line) together with one-step (a) and five-step (b) ahead experimental predictions (circles).

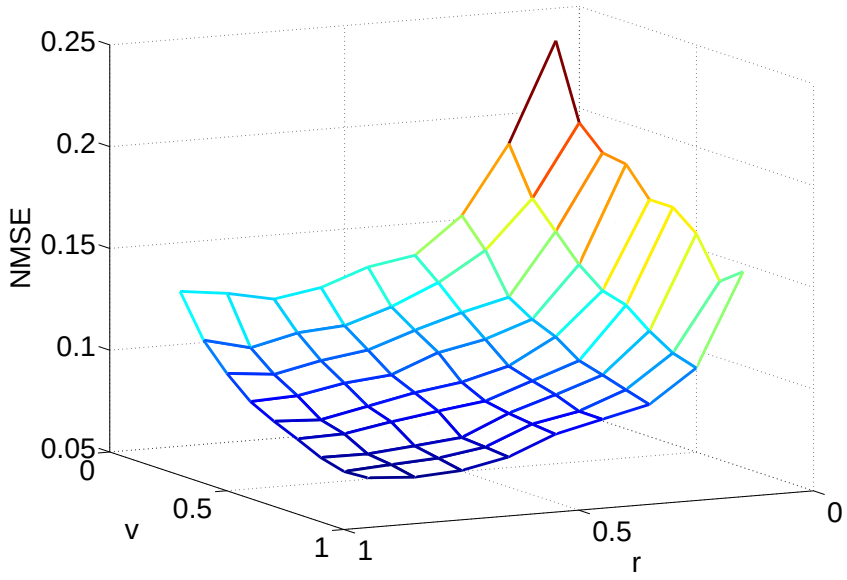


Figure 5.11.: Simulation results for the NMSE in the Santa Fe time-series prediction task to determine the optimum reservoir configuration (parameters r and v).

5.5. Discussion

In this chapter, I have highlighted the similarity of the SC-based continuous-value neuron designs (such as the one presented in chapter 4) with stochastic spiking neuron models. This resemblance results particularly appealing since it allows a compact hardware implementation of a neuron model capable of emulating the basic spiking behavior. More specifically, a variation of the stochastic spiking neuron (SSN, [RCOM14]) has been proposed which allows to make use of the synchronization (de-synchronization) between different units in the network to extend the range of processing capabilities.

The proposed neuron design has been used to implement a reservoir network (LSM) in a medium-cost FPGA. The limited use of hardware resources shows that the present approach is suitable for hardware implementation with low-cost devices. The implemented network has been shown to adequately perform several benchmark time-series prediction tasks. Indeed, the simplified neuron design (with a nonlinear function implemented through a single OR gate) allows considerable hardware resource saving (about 17%) while it provides similar accuracy to that of the SC-based sigmoid unit (as observed for the Santa Fe laser task).

The automated software tool used in chapter 4 to export the stochastic ESN designs to a VHDL hardware description (for any desired network size and weight configuration) can also be conveniently used for the present LSM implementation just by

replacing the sigmoid neuron design with that presented in sec. A.3.

The classical discrete-time (sigmoid) neuron represents an approximation to the spiking neuron when assuming an information codification based on the firing rate. Therefore, an implementation of the classical discrete-time neuron using the SC approach (such as the one proposed in this chapter) can be employed to emulate a spiking node carrying the information on its output signal probability. Despite of being a simplified model, the proposed variant of the SSN retains some general properties of the biological neuron and can be used to describe the transformation of input spike trains into their corresponding output sequences of firing-times through low-cost probabilistic circuitry. A particular benefit of this proposed pseudo-spiking model is the possibility to implement a range of different functions by tuning the correlation/de-correlation between different units, which might improve the network's performance for some particular tasks.

The cyclic architecture employed for the LSM implementations analyzed in this chapter does not allow exploiting the correlation between different neurons (only correlations between the external input and the output of some neurons could be set). Nevertheless, the general reservoir structure with random connections (where each neuron can be connected with any other) enables the possibility to correlate a desired set of neurons and potentially modify the functionality of the network ([RCOM14]). It remains as future research work to analyze whether the use of correlations between different neural units can improve the network's performance for some applications.

The potential applications of the presented LSM implementation are the same ones of the SC-based ESN of chapter 4. Namely, specialized systems where small size, low cost, low power, or soft-error tolerance is required, and limited speed is acceptable. Although it is beyond the scope of this thesis (focused on the engineering application of neural networks), it is worth mentioning that the implementation of biologically plausible reservoir networks (LSMs) can be of high interest to model biological phenomena in the field of neuroscience ([MNM02], [MLM02]).

6. Hardware echo state networks without multipliers

6.1. Overview

The development of efficient circuitry supporting artificial neural networks (ANNs) is crucial to fully exploit the advantages of these systems. However, the hardware realization of massive networks, such as echo state networks (ESNs), is costly mainly due to the requirement of many synapse multipliers as observed in chapter 3. In this chapter, I propose a compact digital hardware design of the ESN based on a very simple concept: reducing the resolution of the synaptic weight values. The limitation of the connection weights to a few discrete values allows simplifying the multiplier design, which can be replaced by a few shift and add operations consuming minimal area. This approach takes advantage of the fixed connectivity structure of the reservoir networks enabling the low cost implementation of large models with hundreds of neurons. The performance of the proposed implementation without multipliers is evaluated for a chaotic time series prediction task (the Santa Fe benchmark). The results show that the hardware-reduced network using low resolution for the weights exhibits practically the same accuracy than the conventional implementation of chapter 3 with full resolution.

6.2. The proposed design without multipliers

Even though a great deal of attention has been directed to the development of efficient designs of the nonlinear sigmoid activation function for digital implementations (as outlined in sec. 3.2.3; see, for example, [BTdC02] and [DCFEB13]), the application of the internal weights present in ANNs sharply constrains the parallel implementation of massive networks in a single chip. Given the large number of products to be implemented between inputs and weights when considering a huge number of synapses, multipliers expend a significant portion of the integrated circuit resources.

In this work, I propose to avoid the use of multipliers by limiting the possible weights to integer powers of two and sums of powers of two so that simple shift registers and adders can be employed instead of multipliers. This technique has

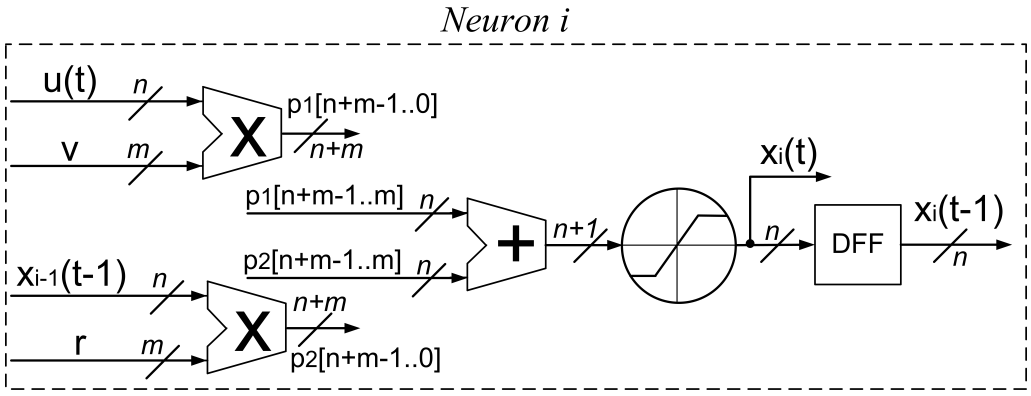
been widely used in other fields such as the digital filter design ([LL88]). In the case of standard ANN implementations that use back-propagation as learning algorithm, the constraint on the weights leads to lower network performance ([MOPU93]). Nonetheless, I show that for reservoir networks with fixed connections, the proposed approach only implies a minor accuracy loss. A similar idea, although not based on the discretization of the connection weights but on the quantization of the neuron's activation function, was proposed for reservoir computing in [BSL10].

As in previous chapters, I focus on the implementation of the echo state network according to the simple cycle reservoir (SCR) topology introduced in sec. 2.3.1.2. Such a deterministic reservoir with cyclic architecture (Fig. 3.13) presents similar performance to the classical random one while it minimizes the number of connections ([RT11]). In addition, the digital design of the SCR network can be easily automated for any number of units since all neurons have the same structure (one connection input from a neighboring neuron and a second one from the input layer) and it is independent on the size of the system. Remind that in the cyclic reservoir, all the connections between internal units have the same weight value r whereas the inputs are connected to the reservoir with a weight that can be either positive or negative (with a 50% probability), but with the same absolute value v. Parameters r and v must be analyzed to find the optimum weight configuration.

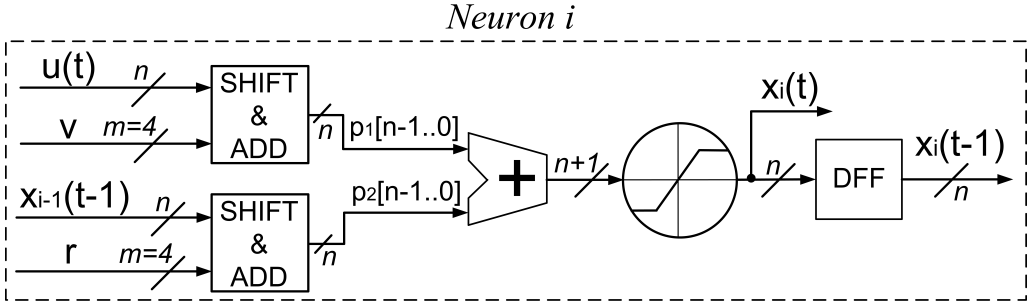
The conventional circuit design of the two-input sigmoid neuron necessary to build the cyclic reservoir is illustrated in Fig. 6.1(a). The first neuron's input value ($u(t)$) refers to the external input signal (to be processed by the network) and the second one ($x_{i-1}(t-1)$) to the output of a neighboring neuron (at the previous time step). A resolution of n bits is given to the input values and of m bits to the weights (with m not necessarily equal to n). The multiplier's output is truncated to n bits taking the most significant bits of the result, but a higher or lower resolution could be employed depending on the desired accuracy. The two's complement notation $s0.n-1$ (that is, one sign bit, no integer bits and $n-1$ bits for the fractional part) is assumed for the neuron inputs (and output) and $s0.m-1$ for the weights so that their values are limited to the $[-1, 1)$ range.

The general implementation of Fig. 6.1(a) using full multipliers allows any weight configuration with a precision of m bits. When the weight resolution is reduced to a few bits, for example $m = 4$, the multipliers can be substituted by shift-and-add blocks as shown in Fig. 6.1(b). The proposed multiplier-less realization enables great hardware saving at the cost of a lower resolution. The shift-and-add block is depicted in Fig. 6.1(c). Basically, it performs a multiplication of the input signal ($u(t)$) by the corresponding weight (v) with a pair of shift registers and an adder. Some additional circuitry is included to perform the negation of the shifted values in case it is necessary. Remind that, in the two's complement notation (sec. 3.2.1), the opposite of a number is obtained by inverting all the bits of that number (with NOT gates) and adding a "1" to the result. A multiplexer is employed to provide either the number that directly results from the shift register or its corresponding opposite value depending on a selection signal. A decoder

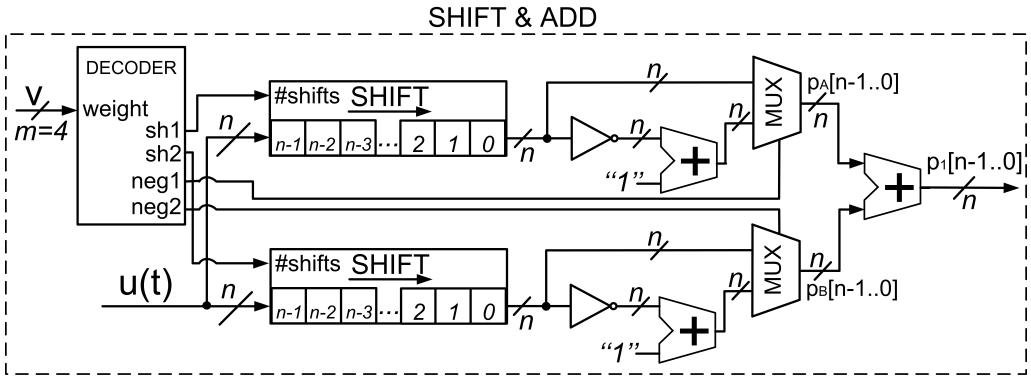
configures the shift registers (with the number of required shifts, *sh1* and *sh2*) and controls the activation of the negations (*neg1* and *neg2*) as a function of the weight value (v).



(a)



(b)



(c)

Figure 6.1.: General circuit design of the neuron (a). Reduced implementation scheme when the weight resolution is limited to a few bits ($m = 4$): the multipliers can be replaced by simple shift-and-add blocks (b). Description of the shift-and-add block (c).

By way of example, a single right shift of the input ($sh1 = 1$) performs a multiplication by 0.5 while two shifts ($sh2 = 2$) are equal to a factor of 0.25. The direct addition (with $neg1 = neg2 = 0$, indicating that no negation of the shifted values is necessary) of these two shifted magnitudes results in a weight $v = 0.75$. The weight value $v = 0.875$ can be implemented by selecting no shifts and no negation for the first shift register ($sh1 = 0$, $neg1 = 0$) and three shifts with a negated output for the second one ($sh2 = 3$, $neg2 = 1$) so that the input signal $u(t)$ is weighted by the factor $v = 1 - 0.125 = 0.875$. A negative factor, for instance $v = -0.5$, may be obtained through the negation of both shifted magnitudes ($neg1 = neg2 = 1$), where each one is obtained with two displacements ($sh1 = sh2 = 2$) so that $v = -0.25 + (-0.25) = -0.5$. Note that, since we are using two's complement signed numbers, the shifting operation must fill the empty positions that result after moving the binary number to the right with 0's when the signal to be shifted ($u(t)$) is positive and with 1's in case it is negative.

The proposed shift-and-add scheme of Fig. 6.1(c) allows a multiplication using any weight value in the range $[-1, 1]$ with steps of 0.125. Therefore, the weight resolution can be considered to be of four bits ($m = 4$). However, it is worth mentioning that this structure with two shift registers can actually be used for many more intermediate values. For example, the weight values 0.03125, 0.0625, 0.1875, 0.3125, 0.4375, 0.5625, 0.9375, etc (as well as their corresponding opposite values) can also be implemented.

The proposed neuron design can be used to build an ESN with cyclic architecture organizing the proposed blocks of Fig. 6.1(b) exactly as shown in Fig. 3.14. A software program has been developed to automatically generate the VHDL hardware description code of the SCR network for any desired number of neurons and weight configuration. The resulting VHDL code can be finally synthesized to an actual hardware implementation. Regarding the nonlinear activation function, the simple piece-wise linear approximation with three segments (sec. 3.2.3) has been used to ensure a compact implementation. More accurate designs of the sigmoid function, such as that of Fig. 3.10, could be used to improve the network's performance. The neuron's input and output resolution has been set to $n = 16$ bits.

The program generating the network's VHDL code has been designed to directly implement the configuration of the shift-and-add blocks that corresponds to the weight values employed for the network. That is to say, the network is not implemented using a generic shift-and-add structure capable of applying any weight value, but the software provides a design that is already configured (with the number of required shifts and negations) according to the selected weights. An example of neuron implementation (VHDL code) is shown for the case of $r = v = 0.875$ in Algorithm 6.1. Note that, in this case, the weighted inputs are obtained summing the result of applying no displacements to the input signal with that corresponding to the negation of the three-position shifted value. The shift register arithmetic operator (*sra*) performs the desired number of displacements of the input signal and fills the empty positions with 0's or 1's depending on whether that signal is posi-

Algorithm 6.1 VHDL code for the neuron design employing shift-and-add operations to perform the multiplications. The weights are set to the same value for both inputs of the neuron: $r = v = 0.875$.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;

ENTITY neuron_without_multipliers IS
  PORT (input1, input2 : IN STD_LOGIC_VECTOR (15 DOWNTO 0); -- s0.15
    clk, reset: IN STD_LOGIC;
    out_x1, out_x1_previous: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)); -- s0.15
END ENTITY neuron_without_multipliers;

ARCHITECTURE behavior OF neuron_without_multipliers IS
BEGIN

  component ffD_16b IS -- 16-bit register
   PORT (input: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
     clk, reset: IN STD_LOGIC;
     output: OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
  END component;

  component f_tanh_aprox_3_segments IS  -- the activation function
   PORT ( x: IN STD_LOGIC_VECTOR (16 DOWNTO 0); -- s1.15
     f: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)); -- s0.15
  END component;

SIGNAL prod11, prod12, x1, x1_previous: STD_LOGIC_VECTOR (15 DOWNTO 0); -- s0.15
SIGNAL prod11b, prod12b, sum1: STD_LOGIC_VECTOR (16 DOWNTO 0); -- s1.15

-- first product (input1 * v)
prod11(15 DOWNTO 0) <= to_stdlogicvector(to_bitvector(input1(15 DOWNTO 0)) sra 0) +
    NOT (to_stdlogicvector(to_bitvector(input1(15 DOWNTO 0)) sra 3)) + '1';

-- second product (input2 * r)
prod12(15 DOWNTO 0) <= to_stdlogicvector(to_bitvector(input2(15 DOWNTO 0)) sra 0) +
    NOT (to_stdlogicvector(to_bitvector(input2(15 DOWNTO 0)) sra 3)) + '1';

-- conversion of prod11 and prod12 from s0.15 to s1.15 notation
prod11b(16) <= prod11(15);
prod11b(15 DOWNTO 0) <= "0" & prod11(14 DOWNTO 0) when (prod11(15)='0') else
                        "1" & prod11(14 DOWNTO 0);

prod12b(16) <= prod12(15);
prod12b(15 DOWNTO 0) <= "0" & prod12(14 DOWNTO 0) when (prod12(15)='0') else
                        "1" & prod12(14 DOWNTO 0);

-- addition of the two previous terms
sum1 <= prod11b + prod12b;

-- assessment of the activation function
f_tanh1: f_tanh_aprox_3_segments PORT MAP(sum1,x1);

-- the 16-bit register holds the neuron output to be used in the next time step
ff1: ffD_16b PORT MAP(x1, clk, reset, x1_previous);

out_x1 <= x1;
out_x1_previous <= x1_previous;

END behavior;
```

tive or negative, respectively. The components employed in the neuron design are a 16-bit register (*ffD_16b*, used to hold the value of the neuron state, $x1$, so that it can be used by another neuron on the next time step), and the nonlinear function block, *f_tanh_aprox_3_segments* described in Algorithm 3.2.

For the case of weight values corresponding to integer powers of two (0.5, 0.25, 0.125, ...) a single shift operation (instead of two) is sufficient. This is illustrated in Algorithm 6.2, where the weight for the first input is set to v=-0.5 and the second one to r=0.75.

Algorithm 6.2 VHDL code for the "multiplier-less" neuron design when the weights are set to $v = -0.5$ and $r = 0.75$.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;

ENTITY neuron_without_multipliers IS
  PORT (input1, input2 : IN STD_LOGIC_VECTOR (15 DOWNTO 0); -- s0.15
    clk, reset: IN STD_LOGIC;
    out_x1, out_x1_previous: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)); -- s0.15
END ENTITY neuron_without_multipliers;

ARCHITECTURE behavior OF neuron_without_multipliers IS
BEGIN

...

-- first product (input1 * v)
prod11(15 DOWNTO 0) <=
  NOT (to_stdlogicvector(to_bitvector(input1(15 DOWNTO 0)) sra 1)) + '1';

-- second product (input2 * r)
prod12(15 DOWNTO 0) <=
  to_stdlogicvector(to_bitvector(input2(15 DOWNTO 0)) sra 1) +
  to_stdlogicvector(to_bitvector(input2(15 DOWNTO 0)) sra 2);

...

END behavior;
```

An example of the VHDL code describing the whole SCR hardware realization is shown in sec. A.4 of Appendix A for the case of $N = 50$ neurons configured with weight values $v = \pm 0.875$ and $r = 0.875$. A network implemented by means of the present approach, based on the use of low-resolution weight parameters, consumes a higher or lower number of hardware resources depending on the selected values of the weights. For instance, a multiplication by a factor of 0.875 (using two shift registers and a negation) requires more logic elements than that corresponding to

a factor of 0.5 (which only needs a shift register). Therefore, the resources of the worst possible configuration must be considered when evaluating the requirements of the network implementation.

6.3. Experimental results

The VHDL-encoded reservoir network obtained according to the proposed methodology has been synthesized on the medium cost Altera Cyclone IV FPGA. The performance of the system has been tested for the Santa Fe time-series prediction benchmark (introduced in sec. 2.3.4.2). A total of 4000 samples of the original laser data set [WG15] are employed; the first 2000 for training, the next 1000 for validation, and the remaining 1000 for testing.

As in chapter 3, a numerical model of the reservoir network hardware implementation, which truncates the resolution of the variables according to the digital design, is employed for training the system (following the standard procedure for ESNs, sec. 2.3.1.3; that is, a linear regression of the desired output on the reservoir states). Such numerical model has been observed to exactly reproduce the FPGA results. That is, a perfect agreement (with zero error) was found between the experimental and the simulated neuron states. This software implementation is also used to determine the configuration parameters that yield the best performance of the system for the validation set. Finally, once the prediction error has been scanned for all the possible weight configurations (values of r and v), the hardware realization is set up with the optimum weights and evaluated with the test set.

Fig. 6.2 shows how the network performance (prediction error, NMSE) is examined for the validation set as a function of the configuration parameters r and v. It can be observed that the discrete weight values allowed by the proposed multiplier-less approach ensure errors that are in close proximity to the best result using conventional multipliers. The constrained values used for r and v are between 0 and 1 with steps of 0.125.

To perform the experimental measurements of the proposed "multiplier-less" ESN implementation configured with the optimum weight parameters r and v, an FPGA's internal RAM memory supplies the input signal to the reservoir network every time step (a single clock cycle). The resulting network outputs (individual neuron states) are monitored using the signal logic analyzer.

Finally, the network's output layer is computed by software as a linear combination of the experimental neuron states, which are read with a precision of 12 bits. The resulting experimental predictions are used to calculate the network's performance as the error between the estimated and targeted values. The hardware implementation test performance results for several sizes of the reservoir are displayed in Fig. 6.3. The results obtained for the present design based on the use of low-resolution weights (multiplier-less approach) are compared to those for the conventional realization of

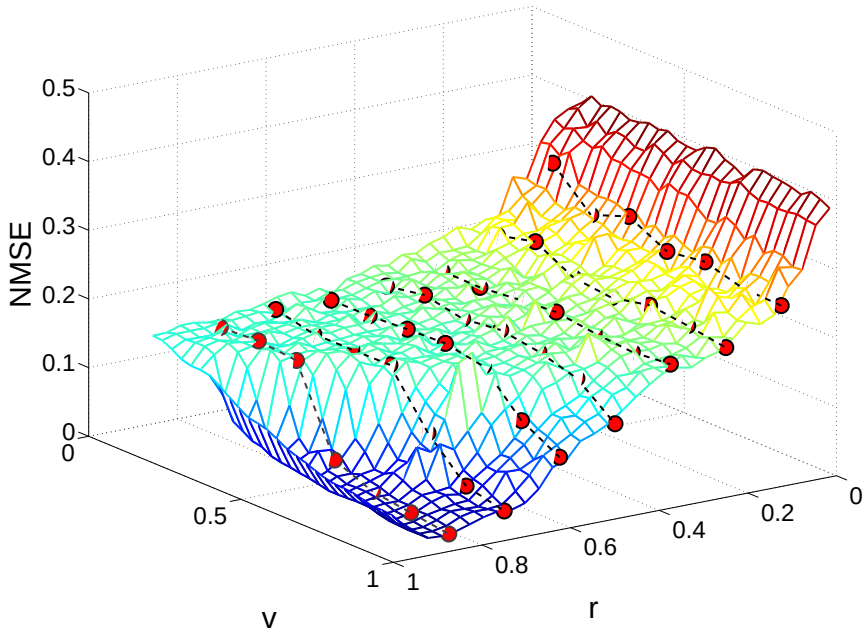


Figure 6.2.: Performance (NMSE) of the ESN implementation with 200 neurons in the Santa Fe time-series prediction task as a function of the weight parameters r and v. The surface is taken from a general realization while the points represent the possible discrete values to which the proposed approach is limited.

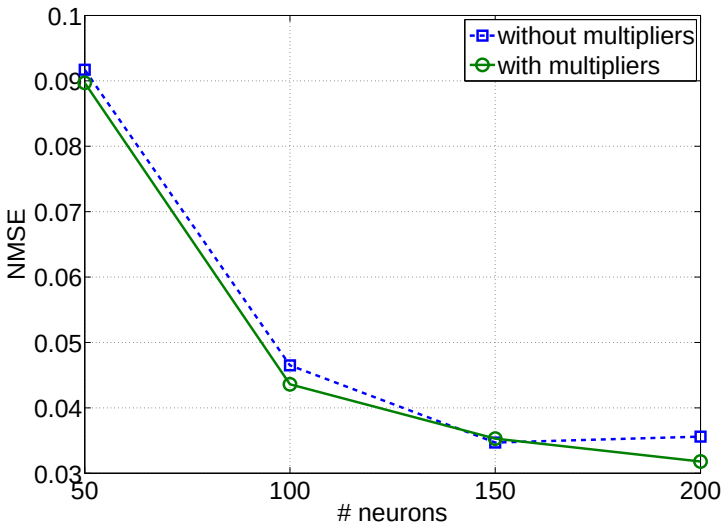


Figure 6.3.: Performance (NMSE) in the time-series prediction task as a function of the reservoir size for a general ESN hardware implementation (with high weight resolution, $m = 16$) and for the proposed realization without multipliers (using limited resolution, $m = 4$).

chapter 3 using high-resolution parameters. Only a slight loss of accuracy is observed in the case of the multiplier-less reservoir being 0.0038 the maximum measured difference in the NMSE. Indeed, in the case of the network with $N = 150$ neurons, the approach without multipliers slightly outperforms the conventional one. This is due to the fact that the network's performance may differ to a small extent for the validation and test set, and therefore the configuration selected for the validation data using low-resolution weights may result (by chance) better than that determined with high-resolution parameters.

The hardware resource utilization for the multiplier-less approach is presented in Tab. 6.1. These results correspond to the network configuration using the weight parameters $v = \pm 0.875$ and $r = 0.875$, which has been observed to be the one requiring the highest number of logic elements. As in chapter 3, the measurements of the spent hardware resources have been taken considering a network circuit with a single output signal corresponding to one of the neuron states (the rest of the states are assigned to intermediate signals). Fig. 6.4 compares the logic elements required by the proposed approach without multipliers with those of the conventional implementation of chapter 3. The proposed design requires the order of 7 times less chip area than the standard realization, thus allowing the massive implementation of reservoir networks using low-cost FPGAs.

It is worth noting that the present implementation allows the parallel computation of all neuron outputs in a single clock cycle (20 ns for a typical clock frequency of 50 MHz).

N (neurons)	**50**	**100**	**150**	**200**
Logic elements	2497 (2.2%)	4947 (4.3%)	7397 (6.5%)	9847 (8.6%)
Registers	800 (0.7%)	1600 (1.4%)	2400 (2.1%)	3200 (2.8%)

Table 6.1.: Hardware resource utilization of the Altera Cyclone IV FPGA for the proposed "multiplier-less" ESN implementation.

6.4. Discussion

In this chapter, I have proposed a simple hardware design of the ESN based on the use of low-resolution weight values. This allows performing the multiplications with shift-and-add blocks that consume minimal area. As a result, this methodology makes possible the implementation of fully parallel reservoir networks with hundreds of neurons using few hardware resources. For instance, an ESN with 200 units requires a approximate number of 10000 logic elements, which is suitable for a low cost implementation (using FPGA devices such as the Cyclone III EP3C16 or the EP3C25, [web16d]).

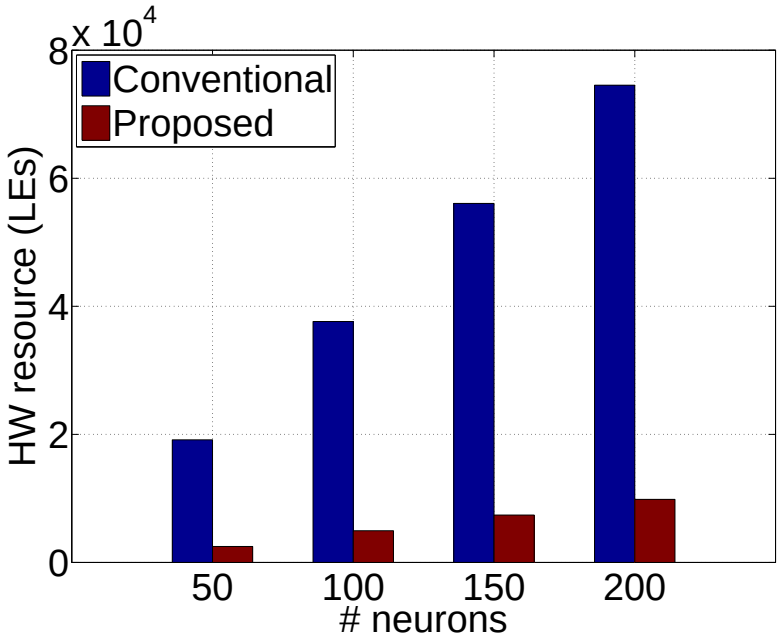


Figure 6.4.: Spent hardware resources of the Cyclone IV FPGA for different sizes of the reservoir using a reduced weight resolution ($m = 4$, implemented with shift-and-add operations) compared to those of the full-resolution case ($m = 16$, implemented with conventional multipliers).

As a demonstration of the validity of the present approach, large reservoir networks (with sizes ranging from 50 to 200 units) have been implemented within a FPGA and evaluated for the Santa Fe time series prediction task. The results show that the hardware-reduced network using low resolution for the weights exhibits practically the same accuracy than the conventional implementation of chapter 3 with full resolution.

To sum up, I have proposed to take advantage of the fixed connectivity structure of the reservoir network (in particular, of the cyclic SCR architecture) by setting the weights to discrete values that are favorable for the hardware implementation. As expected, the limitation of the internal weights (those within the reservoir or connecting the input layer to the reservoir) to powers of two and sums of powers of two has been observed to have little impact on the system's performance. Therefore, the present approach enables the implementation of large reservoir networks with minimal hardware at the cost of a minimum loss of accuracy. It must be noticed, however, that the output layer weights (contrary to the internal ones) must be applied with high resolution for the desired output function to be effectively performed.

It is worth highlighting that a tool (software program) has been developed to automatically generate the hardware structures (such as the shift-and-add blocks, described through a VHDL code) required for any desired configuration of the weights

and size of the reservoir network. This may accelerate the usually long design process of the hardware neural network for a specific application.

The proposed system is suited to supporting the fast processing of temporal information (such as, for example, the processing of video camera pixels, [JWW15]) and also to realize specialized systems implementing computational intelligence techniques and requiring low cost and low power consumption (such as in robotics, wireless sensor networks, predictive controllers, or medical monitoring applications). In addition, it is worth mentioning that the parallel processing capability of the present design (all nodes are computed simultaneously) endows the system with fault-tolerance (it can continue to function, though with reduced performance, in the presence of faults in some components), which may be interesting for safety critical applications.

7. Hardware implementation of delay-based echo state networks

7.1. Overview

In the preceding chapters, I have proposed the use of different approaches to efficiently implement concurrent echo state networks (ESNs) in digital hardware. Stochastic computing (SC) has been viewed as an appealing technique that makes possible the implementation of complex functions using simple circuitry. The reduction of the hardware requirements in SC comes at the cost of the need for long evaluation periods, which leads to systems with limited processing speed. The concept proposed in this chapter consists in sequentializing the operation of the network so that a single neuron (evaluated at different temporal positions) is sufficient to emulate the whole network design. Similarly to SC, this approach relaxes the hardware requirements at the cost of a longer computation time. However, in this case, the implementation cannot benefit from the network's inherent parallelism.

The sequential implementation of reservoir networks with a cyclic structure is the notion behind delay-based RC systems. In this paradigm, the current output of the single neuron is given in terms of that value at previous times. This is usually computed as the solution of a nonlinear delay differential equation (DDE) as described in sec. 2.3.3. Here, I use the sigmoid function as non-linearity (that is, a "static" node is employed instead of a dynamical one) so that the delay-based design replicates an ESN. The proposed digital circuit for such a sequentially-operated network is described and tested for the Santa Fe benchmark task. The resulting system requires few hardware resources, which makes it suitable for a low-cost implementation. Despite the reduced computation speed compared to a parallel implementation scheme, the system is still compatible with numerous real-life applications.

7.2. The proposed delay-based design

7.2.1. Introduction

The present design is based on the sequential implementation of the ESN with cyclic topology (the SCR architecture). The SCR structure has been introduced in

sec. 2.3.1.2 and used in the previous chapters for a parallel implementation of the ESN. That is, in the previous network implementations, the design is composed of a number of separate blocks representing the individual neurons that compute their output simultaneously. A SCR network where all individual nodes simultaneously process the incoming signal $u(t)$ in a single time step is depicted in Fig. 7.1. The distributed computing in ANNs offers advantages in terms of reliability and processing speed that can be exploited in hardware. Here, however, I propose a scheme where the neuron outputs are computed one after each other. In this approach, the whole recurrent network is substituted by a single unit that calculates the network outputs at different times. Such a serial network design highly relaxes the hardware requirements.

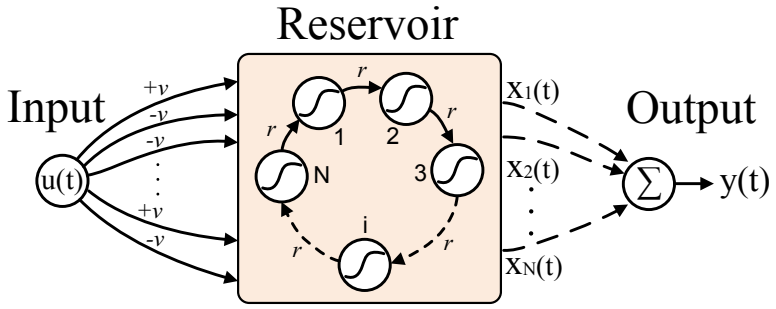


Figure 7.1.: Simple cycle reservoir (SCR) network where all nodes are computed in parallel.

The general concept of emulating the SCR network with a single node by serializing the computation of each neuron in the network is illustrated in Fig. 7.2. We define N equidistant points in the reservoir, separated in time by $\theta = \tau/N$ within one delay interval of length τ. We refer to these N equidistant points as "virtual nodes" since they play an analogous role to the nodes in the traditional concurrent reservoir. The values of the delayed variable x at each of the N points define the states of the virtual nodes ($x(\theta)$, $x(2\theta)$ ... $x(N\theta)$). That is, the neuron outputs that result from the nonlinear transformation of the input signal at a given time.

The input signal to be processed is represented as a time-varying scalar variable, but a vector of any dimension can be used to drive the system. The feeding to the individual virtual nodes is achieved by time multiplexing the input signal. This pre-processing is illustrated in Fig. 7.2. Firstly, the input stream ($u(t)$) undergoes a sample and hold operation to define a stream that is constant during one delay interval τ before it is updated with the next sample in the input signal. Then, a random mask vector is applied over the resulting signal $I(t)$. This mask emulates the random weights from the input layer to the reservoir in the concurrent RC design (that is, the vector of weights $\mathbf{v} = (v_1, v_2, \ldots v_N)^T$). The multiplication of the mask ($\mathbf{M} = \mathbf{v}$) with the sample of the input signal $I(t)$ at a certain time t_0 results in an N-dimensional vector ($\mathbf{M}\, I(t_0)$) that represents the temporal input sequence, $J(t)$,

within the interval $(t_0,\, t_0 + \tau]$ with time steps separated by θ. Each virtual node is updated every time τ.

For the SCR architecture, all the quantities in the vector of input weights have the same absolute value and a sign that varies randomly (with equal probability). For example, as illustrated in Fig. 7.1, the "mask" may take the form $\mathbf{M} = (+v,\, -v,\, -v,\, \ldots\, +v,\, -v)^T$. In the case of a multidimensional input signal ($\mathbf{u}(t)$), the mask consists in a matrix of such randomly chosen values (instead of a single vector) so that its application over the multidimensional input at a given time ($\mathbf{M}\,\mathbf{I}(t_0)$) again provides an N-dimensional vector representing the temporal input sequence, $J(t)$, within the interval $(t_0,\, t_0 + \tau]$ with time steps separated by θ.

Note that the values of the delayed variable x at each time $t_0 + i\theta$ (with $i = 1,\, \ldots\, N$) correspond to the outputs of the individual neurons expressed as $x_1(t_0),\, \ldots\, x_N(t_0)$ in the general notation of the concurrent reservoir.

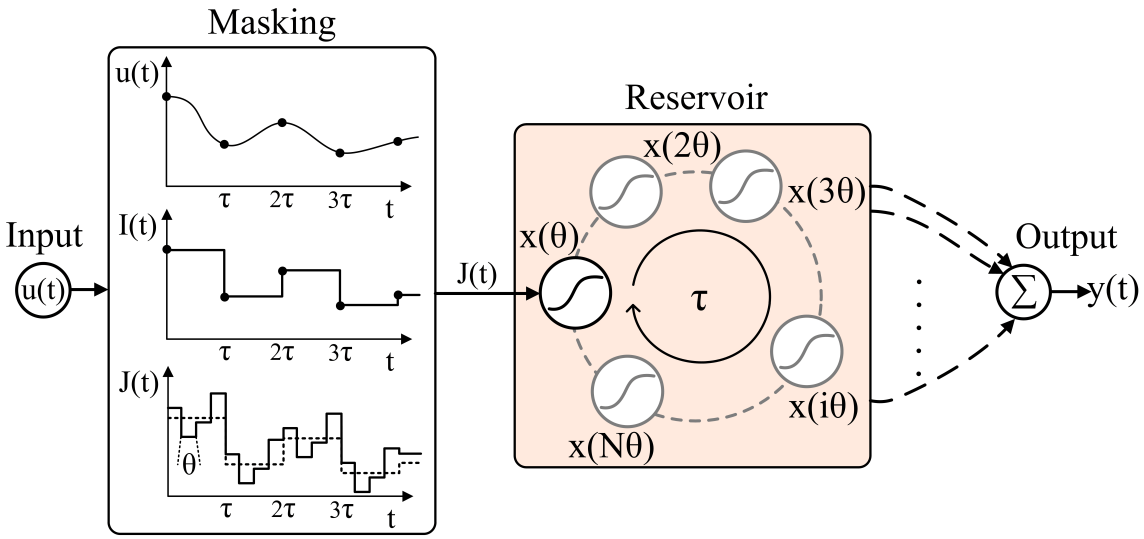


Figure 7.2.: Emulation of the SCR network by means of a single node. The masking process sequentializes the application of the input weights corresponding to each one of the neurons in the network. The output of the single node at different temporal positions ($x(\theta)$, $x(2\theta)$ $\ldots$ $x(N\theta)$) represents the "virtual" neurons of the network (x_1, x_2 $\ldots$ x_N).

The input sequence $J(t)$ drives the single-node reservoir providing a new value to the system each period θ. In addition, the value of the neuron state x at a previous time step is also necessary to compute the current state. Therefore, the neuron state values must be stored for future evaluations of the neuron output. The processing scheme of this "delay-based" system is illustrated in Fig. 7.3 showing the necessary connections between the virtual nodes to reproduce the cyclic structure of Fig. 7.1. As can be appreciated, the neuron output at a given time t ($x(t)$) depends on the value at the previous instant $t - \tau - \theta$ ($x(t - \tau - \theta)$). This is true for all virtual

neurons except the first one, whose result at time t is given in terms of that at time $t-\theta$ ($x(t-\theta)$).

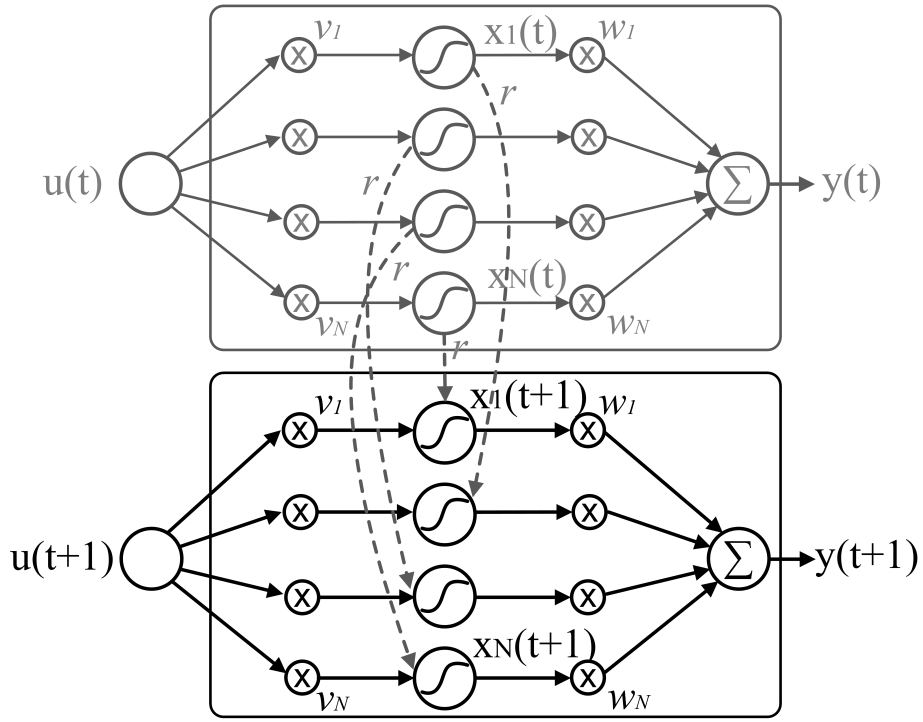


Figure 7.3.: Computing process scheme of the proposed delay-based system.

This way, the connectivity of Fig. 7.1 is reproduced, where the output of a neuron ($x_i(t)$) is computed as a function of the current input sample ($u(t)$) and of the output of the neighboring unit at the previous time step (for the previous input sample, $x_{i-1}(t-1)$). Note that the notation of the concurrent reservoir is employed in Fig. 7.3 ($x_1(t_0)$, ... $x_N(t_0)$) and the delayed period is assumed to be $\tau = 1$. Consequently, $x_1(t)$, $x_2(t)$, $x_3(t)$, ... $x_N(t)$ correspond to $x(t+\theta)$, $x(t+2\theta)$, $x(t+3\theta)$, ... $x(t+N\theta)$ while $x_1(t+1)$, $x_2(t+1)$, ... $x_N(t+1)$ refer to $x(t+\tau+\theta)$, $x(t+\tau+2\theta)$, ... $x(t+\tau+N\theta)$, respectively. The state of each of the virtual nodes ($i = 1, \ldots N$) in the emulated SCR network is calculated according to the following recurrent expression:

$$x_i(t+1) = f[v_i \cdot u(t+1) + r \cdot x_{i-1}(t)] \tag{7.1}$$

where f denotes the sigmoid transfer function, v_i the weight connecting the input stream $[u(t)]$ with the i-th neuron, and r the constant inter-neuronal weight. Note that, as indicated in Fig. 7.3, the value $x_N(t)$ is employed for the calculation of $x_1(t+1)$; that is, $x_0(t)$ corresponds to $x_N(t)$ in equation 7.1.

The output layer is given as a linear combination of the neuron states:

$$y(t+1) = \sum_{i=1}^{N} w_i \cdot x_i(t+1) \tag{7.2}$$

This final readout can also be computed sequentially. Once the output of each virtual node $[x_i(t)]$ is obtained (every time θ), it is multiplied by the corresponding output weight (w_i). The result is added for all the nodes $(i = 1, \ldots N)$ and provided as final readout output $[y(t)]$ after the calculation of the last node $x_N(t)$ (every period τ).

The sequential computing scheme of the ESN with cyclic connections (Fig. 7.3) is notably similar to that of the single dynamical node RC introduced in sec. 2.3.3 (Fig. 2.14). Indeed, the dynamical-node based design can be viewed as a variation of the present implementation where the nonlinear transformation of the node's input signals is realized through the numerical integration of a DDE (instead of directly applying a sigmoid function, [GSMOP12]). In addition, the DDE implements the dependence of the current state $[x(t)]$ on the value at a previous time $[x(t-\tau)]$. The iterated solution of the differential equation (the value for the current integration step is given in terms of the immediately previous one) forces a connection between each virtual node $[x(t)]$ and the immediately previous one $[x(t-\theta)]$ as can be appreciated in Fig. 2.14.

An apparent difference can be observed in the connections present in both designs (Fig. 7.3 and Fig. 2.14): the dynamical system links each node at a time t with that at $t-\tau$ while the "static" design connects the neuron at t with that at $t-\tau-\theta$. Nonetheless, both schemes can be considered equivalent due to the fact that the nodes in the dynamical system (Fig. 2.14) at t indirectly depend on the states at $t-\tau-\theta$ through the connection of the immediately previous node $x(t-\theta)$. Therefore, the information entering each node from previous states can be considered similar in both cases. A formal description of the equivalence between the reservoir map obtained from sampling the solutions of a DDE (equation 2.23, single dynamical node RC) and that corresponding to a discrete-time dynamical system (equation 2.14) employed for ESNs is given in [GHLO15].

The main advantage of the design based on a "static" node (ESN) is that the nonlinear function can be evaluated in one (or a few) clock cycles. On the other hand, the dynamical-node design requires to calculate the evolution of the state variable x over a period of time, which requires a sufficiently large number of intermediate integration steps. Consequently, the dynamical node RC involves a lower processing speed. However, it allows the implementation of more complex non-linear transformations.

7.2.2. Hardware implementation

The proposed circuit design sequentially implementing an ESN with cyclic topology is depicted in Fig. 7.4. Basically, the hardware required to perform the operations of a single neuron unit (7.1) along with a number of registers serving as memory for the delayed variable emulate the whole network. The incoming signals of the single neuron and their corresponding weights need to be properly modified each time step for the outputs corresponding to a number of virtual neurons to be generated. Therefore, a few additional control signals are also necessary.

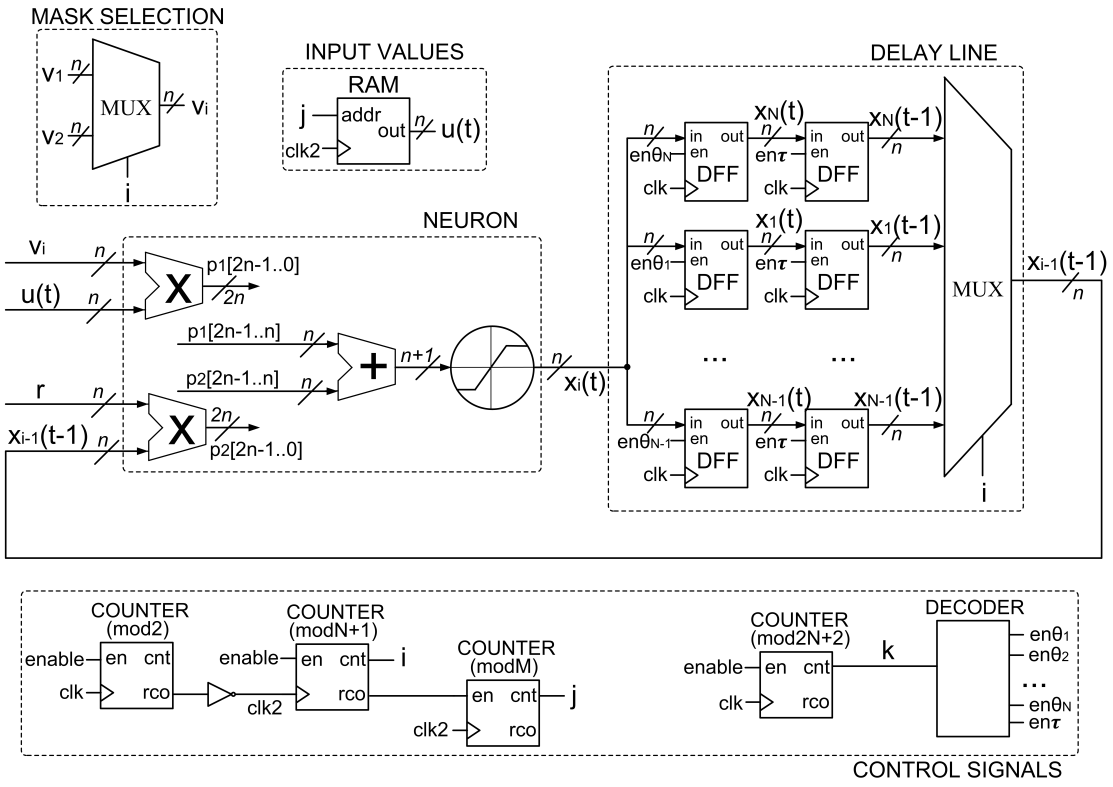


Figure 7.4.: Schematic circuit design of the proposed delay-based ESN implementation.

The simple three-segment piece-wise linear approximation to the sigmoid function (sec. 3.2.3) is employed as transfer function. More accurate approximations could be used to improve the network's performance. A number of 2 clock cycles has been set as necessary for each time step θ (the first cycle is used to assess the neuron output and the second one to store the resulting value in a register). A higher number of clock cycles may be necessary in case of using a more accurate approximation to the sigmoid function. The neuron's input and output resolution has been set to $n = 16$ bits.

A first counter (which is enabled every period θ, set as 2 clock cycles, $\theta = 2\,T_{clk}$)

provides the signal i indicating the node to be evaluated (ranging from $i = 1, \ldots N+1$). The last extra period ($i = N + 1$) is not employed to calculate any neuron's output but to store all the calculated states ($x_1(t)$, $\ldots$ $x_N(t)$) on a group of registers. A second counter is activated every τ period, defined as $\tau = (N + 1)\,\theta$, generating the signal j, which denotes the sample number of the input stream (ranging from $j = 1, \ldots M$). A last counter, which is enabled every clock cycle, provides the signal k (in the range $k = 1, \ldots 2N + 2$) in charge of generating (through a decoder) a number of enabling signals for the registers ($en\tau$, $en\theta_1$, $en\theta_2$, $\ldots$ $en\theta_N$).

The signal i is used to select (by means of a multiplexer) the weight v_i that corresponds to each node's evaluation step. It is also employed to select the state value of a previous time step ($x_{i-1}(t-1)$) necessary for the computation of the current state ($x_i(t)$). Each time step $i\theta$, the multiplexer supplies the state value corresponding to the preceding neuron ($i-1$) at the previous period τ ($t-1$). For the case of $i = 1$, the N-th neuron state $x_N(t-1)$ is provided.

The signal j selects the input sample (stored in an internal RAM memory) that must enter the system each period τ. Consequently, the input value is kept constant throughout the whole sampling time τ. Every time period θ, the value $x_i(t)$ resulting from the neuron block is stored in the corresponding register. There is a pair of registers for each virtual node ($i = 1, \ldots N$). The first stage of registers holds the state values computed at the current step t while the second stage stores the values corresponding to the previous step $t - 1$ (necessary for calculating the neuron's output at the present time). The temporal evolution of the control signals is shown in Fig. 7.5 for the case of $N = 3$. The enabling signals ($en\theta_1$, $\ldots$ $en\theta_N$) habilitate holding the current neuron state at the output of the corresponding register of the first group. When $i = N+1$, the values of the second stage of registers are updated (through signal $en\tau$) for the calculations of the next time period τ ($t + 1$).

As regards the output layer (expression 7.2), it is realized by means of a multiply-accumulate circuit (MAC) as shown in Fig. 7.6. Every time θ, the neuron state value $[x_i(t)]$ is multiplied by the corresponding weight (w_i) and the result (z_i) is added to the previously accumulated value (y_{i-1}). The resulting value (y_i) is re-stored in a register for future accumulations. When the operation has been performed for all nodes (that is, when the index i reaches $i = N + 1$), the final result is stored in a second register (y_j) so that it can be visualized. Afterwards, the first register is reset to an initial accumulation zero value before restarting the multiply-accumulate operations for the next time step j. The evolution of the control signals enabling the habilitating and resetting of the registers ($en\theta$, $en\tau_2$, $en\tau$) is depicted in Fig. 7.7.

The numerical quantities in the network design of Fig. 7.4 are represented according to the two's complement notation $s0.15$ for the input values ($u(t)$), internal reservoir weights (v_i, r) and neuron states ($x_i(t)$) since their values are bound in the $[-1, 1]$ interval. For the output weights (w_i) in the MAC of Fig. 7.6, the $s5.14$ notation is employed so that values out of that range can be used. The notation employed for the final readout (y_j) is $s6.13$, which ensures that overflow does not occur even after

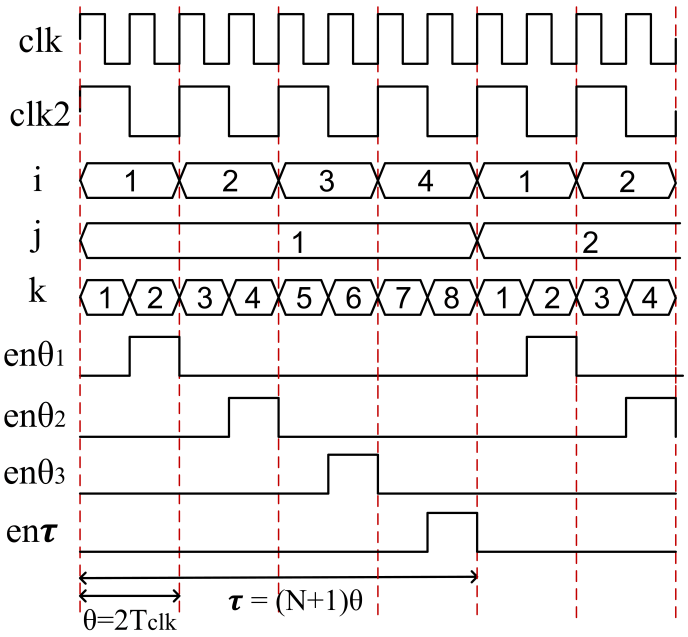


Figure 7.5.: Evolution of the control signals of Fig. 7.4 for an exemplary system using three neurons ($N = 3$). θ denotes the required time to process each virtual node (set to two clock cycles). τ stands for the time needed to process all the nodes in the network.

the accumulated addition of a high number of node states (N).

The hardware resource utilization for the present delay-based approach is presented in Tab. 7.1 for different sizes of the system (N). These results correspond to a network configuration using generic values for the weight parameters v_i and r. That is, the reservoir weights ($v = |v_i|$ and r) are defined as input parameters that may take any desired value. As in chapter 3, the measurements of the spent hardware resources do not include the output layer and have been taken considering a network circuit with a single output signal corresponding to one of the neuron states (the rest of the states are assigned to intermediate signals). The implementation makes use of 12-bit counters to generate the signals i and k so that the circuit can emulate a number of approximately 2000 neurons. Greater counters can be used in case of requiring a higher number of nodes.

N (neurons)	**50**	**100**	**150**	**200**
Logic elements	3085 (2.7%)	5363 (4.7%)	7597 (6.6%)	9821 (8.6%)
Registers	1626 (1.4%)	3226 (2.8%)	4826 (4.2%)	6426 (5.6%)

Table 7.1.: Hardware resource utilization of the Altera Cyclone IV FPGA for the proposed delay-based ESN implementation.

The use of registers holding the neuron states allows a fast access to the delay

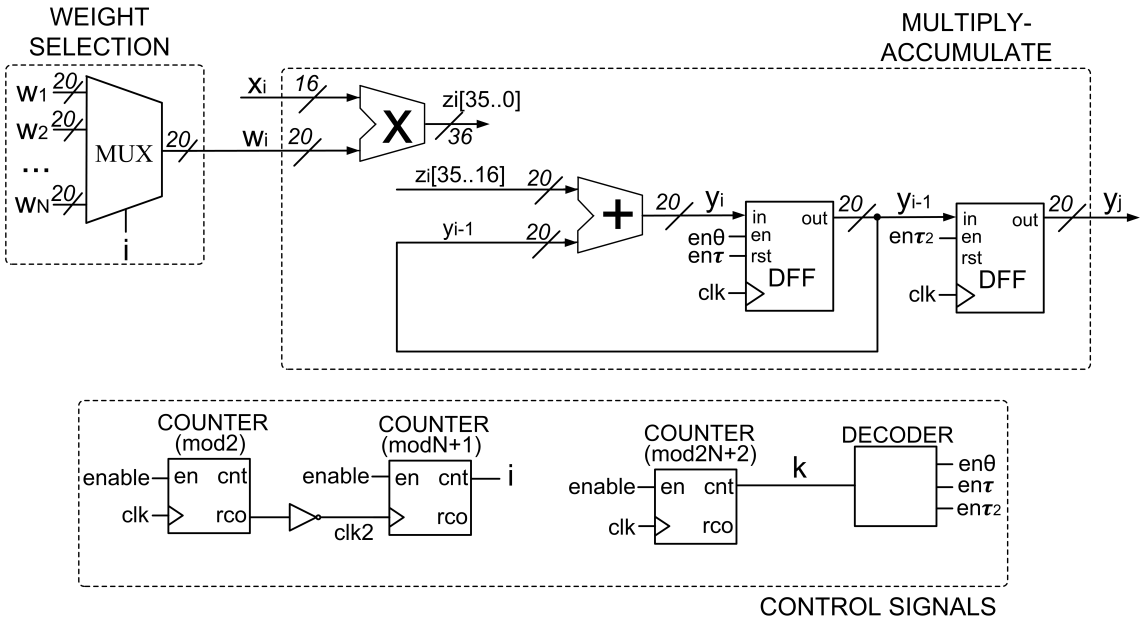


Figure 7.6.: Circuit design of the ESN's output layer (linear combination of the neuron states) realized by means of a multiply-accumulate circuit

line, which enables a system with high processing speed. In particular, the present design, only requires two clock cycles to compute the output of each individual neuron. However, it is worth mentioning that the replacement of the registers by an internal RAM memory (which also ensures a high access speed) would be more efficient significantly reducing the requirement of logic elements.

The time required for the network processing an input sample (τ) in terms of the clock frequency (f_{clk}) and number of nodes (N) is given by 7.3. Tab. 7.2 presents the processing time (τ) and maximum frequency at which data can be processed (f_{max}) for different values of the number of units considering a system's clock frequency of 50 MHz.

$$\tau = 2 \cdot (N + 1) \cdot \frac{1}{f_{clk}} \tag{7.3}$$

A software program has been developed to automatically generate the VHDL code of the proposed delay-based implementation for a network of any desired size N. Indeed, the core of the implementation is not affected by the number of nodes in the system as it calculates the output of a single sigmoid neuron each time step θ, but the block of registers employed as memory for the neuron states as well as the control signals need to be adapted with the reservoir size. An example of the VHDL code describing the SCR hardware realization for 200 neurons (including the output layer) is shown in sec. A.5.

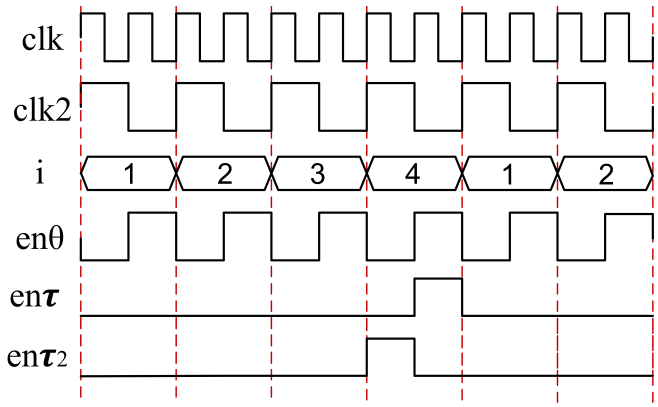


Figure 7.7.: Evolution of the control signals of Fig. 7.6 for an exemplary system using three neurons ($N = 3$).

N (neurons)	τ (μs)	f_{max} **(kHz)**
50	2.04	490.2
100	4.04	247.5
150	6.04	165.6
200	8.04	124.4

Table 7.2.: Processing speed of the proposed FPGA-based ESN implementation as a function of the number of nodes.

7.3. Experimental results

The proposed hardware design has been synthesized on the medium cost FPGA (Altera Cyclone IV). As in the preceding chapters, the system's performance is analyzed for the Santa Fe time-series prediction benchmark (sec. 2.3.4.2) employing 4000 samples of the original laser data set ([WG15], the first 2000 for training, the next 1000 for validation, and the remaining 1000 for testing).

A software program emulating the behavior of the hardware implementation (by truncating the resolution of the variables accordingly to the hardware) is employed for training the system. The standard training procedure for RC (linear regression of the desired output on the reservoir states) assigns an output weight to each virtual node, such that the weighted sum of the states approximates the desired target value as closely as possible. The numerical model employed for training the system exactly reproduces the FPGA experiments. This is shown in Fig. 7.8, where the evolution of some selected neuron states is depicted for both the experimental and numerically simulated system (composed of 200 units).

As in chapter 3, a number of simulations have been carried out for each possible configuration of the reservoir weights (r and v) using the training data to determine the output weights and the validation set to assess the prediction error (NMSE). A

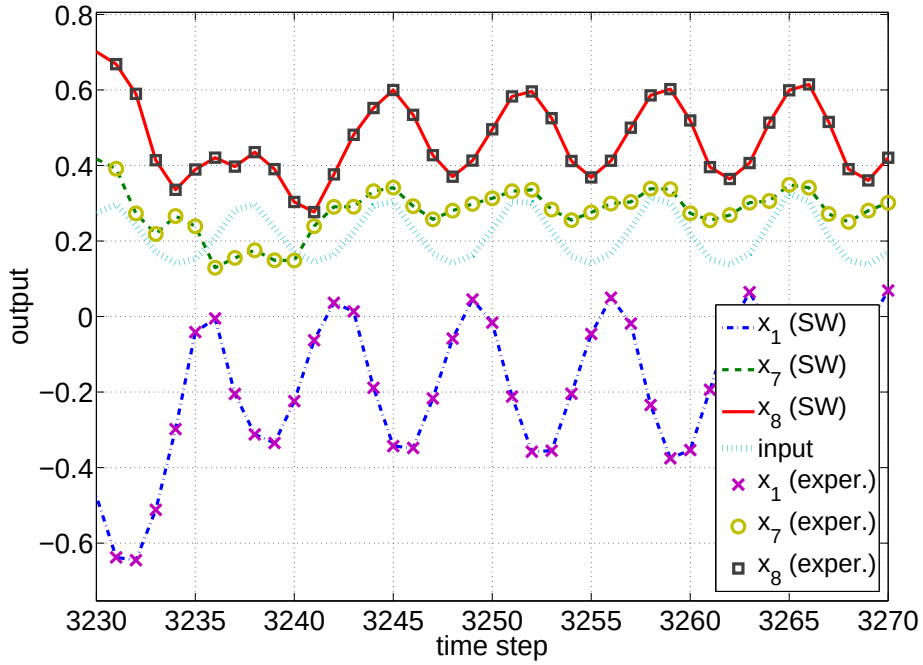


Figure 7.8.: Evolution of some selected neuron states along with the input of the system (Santa Fe laser intensity values). The neuron outputs are depicted for both the software (lines) and experimental hardware realization (symbols). A perfect match can be observed. The graph corresponds to a fragment of the test data set using a network configuration with $N = 200$ neurons, $r = 0.8906$ and $v = 0.99997$.

similar plot to that of Fig. 3.16 is obtained (for each value of the network's size, N), which allows to select the optimum values of r and v.

Finally, the hardware implementation is configured with the optimum weight parameters r and v and experimentally tested using previously unseen input data of the same kind of those used for training (the test set). An FPGA's internal RAM memory supplies the input stream to the reservoir network (a new sample every time period τ). The resulting network output (the final readout, y_j) is monitored using a logic analyzer.

The experimental predictions performed by the 200-unit network are depicted in Fig. 7.9. The test prediction error (NMSE) is presented in Fig. 7.10 for different values of N. The error values obtained for the present serial implementation practically coincide with those of the parallel design of chapter 3. A slight loss of accuracy is due to the fact that the output layer operations have been implemented in hardware with limited resolution for the present serial implementation, but they were performed in software (using a higher resolution) in the case of the parallel design of chapter 3.

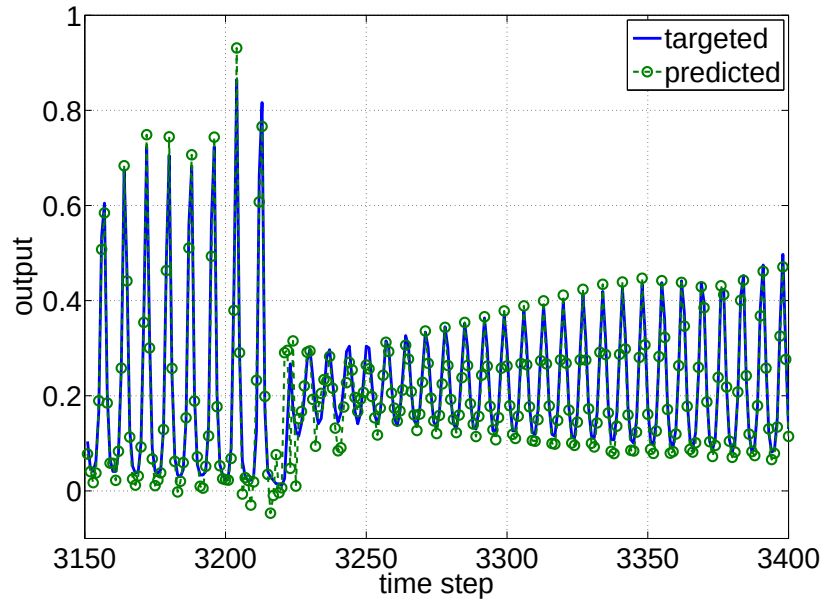


Figure 7.9.: Fragment of the Santa Fe laser time-series: original values and one-step ahead predictions performed by the proposed delay-based ESN implementation with 200 neurons.

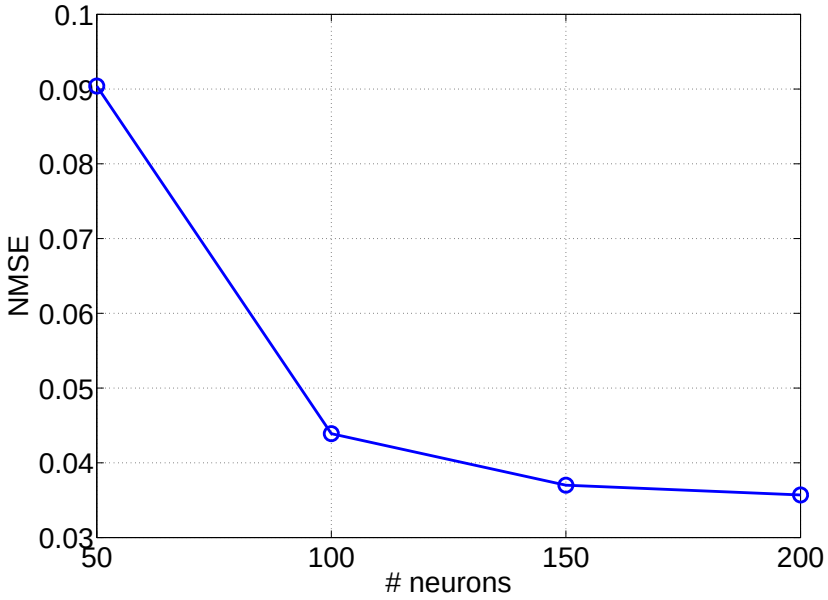


Figure 7.10.: Performance (NMSE) in the time-series prediction task as a function of the reservoir size N for the proposed delay-based hardware ESN.

7.4. Discussion

The proposed digital hardware design of the ESN based on the serial processing of the individual nodes leads to a significant reduction of the hardware resources (number of logic elements) compared to the fully-parallel realization (chapter 3). Approximately, the present implementation requires between six and seven times less area than the conventional concurrent design. Obviously, the sequential design involves a slower throughput. More specifically, the time needed to process all the nodes in the network is increased by a factor equal to the number of neurons. For example, a system with a few hundreds of nodes can process each input sample in a period of the order of $10\mu s$ assuming a clock frequency of 50 MHz.

The organization of the neurons in a cycle (SCR chain structure) simplifies the network implementation compared to the classical ESN with random connections. In the SCR, every neuron is only connected to the preceding one. However, in the classical ESN, the number of connections may vary from one neuron to another. In addition, the neurons to which a unit is connected need to be stated for each individual node. This would make more complex the implementation of the ESN classical random design.

It is worth mentioning that the proposed design could be improved replacing the block of registers implementing the delay line by an internal RAM memory. This would reduce the number of required logic elements. On the other hand, the proposed sequential scheme could be combined with the multiplier-less approach of chapter 6 (selecting a few possible discrete values for the input weights), which would further reduce the hardware requirements.

Regarding the system's accuracy, it could be improved selecting a more accurate approximation for the sigmoid function. As the physical implementation of a single neuron is used to emulate the whole network, the higher number of resources required for a more complex activation function is not as critical as in the case of a parallel design.

The proposed design has been synthesized and tested on a FPGA device. The processing speed of such an FPGA-based implementation (even with a low frequency clock of 50 MHz) is compatible with a number of real-time applications requiring sampling times no lower than 10 μs, such as medical monitoring applications ([EMSFM15]), control units ([LZF06], [ZL08]), or handwriting ([PS00]) and speech ([SDVC07]) recognition systems. Nevertheless, it must be noted that an FPGA-based implementation of the present design is unlikely to be faster than a microprocessor-based implementation given the lower operation frequency and the sequential nature of the proposed delay-based approach.

The design presented here seems particularly suitable for building a low-power low-cost neuromorphic processor chip capable to implement a large neural processing system. The implementation on a specific chip could present a similar processing speed to that of a general-purpose processor and would probably be advantageous

in terms of power consumption. A higher energy efficiency would be expected due to the fact that the proposed design has been tailored for the particular purpose of the ESN realization (using limited resolution for the variables) and the delay line (memory) could be efficiently accessed on the same chip.

In general, the proposed implementation (either in a specific fabricated chip or in a low-cost FPGA) can be of interest for applications requiring the implementation of a machine learning technique and demanding low power consumption, such as mobile/autonomous objects (robotics, wireless sensor networks, monitoring medical devices, medical implants, etc). Such an implementation may also be used as a co-processor supporting software-based computations. For instance, the ESN hardware might be employed to perform handwriting recognition (sec. 9.2) in a PDA allowing to release potential resources on the main processor for other applications.

Compared to the stochastic computing approach, the presented delay-based design allows a faster system (about 100 times faster considering the case of a 16-bit precision for the SC-based operations), a more compact implementation (about three times less logic elements are required) and does not involve any loss in the accuracy of the results. The major shortcoming of the proposed serialized ANN processing seems to be the lack of fault-tolerance in case of errors occurring in any of the system's components.

Finally, it is worth highlighting that the development of the present sequential ESN design has given some insight into the origin of the time-delay RC structure (sec. 2.3.3). Such delay-based dynamical systems essentially reproduce the connectivity of the cyclic ESN through the use of a DDE. The digital implementation of this type of dynamical-node systems is tackled in next chapter.

8. The single dynamical node reservoir computer

8.1. Overview

Previous chapters have been focused on the implementation of echo state networks (ESNs) using digital circuits, which represent the classical reservoir computing (RC) approach based on discrete-time dynamical systems. The present chapter describes a hardware implementation (the first digital realization, to the best of my knowledge) of a time-delay reservoir (TDR) system; that is, a reservoir computer that employs a single nonlinear oscillator with delayed feedback as dynamical node. As introduced in sec. 2.3.3, the TDR emulates the nodes of a recurrent network by sampling the solutions of a delay differential equation (DDE). Consequently, it consists in a serial computation scheme similar to the design of chapter 7 where the output of each virtual node of an ESN is computed as a function of a previous node (i.e., the state of the system at a previous time step) and of the input at current time. In the case of the TDR, however, the nonlinear output of each virtual node is not computed through a direct discrete-time recursive formula, but by finding the solution of a differential equation after an appropriate number of integration steps. As a serial design, the single-dynamical-node approach emulates a network structure with multiple nodes using minimal hardware resources. Nonetheless, this comes at the cost of a slow-down of the information processing compared to a parallel design. The proposed digital implementation is applied to perform a temporal pattern classification task and a chaotic time-series prediction.

8.2. The proposed implementation

The proposed implementation of the single-dynamical-node reservoir computer consists of a self-contained system realized in an Altera DE0 development board, which makes use of the Cyclone III low-cost FPGA ([web16d]). The final solution is hosted within the FPGA chip with the delay line implemented using a RAM memory, but an external SDRAM memory is also employed to store the mask and output weights. The system receives the external input signal and the desired configuration parameters from a PC through an RS-232 serial port. The implementation has been conceived so that it can be operated using different configurations without the need

of reprogramming the device. The values of the configuration parameters just need to be modified in a text file and sent to the system that is adapted to operate according to them. The design was originally aimed at classifying temporal signals, but it can also be employed for time-series prediction or function approximation tasks. For this particular implementation, the design of the different components has been developed using schematic blocks available with the Quartus II software instead of directly describing the system through a hardware description language (VHDL) as proceeded for the digital designs presented in previous chapters.

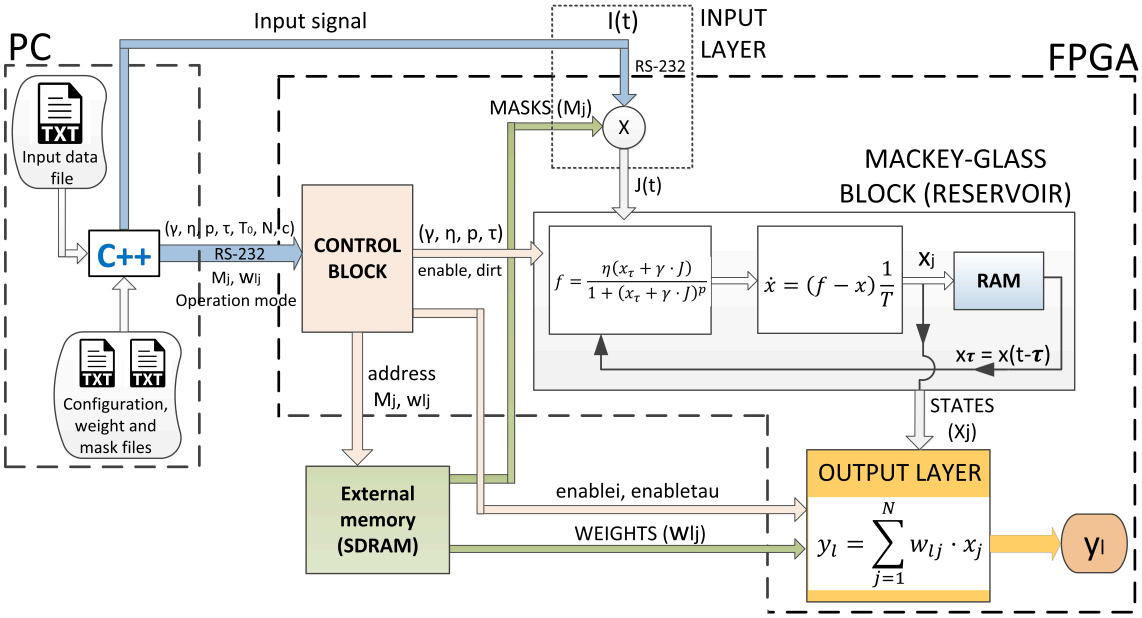


Figure 8.1.: Diagram of the single-dynamical-node RC implementation.

Fig. 8.1 schematically shows the proposed digital implementation. The core of the design is a differential equation solver that emulates the Mackey-Glass oscillator (equations 2.23 and 2.25) as the nonlinear node. This oscillator along with the RAM block represents the delay-based reservoir depicted in Fig. 2.13. The Mackey-Glass node receives the external input ($J(t)$), after masking the signal ($I(t)$) in the input layer. In the global implementation, a control block configures both the dynamical system with the desired parameters as well as the external memory with the mask (M_j) and weight values (w_{lj}) for each node (j) and output class (l). In addition, it states which memory value (*address*) should be provided at each time step. An external C++ program specifies to the control block whether the system must enter into a configuration mode (in which it is arranged with the proper parameters and memory values) or into the operation mode to process the inputs. The configuration parameters, mask, weights, and input values are all stored in external data files. The orders to choose the system's mode and all of the data are sent to the system by the C++ program via the serial port.

Once the system has been configured, it is provided with an input value every delay interval τ, as described in sec. 2.3.3. The input signal is multiplexed in time within each τ through the application of a mask (M_j) in time steps $\theta = \frac{\tau}{N}$ (where N denotes the considered number of virtual nodes) before feeding the dynamical system. The values of the delayed variable x at each time $j\theta$ (representing the output of the individual virtual nodes with $j = 1, ..., N$) are stored in the internal memory of the FPGA (so that they can be used for subsequent iterations of the differential equation) and supplied to the output layer block. Finally, the output layer performs every time θ a multiplication of the state x_j by the corresponding weight w_{lj} for each one of the c output classes ($l = 1, ..., c$). Only one class needs to be considered ($c = 1$) when the system performs a prediction task.

To perform temporal classification, the result of this product is sequentially added for a given number (α) of intervals τ:

$$y_l = \sum_{j=1}^{\alpha \cdot N} w_{lj}\, x_j \tag{8.1}$$

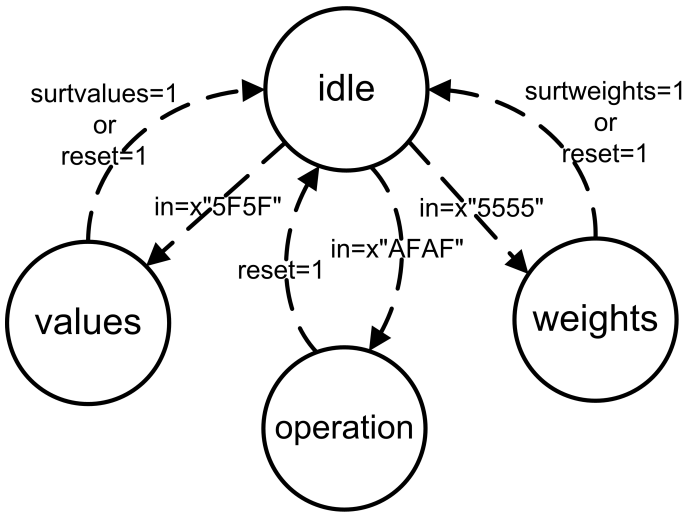
The results of such weighted sum of the states y_l for each category l are compared after α delay intervals. The greatest of these values (y_k) determines the output category ($l = k$) that matches the input signal. That is, our system follows a winner-take-all strategy. The length of the period that is necessary to classify the input signal ($\alpha \cdot \tau$) depends on the particular task, but it is usually set equal to the whole duration of the input sample (e.g., the extent of a spoken digit recording in a speech recognition task). After each classification period $\alpha \cdot \tau$, the output signal y_l is reset so that a new input time trace can be processed. In the case of time-series prediction, the weighted sum of equation 8.1 is computed with $\alpha = 1$, providing a predicted value at each interval τ.

As usual, the numerical quantities are represented in the digital format adopting the fixed-point notation. In particular, for the present implementation, the unsigned notation is used to represent all variables except the output weights, which require an additional sign bit as I describe in sec. 8.2.5. A number of 8 fractional bits and no integer bits are used for the input signal as well as for the mask values and the delayed variable x. Consequently, their working range is limited to the $[0, 1)$ interval. Intermediate variables are given a greater resolution when necessary. The output weights w_{lj} and the final classification outputs y_l are provided with a 16-bit resolution.

In the next subsections, I describe in more detail the design and operation of the different components in the implementation.

8.2.1. The control block

This block determines the operation mode of the system and generates a number of signals that are necessary to carry out each one of the possible functions. It is externally controlled by a PC that sends the orders and required data through the serial port. A finite-state machine (FSM) is employed to select the system's desired operation mode. A diagram of such FSM is illustrated in Fig. 8.2, which shows the transitions between the different states and the true table for the output variables as a function of the state. Three possible states are considered for the machine (apart from the "idle" state of no activity): "values", "weights" and "operation". The "values" mode corresponds to the function of configuring the system with the desired parameters (i.e., the assignation of values for γ, η, p, τ, N, ...) while the "weights" mode enables the loading of the mask and output weight values (for each node and output class) in an external memory. Finally, in the "operation" mode, the system processes the input signal through the Mackey-Glass oscillator and the linear readout.



	outputs		
state	**envalues**	**enopera**	**enweights**
idle	0	0	0
values	1	0	0
operation	0	1	0
weights	0	0	1

Figure 8.2.: Finite-state machine employed to select the system's operation mode: idle, operation, value configuration or weight loading.

The change from one state to another is determined by the sequence of input values that are presented to the system. Such input signal (noted as *in* in the diagram) is provided in a "byte" format, that is, as a series of 8-bit digital numbers. The state changes from "idle" to "values" when the sequence of bytes x"5F" and x"5F"

(where x" · " denotes the hexadecimal representation) arrives to the circuit. After receiving 11 bytes, which correspond to the system's configuration parameters, the variable *surtvalues* is activated and the state moves back to "idle". Similarly, a transition from "idle" to "weights" occurs when the numbers x"55" and x"55" are sequentially received through the serial port. Once in this state, sequences of five bytes are used to indicate the weight and mask values as well as the memory address where they must be stored (the first two bytes represent the address while the other three codify the numerical quantity). To exit this state, the system must receive the following sequence of five bytes: x"00", x"00", x"FF", x"FF", x"FA", after which the variable *surtweights* is automatically activated and the state goes back to "idle". The system enters the "operation" mode when it receives the input sequence x"AF", x"AF". At this point, the quantities that are sent through the serial port correspond to the input signal to be processed and the system will only change to the inactive (idle) state if the reset is enabled. Each value of the input stream must be provided every time interval τ given by the number of clock cycles that the circuit requires to evaluate all the considered virtual nodes.

The output signals of the FSM (*envalues*, *enopera*, *enweights*) are used in the control block (either directly or through other intermediate signals) to enable the operation of the whole system in each different mode. For example, *enweights* is employed as the "read/write" control signal of the memory indicating whether the external memory must save the incoming quantities in a particular location (i.e., "write", enabled when $enweights = 1$) or access the stored values (i.e., "read", enabled when $enweights = 0$).

When the system is in the "operation" mode, the control block generates a number of signals that are necessary for the functioning of the system. For instance, the signal *enable* activates the reading of the output of the differential equation solver (Mackey-Glass block) for a clock cycle once every Δt period, which is given by a certain number (T_0) of clock cycles and corresponds to the integration time step ($\Delta t = T_0\, T_{clk}$). Such period of time within each time step (Δt) is employed to calculate the result of each iteration of the differential equation. In addition, it must be considered that a minimum waiting time of a few clock cycles ($T_0 \sim 5$) before enabling each iteration is necessary to ensure that the external memory has enough time to correctly provide the corresponding value of the mask to the Mackey-Glass block. The parameter T_0 can be conveniently used to adjust the processing rate of the circuit so that it coincides with the rate at which the system receives the external input stream through the serial port. On the other hand, the signal *dirt* indicates the address of the internal RAM memory from which the Mackey-Glass block must read the value of the delayed variable $x(t - \tau)$ (and where the current state, $x(t)$, must be stored for subsequent iterations). The circuit employed to generate these signals is depicted in Fig. 8.3 along with an example of the signal evolution considering an integration step equal to three clock cycles ($T_0 = 3$). As it can be appreciated, the circuit is based on binary counters (using 8 or 16 bits) that count up to a predefined value after which they are restarted. The ripple-carry

output (rco) of the first counter activates the *enable* signal every T_0 clock cycles and enables the second counter, which provides the memory address $dirt[15..0]$. The system's configuration parameter ri defines how many integration steps (Δt) are considered in a delay period τ. Likewise, the control block generates the signals *enablei* and *enabletau* that are enabled for a clock cycle once every time period $\theta = \tau/N$ and τ, respectively. The parameter ti indicates how many clock cycles compose the θ period and n refers to the number of virtual nodes (N). The signal *enablei* activates the capturing of the differential equation state x as a virtual node and sending it to the output layer (classification block) while *enabletau* indicates to the output layer when the computation of all virtual nodes is concluded (for each incoming input value). Note that the counters only operate if $enopera = 1$ (that is, when the system is in the operation mode).

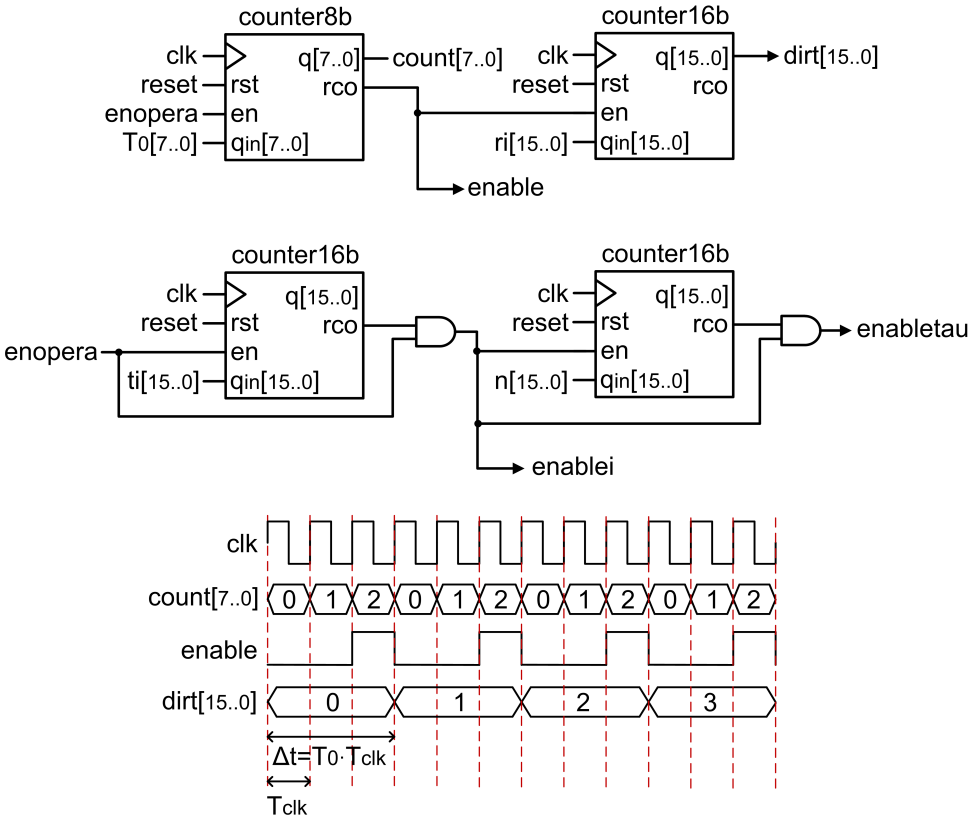


Figure 8.3.: Circuits employed in the control block to generate some necessary signals for the functioning of the system. The evolution of some of these signals is illustrated for the case $T_0 = 3$.

In a similar manner (using counters), a few more signals are generated by the control block to indicate the address (*address*) that the external memory must access at each proper time and to enable the use of the memory values as masks or output weights. The system has been designed so that a new mask value is provided to the input layer at the beginning of each θ period. Afterwards, the memory supplies a

series of c values that correspond to the output weights for each one of the different classification categories. A period of 25 clock cycles is waited before sending each output weight so that the classification block can receive the correct memory value and perform the required operations within this time.

8.2.2. The external C++ program and the configuration files

A C++ program is used to transfer the data from an external PC to the FPGA board through the serial port. Such program reads the data from four different text files as illustrated in Fig. 8.1 (one for the configuration parameters, one for the mask values, another one for the weights and the last one for the input stream), converts the values to a proper byte format, and finally sends them to the FPGA. Firstly, the program drives the system (by sending an adequate value to the control block, as described in sec. 8.2.1) into the configuration mode. Then, it provides a number of parameters that set up the system. Subsequently, the same process is followed to arrange the external memory with the desired masks and weights. Finally, the system is led to the operation mode and the program sends a new value of the input signal every time step.

An example showing the information included in the configuration file is provided in Algorithm 8.1. The C++ program uses the values indicated in the file to extract the parameters that are required for the operation of the system. The values of *gamma*, *eta*, p, *Number of nodes* and *Number of categories* directly determine the corresponding quantities for γ, η, p, n and c, respectively. γ and η are converted into 8-bit binary numbers according to the unsigned fixed point notation using no integer bits (thus, one byte is employed to represent each of them and their range is the [0, 1) interval). On the other hand, the corresponding binary values of p, n and c are represented as unsigned integer numbers with no fractional part. A single byte (8 bits) is employed for p and c while two bytes are employed for n. However, the range of n is limited by the memory capacity to [1, 1023]. The limitation of the exponent p in the [2, 20] interval is due to the approximation employed to implement the Mackey-Glass nonlinearity (sec. 8.2.3).

Algorithm 8.1 Text file defining the system's desired configuration. A range of allowed values is given for each parameter.

```
gamma=0.9 [0.0, 0.996]
eta=0.9 [0.0, 0.996]
p=7 [2, 20]
T=86.7us [51us, 1305.6us)
tau=10T [0T, 128T]
Number of nodes=50 [1, 1023]
Number of categories=3 [1, 255]
```

The circuit parameters T_0, ri and ti are determined by the system's characteristic

time scale T (expressed in μs) and the delay interval τ. The number of integration time steps for solving the differential equation has been set to 255 within each characteristic time T (equation 8.2). Such number of iterations is a fixed system parameter, independent of the number of nodes and of the delay period.

$$T = 255\,\Delta t = 255\,T_0\,T_{clk} \tag{8.2}$$

The C++ program determines the value of T_0 from expression 8.2 assuming a clock frequency $f_{clk} = \frac{1}{T_{clk}} = 50\,MHz$.

The parameter ri defines the number of integration steps (Δt) considered in a delay period τ. That is,

$$\tau = ri\,\Delta t \tag{8.3}$$

On the other hand, the delay period τ can be expressed as an integer number a of characteristic time periods T:

$$\tau = a\,T = a \cdot 255\,\Delta t \tag{8.4}$$

Consequently, ri can be determined by the following formula:

$$ri = a \cdot 255 = \frac{\tau \cdot 255}{T} \tag{8.5}$$

The parameter ti is defined as the number of clock cycles that compose a $\theta = \frac{\tau}{N}$ period, and therefore it is given by

$$ti = \frac{ri\,T_0}{n} \tag{8.6}$$

A precision of 16 bits (two bytes) is used to codify the integer numbers ri and ti while a single byte is employed for T_0.

Similarly, the C++ program reads the files containing the weight, mask and input signal values [mask and input normalized to the $[0, 1)$ interval and weights to the $(-1, 1)$ range], which are converted into binary numbers (with a precision of 2 bytes for the weights and of a single byte for the mask and input values) before being sent through the serial port.

By simply editing the text file shown in Algorithm 8.1, the user can conveniently configure the system with the desired setup. It is not necessary to modify the implementation design, but it is enough to run the C++ program, which arranges the circuit with the indicated parameters. Likewise, adequate weight and input data files must be employed for the system to perform the desired task.

8.2.3. The Mackey-Glass block

This block is devoted to solve the Mackey-Glass differential equation ([MG77]), which is given by

$$\frac{dx(t)}{dt} = (-x(t) + \frac{\eta \cdot [x(t-\tau) + \gamma \cdot J(t)]}{1 + [x(t-\tau) + \gamma \cdot J(t)]^p})\frac{1}{T} \tag{8.7}$$

The output of the block is provided every integration time step given by $\Delta t = T_0\, T_{clk}$. The circuit employed as differential equation solver is schematically illustrated in Fig. 8.4. The *enable* signal activates (for a clock cycle every Δt) reading out the iterated solution of the state variable x_i (at the output of the register) and initiates the calculation of a new iteration (x_{i+1}) using the obtained value x_i. The period T_0 indicates the number of clock cycles between a time step and the next one. It is used to find the solution of the state variable (x_{i+1}) as a function of the preceding value (x_i), of the state a delay period τ before (noted as $x_\tau = x(t-\tau)$) and of the current external input value multiplied by the corresponding mask (J). The value of the parameters η and γ is specified by the control block. The value of the delayed state x_τ is provided by an internal RAM memory that stores the values of x for every integration step throughout a whole τ period. The content of the memory is updated every time step with the result of the differential equation solver. The address (referred to as *dirt* in Fig. 8.1) that must be accessed each time step (for reading the delayed solution, x_τ, and writing the current one, x_i) is supplied by the control block. A clock signal with higher frequency ($clkfast$ with $f_{clkfast} = 200$ MHz) is employed within the Mackey-Glass block to perform some of the necessary calculations.

Equation 8.7 can be rewritten as

$$\frac{dx}{dt} = [f(x_\tau,\, J) - x]\frac{1}{T} \tag{8.8}$$

where f takes the form

$$f(x_\tau,\, J) = \frac{\eta(x_\tau + \gamma \cdot J)}{1 + (x_\tau + \gamma \cdot J)^p} \tag{8.9}$$

The differential equation 8.8 can be replaced by the following recurrent expression assuming the first-order Euler numerical method (i.e.: the approximation $\frac{dx}{dt} \simeq \frac{x_{i+1}-x_i}{\Delta t}$):

$$x_{i+1} = x_i + (f - x_i)\frac{\Delta t}{T} \tag{8.10}$$

Since the system employs an integration time step that is fixed to $\Delta t = {}^{T}/_{255}$ (equation 8.2), expression 8.10 results in

$$x_{i+1} = x_i + \frac{f - x_i}{255} \tag{8.11}$$

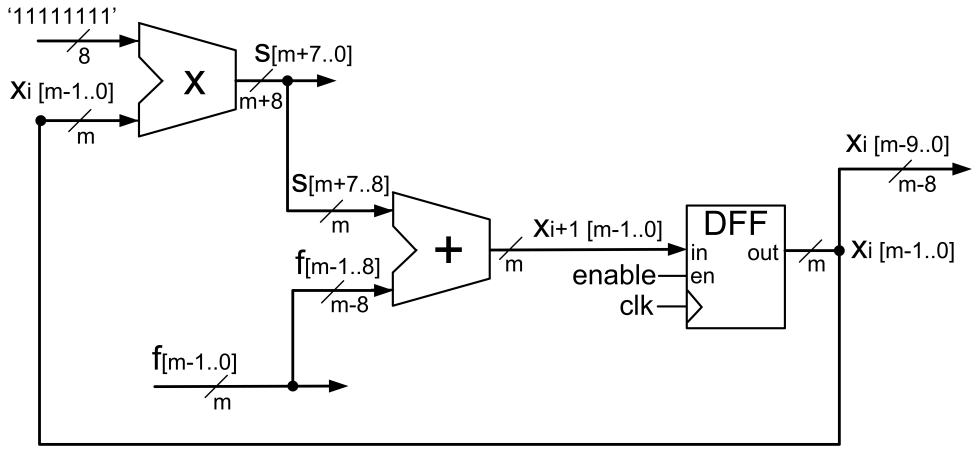


Figure 8.4.: Schematic of the differential equation solver. First-order Euler method is implemented by means of an adder, a multiplier, and a register. A precision of 16 bits ($m = 16$) is used within the solver.

This equation is implemented digitally by only using an adder, a multiplier, and D flip-flops (DFF) as shown in Fig. 8.4. Although an amount of 16 bits ($m = 16$) is used to codify the variable x within the differential equation solver (8 bits for the integer part and 8 bits for the fractional part), the output (x_i) is provided to the classification block with 8-bit resolution (8 fractional bits and no integer ones). The multiplication of the current state value x_i by 255 is performed in order to obtain the value of s, defined as $s = 256\, x_i - x_i$. Division by 256 is performed by shifting the binary numbers eight positions to the right. The addition of f and s after being shifted results in the following expression for x_{i+1}, which is a good approximation

to equation 8.11:

$$x_{i+1} = x_i + \frac{f - x_i}{256} \tag{8.12}$$

The output of the register in Fig. 8.4 is updated every Δt (through the *enable* signal) so that the resulting value of x_i can be used as input for the next iteration, x_{i+1}. The period of time within each time step (Δt) is necessary to calculate the result of each iteration of the differential equation, especially for the calculation of f. Regarding this function, it is implemented by an appropriate block that employs a 200-MHz clock to iteratively find an approximate solution. Such block first computes the value of an intermediate variable $z = x_\tau + \gamma \cdot J$ by means of an adder and a multiplier, and then estimates the function $f(z) = \eta \frac{z}{1+z^p}$ as I detail below. The function f is represented in Fig. 8.5 for different values of p (assuming $\eta = 1$). It can be observed that f is bound to the $[0, 1]$ interval, it behaves linearly ($f(z) \simeq z$) for small values of z while it asimptotically tends to zero for high values of z. The circuit employed to estimate the value of f provides an initial guess of the function (we name it *ratio*) that is recursively modified until it approximates the desired function (i.e., $ratio \simeq \frac{z}{1+z^p}$). More specifically, the value of *ratio* is supplied by an up-down counter (with 8-bit precision). This counter is gradually increased or decreased (by a unity each clock cycle) depending on the result of the product $(1+z^p) \cdot ratio$. In case $(1+z^p) \cdot ratio > z$, the value of *ratio* is decreased (by enabling the counter's *down* signal) while it is increased otherwise (by enabling the counter's *up* signal). This procedure allows to obtain an estimation of the desired function after a sufficiently large number of iterations. The value of z^p is evaluated through a serial implementation of the product of z by itself a number of p times. Therefore, the proposed implementation is limited to positive integers for p. In addition, large values of p cannot be employed as they would require too long evaluation times. Finally, the resulting value of *ratio* is multiplied by η to obtain the estimated $f(z)$.

8.2.4. The external memory

The values for the masks and weights are stored in a SDRAM memory chip on the Altera DE0 board (external to the FPGA, as illustrated in Fig. 8.1). The control block provides the *address* signal to the memory, which indicates the location where each value must be stored when the system is in the "weight loading" mode. When the system is in "operation", the *address* signal provides the memory location that must be accessed each time to obtain the appropriate value. Each value has 16 memory bits available although the masks only make use of 8 bits. Fig. 8.6 shows how the weight and mask values are organized in the memory; that is, the address ($address[17..0]$) that corresponds to each weight (w_{lj}) and mask value (M_j). The addresses are encoded using 3 bytes (24 bits) of which 18 are actually employed (the

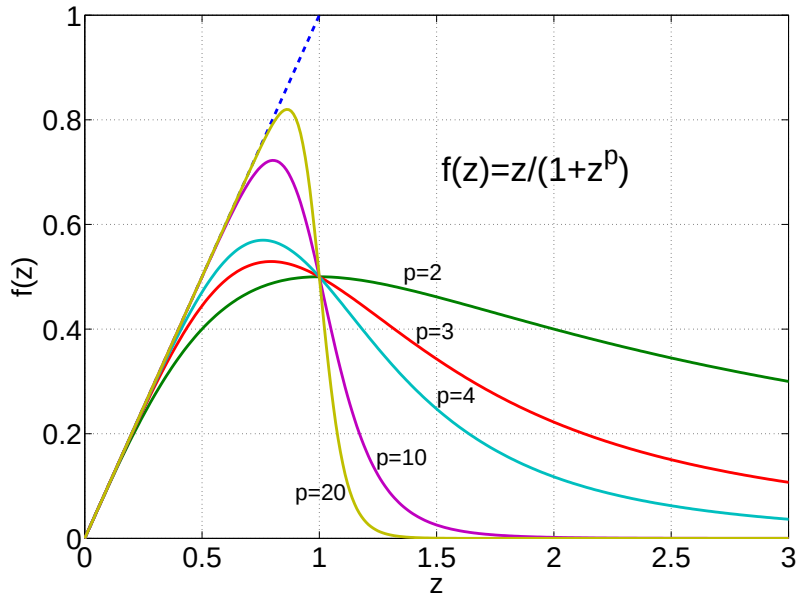


Figure 8.5.: Representation of the nonlinear function $f(z) = \frac{z}{1+z^p}$ employed in the Mackey-Glass differential equation.

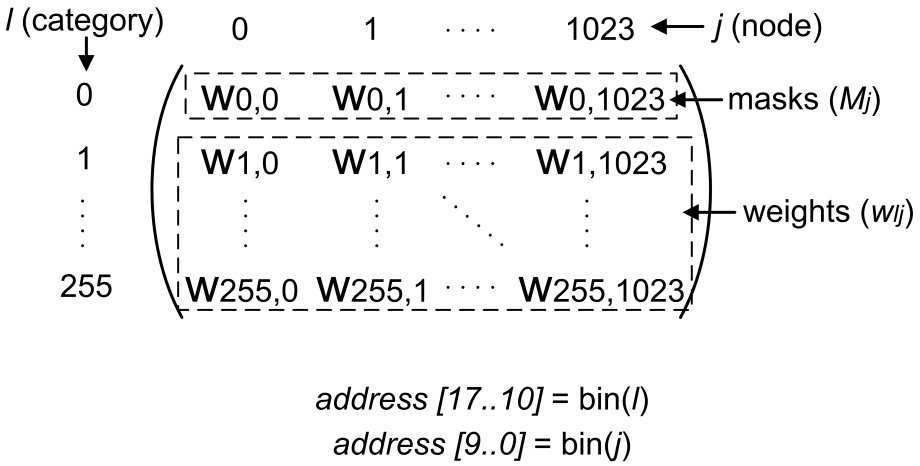


Figure 8.6.: Organization of the weight and masks in the external memory. The address for each value is obtained concatenating two binary numbers ($address[17..10]$ and $address[9..0]$). The most significant one corresponds to the weight category number l ($l = 0$ indicates that the value corresponds to a mask) and the least significant part to the virtual node number j.

first 6 are set to 0). The 10 least significant bits indicate the virtual node number j (with $j = 0 \ldots N-1$) while the other 8 bits represent the category number of the weight l (with $l = 1 \ldots c$). The case $l = 0$ indicates that the value corresponds to a mask.

8.2.5. The classification block

The output layer block performs every time θ the multiplication of the virtual node state x_j ($j = 1 \ldots N$) by the corresponding output weight w_{lj} for each one of the c output classes ($l = 1, ..., c$). The resulting product for each category is sequentially added for a given number (α) of intervals τ as indicated in equation 8.1. That is, the product is cumulatively added throughout αN θ periods after which the output value is reset. In case of the system performing a prediction or function approximation task instead of pattern classification, α is set to 1 so that a new output value is provided every interval τ. This block is implemented through a multiply-accumulate circuit (MAC) equivalent to that of Fig. 7.6. In this case, however, a precision of 8 bits is employed for the node states (x_j) and of 16 bits for the output weights (w_{lj}). The unsigned 0.8 notation is used for the reservoir states and a particular codification (different from the two's complement notation) is selected for the weights: the first bit indicates the sign of the value (0 means positive and 1 negative) and the rest of bits represent the absolute magnitude as in the unsigned 0.15 notation (thus, the actual weight values must be normalized to the $(-1, 1)$ range). This representation allows to operate with negative quantities while still making use of the unsigned notation. More specifically, the product $x_j \cdot w_{lj}$ is performed according to the unsigned notation (the first bit of the weight that indicates the sign is excluded to correctly carry out the multiplication) and the summation in the MAC circuit (see Fig. 7.6) is chosen to perform either an addition or a subtraction depending on the sign bit of the weights (addition for positive weights and subtraction for negative ones). The output of the block providing the accumulated value (y_l, chosen with a 16-bit precision) is initialized to an intermediate preset value of 32768 to avoid negative results. The use of an internal RAM memory is necessary to store (each time θ) the cumulatively added values that correspond to the different categories ($l = 1, ..., c$).

The classification block receives the virtual node values x_j from the Mackey-Glass block. The capturing of the differential equation state x as a virtual node is activated with the *enablei* signal (that is, every time θ). On the other hand, the weight values are provided by the external memory. During each inter-neuronal period θ, the memory supplies a series of c values that correspond to the output weights for each classification category. A period of 25 clock cycles is waited before sending each output weight so that the classification block can receive the correct memory value and perform the required operations within this time. The signal *enabletau* indicates when the computation of all virtual nodes (for each incoming input value)

is concluded. Another signal (*resets*) that is activated after every $\alpha\tau$ interval states when the MAC circuit must be reset.

To perform temporal classification, the output layer is trained using a target function that is -1 if the signal does not correspond to the sought category, and +1 if it does. In operation, the output of the classification block provides a time trace (y_l) for every target l, and a winner-take-all is applied to select the actual category. In other words, the values of y_l are compared for the different categories after every $\alpha\tau$ interval and the greatest of these values (y_k) determines the output category ($l = k$) that matches the input signal. Such classification process is illustrated in next section (Fig. 8.9) for different types of temporal input signals. The normalization factor of the output weights does not affect the classification result since the outputs for the different classes (y_l) are compared among them. Nonetheless, in the case of the system performing a prediction or function approximation task (only one class is considered, $c = 1$), the resulting output must be re-scaled according to the normalization factor employed for the output weights.

8.2.6. System limitations

The processing rate of the system is determined by the parameter T_0, which defines the number of clock cycles within an integration time step ($\Delta t = T_0\, T_{clk}$). The period of time within each time step (Δt) is employed to calculate the result of each iteration of the differential equation in the Mackey-Glass block. It has been observed that about 10 clock cycles ($T_0 \sim 10$) are usually enough to correctly determine the solution of the differential equation. On the other hand, the value of T_0 is conveniently used to adjust the processing rate of the circuit so that it coincides with the rate at which the system receives the external input stream through the serial port. That is to say, it helps to couple the input sampling time and the system's τ period given by

$$\tau = a\, T = a \cdot 255\, \Delta t = a \cdot 255 \cdot T_0 \cdot T_{clk} \tag{8.13}$$

where a defines the length of the delay interval τ. The required time to send each data point of the input signal from the PC to the FPGA through the RS-232 serial port has been observed to be $t_{RS232} = 8678\, T_{clk}$. The same data value can be sent a number of times (l_{repeat}) to approximately couple the input-supply time with τ:

$$\tau = l_{repeat} \cdot t_{RS232} = l_{repeat} \cdot 8678\, T_{clk} \tag{8.14}$$

The values of T_0 and l_{repeat} can be tuned so that equations 8.13 and 8.14 are made equal. For example, for the case of a system with $\tau = 10\, T$ ($a = 10$, which is the

Parameter	Allowed range
$\gamma = 0.9$	[0, 0.996]
$\eta = 0.9$	[0, 0.996]
$p = 7$	[2, 20] only integers
$\tau = 10\,T$	$[0T, 128T]$
$T = 86.7\,\mu s$	$T = 255\,T_0\,T_{clk}$
$T_0 = 17$	[10, 255]
$N = 50$	[1, 1023]
$c = 3$	[1, 255]
Input	[0, 0.996]

Table 8.1.: Possible ranges of the different parameters in the proposed FPGA implementation and specific values employed in the pattern recognition example application.

configuration used for the tasks presented in next section), setting $l_{repeat} = 5$ makes possible an approximate synchronization of the input-supply time with the system's delay period (τ) using the parameter $T_0 = 17$. This value results in a necessary time to process each input data point of $\tau = 0.867$ *ms* (equation 8.13 assuming a clock frequency of 50 MHz).

It is worth mentioning that the required time for sending data through the serial port (t_{RS232}) occasionally varies from the observed value, which may lead to isolated errors in the received input signal.

The presented implementation has been designed to use a number of categories up to $c = 255$, a maximum number of 1023 nodes and delay values up to $\tau = 128\,T$. However, it must be noted that the the multiplications of the state x_j by the corresponding weight w_{lj} for each one of the c output classes ($l = 1 \ldots c$) are performed sequentially in the classification block, and therefore the time θ between nodes must be high enough to perform the multiplications for all the classes. Indeed, the maximum possible number of classes (c_{max}) is given by the following formula:

$$c_{max} = \lfloor \frac{\theta}{25\,T_{clk}} \rfloor = \lfloor \frac{255 \cdot a \cdot T_0}{25 \cdot N} \rfloor \tag{8.15}$$

since 25 clock cycles have been set as necessary to wait for receiving the proper weight value (from the external memory) and to perform each multiplication. For instance, in case of using the parameters $a = 10$, $T_0 = 17$ and $N = 50$, the allowed number of classes is limited to $c_{max} = 34$. A greater number of categories would require increasing the evaluation time (T_0) accordingly.

In Tab. 8.1, I show the limiting ranges that can be used for the different parameters in the presented FPGA implementation of the single dynamical node reservoir com-

puter. As in previous implementations, the design has been developed to deal with one-dimensional input signals.

8.3. Experimental results

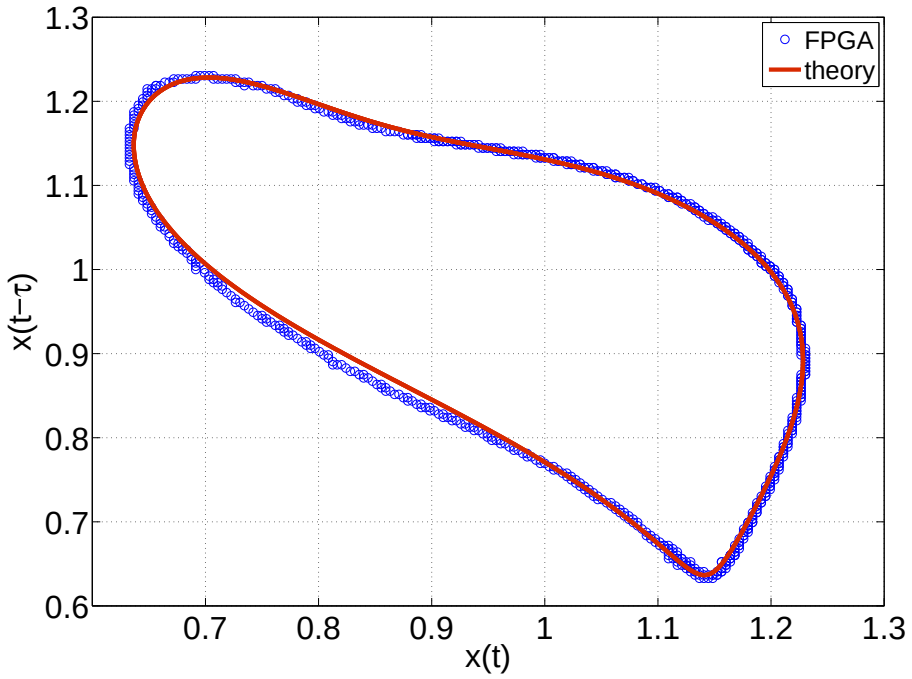
After synthesizing the proposed digital circuitry on a low-cost FPGA (an Altera Cyclone III), the system has been tested for a temporal pattern classification task and for chaotic time-series prediction. Before showing the performance results and the required hardware, I test the correct operation of the Mackey-Glass differential equation solver.

8.3.1. Validation of the differential equation solver

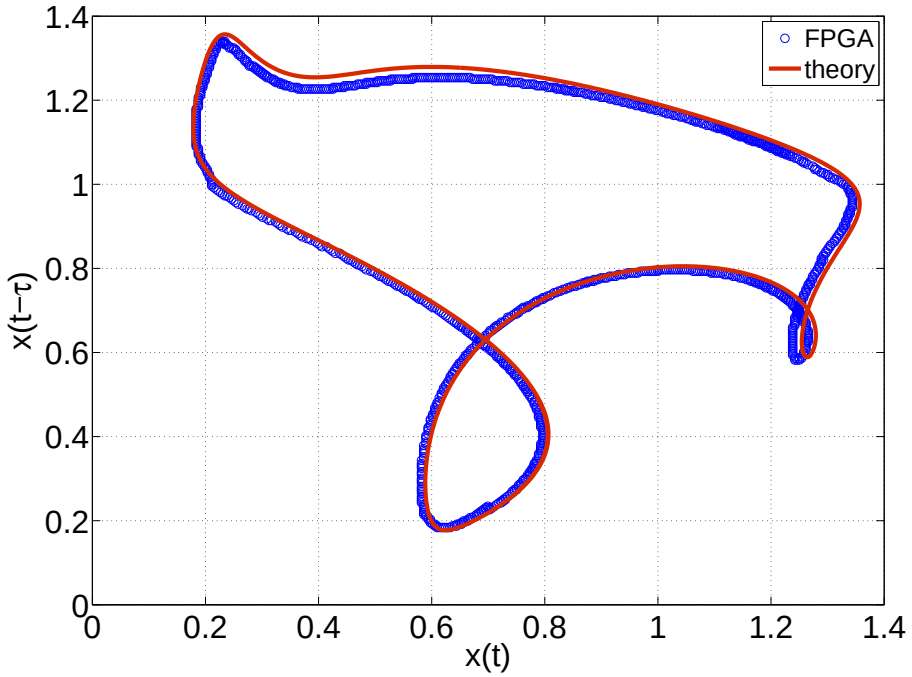
The Mackey-Glass equation (8.7) in the absence of external input (i.e., $\gamma = 0$) displays a range of periodic and chaotic dynamics depending on the values of the parameters T, η, τ and p ([GM10]). In Fig. 8.7, the periodic solutions that result for two different configurations are shown. The solutions obtained using the FPGA implementation are compared with the expected numerical results acquired by software (MATLAB). The fourth-order Runge-Kutta method (MATLAB code available at [Coc09]) with a time step of 0.01 was used to find the software-based numerical solutions. The Mackey-Glass solutions are represented in the phase space plotting the value of x_τ as a function of x. It can be observed that the experimental solutions provided by the FPGA solver are qualitatively equal to the expected results obtained by software. The small deviations are due to the approximated methods employed in the hardware implementation (a simpler numerical integration approach, limited resolution of the variables, approximations to estimate the nonlinear function of equation 8.9, etc). Such discrepancies in the values of the differential equation state do not have a major impact on the final performance of the system as long as it can be trained using the experimental solutions.

8.3.2. Classification task

As a proof-of-concept and to illustrate the real-time classification capabilities of the proposed FPGA implementation, we devised a simple benchmark task. The system was trained to differentiate between three different noisy input signals, namely, sawtooth, sine, and square input waveforms. A random noise with 2% relative amplitude is added to the input signals to test the robustness in the classification. The input values are restricted to a resolution of 8 bits (that is, their binary range is [0, 255]) and the noise is uniformly distributed (adding a random value in the [−5, 5] range to the noise-free signal). Each cycle of the input signal contains 20 time steps.



(a)



(b)

Figure 8.7.: FPGA experimental and theoretical solutions of the Mackey-Glass differential equation using two different configurations: $T = 1$, $\eta = 2$, $\tau = 2$, $\gamma = 0$ and $p = 7$ (a) and $T = 1$, $\eta = 2$, $\tau = 2$, $\gamma = 0$ and $p = 20$ (b). A value of $T_0 = 10$ has been employed in the hardware implementation.

The behavior of the Mackey-Glass dynamical system ($x(t)$) with an external forcing function ($I(t)$, in particular, the sawtooth input signal) is illustrated in Fig. 8.8. The bottom panel of Fig. 8.8 illustrates how each sample of the input signal is expanded over the N virtual nodes through the multiplication by a random mask and the transformation produced by the Mackey-Glass non-linearity.

As described in sec. 8.2.5, the system's output layer is trained to provide a value of 1 for the classification output corresponding to one of the possible categories when the input is of that particular type, and a -1 otherwise. To carry out the classification, the network readout signals (y_1, y_2 and y_3 referring to the classification outputs of the square, sine and sawtooth, respectively) are cumulatively added throughout a certain number of time steps (α) according to equation 8.1. For the present example, α has been set equal to the number of points in the input cycle period (i.e., $\alpha = 20$). The output classifiers (y_k, $k = 1, \ldots, 3$) are computed for each input step and updated (reset to a value of $y_k = 32768$, $k = 1, \ldots, 3$) at the end of each cycle (after 20 intervals τ). The greatest value of the y_k sums at the end of the cycle determines the output category that matches the input signal. This classification process performed by the experimental system (using the set of parameters $p = 7$, $\gamma = \eta = 0.9$, $\tau = 10T$, $N = 50$ and $T = 1$) is depicted in Fig. 8.9 for 2 cycles of the input signal when it corresponds to a square (Fig. 8.9a), sinusoidal (Fig. 8.9b) and sawtooth pattern (Fig. 8.9c). The represented results are for the case of input signals without noise. Similar results are found when noise is added to the input patterns. A value of $T_0 = 17$ has been employed in the implementation so that the delay period τ corresponds to an evaluation time of 0.867 *ms*. It can be observed that the input signals can be clearly differentiated.

In order to select the optimum configuration parameters that yield a good experimental classification performance, we carried out numerical simulations first. Using the software implementation of the Mackey-Glass reservoir system, the performance was evaluated in terms of the average error rate in the classification of the three input waveforms and the confidence margin, defined as the distance between the reservoir's best guess of the target and the closest competitor. A similar analysis to that performed in previous chapters for optimizing the parameters r and v of the cyclic reservoir (see, for example, Fig. 6.2) was carried out to determine the optimum values for γ and η in the Mackey-Glass node. The delay ($\tau = 10$), number of nodes ($N = 50$) and the exponent ($p = 7$) were kept fixed during iterations. Such configuration (with $\theta = 0.2$ and $p = 7$) is similar to that successfully employed in [ASVDS$^+$11] to perform a speech recognition task. It guarantees that the system operates in a stable fixed point in the absence of external input ($\gamma = 0$). With input, however, the system might exhibit complex dynamics. Both the error rate and the margin in the classification task were examined for different combinations of γ and η searching for a configuration yielding low error rates and high separation between the three classes, which allow good classification results since these conditions are the most robust to fluctuations. The optimum values for the configuration parameters compatible with the digital realization (that is to say, within the allowed ranges

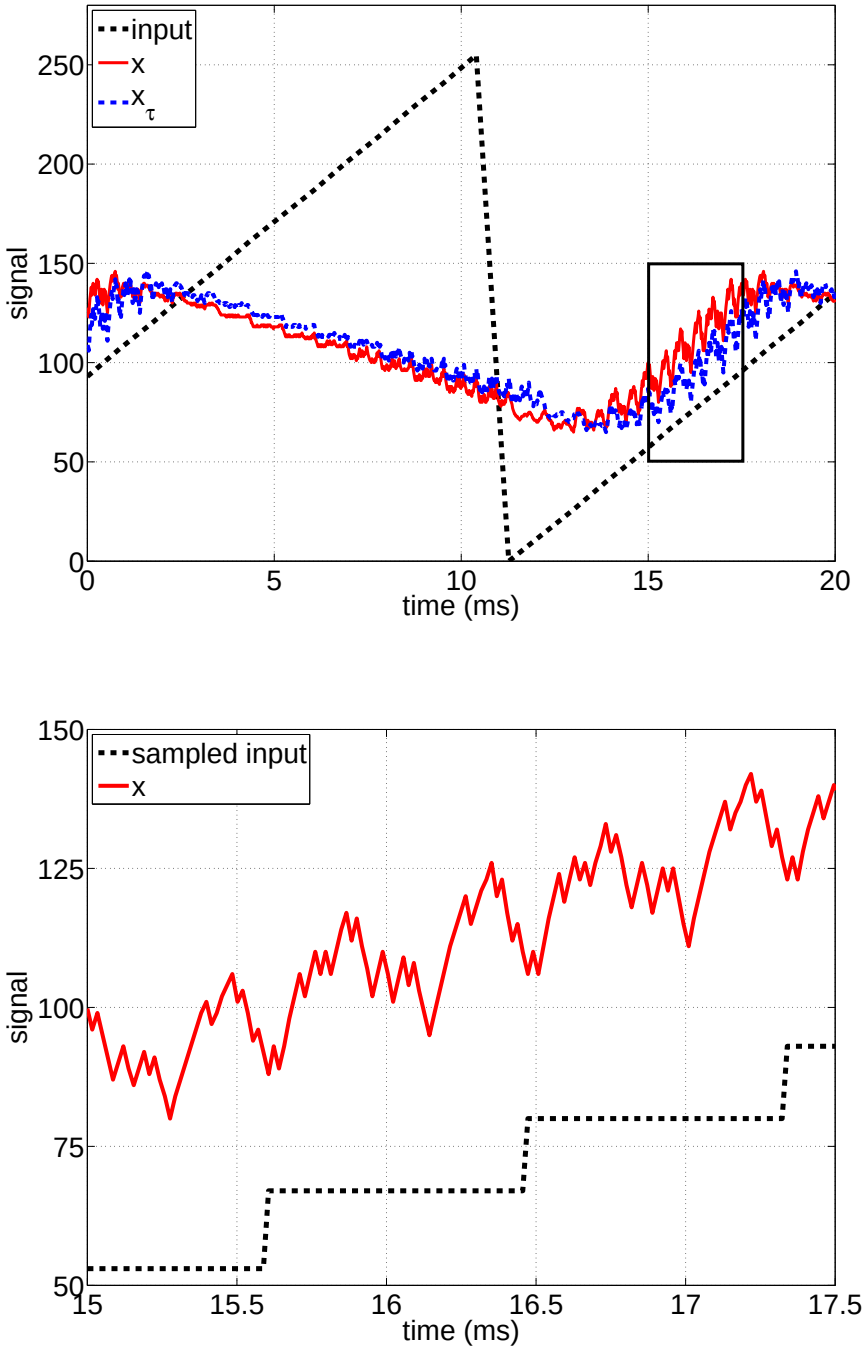
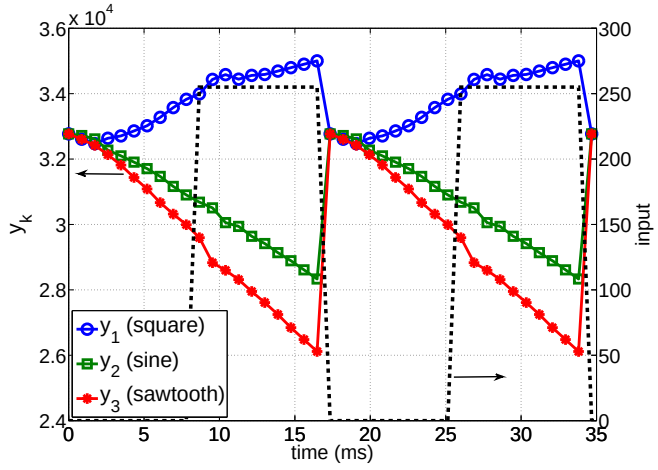
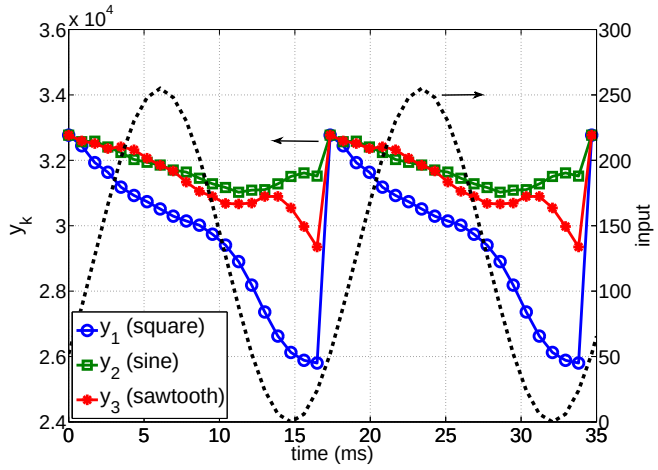


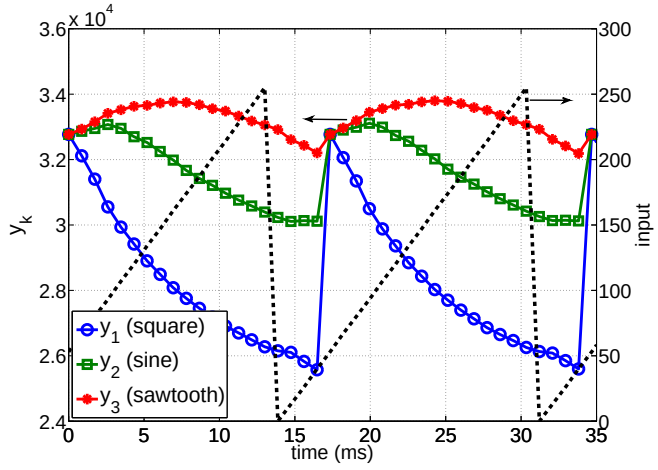
Figure 8.8.: Sawtooth input signal and its corresponding states $x(t)$ for the case of the system configured with the following algorithmic parameters: $p = 7$, $\gamma = \eta = 0.9$, $\tau = 10T$, $N = 50$, $\theta = 0.2T$ and $T = 1$. A value of $T_0 = 17$ has been employed in the implementation so that a delay period τ corresponds to an evaluation time of 0.867 *ms*. The dynamical system nonlinearly maps each value of the sampled input into an N-dimensional state that facilitates classification. The bottom panel is an enlargement of $x(t)$, as indicated by the closed black window above, and the input expanded over a τ interval.



(a)



(b)



(c)

Figure 8.9.: Pattern recognition system behavior when the input is a square (a), sinusoidal (b) and sawtooth (c) signal. I show the input signal (right y-axis) along with the output classifiers (left y-axis) for the three possible patterns to be recognized. A clear recognition of the input type is obtained.

as indicated in Tab. 8.1 and ensuring that the state x is also kept within the limits given by the 8-bit resolution $[0, 1)$) are $\gamma = 0.9$ and $\eta = 0.9$ (for more details, see the plots presented in [ASEM+15]). A random mask vector allowing two possible values (0.1 and 0.9) with equal probability was employed.

Experimental results of the Mackey-Glass reservoir states $x(t)$ when the different types of signals (with 2% of noise) were entered into the system (using the optimum configuration) were employed for training. In fact, a number of variations of the experimental states were also generated (by software) adding small amounts of noise over the original $x(t)$ signals in order to obtain a more complete training data set. As usual, a standard linear regression of the desired output on the reservoir states was used to train the system assigning an output weight to each virtual node, such that the weighted sum of the states approximates the desired target value as closely as possible.

After configuring the hardware implementation with the obtained output weights, the testing was carried out monitoring the experimental results of the classification outputs (y_k) when entering previously unseen input data of the same kind and with the same noise level as those used in the training process. An error-free classification (100% of accuracy) was found for a test set containing about 1000 cycles (including the same amount of data points for each type of the input signals).

8.3.3. Time-series prediction task

The second task that was evaluated is the Santa Fe time-series prediction benchmark already used in the previous chapters and described in sec. 2.3.4.2. It consists in the one-step ahead prediction of the Santa Fe laser data set containing 10000 samples ([WG15]). In this case, the first 9000 points were used for training and the remaining 1000 for testing.

Similar simulations to the ones performed for the classification task were carried out to determine the configuration that gives the best performance. In this case, a numerical model emulating the behavior of the hardware implementation using the same integration method (first-order Euler approach) and truncating the resolution of the variables accordingly to the hardware (8 bits for the reservoir states and 16 bits for the output weights) was employed for training the system and to find the best-performing configuration. Such numerical model approximately reproduces the FPGA experimental results as illustrated in Fig. 8.10, where the time trace $x(t)$ is depicted for both the experimental and numerically simulated system when the Santa Fe time series is used as driving input signal. As for the previous task, some of the parameters were kept fixed ($\tau = 10$, $N = 50$ and $p = 7$) while γ and η were varied to find the values providing a minimum quantity of the normalized mean square error (NMSE). The same binary mask vector used for the classification task was employed here. An optimum error of the prediction $NMSE = 0.1069$ was obtained for the parameters $\gamma = 0.75$ and $\eta = 0.86$.

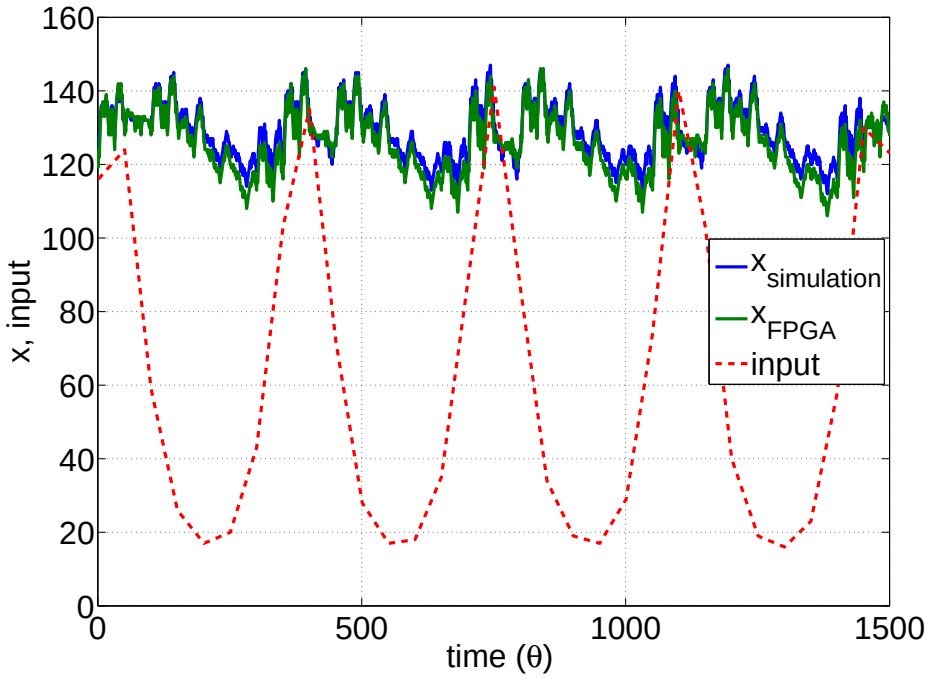


Figure 8.10.: Evolution of the reservoir state $x(t)$ along with the input of the system (Santa Fe laser intensity values). The reservoir states are depicted for both the experimental hardware realization (x_{FPGA}) and for the numerical model used to train the system ($x_{simulation}$). A good match can be observed. The graph corresponds to a fragment of the test data set using the configuration $\tau = 10$, $p = 7$, $\gamma = 0.9$ and $\eta = 0.9$.

Finally, after configuring the hardware implementation with the optimum configuration parameters and with the trained output weights, the system was tested monitoring the experimental results of the prediction output. As in the previous task, a value of $T_0 = 17$ was used to run the system. A fragment of the FPGA predicted and targeted laser intensity values is shown in Fig. 8.11. A prediction error of $NMSE = 0.1312$ was obtained with the FPGA implementation, which is comparable to the expected numerical result.

It must be noted that an error of $NMSE = 0.0506$ is found when the system is simulated applying no restriction on the resolution of the variables, which indicates that an implementation using higher resolution for the variables could present a significantly better performance. On the other hand, it has been observed that a more complex integration method (fourth-order Runge-Kutta approach) only represents a minor increase in the accuracy with a prediction error of $NMSE = 0.0504$.

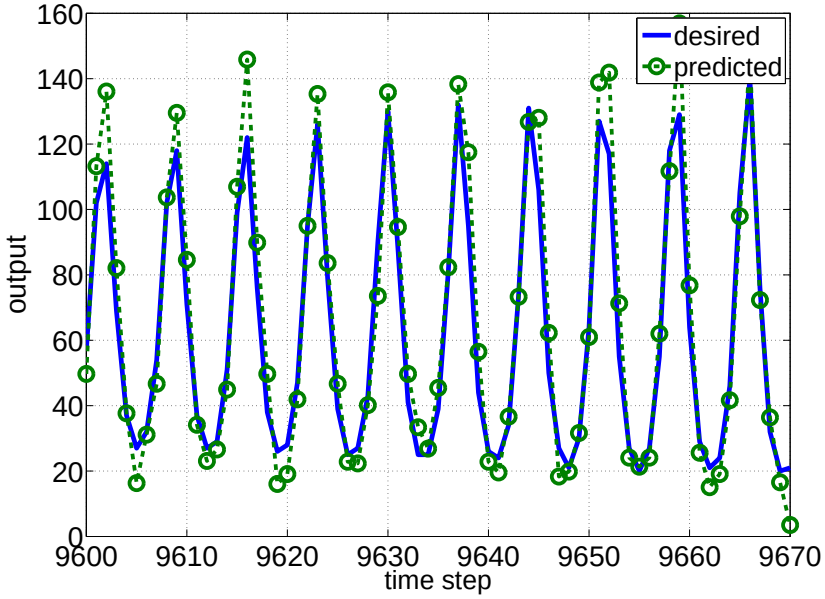


Figure 8.11.: Segment of the Santa Fe laser time series (FPGA predictions and targeted values).

8.3.4. Hardware resources

The hardware resource utilization of the proposed single dynamical node RC implementation is presented in Tab. 8.2. The number of logic elements needed for the implementation represents only a 10% of the available number in the Cyclone III low-cost FPGA. A 14% of the total number of embedded multipliers (of 9-bit elements) is also required. On the other hand, only sufficient memory is needed to allocate the delayed signals. The formula for the required RAM bits in Tab. 8.2 results from the fact that a number of 256 values of $x(t)$ (one for each integration time step) with 11-bit resolution need to be stored within each delayed period τ. In addition, 16 bits of RAM memory for each classification category are employed in the classification block. The design can use delay values up to $\tau = 128T$ and a number of categories up to $c = 255$ using 71% of the RAM capacity. The limited use of resources highlights the advantages, from the hardware implementation point of view, of using a single nonlinear node approach based on time multiplexing.

Regarding the use of the external memory, it depends on the number of 16-bit weight and mask values (Fig. 8.6) as follows:

$$Memory\ bits\ (SDRAM) = 16 \cdot (c+1) \cdot N \tag{8.16}$$

where c denotes the number of categories and N the number of nodes. For the case

Logic elements	1605 (10.4%)
Registers	882 (5.7%)
Memory bits	$768 + 16\,c + 2816\,\tau/T$
Embedded multipliers	16 (14.3%)

Table 8.2.: Hardware resource utilization of the Altera Cyclone III FPGA for the proposed single dynamical node RC implementation.

of an implementation using the maximum allowed values of c and N ($c = 255$ and $N = 1023$), 523.8 kbytes of the 8-Mbyte SDRAM are required.

The power consumption of the implementation was measured for the system in operation (performing the prediction task). The measured power consumption of the whole FPGA development board (including peripherals such as some LEDs, the SDRAM module, the serial port communications, etc) was found to be 1197 mW, of which 83 mW were estimated to be due to the configured circuit (simulation result of the Quartus power analyzer software).

8.4. Discussion

In this chapter, I have presented the first digital implementation of the RC approach using a single nonlinear oscillator with delayed feedback as dynamical node. A self-contained system has been realized in a low-cost FPGA board with the delay line implemented using a RAM memory. The utilization of an on-chip memory to store the reservoir states enables a high throughput and implies lower power consumption than using on-board memories. The idea of replacing a recurrent network with a large number of nodes by a single node subject to delayed feedback relaxes the hardware requirements for the FPGA as observed in Tab. 8.2. The properties of this computationally low-cost method are particularly suited to process temporal information. More specifically, I have shown that the proposed system is capable of classifying different patterns and to perform time-series forecasting. Importantly, the implementation can be used to differentiate a larger number of input classes.

A hardware implementation specifically designed for a particular purpose is expected to be more energy-efficient than the standard general-purpose microprocessor-based alternative. Therefore, the use of a compact implementation for the whole system in a single integrated circuit is interesting from the energy efficiency point of view. It can be a solution for those electronic systems implementing computational intelligence techniques and requiring low power dissipation such as wireless sensor networks ([KFV11]), predictive controllers or monitoring medical devices. In the latter case, a delay-based software implementation of RC was found to achieve state-of-the-art performance in the classification of electrocardiographic (ECG) signals of cardiac arrhythmia ([EMSFM15]). Since this medical application requires

a sampling time of about 1 ms, ECG classification is fully compatible with our FPGA-based implementation of RC in real time.

Although for the present design I have not developed an automatic VHDL code generator to systematically produce the code for any desired configuration, the implemented system is quite flexible as it allows a range of values for the different configuration parameters as indicated in Tab. 8.1. It is worth noticing that once the design is synthesized on the FPGA, it is possible to change the system's configuration without the need of reprogramming the device. It is enough to reset the circuit and configure it again with the desired parameters (indicated on a text file as illustrated in Algorithm 8.1).

Regarding the system's performance, the presented implementation of the single dynamical node reservoir computer is not advantageous over the ESN realizations described in previous chapters. For example, it has yielded an error $NMSE = 0.13$ in the Santa Fe time-series prediction task while the ESN implementations of chapter 6 (multiplier-less approach) and chapter 7 (delay-based approach) provide error values of about $NMSE = 0.09$ when using the same number of nodes ($N = 50$) and a very simple approximation for the sigmoid function. In addition, the performance for the preceding multiplier-less and delay-based ESN designs is improved when increasing the number of nodes (e.g., $NMSE \simeq 0.036$ for $N = 200$) whereas no improvement has been observed for higher values of N in the single-dynamical-node reservoir. However, simulations of the present system have shown that the prediction error might be decreased down to values of $NMSE \simeq 0.05$ employing a higher resolution for the variables (remind that only 8 bits have been used for the network states, $x(t)$, in the current design). A similar error is expected for an ESN implementation with $N = 50$ if using a more accurate nonlinear function (see the deterministic result in Fig. 4.18). Nonetheless, it must be highlighted that some of the parameters in the present implementation have been set to fixed values (for instance, $\tau = 10$ and $p = 7$). An exploration of different parameters might provide better results.

The single dynamical node reservoir approach (often referred to as the time-delay reservoir, TDR) essentially follows a serial computation scheme similar to the delay-based ESN design of chapter 7 where the output of each virtual node in the network is computed as a function of a previous node (that is, the state of the system at a previous time step) and of the input at current time. In the case of the former system, however, the nonlinear output of each virtual node is not computed through a direct discrete-time recursive formula, but it is obtained by finding the solution of a differential equation after an appropriate number of integration steps. This implies a slow-down of the information processing compared to the serial ESN implementation. For the case of the configuration employed in the experiments, the TDR system requires a time of the order of 1 ms (0.867 ms) to process each data point of the input signal while the evaluation time is three orders of magnitude smaller for the delay-based ESN design (approximately 2 μs for the case of $N = 50$). On the one hand, the system's processing rate is limited by the communications through the serial port as the input stream must be sent at a rate that is compatible with

the FPGA implementation. For the system tested in the experiments, for example, a value of the parameter $T_0 = 17$ has been used to ensure a correct reception of the input, but a value $T_0 = 10$ would be enough to run the system. On the other hand, it is worth noticing that the integration time step has been fixed to a small quantity ($\Delta t = T/255$) to guarantee that the system can operate properly (with a sufficient number of integration steps within each θ period) even for a configuration using a small value of τ and a large number of nodes N. For the case of our implementation with $\tau = 10$ and $N = 50$, however, the time step could be reduced to $\Delta t = T/50$, which still ensures a correct functioning (with a number of 10 intermediate steps within each θ) while increasing the processing rate by a factor of 5. Nevertheless, it must also be reminded that a fast clock frequency (200 MHz) has been used to perform the calculations in the differential equation solver of the TDR implementation but a plain clock frequency of 50 MHz was used in the ESN implementations of previous chapters.

Among the different RC techniques, the TDR approach has attracted much attention in recent years as a machine learning method that makes possible the use of dynamical systems for computation on sequential data. The tolerant requirements for the reservoir have led to implementations on several hardware platforms as described in sec. 2.3.3. Here, we have shown that a digital hardware implementation of the concept is possible. Despite the potential advantages of a digital hardware realization compared to an equivalent software-based implementation using conventional general-purpose microprocessors, the particular TDR methodology seems more adequate for physical implementations through optoelectronic systems, where a relatively simple design of the optical reservoir is possible based on a fiber and a single dynamical node ([ASVDS+11], [LSB+12], [DSS+12], [BSMF13]). This seems particularly interesting when the information is already in the optical domain as in the case of many telecommunications and image processing applications ([VMVV+14]). Although the presented digital design might be improved in some aspects (precision of the variables, employed approximations, search for a better configuration), it seems unlikely to outperform the ESN implementations described in preceding chapters. For example, the sequential design proposed in chapter 7 (delay-based approach using a "static" node instead of a dynamical one) exhibits a higher accuracy in time-series prediction (Santa Fe task) with the advantage of a much simpler design (a differential equation does not need to be implemented) and a higher processing rate.

9. Applications

In this chapter, I present some engineering problems where the hardware implementation of a reservoir computing (RC) system can be applicable. The first task (sec. 9.1) is an academic example that has already been employed in chapter 8 to show the capabilities of the single dynamical node reservoir implementation. Here, it is used to test the stochastic echo state network (ESN) design (chapter 4) for temporal pattern recognition, and specially to illustrate the high fault tolerance of the implementation based on stochastic computing (SC). The following tasks are two real-life applications of practical relevance: handwriting recognition (sec. 9.2) and equalization of a nonlinear communication channel (sec. 9.3). These exemplary tasks are analyzed for the stochastic approach although any other of the digital hardware designs proposed throughout the thesis could also be used. The fault-tolerant stochastic methodology seems of particular interest for the equalization of a satellite communication channel, which requires to be performed in a severe environment (space) likely to present soft errors induced by radiation. Finally, sec. 9.4 describes additional potential applications to different engineering fields.

9.1. Noisy signal classification

This section evaluates the SC-based ESN implementation presented in chapter 4 for a task consisting in the classification of temporal signals. Stochastic computing might be of utility in specialized systems where small size, low power, or soft error tolerance is required and limited precision or speed is acceptable. As observed in chapter 4, the high precision required by time-series prediction tasks makes necessary to employ long evaluation periods for the stochastic computations to achieve the desired accuracy, which limits the system's processing speed. In this section and the next one (sec. 9.2), I focus on pattern recognition tasks since this type of applications seems to be more suitable for probabilistic computations using low precision signals, and therefore may require a shorter computation time than in time-series forecasting. On the other hand, I illustrate the high robustness to soft errors of the stochastic computing approach. This characteristic can be of great interest for applications that need to be operated in severe environments such as the one presented in sec. 9.3.

More specifically, the present task consists in differentiating between three different noisy input signals, namely, saw-tooth, sine, and square input waveform. As in

chapter 8, random noise with a 2% of relative amplitude is added to the input signals to test the robustness in the classification. The input values are restricted to a resolution of 8 bits (that is, their binary range is [0, 255]) and the noise is uniformly distributed (adding a random value in the [−5, 5] range to the noise-free signal). Each cycle of the input signal contains 20 time steps. A number of 300 cycles (100 for each input type; that is, totally, 6000 input values) are used to train the system. The same amount of data is employed for testing. To evaluate the performance of the stochastic ESN implementation, I employ a numerical model that conveniently emulates the hardware as shown in sec. 4.4.1. The ESN's output layer is trained to provide a value of 1 for the classification output corresponding to one of the possible categories when the input is of that particular type, and -1 otherwise. Remind that to carry out the classification, the network readout signals (y_1, y_2 and y_3 referring to the classification outputs of the square, sine and saw-tooth, respectively) need to be cumulatively added throughout a certain number of time steps; for example, the input cycle period (see 9.1 where α denotes the number of time steps in the cycle, and k indicates the classification category).

$$out(classification)_k = \sum_{j=1}^{\alpha} y_k(t_j) \tag{9.1}$$

The greatest value of this sum at the end of the cycle determines the output category that matches the input signal. In other words, the system follows a winner-take-all strategy. The value of this addition is reset to zero at the beginning of each cycle period. This classification process performed by an SC-based ESN (with $N = 200$ neurons and an evaluation period of $T_{eval} = 2^8 - 1$ clock cycles) is depicted in Fig. 9.1 for some cycles of the input signal. The input initially has a square form, and then it changes to the sine shape. The time step in this graph corresponds to $5.1\,\mu s$ (assuming a clock cycle frequency of 50 MHz). It can be observed that the classification is correctly performed for the square and sine input signals. The output classifiers exhibit a short undesired transitory behavior at the point in which the input signal changes from the square to the sine shape. This is due to the network "remembering" the states of the previous cycle. Similar results are found for the saw-tooth waveform.

I have analyzed the performance of the SC-based ESN in this simple classification task as a function of the system's evaluation time. The classification results (accuracy, as defined in 9.2) are presented in Tab. 9.1 for a network of fixed size ($N = 200$). The configuration parameters have been set to $r = v = 0.95$. The results show that a number of $2^{10} - 1$ clock cycles is enough to achieve a classification accuracy equivalent to that of a software implementation based on conventional computing (perfect recognition of the input signal). A shorter evaluation time of $2^8 - 1$ clock cycles still allows a classification with high accuracy (97.7% of correct answers) for this

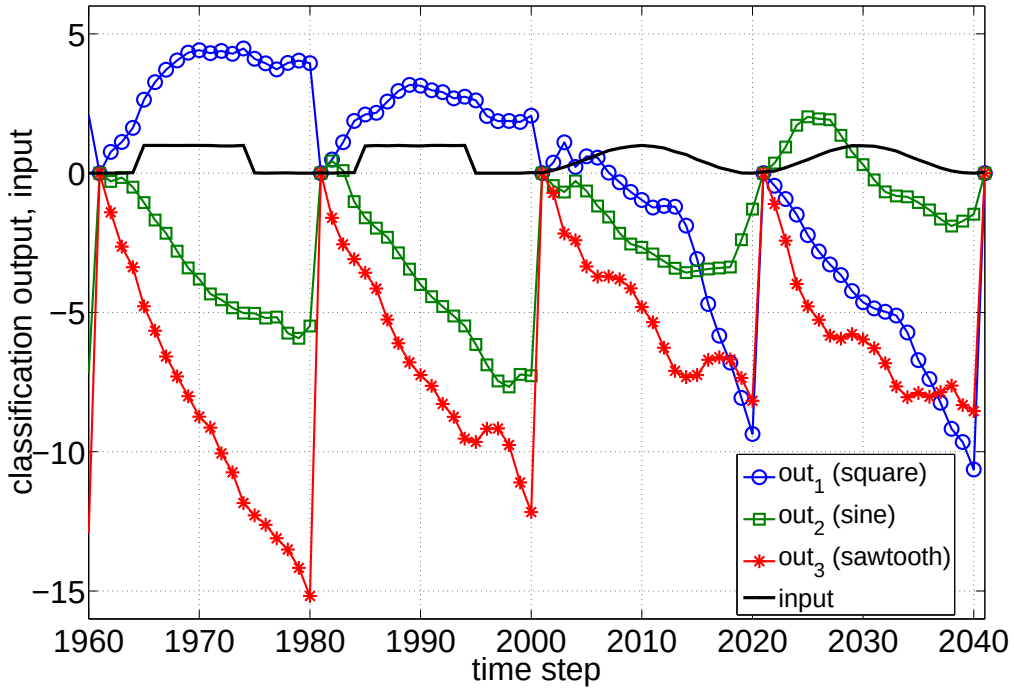


Figure 9.1.: Pattern recognition system behavior for a SC-based ESN with $N = 200$ neurons and $T_{eval} = 255\,T_{clk}$ ($time\,step = 5.1\,\mu s$ @ $50\,MHz$). The input signal is represented along with the output classifiers for the three possible patterns to be recognized (square, sine and saw-tooth wave forms). A clear recognition of the input type is obtained.

particular task.

$$accuracy\,(\%) = \frac{\#\,correct\,classifications}{total\,\#\,classifications} \times 100 \tag{9.2}$$

9.1.1. Fault-tolerance analysis

One of the most appealing features of SC implementations is their high degree of error tolerance. Stochastic circuits tolerate environmental errors that seriously affect the behavior of conventional circuits. A single bit flip (especially of a high significance bit) causes a huge error on a binary circuit, but flipping a few bits in a long random bit-stream has little effect in the value of the stochastic number represented. Therefore, SC can be interesting for applications that need to be operated in error-prone environments. In this subsection, I briefly introduce the issue of circuit reliability (soft errors in digital circuits) and analyze the fault-tolerance of the SC-based ESN implementation for the simple 3-pattern classification task.

$T_{eval}\,(T_{clk})$	***accuracy (%)***
$2^4 - 1$	45
$2^6 - 1$	67.3
$2^8 - 1$	97.7
$2^{10} - 1$	100

Table 9.1.: Classification results of the SC-based ESN with $N = 200$ neurons in the 3-pattern recognition task as a function of the evaluation time (number of clock cycles).

9.1.1.1. Introduction

In general, digital systems are reliable when the internal circuits operate within specifications. Nevertheless, they become vulnerable if something unforeseen occurs, such as a deviation of the signals from the specified voltage range for a logic 1 or 0. In fact, a single unexpected logic inversion can halt an entire system ([HH04]).

The increasing density of modern chips have resulted in the current technology being much more sensitive to noise. Decreasing the CMOS feature sizes causes the devices to be less reliable ([MF04]). Digital circuits are sensitive to noises due to soft errors ([ZS06]), environmental noises, or process ([BMR07]), voltage, and thermal variations ([San00]).

Soft errors refer to non-permanent errors that can severely limit the reliability of CMOS circuits. They are produced by the charge injection due to a particle hit ([SKK+02]) from an alpha particle ([MW79]) (from impurities present on packaging materials) or a neutron ([HS00], present in terrestrial cosmic radiation even at the sea level). When the high energy particles penetrate the silicon substrate, they generate electron-hole pairs along their tracks that may be collected by p-n-junctions. If enough critical charge is collected in a node within a combinational gate, a short current pulse is performed that may invert the logic state at the gate output ([BB97]). On the other hand, the charge collected in an internal node within a latch (memory element) may cause to flip the value stored in the cell ([DS95]). Traditionally, soft errors in memories have been a much greater concern than in combinational circuits since the memories contain many more bits that are susceptible to particle strike (for the same circuit area). However, the current technology down-scaling has caused soft errors in combinational logic gates to become as frequent as those observed in unprotected SRAM cells ([SKK+02], [Key01]).

To sum up, soft errors caused by ionizing radiation have become a major concern (specially in severe environments, such as space) due to the continuous scaling of semiconductor devices. Numerous design methodologies have been developed throughout the last decades to overcome the loss of reliability ([MSZ+05]) as can be the SRAM ([Nic05]) and combinational cells ([ZM06]) hardening, the on-chip error checking and correction circuits ([OYSK04]), space redundancy and time re-

dundancy. Space redundancy mainly includes dual modular redundancy (DMR) and triple modular redundancy (TMR). The latter is the most common scheme to perform single event upset (SEU) hardening ([HL09]).

Stochastic computing involves redundancy in the encoding of the signal values. As a result, it represents an alternative technique capable to cope with errors. The probabilistic architecture can easily adjust the level of desired immunity by means of the evaluation time, thus allowing to easily manage the trade-off between the probabilistic circuit performance (computation speed) and noise immunity. The SC approach is a system-level hardening architecture and not a silicon-level one, and therefore it operates with standard CMOS technologies. SC is compatible with the different gate level hardening and noise improvement designs mentioned above, being highly recommended their use. In fact, the SC-based designs contain some parts that are operated through conventional Boolean logic, such as the B2P and P2B converters. This makes necessary to combine SC with other methods that mitigate bit-flips for the whole system's reliability to be ensured.

9.1.1.2. Methodology

The noise has been emulated by introducing random changes into the temporal signals. That is to say, generating random flips on the stochastic bit-streams. For this purpose, I have used a B2P block together with an XOR gate as illustrated in Fig. 9.2. The input to the B2P block is the desired fraction of noise injection (r_{noise}) encoded according to the unipolar representation (sec. 4.2.2.2). The XOR gate produces a flip on the input bit-stream every time the B2P provides a high value. In other words, the bits of the input signal are flipped with a probability equal to the noise fraction value.

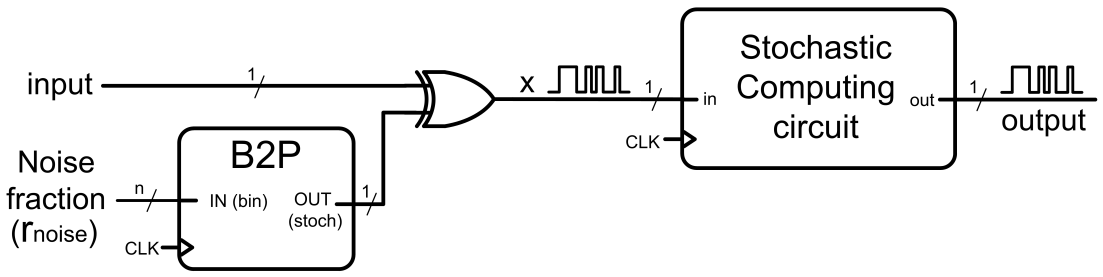


Figure 9.2.: Noise injection setup.

Although the stochastic circuit may be affected by radiation at different locations, I only analyze the case of errors (bit flips) being produced at a single point of the circuit. In particular, I have chosen to introduce noise in the signal that corresponds to the network's external input (which feeds all the neurons) as proposed in [CMO+16].

I have analyzed the effect of different degrees of noise on the performance (classification accuracy) of the system using the simulation model of the SC-based hardware implementation.

As proposed in [CMO+16], I state that the maximum allowable noise (100% noise) occurs when the probability of having a flip bit due to noise is 50%, that is $r_{noise} = 0.5$. Note that, in this situation, the information carried by the stochastic bit-stream is completely lost since it has the same probability of obtaining a 0 or a 1, which implies that after the noise injection the resulting signal provides a constant value ($x^* = 0$ in the bipolar codification). Therefore, the noise rate is given in the range from 0% to 100%, which corresponds to a swap of the probability r_{noise} in the [0, 0.5] range according to 9.3.

$$Noise\,rate\,[\%] = 200 \times r_{noise},\ \forall r_{noise} \in [0,\ 0.5] \tag{9.3}$$

The weight values obtained from the previous training process (with 0% noise) are employed for testing the network with the same test set used above, but injecting different fractions of errors to the input signal.

It is worth noting that if a stochastic signal is affected by a noise of low intensity, it will continue to operate correctly since a stochastic bit-stream is essentially a signal generated by noise. This is the case of the SEU hit at sea level, which present a frequency of the order 10^{-12} upset/(bit-hr) for memories ([Nor96]). I evaluate the effect of higher noise rates (due to electromagnetic waves, SEUs at high altitudes or in the space) on the network's classification accuracy.

9.1.1.3. Results

The classification results are presented in Tab. 9.2 for different degrees of noise injection. The chosen system employs $N = 200$ neurons and $T_{eval} = 2^8 - 1$ clock cycles. It can be observed that small fractions of injected soft errors practically do not alter the classification accuracy. The system can tolerate up to a 15% of noise injection keeping an accuracy greater than 90%. These results illustrate the inherent robustness towards noise of the SC approach.

9.2. Handwriting recognition

9.2.1. Introduction

Even in the age of the digital computer, handwriting persists in many everyday situations where it is more convenient than using a keyboard. The widespread use

noise rate (%)	***accuracy (%)***
0	97.7
5	97.3
10	95.7
15	90.3
20	82
30	72
40	59.7

Table 9.2.: Classification results of the SC-based ESN with $N = 200$ neurons and $T_{eval} = 2^8 - 1$ clock cycles in the 3-pattern recognition task as a function of the noise injection.

of handwriting makes its automatic recognition a task of practical importance. It can be employed, for example, for reading handwritten fields in forms or for interpreting notes in a PDA (personal digital assistant) or tablet PC.

There are two approaches for converting handwriting to a digital form: offline and online. The offline recognition is based on the scanned image of the writing while the online case employs the trajectories of the pen tip, that is, the writing coordinates as a function of time. The data requirements for an average written word are of the order 100 bytes (typically sampled at 100 samples per second) in the online case and of the order of 100 kilo-bytes in the offline handwriting (typically sampled at 300 dots per inch). The recognition rates, in general, are higher for the online case ([PS00]).

Other tasks associated with handwriting are the interpretation, identification and signature authentication. Handwriting interpretation consists of determining the meaning of a body of handwriting. Handwriting identification is a process to determine the author of a sample from a set of writers. Signature authentication is aimed at verifying if a signature is that of a given person.

Automatic recognition of handwriting has found commercial uses in hand-held computers such as PDAs (using online systems) and for interpreting handwritten postal addresses or amounts on bank checks (through offline systems). Nevertheless, the recognition rates achieved until now still leave room for improvement ([GLF+09]), specially in the case of mathematical symbols ([ASB13]) and of non-Latin characters with specific features that hinder the symbol recognition ([SNJ14]).

In this section, I focus on the online classification of handwritten characters. Different methodologies have been used to tackle this problem such as Support Vector Machines (SVM, [KW07]), Hidden Markov Models (HMM, [SNJ14]) and artificial neural networks ([ASB13]). In particular, recurrent neural networks (RNNs) have been shown to outperform classical HMM classifiers ([GLF+09]). Nonetheless, the recognition performance numbers are dependent on the particular set employed to test the system.

In general, handwriting recognition is implemented in software using general-purpose processors. However, custom hardware implementations can support more efficiently the operations required by this type of systems. PDAs require to perform accurate handwriting recognition using limited processor area, power and memory. The hardware implementation of the character recognition system makes possible to release potential resources on a PDA for other applications. Several digital hardware designs have been proposed to support and speed up handwriting recognition ([MR99], [SJM04], [MPR09], [IBA+10]). The work presented in [MR99], for example, proposes a specific hardware architecture targeted at the pre-processing of the online handwriting recognition signals.

Here, I propose the use of an ESN, in particular the hardware implementation based on SC (chapter 4), as main processing core of the online handwriting recognition task (the input data are pre-processed by software before entering the ESN-based classifier). I show that this hardware realization is compatible with the recognition task (the sampling rate of the input signals is 5 ms while the implementation requires a maximum evaluation time of 1.3 ms to process each input value) and achieves similar accuracy to that of a software implementation with full precision.

9.2.2. Methodology

The character trajectories data set has been obtained from the Machine Learning Repository ([Lic13]). These data were used in [WTS08] to model the biological movements and produce reconstructions of handwriting in robotic applications. The signals consist of a number of labeled samples of pen tip trajectories recorded whilst writing individual characters. All samples are from the same writer and captured at 200Hz using a WACOM tablet. Three dimensions were kept: x, y, and pen tip force. The data were numerically differentiated, smoothed and normalized. The handwritten characters corresponding to three sample letters are illustrated in Fig. 9.3. Some example trajectories representing different temporal patterns that need to be learnt for the character classification have been shown in Fig. 2.15 (sec. 2.3.4.1).

I have limited the task to the recognition of three classes corresponding to the first three letters of the alphabet. However, it could be exported to discriminate a higher number of categories. The proposed hardware design for the neuron, based on SC, is depicted in Fig. 9.4. It is slightly different from that presented in chapter 4 since the present task requires dealing with a three-dimensional input (handwriting velocities v_x and v_y, and temporal variation of the pen tip pressure). Therefore, the neuron includes four inputs (three for the external inputs and one for the internal connections with other neural units). The numerical quantity resulting from the 4-input multiplexer needs to be multiplied by 4 (after being converted to a binary value) in order to compensate the $1/4$ scaling factor of the stochastic addition. This operation is implemented within the activation function block by means of a simple shift (two positions to the left) of the binary number. It is worth noting that the 4-input

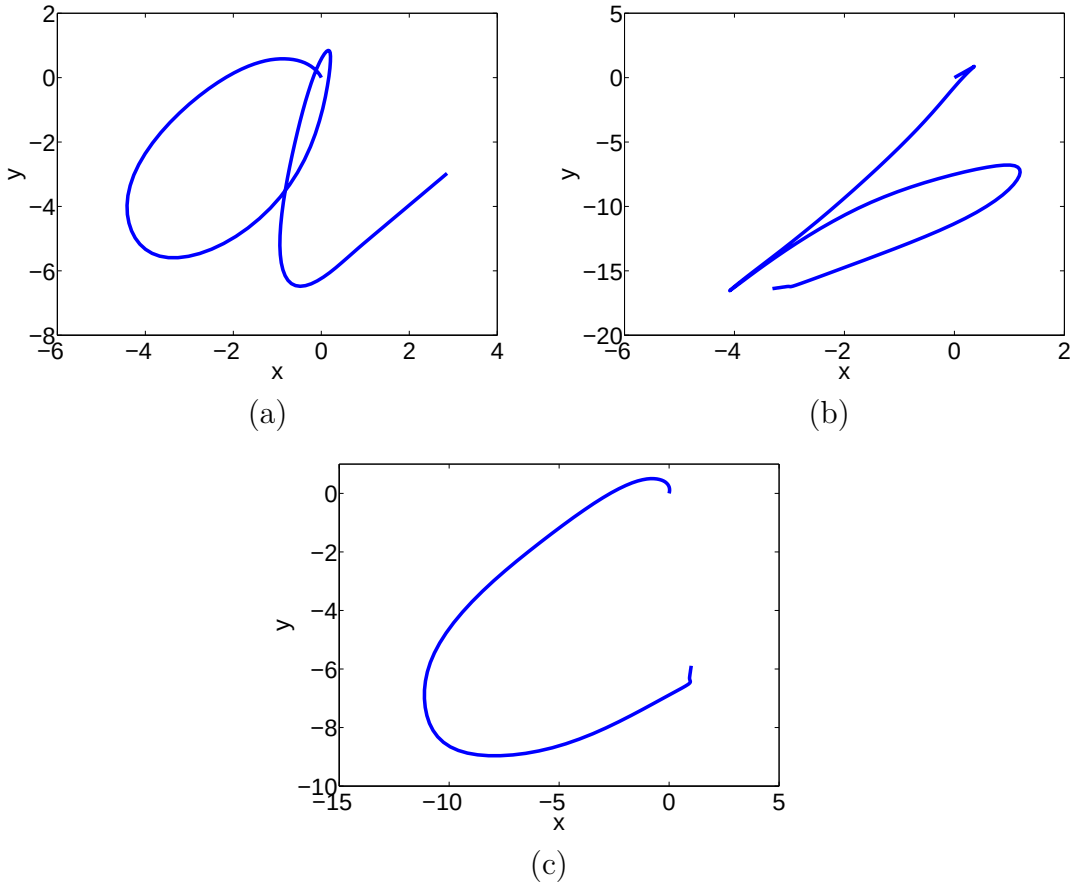


Figure 9.3.: Illustration of three handwritten characters corresponding to the letters a (a), b (b) and c (c).

neuron only requires a few more logic elements than the 2-input design since the 4 additional B2P converters can be shared by all neurons in the network (as described in chapter 4 for the ESN with cyclic topology). The online handwriting classification task has been evaluated through simulations of the SC-based ESN implementation employing a numerical model that limits the resolution of the variables according to the hardware (8 bits for the neuron states and for the output weights, and 16 bits for the final classification readouts).

9.2.3. Results

Fig. 9.5 illustrates the evolution of a neuron of the SC-based ESN implementation (with $N = 200$) for an input signal corresponding to the handwriting trajectories of the letter "a". The neuron state is compared to that of an equivalent conventional software implementation. The SC-based signal presents noisy fluctuations around the deterministic (conventional, non-probabilistic) one. Since the handwriting trajectories present significant variations from one sample to another (of the

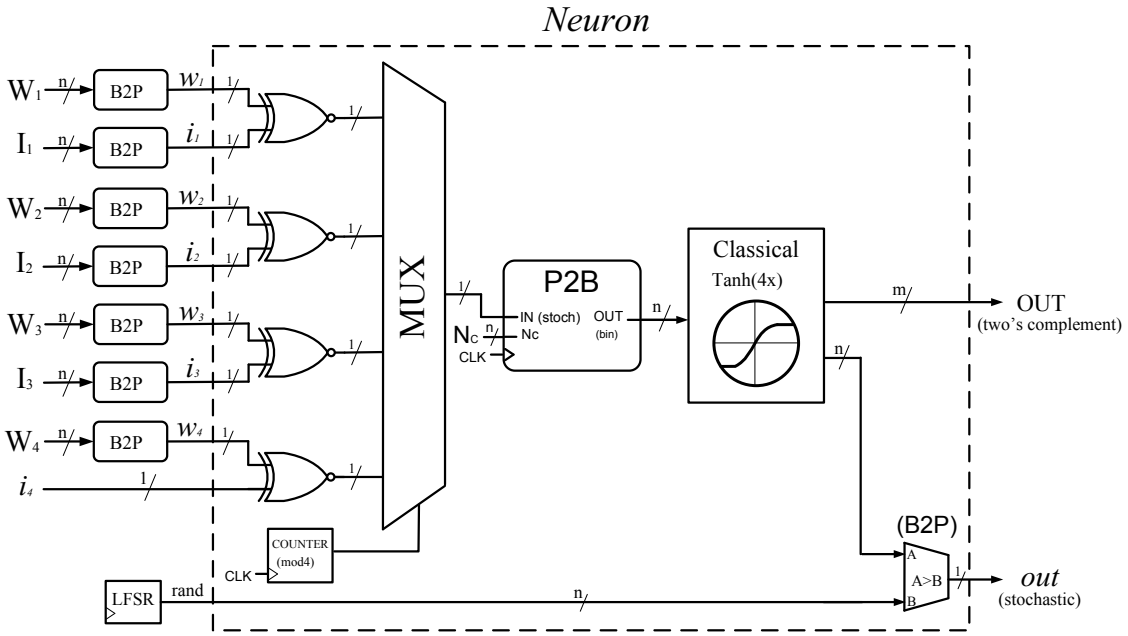


Figure 9.4.: SC-based four-input sigmoid neuron.

same character class), the probabilistic deviations of the stochastic approach do not severely hamper the classification task.

The classification behavior of the SC-based ESN (using $N = 100$ and $T_{eval} = (2^{12} - 1)\, T_{clk}$) is illustrated in Fig. 9.6 for two input samples (corresponding to the characters "a" and "b"). A clear recognition of the input category can be observed. The training of the system and calculation of the classification outputs (9.1) has been carried out as described in sec. 9.1. A set of 234 samples (containing the input signals of the three different characters "a", "b" and "c") has been used for training and a similar number, 220 samples, for testing. No validation set was considered due to the relatively low number of samples. The internal weight parameters have been fixed to $r = |v| = 0.95$ since the optimum weights in the SC-based ESN are usually near to 1. Nonetheless, an optimization process using a validation set might have improved the network's classification results.

Tab. 9.3 shows the classification accuracy of the SC-based ESN implementation (with 100 neurons) for different evaluation periods. A conventional software implementation employing the same number of neurons and the same internal weight configuration than the SC-based design performs the classification with a 96.8% of accuracy. The stochastic implementation tends to this value as the system's evaluation period (precision) is increased.

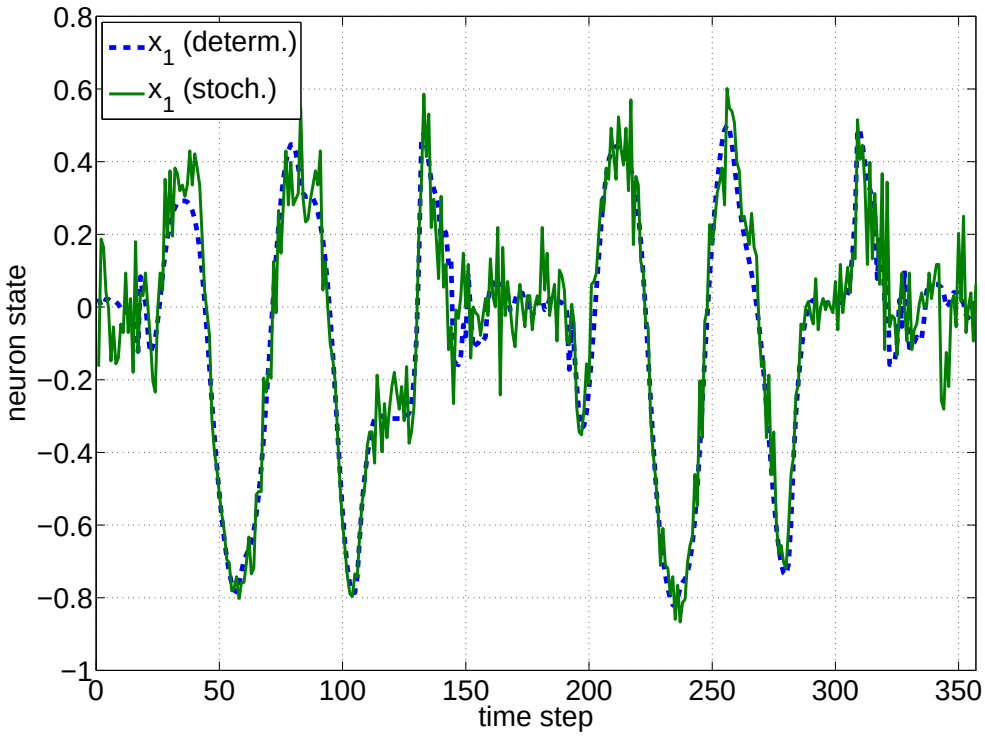


Figure 9.5.: Evolution of one of the neuron states (x_1) in a 200-unit ESN driven by the handwriting trajectories of the letter "a". The signal is represented for two input samples (178 time steps each) of the same character employing SC (for an evaluation time $T_{eval} = (2^{12} - 1)\, T_{clk}$, solid line) and a conventional software approach (deterministic, dashed line). The SC-based signal presents noisy fluctuations around the deterministic one.

$T_{eval}\,(T_{clk})$	***accuracy (%)***
$2^{10} - 1$	82.3
$2^{12} - 1$	87.3
$2^{14} - 1$	93.6
$2^{16} - 1$	95.4

Table 9.3.: Classification results of the SC-based ESN with $N = 100$ neurons in the handwriting recognition task as a function of the evaluation time (number of clock cycles).

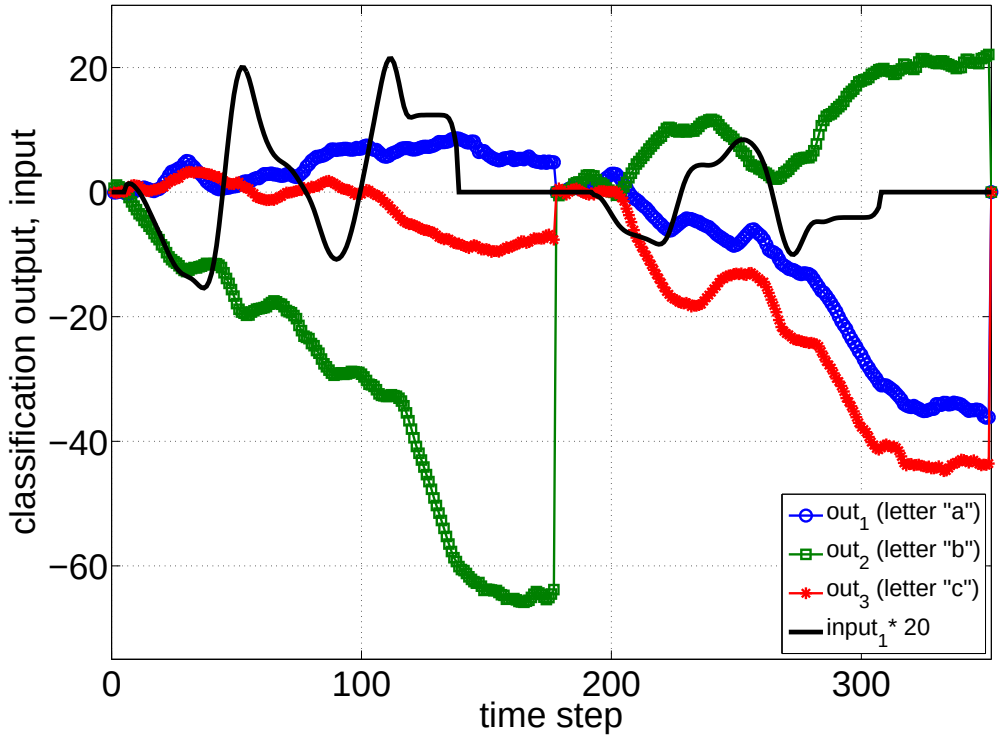


Figure 9.6.: Pattern recognition system behavior for a SC-based ESN with $N = 100$ neurons and $T_{eval} = (2^{12} - 1)\, T_{clk} = 4095\, T_{clk}$ ($time\ step \simeq 82\,\mu s$ @ $50\,MHz$). Two samples of the input signal (corresponding to the letters "a" and "b") are represented along with the output classifiers for the three possible patterns to be recognized (characters "a", "b" and "c"). A clear recognition of the input type is obtained.

9.3. Equalization of a wireless communication channel

9.3.1. Introduction

Communication systems are aimed at efficiently sending information from a transmitter to a receiver using an available channel. This requires a processing of the received data as the channel is always responsible for distorting to some degree the transmitted signals ([BB99]). The nature of the modifications suffered by the sent data depend on the particular features of the channel model, which can be either linear or nonlinear. In the case of satellite and mobile-phone communications, the sender's power amplifier must work in the high-gain region, close to the saturation point, in order to ensure a low use of energy. This adds important nonlinear distortions in the communication channel, which may be compensated either at the

transmitter side with pre-distortion or at the receiver side with equalization. Here, I focus on the digital compensation of the nonlinear channel at the receiver side. To sum up, the problem of channel equalization consists of designing a device (the equalizer) that is present in the receiver and is intended to cancel the distortions introduced by the physical environment used for transmission, thus enabling the correct recovery of the original information ([BLAZ11]).

The schematic of a communication system using a channel equalizer is illustrated in Fig. 9.7. The transmitter communicates the symbol sequence $d(t)$ as an analog envelope signal modulated on a high-frequency carrier signal. Then, it is received and demodulated into the analog signal $s(t)$, which is a corrupted version of $d(t)$. The main sources of corruption are the linear superposition of adjacent symbols ($q(t)$, intersymbol interference), nonlinear distortion induced by operating the sender's power amplifier in the high-gain region, and noise ($\nu(t)$, thermal or due to interfering signals). The corrupted signal $s(t)$ is then passed through the equalizer with the purpose of recovering the transmitted sequence ($d(t)$) or its delayed version $d(t-\tau)$, where τ represents here the propagation delay associated with the physical channel. Finally, the equalized signal $y(t)$ is converted back into a symbol sequence.

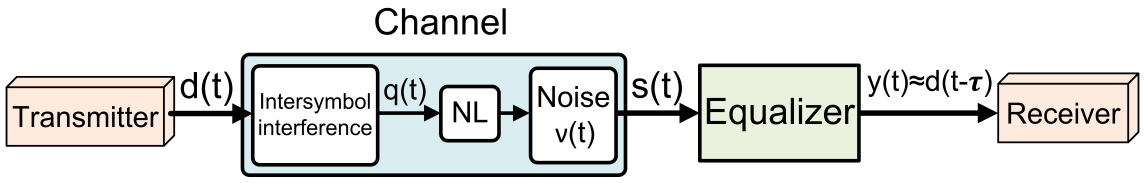


Figure 9.7.: Schematic diagram of a wireless communication system with a channel equalizer.

The function performed by the equalizer is adapted during the training stage so that the output $y(t)$ can restore $s(t)$ to $d(t-\tau)$ as closely as possible. Using the training data ($s(t)$ as input and $d(t-\tau)$ as desired output), the equalizer parameters (weights) are adjusted so as to minimize the error $e(t)$, defined as the difference between the target output $d(t-\tau)$ and the equalization output $y(t)$: $e(t) = d(t-\tau) - y(t)$. Once the training has been completed, the equalizer weights are fixed and used to estimate the transmitted sequence.

Linear filters have been widely used as equalizers due to their simplicity and mathematical tractability ([Luc65], [GL81], [Shi82]). However, their performance is not satisfactory for highly nonlinear and dispersive channels. Channel equalizers based on more complex approaches, such as polynomial filters (using Volterra series expansions, [KS89], [Mat91], [GR95], [MW11]) and artificial neural networks ([CGC90], [MP93], [PPBP99], [PMC09]), were developed showing a higher performance than the linear channel equalizers.

Among ANN models, echo state networks (ESNs) represent an attractive equalization solution since they are both nonlinear and recurrent, which makes possible to

meet the memory and mapping requirements of this particular task with the advantage of not posing complex optimization problems. The ESN approach has been proposed and investigated for the nonlinear channel equalization task using channel models of diverse complexity ([JH04], [SOP07], [BLAZ11], [RT11], [BSMH15]). The results presented in [BSMH15] show that the ESN is able to reach the same performance as a state-of-the-art Volterra equalizer and has similar complexity. Here, I propose and analyze the use of a digital hardware implementation of the ESN algorithm to perform the equalization of a wireless communication channel. More specifically, I select the implementation design based on stochastic computing (stochastic ESN, chapter 4). A low-power hardware implementation of the channel equalizer seems of great interest in wireless communications given the limited available power in mobile phone devices and aboard satellites. On the other hand, the high tolerance to soft errors provided by the stochastic computing approach is also desirable for an equalizer operating in harsh environments.

9.3.2. Methodology

To create the data set, I have taken the channel model of a nonlinear wireless transmission system from [JH04]. This model only considers real inputs. A more realistic extension making use of complex symbols can be found in [SOP07]. The channel is represented as a linear system with memory length 10 followed by a memory-less noisy nonlinearity. The input to the channel is an independent and identically distributed random sequence $d(t)$ with values from $\{-3, -1, 1, 3\}$. Then, $d(t)$ values are used to form a sequence $q(t)$ through a linear filter as follows:

$$\begin{aligned} q(t) = 0.08\, d(t+2) - 0.12\, d(t+1) + d(t) + 0.18\, d(t-1) \\ -0.1\, d(t-2) + 0.09\, d(t-3) - 0.05\, d(t-4) \\ +0.04\, d(t-5) + 0.03\, d(t-6) + 0.01\, d(t-7) \end{aligned} \tag{9.4}$$

Finally, a noisy nonlinear transformation is applied to generate $s(t)$:

$$s(t) = q(t) + 0.036\, q(t)^2 - 0.011\, q(t)^3 + \nu(t) \tag{9.5}$$

where $\nu(t)$ is an independent and identically distributed Gaussian noise with zero mean. More specifically, the noise has been adjusted in power to yield a signal-to-noise ratio (SNR) of 20 dB.

The equalization task consists in getting the output $y(t) = d(t-2)$ when the corrupted signal $s(t)$ is presented at the network input. A sequence example of symbols

entering the channel $[d(t)]$ along with the corresponding distorted signal $[s(t)]$ is shown in Fig. 9.8.

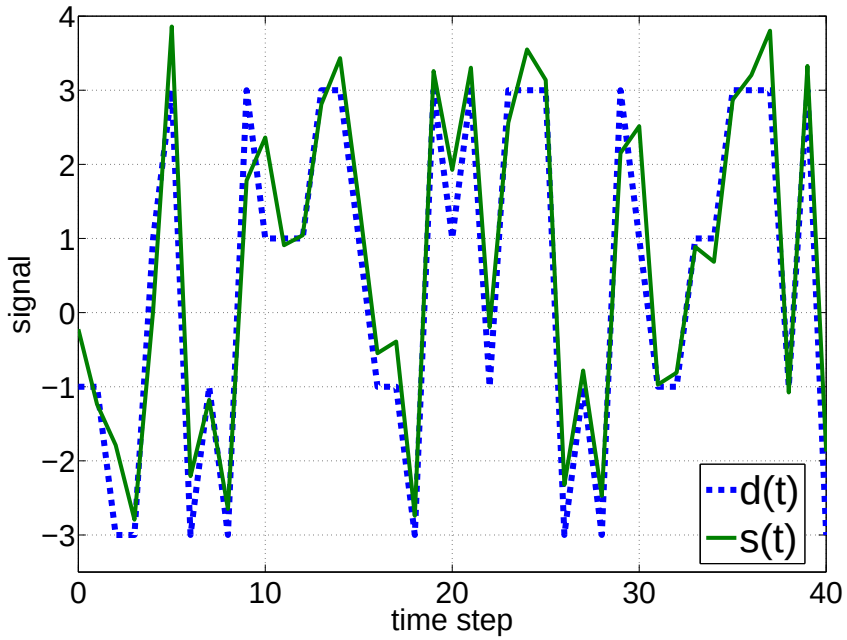


Figure 9.8.: Sequence example of input $(s(t))$ and desired output values $(d(t))$ for the channel equalization problem. The input signal corresponds to the corrupted values received after passing through the channel and the target outputs are the original values emitted by the source.

The equalized signal $y(t)$ needs to be finally converted back into a 4-symbol sequence. This is done by equidistant thresholding. That is to say, the symbol "+3" is chosen if $y(t) > 2$, "+1" if $2 \geq y(t) > 0$, etc (the limits employed for such symbol classification are illustrated in Fig. 9.9). Therefore, the equalization task is actually a classification problem where deviations of the estimated output with respect to the target signal are acceptable as long as the values are kept within the correct classification boundaries.

As in the two previous sections, the present task has been evaluated through the simulation of the SC-based ESN implementation (chapter 4) employing a numerical model that limits the resolution of the variables according to the hardware (8 bits for the neuron states and for the output weights, and 16 bits for the final classification readouts). The input signal $s(t)$ was normalized to the $[-1, 1]$ range before being entered into the network.

The quality for the equalization process is usually measured as the fraction of incorrect symbols finally obtained (symbol error rate, SER) although the error (e.g., the NMSE) between the desired and predicted signals can also be employed.

9.3.3. Results

The equalized signal provided by the SC-based ESN (using $N = 50$ and $T_{eval} = (2^{10} - 1)\,T_{clk}$) is illustrated in Fig. 9.9 for a fragment of the test sequence. It can be observed that the estimated output allows a correct classification of the input values in most cases. As usual, the system has been trained through conventional linear regression minimizing the square error $[y(t)–y_{target}(t)]^2 = [y(t)–d(t–2)]^2$. A sequence of 5000 values has been used to perform the experiment, of which the first 2000 samples were employed for training (disregarding an initial washout period of 200 points) and the remaining 3000 for testing. The internal weight parameters of the network have been fixed to $r = |v| = 0.95$ since the optimum weights in the SC-based ESN are usually near to 1. However, a different configuration could provide better results.

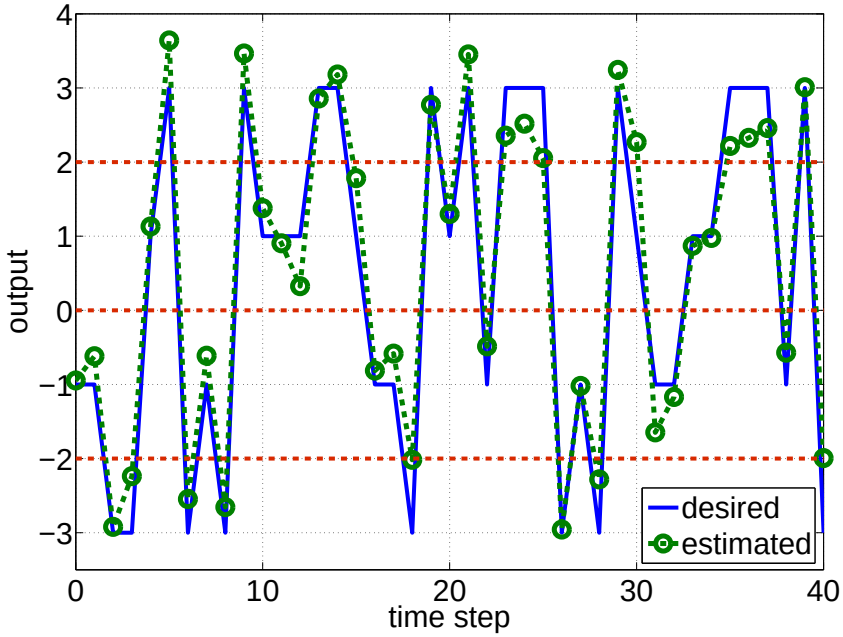


Figure 9.9.: Network estimations along with the targeted values. These results correspond to a fragment of the test set obtained with $N = 50$, $v = r = 0.95$ and $T_{eval} = (2^{10} - 1)\,T_{clk}$. The red dashed lines indicate the limits employed to classify the symbols (e.g., those values within the [0, 2] range correspond to the symbol "+1" while the values that are greater than 2 are classified as a "+3").

Tab. 9.4 shows the equalization error (NMSE) of the SC-based ESN implementation (with 50 neurons) for different evaluation periods. A conventional software implementation of the SCR network employing the same number of neurons and the same internal weight configuration than the SC-based design performs the task with an error $NMSE = 0.052$. It can be observed that the stochastic implementation reaches

this value with a system's evaluation period of 2^{12} clock cycles and that a shorter evaluation time of 2^{10} clock cycles allows a comparable result.

$T_{eval}\,(T_{clk})$	***NMSE***
$2^8 - 1$	0.091
$2^{10} - 1$	0.061
$2^{12} - 1$	0.052
$2^{14} - 1$	0.051

Table 9.4.: Performance results of the SC-based ESN with $N = 50$ neurons in the equalization task as a function of the evaluation time (number of clock cycles).

The results reported in [RT11] indicate that an optimization of the network internal weights (v and r) can lead to a higher performance than the one found here (NMSE of an order of magnitude smaller). Nonetheless, it must be mentioned that such small errors are unlikely to be obtained with our hardware implementations. This is due to the fact that the optimum network configurations correspond to very small values of the parameter v, which requires a high resolution for the variables and is not compatible with our limited-precision design.

Typical values of the symbol rate in satellite communications are about 20000-30000 symbols per second (usually noted as bauds, Bd, indicating the number of symbol changes across the transmission medium per time unit using a modulated signal). For example, the majority of TV channels of Astra ([web16a]) and Eutelsat Hot Bird ([web16b]) satellites use a symbol rate of either 22 kBd or 27.5 kBd. It is worth noting that the processing speed of our SC-based ESN hardware realization is compatible with these rates using an evaluation period of 2^{10} clock cycles, which implies a required time of 20.5 μs to equalize each value of the input sequence (assuming a clock frequency of 50 MHz) while the duration time of each symbol in the communication process is of 36.4 μs (assuming the rate of 27.5 kBd). In case of requiring a higher accuracy (longer evaluation times), it would be necessary to increase the system's clock frequency for the task to be performed on real-time.

9.4. Other applications

9.4.1. Control

Real-time control of nonlinear systems is often a complex and computationally intensive task. ANNs are attractive for the field of intelligent control given their parallelism, learning and adaptation capabilities, and fault tolerance. Usually, neural networks are combined with other approaches (e.g., finite-state automata, FSA) for the control of physical plants, machines or industrial processes ([CR95]). In some

cases, ANNs are trained to emulate existing controllers, and afterwards used to replace the conventional controller ([BL91], [KHB+98], [Bur05b]) with the advantage of increased speed of execution and fault tolerance. The selected hardware implementation of the ANN depends on the particular requirements of each application.

Model predictive control (MPC) is based on the use of an internal nonlinear plant model, which captures the main process characteristics, followed by a dynamic optimization that provides the optimal manipulated variables. The process models may be obtained from physical principles (e.g., [CdPdK05]) or be black-box data-driven models, which are mostly based on ANNs ([CRBV07]). In particular, reservoir computing, as a RNN approach, is appropriate for modeling dynamical systems. For example, in [WSS08b], RC has been successfully employed to model the output flow of a heating tank with variable dead-time. Details of such plant are given in [CdPdK05]. On the other hand, the works presented in [PCK+14] and [PLK+16] illustrate the use of ESNs for the position control of hydraulic excavators.

An example of neural-network digital chip implementation aimed at the real-time control of nonlinear plants and appropriate for low-power embedded applications requiring small size, low power consumption, and high reliability is presented in [CR95]. An analog neural network chip employed as controller for the unstable oscillations of a combustion engine is proposed in [LBH02]. An illustrative FPGA realization of an ANN for the control unit of an induction motor is found in [ZL08], where the hardware implementation (using the stochastic computing approach) is reported to exhibit lower hardware cost than a conventional microprocessor-based design for the same application.

9.4.1.1. Sensorless control of a wind turbine generator

ANNs are often used to implement a particular function required by the control unit. A good example of this is presented in [Qia09], where an ESN is used to estimate the wind speed from the measured electrical power in a wind turbine generator (WTG). The estimated wind speed is then used for controlling the WTG system. That is, the controller implements two functions: first, it estimates the wind speed (through the neural network); secondly, based on the derived wind speed, it generates the optimum rotor speed profile for maximal energy utilization and aerodynamic efficiency. Most controller designs employ anemometers to measure wind velocity in order to derive the desired generator speed. However, these sensors increase the equipment and maintenance costs of the WTG system. In addition, the potential failure of the sensors reduces the reliability of the overall system. Given the importance of high reliability, little maintenance and low cost (specially in the case of small wind turbines), a number of solutions have been proposed for the sensorless control of WTG systems through the real-time estimation of wind velocity from other available measurements, such as the turbine's produced mechanical or electrical power ([BSE99], [TI04], [BS05], [LSM05], [LZF06], [QZAH08], [Qia09]).

Among the different algorithms for wind speed calculation in WTG systems, ANNs present the advantage of learning capability, fast parallel computation and fault tolerance. In particular, ESNs seem more adequate than feed-forward neural networks to approximate the nonlinear dynamics of complex WTG systems ([Qia09]). On the other hand, although the proposed methods are usually implemented in software using the same microprocessor or digital signal processor (DSP) employed as controller ([LSM05]), hardware realizations are crucial to exploit the parallelism and fault tolerance of ANNs. An example of FPGA-based neural network implementation for this task is given in [LZF06]. Therefore, the RC hardware implementation designs presented throughout this thesis could be extremely useful for the present application. The possibility to implement the RC designs in low cost and low power devices (either FPGAs or application-specified integrated circuits, ASICs) seems particularly interesting for small wind turbines (less than 15 kW) given their requirements of reduced cost and capability to operate autonomously in remote places without power grid access.

9.4.2. Robotics

Reservoir Computing has been successfully applied in the field of robotics for mobile robot control ([SP05], [Bur05b]), event detection and robot localization ([ASD+07], [ASS08b]). In this context, several works have proposed the use of ESNs to learn navigation behaviors for mobile robots in different environments ([ASS08a], [AS15]) so that the high-dimensional reservoir space allows to learn multiple dynamic robot behaviors consisting of sensory-motor sequences based on examples of navigation behaviors (generated, for instance, by a supervisor). In the road sign problem ([ASS07]), for example, an artificial agent (robot) which is driving along a corridor receives a temporary sign at some specified time which must be remembered at a later moment in order to take the correct decision (to turn right or left). The difficulty of this task relies on the time gap existing between the sign and the decision. That is, the robot has to navigate holding the information (the sign) gathered in the past. RC is well suited to handle such type of temporal problems. The work presented in [ASVC07] shows how ESNs are used as an implicit model for robot localization based on the robot's sensory input history. In addition, the same reservoir setup is employed to model the robot controller. Liquid state machines (LSMs, [WHS13], [dACA16]) and Physical RC ([PVCL06], [HIF+11], [SHP11], [ZNS+13], [NHK+13]) have also been proposed for various robotic applications.

Small and energy efficient autonomous mobile robots are actively investigated for their potential application as domestic service robots ([SNS11]). Such robots must be easy to teach using training sequences and must perform their task (maintain a sense of position and navigate without human intervention) with a restricted amount of power. The RC hardware implementations proposed in this thesis seem adequate for the control of autonomous mobile robots given their capability of learning by example, their low cost and low power consumption.

Another important application of mobile robots is their use in dangerous and inhospitable environments. For example, they might be used in a nuclear disaster to access areas with high levels of radiation. A robot controller immune to non-destructive radiation (SEUs or soft-errors) is necessary in that case. The use of the LSM approach for this purpose has been proposed and investigated in [dAKS+16]. It has been observed that the system can withstand different noise levels, which could be seen as an example of the result from the non-destructive effects of radiation. In this context, a hardware implementation of the RC approach based on stochastic computing (SC, such as those described in chapter 4 and chapter 5) could be of great interest improving the system's reliability.

9.4.3. Wireless sensor networks

A wireless sensor network (WSN) consists in a network of distributed autonomous devices that can sense or monitor physical or environmental conditions cooperatively ([ASSC02], [KFV11]). Example applications of WSNs include agricultural monitoring ([LBV06]), habitat surveillance ([MHO04]), prediction and detection of natural calamities ([DVT+15]), military operations, medical and structural health ([WDH15]) monitoring among others ([KFV11]). A sensor network is usually composed of a large number of low-cost, low-power sensor nodes that are small in size and communicate with its neighbors without cables. Such sensor nodes are often grouped in clusters, and each cluster has a node that acts as the cluster head. All nodes forward their sensor data to the cluster head, which in turn routes it to a specialized node (the base station). Sensor nodes perform functions of sensing, data processing, and communicating, but are severely constrained in terms of storage resources and power supply, which also limits their computational capability. Actuators such as alarms or automatic irrigation systems (in agricultural applications) are often connected with WSNs.

Interpreting the information collectively gathered by sensor nodes in WSNs is a challenging task, specially considering that some sensor nodes may fail and such failure should not affect the overall task of the sensor network. Computational intelligence (CI) techniques, such as ANNs, are often used in WSNs for processing the sensory data and generating the desired responses while withstanding high fault tolerance ([CSC13], [AMN14]). In [SK08], the ESN learning concept is proposed to infer the spatio-temporal dynamics of the data collaboratively measured by sensors in WSNs. Such data processing could be performed either at a fusion center or distributively by sensors nodes.

An exemplary application using ESNs for processing the sensor data in a WSN is presented in [MP08], where the design and implementation of an intelligent pedestrian counter system is described. A pedestrian counter is a device used to measure the number of people traversing a certain passage or entrance per time unit. It can be applied to effective resource utilization, planning of service activities and ensuring safety and convenience. The proposed system employs low-cost passive infrared

sensors to detect the pedestrian movements. The wireless sensor nodes handle the sensor data acquisition and transmission, they all communicate (over a radio channel) to the base station computer, which processes the data. The ESN approach is the machine learning technique used to learn the motion patterns from the noisy sensors and predict the counts. In this case, the ESN algorithm is implemented in software on the base station computer. However, a hardware realization like the ones proposed throughout this thesis (using a low-cost FPGA or an specific chip) might be interesting for further reducing the overall cost of the system. Any of the different RC hardware designs of this thesis are, in principle, compatible with the 40-Hz sampling frequency of the sensors suggested for the pedestrian counter system ([web17b]), which corresponds to an available time of 25 ms to process each input value.

ESNs have also been used in a WSN for structural health monitoring ([WDH15]). The array of sensors was placed onto a test footbridge subjected to potentially damaging interventions. The measurements from a number of temperature sensors were employed as inputs to an ESN, which was tasked with estimating the expected output signal from several tilt sensors that were also placed on the footbridge. After training the ESN with the temperature and tilt sensor data (obtained before performing any intervention), the ESNs' prediction accuracy allowed inferences to be made about when interventions occurred and also the level of damage caused. In addition, the damaged regions could be determined using the error in signals and the location of each of the tilt sensors. Such a system appears to be of great value to industry.

The above-mentioned examples of sensor networks using the ESN approach were designed to directly transmit data from the array of sensors to a server data repository. Nevertheless, the gathering of the distributed data at a central station requires high communication costs, and therefore collaborative information processing through distributed algorithms is highly desirable. That is, an interesting alternative is to implement more "intelligence" on the sensor nodes (rather than at the base station) prior to sending messages since it can increase the efficiency of communication. For instance, the elements of the network deciding what data to pass on, using local area summaries and filtering may minimize the power use while maximizing the information content. The RC hardware designs proposed in this thesis could be particularly useful to implement data processing functionalities in the sensor nodes given their powerful computational capabilities using limited resources (power and chip area). For example, in the pedestrian counter system, each sensor node could implement an ESN and count locally.

9.4.4. Medical applications

RC has been used in biomedical applications with great success. For example, ESNs have been used for the analysis of biological signals to detect epileptic seizures

on real-time with high accuracy ([BVvM+11], [BVN+13]). Epileptic seizures are characterized by recurrent atypical brain activities with unusual excessive electrical discharges ([SAC+10]). One of the commonly used methods for diagnosing epilepsy is analyzing the brain electrical activity (electroencephalography, EEG, signals) recorded over a prolonged period of time. An automated detection method for epileptic seizure can perform such lengthy inspection process without human intervention, thus saving a significant amount of doctors' time. Furthermore, an automated system can generate an alarm requesting for medical help in case a positive detection is made. Usually, sophisticated and computationally intensive signal processing tools (e.g., ANN-based classifiers, [AKS05] and [SE05]) are used for software-based epileptic seizure detection, which is mainly aimed at obtaining high accuracy without regard to detection latency, hardware cost, or power consumption.

On the other hand, hardware-based detectors (appropriate for wearable embedded devices) must take into account the real hardware implementation constraints, and therefore tend to focus on low power consumption to achieve extended battery life often at the cost of using less complex algorithms with lower overall accuracy. See, for example, [PCFB09], [RWRI09], [RGW+09] and [SAC+10], which present ASIC and FPGA-based implementations for the EEG signal processing and automated real-time detection of seizure events. Hardware embedded devices consuming low power and allowing online processing of EEG signals are particularly interesting as they enable treatments for epilepsy that are based on rapidly detecting the seizure and actively counteracting it using medication or brain stimulation. The ESN digital hardware designs presented in this thesis could possibly be good candidates for their use in portable epileptic seizure detectors given their high computational capability (the ESN approach have already been shown to be adequate for EEG signal classification in [BVvM+11] and [BVN+13]) and relatively low power consumption. The sampling frequency in clinical scalp EEG is typically 256-512 Hz, which is compatible with our RC hardware implementations.

Similarly, RC (in particular, the single dynamical node approach, sec. 2.3.3) has been successfully used for the classification of electrocardiograms (ECGs, [EMSFM15]). ECGs are widely used for the detection of cardiovascular diseases, such as arrhythmia. They can be conveniently acquired with low-cost portable devices outside the clinical environments. However, as for the case of EEG signals, the analysis of long-term monitored heartbeat signals is very time-consuming. This motivates the development of algorithms to automate the ECG classification. Several machine learning techniques have been applied to the heartbeat classification problem ([JDP15]), such as ANNs ([GTK+14]) and support vector machines (SVMs, [CLO+13]). Hardware implementations of such algorithms are also desirable for the personalized real-time classification of ECG signals through embedded systems aimed at the long-term patient monitoring (see, e.g., [JC10] and [JC13], where FPGA-based realizations are presented). Therefore, our power-efficient and low-cost FPGA-based implementations of RC seem well suited to real-time ECG classification. ECG signals are usually sampled at frequencies equal to or under 1 kHz ([MJPV15]), which is, in

general, compatible with the implementations proposed in this thesis.

ESNs have also provided good results in the monitoring, parameter estimation and event detection of fetal ECG using noninvasive measurements, which are noisy and contaminated by strong interferences, such as maternal ECG and fetal brain activity ([LM13]). Some more medical applications of the ESN technique related to the processing of ECG signals are presented in [PML11], [PMSL12b] and [PMSL12a].

9.4.5. Image and video processing

Deep neural networks (DNNs) and convolutional neural networks (CNNs) have been widely investigated for visual object recognition ([CMGS10], [CMS12], [CM15], [OZW+16]) becoming state-of-the-art methods for solving classification or recognition tasks on many data sets of images or videos, such as the MNIST digits data set ([LBBH98]). RC has also been proposed and analyzed as an alternative for computer vision applications in [JWW15], [HSD+15] and [SSC16]. In particular, these works have investigated the potential of RC networks for the task of offline handwritten digit recognition. In [JWW15], for example, the authors employed a network structure consisting of three stacked reservoirs, each of them composed of 16K neurons (forming a "deep" RC network), to solve the digit recognition problem on the MNIST data set. They reported a digit error rate (DER) of 0.92%, which is competitive with former state-of-the-art systems (presenting a minimum DER of 0.6%), but cannot outperform them. The main advantage of RC over CNNs and DNNs is the simple and fast training, which allows learning large data sets in reasonable time. In addition, RC presents high robustness generalizing well to noisy conditions.

RC-based image processing systems could be applied to perform more elaborated tasks such as face detection, medical image analysis or plate number identification. It is worth mentioning that ESNs have been used for image restoration presenting better results than other state-of-the-art-methods ([DW16]).

Hardware acceleration of neurocomputing has received much attention in recent years for image recognition and classification. See, for instance, [AKK+16], which presents an image sensor that integrates a CNN for intelligent vision processing. The hardware implementation allows to harness the inherent parallelism of the neural algorithm. A different hardware design is proposed in [UHS16] to reduce the power consumption of CNN-based image recognition. [LGMARB+05] provides another example of hardware implementation (using an FPGA) of a cellular neural network to perform real-time image processing for mobile robot vision.

Among the hardware designs presented in this thesis, that of chapter 6 could be particularly useful to accelerate the RC calculations (compared to a conventional processor) given its highly parallel processing capabilities using minimum hardware resource, which is specially desirable considering the high number of nodes (~ 50000) required in the reservoir to perform image classification tasks.

RC can also be useful for the processing of video sequences. In [YM12], an RC model consisting of multiple sub-reservoirs is proposed for multi-object behavior recognition. Enabling computers to recognize human actions in a video stream is critical in many areas, such as video surveillance, human-computer interaction and clinical diagnosis. In computer vision, different classifiers have been used for analyzing human behaviors from video sequences (e.g., SVMs, [HCL+09], and ANNs, [PDS09]). The RC-based method proposed in [YM12] could be effectively applied to the challenging task of classifying different activities performed simultaneously by multiple objects/persons in the same video.

The work presented in [JWW15] has also investigated the application of RC networks on video processing. The practicality of RC for real surveillance is illustrated with the task of real-time door status (open, closed, half open) detection using very low resolution camera sensors. The proposed RC-based door status detector was shown successful on the real environment in which the surveillance camera was used (e.g., passing people, masked frames, different camera angles and light intensities) indicating the robustness of the system on the unseen conditions. The door state was monitored using low-quality cheap visual sensors that can be built into light switches and power outlets. The implemented function could be useful for the control of the heating or lighting in a building. In addition, such RC-based event detection system using inexpensive sensors could possibly be used for more complex tasks, such as human mobility monitoring ([EBD+14]). A digital hardware design allowing to cheaply integrate the RC algorithm together with the visual sensors might be of great interest for automatically processing the raw pixel data and directly providing the desired information of the environment. Given the low capturing rate of the sensors (90 frames per second), any of the implementations proposed in this thesis would be compatible with the real-time processing of the video sequences.

9.4.6. Speech recognition

In recent years, a great deal of research effort has been directed to the task of automatic speech recognition through different ANN approaches. Convolutional neural networks (CNNs, [YD15]) and long short-term memory (LSTM, [LW15]) recurrent networks are among the best performing techniques. The latter method is used, for example, for voice searches, commands and dictation in smartphones through the Google app ([SSR+15]). Nevertheless, significant improvements are still needed before solutions will be available for voice driven applications with strict specifications such as high accuracy and robustness against confounding factors ([JTDM15]). ESNs and LSMs have also been applied to the domain of speech recognition ([VSSVC05], [VSDS07], [SH07], [web09], [TJSM10], [TDM14], [ZYCS15]) performing particularly well in noisy conditions ([JTVM11], [JTDM15]). Indeed, a few digital hardware implementations (using FPGAs) of the LSM approach have already been realized for real-time spoken digit recognition using limited hardware

resources and presenting a considerable speedup compared to a conventional processor ([SDVC07], [SDVC08], [WLL16]).

9.4.7. Others

The possibility of using RC systems as black-box models to estimate temporal functions that depend on a number of inputs without the need of deriving an analytical model makes them useful for very diverse applications. An illustrative example of the use of RC for function approximation is found in [ACRMM16], where I have employed ESNs for estimating daily global solar radiation from temperature data. This can be interesting for locations where solar radiation measurements (requiring expensive equipment) are not available. The proposed ESN approach has been shown to outperform conventional techniques using explicit model equations.

RC-based black-box models can also be effectively used in forecasting tasks as demonstrated in chapter 5, where the presented implementation has been applied to predict the radar back-scatter from an ocean surface. The difference between predicted values and real observations can be used for the detection of objects in sea clutter. A similar approach based on observing the discrepancies between the system's expected behavior (according to the network predictions) and the real monitored one can be of general utility for fault detection purposes. ANNs have been proposed, for instance, for the detection (and classification) of faults in a photovoltaic (PV) electricity production plant ([JM15]). An alarm is activated when the measured power from the real PV system considerably differs from the network predicted values. As for the case of control applications, hardware implementations enabling the real-time automatic fault detection through low-power and low-cost devices are desirable.

The application field of wireless sensor networks (WSNs) described in sec. 9.4.3 is very related to the concept of the "Internet of Things" (IoT, [AIM10]), which refers to the internetworking of "smart" devices that may be vehicles, buildings, mobile phones or any other objects embedded with electronics, sensors, actuators and network connectivity that enable such items to collect and exchange data. The IoT allows objects to be sensed and/or controlled remotely across the network infrastructure improving efficiency, accuracy and economic benefit. Machine learning (ML) techniques, such as ANNs, are generally employed in IoT ([MRC+16]). In particular, ML hardware implementations (as the RC systems presented in this thesis) are of great interest since low power and low latency "smart" chips have the potential to make a wider type of objects become intelligent things in IoT allowing for a more distributed intelligence.

Financial forecasting is another potential application of ML algorithms, such as recurrent neural networks (RNNs, [ZB01]) and RC ([IJK+07]). For example, [MM15] presents an investment support system based on an ensemble of RNNs. The system processes historical data and makes predictions that can be eventually used to trade

successfully on the currency markets. The information obtained from such support system gives investors an advantage over uninformed market players in making investment decisions. A successful application of ESNs to financial time-series forecasting is presented in [IJK+07]. In this case, the predictions were obtained by combining the outputs of ensembles of 500 independently created reservoirs with sizes ranging around 100 units. Hardware implementations (e.g., using high-performance FPGAs) could be extremely useful to support such massive neural computations (with $\sim$ 50000 neurons) ensuring an efficient and timely data processing. Such hardware-based co-processing might be particularly interesting for high-frequency trading ([Ald09]), which consists in a completely automated execution of trading instructions (i.e., carried out by computers) using sophisticated algorithms and characterized by high speed (operations are often performed in fractions of a second), thus requiring a very fast decision making capacity based on big amounts of complex data.

Finally, it is worth mentioning that the RC implementations developed in this thesis could also be of interest in the bio-metric security sector where authentication algorithms based on the recognition of behavior, face, voice, signature or gait are required.

9.5. Discussion

In this chapter, I have described a number of potential applications of the RC hardware designs developed throughout this thesis, which highlights the usefulness of the proposed systems.

The performance of the SC-based ESN hardware implementation presented in chapter 4 has been evaluated for two real-life engineering applications (handwriting recognition and equalization of a nonlinear communication channel) and satisfactory results have been found. The hardware realization is compatible with the applications in terms of processing speed (they could be performed on real-time) and presents potential advantages over a conventional microprocessor-based implementation, such as a reduced power consumption and a higher reliability thanks to the parallel implementation design. In addition, the particular SC-based system presents a graceful degradation of the classification accuracy with the injection of soft errors (fault-tolerance).

More specifically, the implementation of a handwriting recognition system on a specific integrated circuit can be of interest in order to perform the task more efficiently and to release processor resources on a computer or PDA. The classification task (here limited to the recognition of three characters) could be exported to a higher number of classes. A similar system might be used for other purposes, such as signature authentication. On the other hand, a low-power hardware implementation of the channel equalizer seems of great interest in wireless communications given the limited available power in mobile phone devices and aboard satellites.

In general, the SC-based ESN approach have been found adequate (even using relatively short evaluation periods) for pattern recognition applications that present inherent variations of the input signals and for the equalization of a noise-corrupted signal. Classification tasks allow a considerable reduction of the required evaluation time compared to time-series prediction, specially if a certain degree of inaccuracy is acceptable. Nevertheless, the SC-based design is considerably more time-consuming than its counterparts (for example, compared to the approaches of chapter 6 and chapter 7), and therefore its practical use may be limited to particular applications where reliability is paramount, such as a robot controller or a communication channel equalizer operating in harsh environments.

RC, as a RNN approach, allows to generalize time-dependent relationships between the input data set and the outputs without requiring the explicit physics behind the process. In other words, it can be employed as a black-box model for complex dynamical systems, which enables to perform prediction, classification and control tasks in very different fields, such as robotics, computer vision (image processing), sensor networks, medicine or finances.

The digital hardware designs developed in this thesis allow to address a wide spectrum of applications from mobile/autonomous objects to high performance computing co-processing. That is, on the one hand, the proposed RC designs can be used to efficiently implement a powerful machine learning technique employing few hardware resources and consuming low power (through fabricated ASICs or low-cost FPGAs), which can be useful for wearable physiological monitoring systems, wireless sensor networks or robotics and control applications. On the other hand, the RC systems may be implemented on high-performance devices (e.g., FPGAs) to support and accelerate massive computations required, for example, for image, video or financial data processing.

Part III.

Conclusions

10. Conclusions and future work

10.1. Comparative results

Tab. 10.1 and Fig. 10.1 summarize the results obtained for the different machine learning circuits developed and analyzed in this thesis. The area requirements (evaluated in terms of the FPGA's consumed logic elements), processing speed and prediction accuracy in the Santa Fe benchmark task are presented for the different reservoir schemes using a common number of 50 nodes. It must be noted that each design has been elaborated independently and with different characteristics, such as the nonlinear activation function approach, the precision employed for the variables, or even the system's clock frequency.

Implementation scheme	Logic elements	$\mathbf{T}_{eval}$	NMSE
Conventional ESN	19,147	20 ns	0.090
Stochastic ESN	5,306	1.31 ms	0.077
ESN without multipliers	2,497	20 ns	0.092
Delay-based ESN	3,085	2.04 μs	0.090
Single dynamical node RC	1,605	0.87 ms	0.131

Table 10.1.: Comparative results of the implementations proposed throughout the thesis. We show the hardware resource usage (number of logic elements), the time required to process each input (T_{eval}), and the accuracy (evaluated through the NMSE) in the Santa Fe time-series prediction task. All the reservoir networks are designed containing a total number of 50 nodes.

Firstly, it is worth remembering that a very simple design has been employed to approximate the sigmoid function in the "conventional ESN", the "ESN without multipliers" and the "delay-based ESN" while a more accurate approach was used in the "stochastic ESN". Such more-elaborated implementation of the activation function requires 23 logic elements more than the simpler design. Therefore, the use of this function in the fully-parallel "conventional ESN" and "ESN without multipliers" would require an additional number of 1,150 logic elements (for a network with $N = 50$ nodes), but only 23 additional logic elements for the case of the sequential "delay-based ESN" scheme. On the other hand, the use of a more realistic approximation for the sigmoid function implies a higher prediction accuracy. In chapter 4 (see the "deterministic" results of Fig. 4.18), it has been shown that the

use of an exact sigmoid function, even with a truncated 8-bit resolution, leads to a reduced prediction error of $NMSE = 0.044$. Thus, a value close to this could be expected for the deterministic (i.e., non stochastic) "conventional ESN", "ESN without multipliers" and "delay-based ESN" designs when using a faithful activation function.

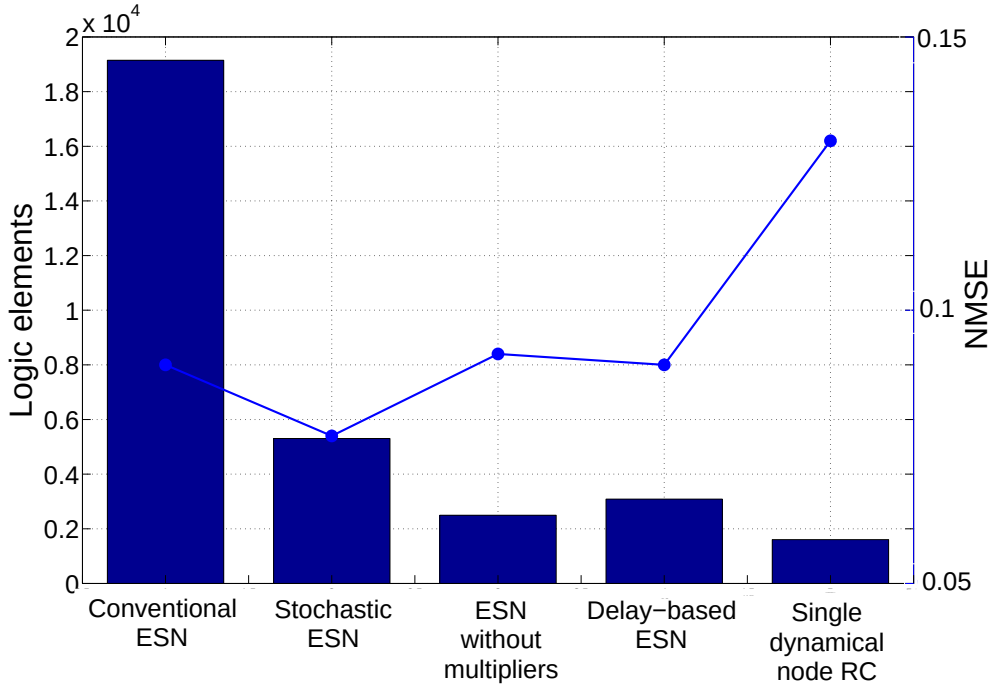


Figure 10.1.: Comparison of the resource usage (logic elements; represented with bars, left axis) and accuracy (NMSE; represented using points, right axis) of the different implementations proposed throughout the thesis. All designs use a number of 50 nodes and are evaluated for the Santa Fe time-series prediction task.

Regarding the "stochastic ESN" approach, it must be reminded that its accuracy to perform a certain task depends on the selected evaluation time. In Tab. 10.1, the values of the prediction error and T_{eval} have been provided for the case of $T_{eval} = 2^{16}\,T_{clk}$ and assuming a clock frequency of 50 MHz. The error is reduced to $NMSE = 0.071$ when the evaluation time is increased to $T_{eval} = 2^{18}\,T_{clk}$, which corresponds to 5.24 *ms* (for a clock frequency of 50 MHz). Likewise, shorter evaluation periods can be used if a lower accuracy is acceptable, as illustrated in Fig. 4.18. It must also be noted that the alternative design for the stochastic ESNs presented in chapter 5 (although only evaluated for a reservoir with 20 neurons) allows a significant hardware resource saving (more than 17% of area reduction) while keeping a similar performance.

As regards the "single dynamical node RC", it is worth mentioning that a number of 16 embedded multipliers of 9-bit elements was employed in addition to the in-

dicated logic elements. The hardware resource of the 16 multipliers approximately corresponds to 400 logic elements (assuming that one of the factors in the multiplications can be set as a constant value). Therefore, the number of logic elements required by this design would be increased to approximately 2005 if no embedded multipliers were used (as is the case of the other implementations). On the other hand, a clock frequency of 200 MHz was used to perform the calculations in this implementation whereas 50 MHz were used for the other designs. Consequently, the time period required to process each input sample (0.87 *ms*) must be multiplied by 4 to be correctly compared with those corresponding to the other approaches. Finally, it must be noted that most of the variables in this implementation have been limited to a resolution of 8 bits, but 16-bit numbers were usually employed in the previous designs. Simulations of the system applying no restriction on the precision of the variables showed that the error could be decreased down to $NMSE = 0.051$ for a high-resolution implementation. Obviously, this would also imply a larger resource requirement.

Considering the previous remarks, we can state that the ESN without multipliers is the best implementation in most aspects. Although it requires slightly larger resource than the "delay-based ESN" and the "single dynamical node RC", it offers the advantage of being a fully-parallel design. The concurrent implementation allows a very high processing speed as the time required to process each input sample is a single clock cycle (20 *ns* considering a 50-MHz clock) whereas the sequential approaches present a much higher latency. Compared to the "conventional ESN" approach, it offers an enormous resource reduction at the only cost of a very small accuracy loss. Therefore, it seems to be the ideal design to support real-time applications requiring very large computational power. Potential examples include high-volume image/video processing ([JWW15]), real-time time-series prediction tasks requiring large reservoirs (e.g., financial forecasting, [IJK+07]) and, in general, applications where specialized neural hardware can conveniently accelerate the required computations. In such co-processing scheme, the reservoir network could be efficiently implemented using a low-cost hardware device while the output layer would be executed through software in the conventional processor.

The "delay-based ESN", as a sequential emulation of the conventional parallel design, allows a significant reduction in the number of logic elements at the cost of a slower throughput. Note that the evaluation time in Tab. 10.1 is given for a reservoir with 50 neurons, but it increases linearly with the number of nodes (as indicated in equation 7.3 and Tab. 7.2). It is worth mentioning that the hardware resources employed for this system could be further decreased combining the serial processing of the nodes with the "multipler-less" technique of chapter 6. In addition, as observed in chapter 7, the implementation could be optimized replacing the block of registers used for the delay line by an internal RAM memory. Therefore, the "delay-based ESN" represents a highly compact design, which makes it particularly well suited to autonomous applications requiring low power consumption. Despite the limited processing speed compared to a concurrent design, this type of imple-

mentation, even using a low 50-MHz clock frequency, is compatible with numerous real-time applications, such as control tasks ([LZF06], [ZL08]), medical signal monitoring (e.g., for epilepsy, [SAC+10], or arrhythmia detection, [JC13]), wireless sensor networks ([MP08]), and online handwriting ([PS00]) and speech ([SDVC08]) recognition systems. All these applications usually require sample times no lower than 10 μs. Obviously, higher clock rates (available, e.g., in ASIC-based implementations) would accordingly increase the system's processing speed.

The "stochastic ESN" approach, based on stochastic computing, reduces considerably the gate count if compared to the "conventional ESN" design at the cost of a lower accuracy and longer evaluation time. Stochastic computing makes possible to trade off the system's computation time and accuracy without hardware changes. That is, stochastic implementations can be relatively fast if a certain loss of accuracy can be tolerated and, on the other hand, they can be quite accurate by setting long enough evaluation periods. Thus, the "stochastic ESN" is potentially useful in specialized systems where low power is required and either limited precision or speed is acceptable. For the case of time-series prediction tasks, where high precision is usually targeted, the required evaluation time is too high if compared to the delay-based and multiplier-less ESN approaches (as shown in Tab. 10.1), and therefore this approach is not appropriate for certain real-time applications. The stochastic implementation seems to be more adequate for applications where accuracy is not critical. This could be the case of temporal pattern recognition or signal equalization tasks where the input streams are often corrupted by noise or present inherent variations. For example, the tasks presented in sec. 9.2 and sec. 9.3 (online handwriting recognition and nonlinear channel equalization) can be performed relatively well with evaluation times of $2^{10}\,T_{clk}$ or even $2^{8}\,T_{clk}$. These time periods correspond to 20.5 μs and 5.1 μs (assuming a 50-MHz clock frequency), which are comparable to that of the "delay-based ESN" approach.

Nevertheless, the most appealing feature of the stochastic computing design is its high degree of error tolerance. On the one hand, contrary to the delay-based approach, the stochastic ESN design performs a parallel execution of all the reservoir neurons, which represents an advantage since the system (a redundant network structure) can operate continuously (although with reduced accuracy) in case of failures in one or some of the network units. On the other hand, stochastic circuits tolerate environmental non-permanent errors that can seriously affect the behavior of conventional circuits, where a single bit flip may cause a severe malfunction, but flipping a few bits of the bit-streams employed for stochastic computing has little impact. The high tolerance to soft errors of the "stochastic ESN" has been illustrated for a time-series classification task in sec. 9.1. To sum up, the stochastic approach can be particularly interesting for applications that need to operate in error-prone environments. Examples of this include mobile robot controllers capable of withstanding the non-destructive effects of radiation (e.g., for being used in a nuclear disaster, [dAKS+16]) and spacecraft or satellite electronics (e.g, for equalization of the communication channel, sec. 9.3).

The "single dynamical node RC" is based on a serial processing scheme similar to the "delay-based ESN" design, which makes possible an implementation consuming very limited resources. In this case, however, the outputs of the virtual nodes are obtained through the solution of a differential equation after an appropriate number of integration steps. This not only increases the complexity of the design, but also implies a slow-down of the information processing compared to the serial ESN implementation. In general, the potential applications of this approach are the same as for the delay-based ESN. That is to say, electronic systems requiring to implement a powerful computational intelligence technique with power consumption constraints, such as wireless sensor networks, predictive controllers and monitoring medical devices. Nevertheless, it must be noted that despite being slower and more complex to implement, the single dynamical node approach does not outperform (in terms of accuracy) the simpler delay-based ESN methodology. In conclusion, the digital hardware realization of this type of system does not appear to be particularly beneficial compared to the other proposed implementations. Such computation scheme seems to be more adequate for optoelectronic implementations, where a relatively simple optical reservoir is possible based on the use of an optic fiber and a single optoelectronic device. This can be particularly useful when the information is already in the optical domain as in the case of telecommunications and image processing applications.

10.2. Conclusions

Today's society requires smart methodologies for extracting useful information from the increasingly large amounts of data it generates. Machine learning techniques, such as artificial neural networks (ANNs) allow automating tasks that are complex to be programmed with sequential computers and represent a powerful tool for numerous engineering applications. More specifically, recurrent neural networks (RNNs), characterized by feedback connections between neurons, are endowed with the ability to process temporal information.

Since its appearance in the last decade, reservoir computing (RC) has attracted the attention of many researchers as a practical approach to implement and train RNNs. By fixing the connections in the recurrent part of the network (referred to as the "reservoir"), the usually complex training process of RNNs is reduced to a simple linear regression problem. More specifically, the reservoir represents a dynamical system that is excited by the input signals. The states of such system are measured for a number of input samples and properly combined to produce the target output. The desired task can be a regression (mapping the input sequences onto real-valued output streams) or a classification of the input time-series. Several versions with subsequent ramifications have been proposed for the RC technique: echo state networks (ESNs), liquid state machines (LSMs) and Physical RC approaches, such as those based on a single dynamical node with delayed feedback. In general,

RC methods represent a powerful tool for processing sequential data. They have been successfully used for speech recognition, chaotic time-series series prediction, robotics, medical, financial and many other applications.

Dedicated hardware realizations of RC can provide high speed gains and power savings compared to software implementations executed on conventional processors. However, methods for such hardware implementation still deserve to be extensively explored. The majority of the previous physical realizations of the RC approach are based on optoelectronic systems using the single dynamical node approach. Only a few digital hardware implementations have been proposed, which are mainly focused on LSMs. To the best of my knowledge, in this thesis I present the first digital hardware realizations of ESNs and of the single dynamical node reservoir computer.

Six different methodologies have been developed aimed at the efficient implementation of RC systems in digital hardware: a conventional reference ESN design, two approaches for ESNs based on stochastic computing, a multiplier-less ESN scheme, a sequentially operated ESN, and the single dynamical node RC approach. The different proposed designs have been implemented in reconfigurable devices showing their functionality and evaluating their performance for time-series prediction and classification tasks.

The results of the various implementations have been analyzed and compared for a benchmark task discussing the advantages and shortcomings of each approach and their suitability for particular applications. This comparison may be useful for potential users of the presented designs to select the most adequate technique according to their specific needs.

To facilitate the usually long development process of hardware implementations, a detailed information of the proposed designs has been provided. In most cases, the corresponding VHDL hardware description codes are given. Such presented codes can be easily adapted to different network configurations with the desired number of nodes and connection weights. Software programs (using the MATLAB language) were employed to generate the VHDL codes of the designs tested throughout the thesis. Such a tool is able to automatically generate a text file containing the VHDL code of the desired reservoir. This way, the hardware design and implementation cycle are accelerated, especially for networks with high number of nodes and when different configurations must be realized. These programs may be easily elaborated (by any potential user of the suggested implementations) by just reproducing the structural codes provided in Appendix A for the desired network configuration.

Similarly, different programs (also using the MATLAB language) were developed for conveniently emulating by software the various hardware implementations, which is useful to adequately train the systems and to evaluate them prior to hardware implementation.

The proposed hardware realizations represent an effective solution for applications where software implementations are not satisfactory, such as those demanding real-time processing capabilities and/or involving power restrictions. In addition, some of

the hardware designs offer a higher reliability than single processor implementations, which may be of interest for safety-critical applications or for those that need to be operated in harsh environments. All in all, the dedicated hardware realizations developed in this thesis broaden the scope of application of the RC approach.

The usefulness of the implementations has been highlighted applying a selected design to perform different tasks of practical relevance, such as online handwriting recognition and the equalization of a communication channel. Furthermore, numerous examples of potential application areas of the developed systems have been described.

10.3. Future work

Throughout the present thesis, I have presented the first digital hardware realizations of the ESN model and of the single dynamical node RC system. For the ESN implementations, I have selected a reservoir network with simple cyclic structure (the SCR), which performs with comparable performance to the classical random architecture while simplifying the design. Nevertheless, the several approaches proposed for reservoir implementation could be extended to realize other architectures, such as the cycle reservoir with jumps (CRJ, [RT12]) and the adjacent-feedback loop reservoir (ALR, [SCL$^+$12b]), which are variations of the SCR construction that allow a superior performance at the cost of slightly higher complexity.

Another interesting variation of the classical ESN consists in a network of multiple reservoirs sparsely interconnected, which has been shown to be more accurate than conventional single-reservoir ESNs ([XYH07], [DZ07], [NR15]). A similar approach (named "deep reservoir computing network" in analogy to deep neural networks, [JWW15]) makes use of multiple layers of reservoirs, which also enables discovering features on different time scales ([Jae07a]) and presents outstanding performance in a variety of tasks ([PDW14], [JWW15]). Since both methods often require a huge number of nodes (see, e.g., [IJK$^+$07] and [JWW15], where approximately 50,000 neurons are employed), a hardware implementation could be particularly useful to accelerate the computations. The multiplier-less ESN design seems a good candidate to efficiently realize this type of massive networks with multiple reservoirs. Another feasible option could be the use of a delay-based ESN sequential structure to implement each one of the multiple sub-reservoirs so that sequential and parallel processing would be combined in the general network (sequential computations to evaluate the nodes within each sub-reservoir and parallel assessment of the different sub-reservoirs).

The developed implementations (or variations of them, such as the mentioned ensembles of reservoir networks) may be employed in a wide spectrum of applications as described in chapter 9. Financial time-series forecasting and image classification tasks (e.g., of handwritten digits) seem particularly interesting due to their apparent suitability to be performed by massive reservoir networks as illustrated by the

promising results reported in [IJK+07] and [JWW15]. Similarly, RC could be useful in the field of renewable energies. For example, it is worth investigating RC systems for the prediction of wind and solar power generation ([COA+16a]), which in turn may be used to estimate the evolution of energy prices. On the other hand, the presented designs could be applied to the field of brain activity monitoring for the real-time detection of epileptic seizures using portable devices, which require low-power hardware designs to ensure an extended battery lifetime. Although several hardware implementations have already been developed for the EEG signal processing and automated real-time detection of seizure events, the prediction of epileptic seizures (even just a few minutes before occurring) is still an open problem, which makes this application very appealing.

Another possible future line of work is the hardware implementation of the RC training algorithm. All the FPGA-based RC systems presented in this thesis use off-chip learning. That is to say, the training of the neural network is performed externally in a PC, and then the FPGA is configured with the corresponding connection weights. The resulting system can be used as a hardware accelerator or as an alternative platform allowing power/cost reduction. Even though RC is characterized by a simple training procedure that can be usually performed in a short time by means of a conventional processor, an on-chip learning implementation including both the training and execution phases of the algorithm may be of high interest so that the system itself could be re-trained and adapted to the changing environment if necessary. Several works have already presented FPGA implementations of the training algorithms for different types of neural networks, such as the widely used backpropagation approach ([OZJM+16]). In the case of RC systems, the training consists in solving a least squares regression problem as described in sec. 2.3.1.3. FPGA-based architectures implementing the least squares method to address linear regression problems with massive data have already been proposed ([VMB15]), and therefore the joined hardware realization of the training and execution of RC models is feasible.

Finally, it is worth mentioning that although the proposed digital hardware designs have been specifically conceived to realize RC systems, they could be adapted to implement other neural network approaches. For example, extreme learning machines (ELMs) follow a similar concept to RC representing a suite of machine and biological learning techniques in which hidden neurons do not need to be tuned ([HWL11]). In this case, though, the networks do not present recurrences. Such approach has been shown powerful in numerous applications and is receiving much attention in recent years (see, e.g., the journal special issues [CHK+13] and [MH16] dedicated to this topic). Given the similarities between ELMs and RC, our RC hardware designs could be easily adapted to execute ELM algorithms.

A. VHDL codes of the digital implementations

A.1. Conventional ESN implementation

VHDL code describing the proposed ESN conventional implementation with 50 neurons. The components employed in the neuron design are a 16-bit register (*ffD_16b*, used to hold the value of the neuron states so that they can be used by another neuron on the next time step), and the nonlinear function block, *f_tanh_aprox_3_segments*, described in Algorithm 3.2.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;

-- classical SCR implementation with 50 neurons

ENTITY conventional_SCR_network_50n IS
  PORT ( input: IN STD_LOGIC_VECTOR (15 DOWNTO 0); -- external input (s0.15)
         clk, reset: IN STD_LOGIC;
         r_input, v1_input: IN STD_LOGIC_VECTOR (15 DOWNTO 0); -- weights r and v (s0.15)
         out_x1: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)); -- neuron outputs x1 ... x50 (s0.15)
         out_x2: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
         ...
         out_x50: OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
END ENTITY conventional_SCR_network_50n;

ARCHITECTURE net OF conventional_SCR_network_50n IS

component ffD_16b IS -- 16-bit register
  PORT ( input: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         clk, reset: IN STD_LOGIC;
         output: OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
END component;

component f_tanh_approx_3_segments IS -- the activation function
  PORT ( x: IN STD_LOGIC_VECTOR (16 DOWNTO 0); -- s1.15
         f: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)); -- s0.15
END component;

SIGNAL r: STD_LOGIC_VECTOR (15 DOWNTO 0); -- interneuronal connection weight (s0.15)
SIGNAL v1: STD_LOGIC_VECTOR (15 DOWNTO 0); -- input connection weights v1 ... v50 (s0.15)
SIGNAL v2: STD_LOGIC_VECTOR (15 DOWNTO 0); --
...
SIGNAL v50: STD_LOGIC_VECTOR (15 DOWNTO 0); -- input connection weights (s0.15)

SIGNAL prod11, prod12: STD_LOGIC_VECTOR (31 DOWNTO 0); -- product results (s1.30)
SIGNAL prod21, prod22: STD_LOGIC_VECTOR (31 DOWNTO 0); --
```

```
...
SIGNAL prod501, prod502: STD_LOGIC_VECTOR (31 DOWNTO 0); -- s1.30

SIGNAL prod11b, prod12b: STD_LOGIC_VECTOR (16 DOWNTO 0); -- truncated products (s1.15)
SIGNAL prod21b, prod22b: STD_LOGIC_VECTOR (16 DOWNTO 0);
...
SIGNAL prod501b, prod502b: STD_LOGIC_VECTOR (16 DOWNTO 0);

SIGNAL sum1: STD_LOGIC_VECTOR (16 DOWNTO 0); -- addition results (s1.15)
SIGNAL sum2: STD_LOGIC_VECTOR (16 DOWNTO 0);
...
SIGNAL sum50: STD_LOGIC_VECTOR (16 DOWNTO 0);

-- neuron states at current and previous step (s0.15):
SIGNAL x1, x1_anterior: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL x2, x2_anterior: STD_LOGIC_VECTOR (15 DOWNTO 0);
...
SIGNAL x50, x50_anterior: STD_LOGIC_VECTOR (15 DOWNTO 0);

SIGNAL v2_input: STD_LOGIC_VECTOR (15 DOWNTO 0); -- s0.15
```

```
begin

v2_input <= NOT(v1_input) + '1'; -- negative value of the input weight v

-- the input connection weights (+v or -v) are assigned randomly for each neuron
v1(15 DOWNTO 0) <= v1_input; -- v1=+v
v2(15 DOWNTO 0) <= v1_input; -- v1=+v
...
v50(15 DOWNTO 0) <= v2_input; -- v1=-v

-- the interneuronal connection is the same for all neurons (r)
r(15 DOWNTO 0) <= r_input;

-- computation of the neuron 1
-- first product:
prod11(31 DOWNTO 0) <= input * v1;
prod11b(16 DOWNTO 0) <= prod11(31 DOWNTO 15); -- truncated value
-- second product:
prod12(31 DOWNTO 0) <= x50_anterior * r;
prod12b(16 DOWNTO 0) <= prod12(31 DOWNTO 15); -- truncated value
-- addition of the two previous terms:
sum1 <= prod11b + prod12b;
-- assessment of the activation function:
f_tanh1: f_tanh_approx_3_segments PORT MAP(sum1,x1);
-- the 16-bit registers holds the neuron output to be used in the next time step:
ff1: ffD_16b PORT MAP(x1, clk, reset, x1_anterior);

-- computation of the neuron 2
-- first product:
prod21(31 DOWNTO 0) <= input * v2;
prod21b(16 DOWNTO 0) <= prod21(31 DOWNTO 15); -- truncated value
-- second product:
prod22(31 DOWNTO 0) <= x1_anterior * r;
prod22b(16 DOWNTO 0) <= prod22(31 DOWNTO 15); -- truncated value
-- addition of the two previous terms:
sum2 <= prod21b + prod22b;
-- assessment of the activation function:
f_tanh2: f_tanh_approx_3_segments PORT MAP(sum2,x2);
-- the 16-bit registers holds the neuron output to be used in the next time step:
ff2: ffD_16b PORT MAP(x2, clk, reset, x2_anterior);

...

-- computation of the neuron 50
```

```
-- first product:
prod501(31 DOWNTO 0) <= input * v50;
prod501b(16 DOWNTO 0) <= prod501(31 DOWNTO 15); -- truncated value
-- second product:
prod502(31 DOWNTO 0) <= x49_anterior * r;
prod502b(16 DOWNTO 0) <= prod502(31 DOWNTO 15); -- truncated value
-- addition of the two previous terms:
sum50 <= prod501b + prod502b;
-- assessment of the activation function:
f_tanh50: f_tanh_approx_3_segments PORT MAP(sum50,x50);
-- the 16-bit registers holds the neuron output to be used in the next time step:
ff50: ffD_16b PORT MAP(x50, clk, reset, x50_anterior);

-- assignation of the neuron states as network outputs
out_x1 <= x1;
out_x2 <= x2;
...
out_x50 <= x50;

END net;
```

A.2. Stochastic ESN implementation

A.2.1. 2-input sigmoid neuron

VHDL code describing the proposed SC-based 2-input sigmoid neuron implementation illustrated in Fig. 4.9. In this code, the *scsl_2in_adder* component performs the stochastic addition operation by means of a multiplexer and a binary counter as shown in Fig. 4.4(c). The P2B block, *p2b_16bits*, is described in sec. 4.2.2.2. The *b2p_16bits_intern* component is just a comparator that provides a "1" when the first input is greater than the second one, and a "0" otherwise. The *Activation_function* component performs several actions over the binary input signal. Firstly, it converts the binary magnitude to its corresponding two's complement format, the resulting value is then multiplied by 2 (this can be performed shifting the binary number one position to the left) and the hyperbolic tangent function is applied. The output of this block is given both as a binary value according to the two's complement format with 8-bit precision and according to the stochastic bipolar notation using 16-bit precision.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

```
-- 2-input stochastic sigmoid neuron
```

```
ENTITY stochastic_sigmoid_neuron IS
  PORT ( in_p: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
         w1, w2: IN STD_LOGIC;
        rnd: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        clk, reset: IN STD_LOGIC;
        out_neuron_8b_Ca2: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        out_p_neuron: OUT STD_LOGIC);
```

```
END ENTITY stochastic_sigmoid_neuron;

ARCHITECTURE bipolar_2in_neuron OF stochastic_sigmoid_neuron IS
-----------------------------------
------ SIGNALS DEFINITION ------
-----------------------------------
SIGNAL in_p_add: STD_LOGIC_VECTOR (1 DOWNTO 0);
SIGNAL lin_p, enable_p2b: STD_LOGIC;
SIGNAL eval_period, lin_bin, out_neuron_bin: STD_LOGIC_VECTOR (15 DOWNTO 0);

-----------------------------------
--- COMPONENTS DEFINITION ---
-----------------------------------
COMPONENT scsl_2in_adder
  PORT ( in_p: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
         clk, reset: IN STD_LOGIC;
         out_p: OUT STD_LOGIC);
END COMPONENT;

COMPONENT p2b_16bits
  PORT ( evaluation_period: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         stoch_in : IN STD_LOGIC;
         clk, reset, en: IN STD_LOGIC;
         binary_out: OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
END COMPONENT;

COMPONENT b2p_16bits_intern IS
  PORT ( binary_in: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         rnd: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         stoch_out: OUT STD_LOGIC);
END COMPONENT;

COMPONENT Activation_function is
  PORT ( input : in  STD_LOGIC_VECTOR (15 downto 0);
         out_Ca2 : out  STD_LOGIC_VECTOR (7 downto 0);
         out_bin : out  STD_LOGIC_VECTOR (15 downto 0));
end COMPONENT;

BEGIN
-----------------------------------------
-- LINEAR PART OF THE NEURON -----
-----------------------------------------
in_p_add(0) <= (in_p(0) XNOR w1);
in_p_add(1) <= (in_p(1) XNOR w2);

add_1: scsl_2in_adder PORT MAP(in_p_add, clk, reset, lin_p);

-----------------------------
-- P2B conversion ---
-----------------------------
enable_p2b<='1';
eval_period (15 DOWNTO 0) <= x"FFFF";
p2b_0: p2b_16bits PORT MAP(eval_period, lin_p, clk, reset, enable_p2b, lin_bin);

-----------------------------------
-- ACTIVATION FUNCTION BLOCK --
-----------------------------------
activation1: Activation_function PORT MAP(lin_bin, out_neuron_8b_Ca2, out_neuron_bin);

-----------------------------
-- B2P conversion ---
-----------------------------
b2p_1: b2p_16bits_intern PORT MAP(out_neuron_bin, rnd, out_p_neuron);

END bipolar_2in_neuron;
```

A.2.2. Neural network

VHDL code describing the proposed SC-based ESN implementation (Fig. 4.12) with 20 neurons. The code for the *stochastic_sigmoid_neuron* component is presented above. The B2P converter, *b2p_16bits*, is described in sec. 4.2.2.2, and the LFSR (*lfsr_26bits*) in Algorithm 4.2.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;
--------------------------------------------------------------------------------
-- 20-neuron stochastic ESN with cyclic architecture (SCR)
--------------------------------------------------------------------------------
ENTITY stochastic_ESN_20_neurons IS
  PORT ( input_p: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         v: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         r: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         clk, reset: IN STD_LOGIC;
         out_p_1: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
         out_p_2: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
         ...
         out_p_20: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
         readout: OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
END ENTITY stochastic_ESN_20_neurons;

ARCHITECTURE stoch_network OF stochastic_ESN_20_neurons IS
-----------------------------
------ SIGNALS DEFINITION ------
-----------------------------
SIGNAL enable_b2p: STD_LOGIC;
SIGNAL seed0, seed1, seed2, seed3 : STD_LOGIC_VECTOR (25 DOWNTO 0);
SIGNAL rand: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL input_p_stoch, v_stoch, r_stoch, v1_stoch, v2_stoch: STD_LOGIC;
SIGNAL w1_1, w1_2, ... , w1_20, w2: STD_LOGIC;
SIGNAL pulsed_p_1, pulsed_p_2, ... , pulsed_p_20: STD_LOGIC_VECTOR (1 DOWNTO 0);
SIGNAL out11p, out22p, ... , out2020p: STD_LOGIC;
SIGNAL out_p_1b, out_p_2b, ... , out_p_20b: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL w_out_1, w_out_2, ... , w_out_20: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL readout0: STD_LOGIC_VECTOR (15 DOWNTO 0);

TYPE weights IS ARRAY (1 TO 20) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
TYPE inputs IS ARRAY (1 TO 20) OF STD_LOGIC_VECTOR(7 DOWNTO 0);

-------------------------------
--- COMPONENTS DEFINITION ---
-------------------------------
COMPONENT b2p_16bits
  PORT (binary_in: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    clk,rst,en: IN STD_LOGIC;
    seed: IN STD_LOGIC_VECTOR (25 DOWNTO 0);
    stoch_out: OUT STD_LOGIC);
END COMPONENT;

COMPONENT lfsr_26bits IS
  PORT (clk, reset, enable: IN STD_LOGIC;
    seed: IN STD_LOGIC_VECTOR (25 DOWNTO 0);
    pseudorandom_out: OUT STD_LOGIC_VECTOR (15 DOWNTO 0) );
END COMPONENT;

COMPONENT stochastic_sigmoid_neuron IS
  PORT (in_p: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
```

```
    w1, w2: IN STD_LOGIC;
    rnd: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    clk, reset: IN STD_LOGIC;
    out_neuron_8b_Ca2: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    out_p_neuron: OUT STD_LOGIC);
END COMPONENT;

BEGIN
-------------------------------------------------------------------------------
-- B2P conversion (external input and weights)-----
-------------------------------------------------------------------------------
enable_b2p<='1';

seed0(25 DOWNTO 0)<="00101011101111011100110001";
seed1(25 DOWNTO 0)<="00000000000000000000111110";
seed2(25 DOWNTO 0)<="00000100000000100101010010";
seed3(25 DOWNTO 0)<="00110101000001010110110110";

lfsr: lfsr_26bits PORT MAP(clk,reset,enable_b2p,seed0,rand);

b2p_1: b2p_16bits PORT MAP(input_p, clk, reset, enable_b2p, seed1, input_p_stoch);
b2p_2: b2p_16bits PORT MAP(v, clk, reset, enable_b2p, seed2, v_stoch);
b2p_3: b2p_16bits PORT MAP(r, clk, reset, enable_b2p, seed3, r_stoch);

v1_stoch <= v_stoch;
v2_stoch <= NOT(v_stoch);

-------------------------------------------------------------------------------
-- distribution of weight values for each neuron ----
-------------------------------------------------------------------------------
-- random distribution of the first weight values
w1_1 <= v1_stoch;
w1_2 <= v2_stoch;
w1_3 <= v2_stoch;
...
w1_20 <= v1_stoch;

-- constant value of the second weight for all neurons
w2 <= r_stoch;

-------------------------------------------------------------------------------
-- definition of neuron connection signals --
-------------------------------------------------------------------------------
pulsed_p_1(0) <= input_p_stoch;
pulsed_p_1(1) <= out2020p;

pulsed_p_2(0) <= input_p_stoch;
pulsed_p_2(1) <= out11p;

pulsed_p_3(0) <= input_p_stoch;
pulsed_p_3(1) <= out22p;

...

pulsed_p_20(0) <= input_p_stoch;
pulsed_p_20(1) <= out1919p;

-------------------------------------------------------------------------------
-- declaration of neurons with their proper connections ----
-------------------------------------------------------------------------------
neuron_1: stochastic_sigmoid_neuron
   PORT MAP(pulsed_p_1, w1_1, w2, rand, clk, reset, out_p_1b, out11p);

neuron_2: stochastic_sigmoid_neuron
   PORT MAP(pulsed_p_2, w1_2, w2, rand, clk, reset, out_p_2b, out22p);
```

```
...

neuron_20: stochastic_sigmoid_neuron
   PORT MAP(pulsed_p_20, w1_20, w2, rand, clk, reset, out_p_20b, out2020p);

out_p_1 <= out_p_1b;
out_p_2 <= out_p_2b;
...
out_p_20 <= out_p_20b;

------------------------------------------------------------------------------
------ OUTPUT LAYER ------
------------------------------------------------------------------------------
-- output layer's weight values (two's complement format)
w_out_1(7 DOWNTO 0) <= x"C0";
w_out_2(7 DOWNTO 0) <= x"0E";
...
w_out_20(7 DOWNTO 0) <= x"FC";

-- multiply and accumulate circuit to calculate the final output as a weighted
-- addition of the individual neuron states
process (clk)
VARIABLE prod, acc: STD_LOGIC_VECTOR(15 DOWNTO 0);
VARIABLE weight: weights;
VARIABLE input: inputs;

BEGIN

weight(1) := w_out_1;
weight(2) := w_out_2;
...
weight(20) := w_out_20;

input(1) := out_p_1b;
input(2) := out_p_2b;
...
input(20) := out_p_20b;

if (reset='1') then
   acc := x"0000";
end if;

L1: FOR j IN 1 TO 20 LOOP
   prod := input(j)*weight(j);
   acc := acc + prod;
END LOOP L1;

readout0 <= acc;

end process;

readout <= readout0;

END stoch_network;
```

A.3. Proposed SSN implementation

VHDL code describing the proposed variant of the SSN model (Fig. 5.5), which represents a version of the previous SC-based sigmoid neuron (Fig. 4.9) where the nonlinear function is implemented by means of a simple OR gate of the weighted

inputs. Note that here, a P2B block of 20 bits is used (*p2b_20bits*), which allows a higher precision of the stochastic computations. In particular, the evaluation period is set to $T_{eval} = (2^{19} - 1) \cdot T_{clk}$. The neuron's binary output is given (in the two's complement format) with a precision of 16 bits (the less significant bits are disregarded).

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
-------------------------------------------------------------------------------
-- 2-input stochastic "spiking" neuron
-------------------------------------------------------------------------------
ENTITY stochastic_or_neuron IS
  PORT ( in_p: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
         w1, w2: IN STD_LOGIC;
         rnd: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         clk, reset: IN STD_LOGIC;
         out_neuron_16b_Ca2: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
         out_p_neuron: OUT STD_LOGIC);
END ENTITY stochastic_or_neuron;

ARCHITECTURE bipolar_2in_neuron OF stochastic_or_neuron IS
-----------------------------------
------ SIGNALS DEFINITION ------
-----------------------------------
SIGNAL in_p_or: STD_LOGIC_VECTOR (1 DOWNTO 0);
SIGNAL out_p_or, enable_p2b: STD_LOGIC;
SIGNAL eval_period, out_or_bin, out_neuron_bin: STD_LOGIC_VECTOR (15 DOWNTO 0);

-----------------------------------
--- COMPONENTS DEFINITION ---
-----------------------------------
COMPONENT p2b_20bits
  PORT ( evaluation_period: IN STD_LOGIC_VECTOR (19 DOWNTO 0);
         stoch_in : IN STD_LOGIC;
         clk, reset, en: IN STD_LOGIC;
         binary_out: OUT STD_LOGIC_VECTOR (19 DOWNTO 0));
END COMPONENT;

COMPONENT b2p_16bits_intern IS
  PORT ( binary_in: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         rnd: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         stoch_out: OUT STD_LOGIC);
END COMPONENT;

BEGIN
-------------------------------------------------------------------------------
-- MULTIPLICATION OF THE INPUTS BY THEIR CORRESPONDING WEIGHTS ------
-------------------------------------------------------------------------------
in_p_or(0) <= (in_p(0) XNOR w1);
in_p_or(1) <= (in_p(1) XNOR w2);

-----------------------------------------
-- NONLINEAR FUNCTION (OR GATE) -------
-----------------------------------------
out_p_or <= in_p_or(0) OR in_p_or(1);

---------------------------
-- P2B conversion ---
---------------------------
enable_p2b <= '1';
eval_period (19 DOWNTO 0) <= x"7FFFF";
p2b_0: p2b_20bits PORT MAP(eval_period, out_p_or, clk, reset, enable_p2b, out_or_bin);
```

```
-------------------------------------------------------------------------------
-- CONVERSION FROM STOCHASTIC BIPOLAR CODING TO TWO'S COMPLEMENT NOTATION --
-------------------------------------------------------------------------------
-- the variable's precision is reduced to 16 bits disregarding the less significant bits:
out_neuron_bin <= out_or_bin(18 DOWNTO 3);

out_neuron_16b_Ca2(15) <= NOT out_neuron_bin(15);
out_neuron_16b_Ca2(14 DOWNTO 0) <= out_neuron_bin(14 DOWNTO 0);

-------------------------
-- B2P conversion ---
-------------------------
b2p_1: b2p_16bits_intern PORT MAP(out_neuron_bin, rnd, out_p_neuron);

END bipolar_2in_neuron;
```

A.4. ESN implementation without multipliers

VHDL code describing the proposed ESN hardware realization where the multiplications can be simply performed through shift-and-add operations (by virtue of the low resolution considered for the weights). The size of the system is fixed to $N = 50$ neurons and the network is configured with weight values $v = \pm 0.875$ and $r = 0.875$. The components employed in the neuron design are a 16-bit register (*ffD_16b*, used to hold the value of the neuron states so that they can be used by another neuron on the next time step), and the nonlinear function block, *f_tanh_aprox_3_segments*, described in Algorithm 3.2.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;
-------------------------------------------------------------------------------
-- classical SCR implementation "without" multipliers (50 neurons)
-------------------------------------------------------------------------------
ENTITY multiplierless_SCR_network_50n IS
  PORT ( input: IN STD_LOGIC_VECTOR (15 DOWNTO 0); -- external input (s0.15)
         clk, reset: IN STD_LOGIC;
         out_x1: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)); -- neuron outputs x1 ... x50 (s0.15)
         out_x2: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
         ...
         out_x50: OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
END ENTITY multiplierless_SCR_network_50n;
-------------------------------------------------------------------------------
ARCHITECTURE net OF multiplierless_SCR_network_50n IS

component ffD_16b IS -- 16-bit register
  PORT ( input: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         clk, reset: IN STD_LOGIC;
         output: OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
END component;

component f_tanh_approx_3_segments IS -- the activation function
  PORT ( x: IN STD_LOGIC_VECTOR (16 DOWNTO 0); -- s1.15
         f: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)); -- s0.15
END component;
```

```
SIGNAL prod11, prod12: STD_LOGIC_VECTOR (15 DOWNTO 0); -- product results (s0.15)
SIGNAL prod21, prod22: STD_LOGIC_VECTOR (15 DOWNTO 0); --
...
SIGNAL prod501, prod502: STD_LOGIC_VECTOR (15 DOWNTO 0);

SIGNAL prod11b, prod12b: STD_LOGIC_VECTOR (16 DOWNTO 0); -- products with s1.15 format
SIGNAL prod21b, prod22b: STD_LOGIC_VECTOR (16 DOWNTO 0);
...
SIGNAL prod501b, prod502b: STD_LOGIC_VECTOR (16 DOWNTO 0);

SIGNAL sum1: STD_LOGIC_VECTOR (16 DOWNTO 0); -- addition results (s1.15)
SIGNAL sum2: STD_LOGIC_VECTOR (16 DOWNTO 0);
...
SIGNAL sum50: STD_LOGIC_VECTOR (16 DOWNTO 0);

-- neuron states at current and previous step (s0.15):
SIGNAL x1, x1_anterior: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL x2, x2_anterior: STD_LOGIC_VECTOR (15 DOWNTO 0);
...
SIGNAL x50, x50_anterior: STD_LOGIC_VECTOR (15 DOWNTO 0);
```

```
begin

-- COMPUTATION OF THE NEURON 1
-- first product (input1 * v; with v=-0.875)
prod11(15 DOWNTO 0) <= NOT (to_stdlogicvector(to_bitvector(input(15 DOWNTO 0)) sra 0) +
    NOT (to_stdlogicvector(to_bitvector(input(15 DOWNTO 0)) sra 3)) + '1') + '1';

-- second product (input2 * r; with r=0.875)
prod12(15 DOWNTO 0) <= to_stdlogicvector(to_bitvector(x50_anterior(15 DOWNTO 0)) sra 0)
    + NOT (to_stdlogicvector(to_bitvector(x50_anterior(15 DOWNTO 0)) sra 3)) + '1';

-- conversion of products from s0.15 to s1.15 notation
prod11b(16) <= prod11(15);
prod11b(15 DOWNTO 0) <= "0" & prod11(14 DOWNTO 0) when (prod11(15)='0') else
    "1" & prod11(14 DOWNTO 0);

prod12b(16) <= prod12(15);
prod12b(15 DOWNTO 0) <= "0" & prod12(14 DOWNTO 0) when (prod12(15)='0') else
    "1" & prod12(14 DOWNTO 0);

-- addition of the two previous terms
sum1 <= prod11b + prod12b;

-- assessment of the activation function
f_tanh1: f_tanh_approx_3_segments PORT MAP(sum1,x1);

-- the 16-bit register holds the neuron output to be used in the next time step
ff1: ffD_16b PORT MAP(x1, clk, reset, x1_anterior);

-- COMPUTATION OF THE NEURON 2
-- first product (input1 * v; with v=+0.875)
prod21(15 DOWNTO 0) <= to_stdlogicvector(to_bitvector(input(15 DOWNTO 0)) sra 0) +
    NOT (to_stdlogicvector(to_bitvector(input(15 DOWNTO 0)) sra 3)) + '1';

-- second product (input2 * r; with r=0.875)
prod22(15 DOWNTO 0) <= to_stdlogicvector(to_bitvector(x1_anterior(15 DOWNTO 0)) sra 0)
    + NOT (to_stdlogicvector(to_bitvector(x1_anterior(15 DOWNTO 0)) sra 3)) + '1';

-- conversion of products from s0.15 to s1.15 notation
prod21b(16) <= prod21(15);
prod21b(15 DOWNTO 0) <= "0" & prod21(14 DOWNTO 0) when (prod21(15)='0') else
    "1" & prod21(14 DOWNTO 0);
```

```
prod22b(16) <= prod22(15);
prod22b(15 DOWNTO 0) <= "0" & prod22(14 DOWNTO 0) when (prod22(15)='0') else
    "1" & prod22(14 DOWNTO 0);

-- addition of the two previous terms
sum2 <= prod21b + prod22b;

-- assessment of the activation function
f_tanh2: f_tanh_approx_3_segments PORT MAP(sum2,x2);

-- the 16-bit register holds the neuron output to be used in the next time step
ff2: ffD_16b PORT MAP(x2, clk, reset, x2_anterior);

...

-- COMPUTATION OF THE NEURON 50
-- first product (input1 * v; with v=+0.875)
prod501(15 DOWNTO 0) <= to_stdlogicvector(to_bitvector(input(15 DOWNTO 0)) sra 0) +
    NOT (to_stdlogicvector(to_bitvector(input(15 DOWNTO 0)) sra 3)) + '1';

-- second product (input2 * r; with r=0.875)
prod502(15 DOWNTO 0) <= to_stdlogicvector(to_bitvector(x49_anterior(15 DOWNTO 0)) sra 0)
    + NOT (to_stdlogicvector(to_bitvector(x49_anterior(15 DOWNTO 0)) sra 3)) + '1';

-- conversion of products from s0.15 to s1.15 notation
prod501b(16) <= prod501(15);
prod501b(15 DOWNTO 0) <= "0" & prod501(14 DOWNTO 0) when (prod501(15)='0') else
    "1" & prod501(14 DOWNTO 0);

prod502b(16) <= prod502(15);
prod502b(15 DOWNTO 0) <= "0" & prod502(14 DOWNTO 0) when (prod502(15)='0') else
    "1" & prod502(14 DOWNTO 0);

-- addition of the two previous terms
sum50 <= prod501b + prod502b;

-- assessment of the activation function
f_tanh50: f_tanh_approx_3_segments PORT MAP(sum50,x50);

-- the 16-bit register holds the neuron output to be used in the next time step
ff50: ffD_16b PORT MAP(x50, clk, reset, x50_anterior);

-- assignation of the neuron states as network outputs
out_x1 <= x1;
out_x2 <= x2;
...
out_x50 <= x50;

END net;
```

A.5. Delay-based ESN implementation

The following VHDL code describes the proposed ESN hardware realization where the nodes are processed sequentially (each neuron is computed after another). The size of the system is fixed to $N = 200$ neurons and the reservoir weights v and r are given as generic values (input signals: *v1_input* and *r_input*, respectively). The components employed in the neuron design are 16-bit and 20-bit registers (*ffD_16b_en* and *ffD_20b_en*), the nonlinear function block *f_tanh_aprox_3_segments*

(described in Algorithm 3.2), 2-bit and 12-bit counters (*counter2b* and *counter12b*). Note that the registers (used to hold the value of the neuron states so that they can be used for future evaluations) include an *enable* signal stating when the input value must be stored. In addition, the network's general signal *rst_ff* is employed to set all the registers to an initial zero value before the first input sample is sent to the system.

The code describes the circuit of Fig. 7.4 except for the RAM memory providing the external input stream driving the network ($u(t)$) and the counter stating the sample (j) that such memory must supply each time step (τ). These two components have been implemented separately from the network. The output layer of Fig. 7.6 (configured with the specific weights obtained from the training process) is included in the design.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;
-------------------------------------------------------------------------------
-- classical SCR sequential implementation with 200 neurons
-------------------------------------------------------------------------------
ENTITY sequential_SCR_network_200n IS
  PORT ( input: IN STD_LOGIC_VECTOR (15 DOWNTO 0); -- external input (s0.15)
         clk, reset, rst_ff, en: IN STD_LOGIC;
         r_input, v1_input: IN STD_LOGIC_VECTOR (15 DOWNTO 0); -- weights r and v (s0.15)
         out_x1: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)); -- neuron outputs x1 ... x200 (s0.15)
         out_x2: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
         ...
         out_x200: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
                yj: OUT STD_LOGIC_VECTOR (19 DOWNTO 0)); -- final network readout (s6.13)
END ENTITY sequential_SCR_network_200n;
-------------------------------------------------------------------------------
ARCHITECTURE net OF sequential_SCR_network_200n IS

component ffD_16b_en IS -- 16-bit register
  PORT ( input: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         clk, reset, enable: IN STD_LOGIC;
         output: OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
END component;

component ffD_20b_en IS -- 20-bit register
  PORT ( input: IN STD_LOGIC_VECTOR (19 DOWNTO 0);
         clk, reset, enable: IN STD_LOGIC;
         output: OUT STD_LOGIC_VECTOR (19 DOWNTO 0));
END component;

component f_tanh_approx_3_segments IS -- the activation function
  PORT ( x: IN STD_LOGIC_VECTOR (16 DOWNTO 0); -- s1.15
         f: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)); -- s0.15
END component;

component counter2b IS  -- 2-bit counter
  PORT ( clk: in std_logic;
         preset : in std_logic;
         preset_value : in integer range 0 to 3;
         enable : in std_logic;
         qin : in integer range 0 to 3;
         q : out integer range 0 to 3;
         rco : out std_logic);
```

```
END component;

component counter12b IS  -- 12-bit counter
  PORT ( clk: in std_logic;
         preset : in std_logic;
                 preset_value : in integer range 0 to 4095;
                 enable : in std_logic;
                 qin : in integer range 0 to 4095;
                 q      : out integer range 0 to 4095;
                 rco : out std_logic);
END component;
```

```
-- reservoir weights
SIGNAL r: STD_LOGIC_VECTOR (15 DOWNTO 0); -- s0.15; interneuronal connection weight
-- input connection weights (s0.15)
SIGNAL vi: STD_LOGIC_VECTOR (15 DOWNTO 0); -- s0.15; general input connection weight
SIGNAL v1, v2, v3, v4, v5: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL v6, v7, v8, v9, v10: STD_LOGIC_VECTOR (15 DOWNTO 0);
...
SIGNAL v196, v197, v198, v199, v200: STD_LOGIC_VECTOR (15 DOWNTO 0);

SIGNAL v2_input: STD_LOGIC_VECTOR (15 DOWNTO 0); -- s0.15

SIGNAL prod11, prod12: STD_LOGIC_VECTOR (31 DOWNTO 0); -- s1.30
SIGNAL prod11b, prod12b: STD_LOGIC_VECTOR (16 DOWNTO 0); -- s1.15
SIGNAL sum1: STD_LOGIC_VECTOR (16 DOWNTO 0); -- s1.15

-- neuron output for each node (s0.15)
SIGNAL x1, x2, x3, x4, x5: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL x6, x7, x8, x9, x10: STD_LOGIC_VECTOR (15 DOWNTO 0);
...
SIGNAL x196, x197, x198, x199, x200: STD_LOGIC_VECTOR (15 DOWNTO 0);

-- neuron states at previous step tau (s0.15)
SIGNAL x1_ant, x2_ant, x3_ant, x4_ant, x5_ant: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL x6_ant, x7_ant, x8_ant, x9_ant, x10_ant: STD_LOGIC_VECTOR (15 DOWNTO 0);
...
SIGNAL x196_ant, x197_ant, x198_ant, x199_ant, x200_ant: STD_LOGIC_VECTOR (15 DOWNTO 0);

SIGNAL xi, x_i_minus1_anterior: STD_LOGIC_VECTOR (15 DOWNTO 0); -- s0.15

SIGNAL q1: integer range 0 to 3;
SIGNAL index_i, index_k, q2, q3: integer range 0 to 4095;
SIGNAL rco, rco2, rco3: STD_LOGIC;

-- control signals
SIGNAL enable_tau, enable_tau2: STD_LOGIC;
SIGNAL enable_theta1, enable_theta2, enable_theta3, enable_theta4, enable_theta5: STD_LOGIC;
SIGNAL enable_theta6, enable_theta7, enable_theta8, enable_theta9, enable_theta10: STD_LOGIC;
...
SIGNAL enable_theta196, enable_theta197, ... , enable_theta200: STD_LOGIC;

-- output layer signals
SIGNAL zi: STD_LOGIC_VECTOR (35 DOWNTO 0); -- s5.30
SIGNAL wi: STD_LOGIC_VECTOR (19 DOWNTO 0); -- s5.14
SIGNAL zib, yi, yi_ant, y_j: STD_LOGIC_VECTOR (19 DOWNTO 0); -- s6.13
SIGNAL enable_thetai: STD_LOGIC;
```

```
begin

v2_input <= NOT(v1_input) + '1'; -- negative value of the input weight v

-- the input connection weights (+v or -v) are assigned randomly for each neuron
v1(15 DOWNTO 0) <= v1_input; -- v1=+v
```

```
v2(15 DOWNTO 0) <= v2_input; -- v2=-v
v3(15 DOWNTO 0) <= v1_input; -- v3=+v
...
v200(15 DOWNTO 0) <= v1_input; -- v200=+v

-- the interneuronal connection is the same for all neurons (r)
r(15 DOWNTO 0) <= r_input;

-- multiplexer choosing the input weight that corresponds to each neuron
WITH index_i SELECT
vi <=
  v1 WHEN 1,
  v2 WHEN 2,
  v3 WHEN 3,
  ...
  v200 WHEN 200,
  x"0000" WHEN OTHERS;
```

```
-- Core of the neuron
-- First product (external input x weight)
prod11(31 DOWNTO 0) <= input * vi;
prod11b(16 DOWNTO 0) <= prod11(31 DOWNTO 15); -- keeps the 17 most significant bits

-- Second product (previous neuron output x weight)
prod12(31 DOWNTO 0) <= x_i_minus1_anterior * r;
prod12b(16 DOWNTO 0) <= prod12(31 DOWNTO 15);

-- Addition of both previous terms
sum1 <= prod11b + prod12b;

-- Assessment of the non-linear activation function
f_tanh1: f_tanh_approx_3_segments PORT MAP(sum1,xi);
```

```
-- Registers storing the output of the neurons
ff1: ffD_16b_en PORT MAP(xi, clk, rst_ff, enable_theta1, x1);
ff1b: ffD_16b_en PORT MAP(x1, clk, rst_ff, enable_tau, x1_ant);

ff2: ffD_16b_en PORT MAP(xi, clk, rst_ff, enable_theta2, x2);
ff2b: ffD_16b_en PORT MAP(x2, clk, rst_ff, enable_tau, x2_ant);

ff3: ffD_16b_en PORT MAP(xi, clk, rst_ff, enable_theta3, x3);
ff3b: ffD_16b_en PORT MAP(x3, clk, rst_ff, enable_tau, x3_ant);

...

ff200: ffD_16b_en PORT MAP(xi, clk, rst_ff, enable_theta200, x200);
ff200b: ffD_16b_en PORT MAP(x200, clk, rst_ff, enable_tau, x200_ant);
```

```
-- Multiplexer choosing the proper neuron state (to be used as input for the next neuron)
-- depending on the current node i

WITH index_i SELECT
x_i_minus1_anterior <=
  x200_ant WHEN 1,
  x1_ant WHEN 2,
  x2_ant WHEN 3,
  x3_ant WHEN 4,
  ...
  x199_ant WHEN 200,
  x"0000" WHEN OTHERS;
```

```
-- control signals (index_i, enable_theta1, enable_theta2, enable_theta3, ..., enable_tau)
```

```
-- First counter to generate de slow clock signal; rco enables the counting of the nodes
counter1: counter2b PORT MAP(clk, reset, 0, en, 2, q1, rco);

-- Second counter; indicates the present node index (i); it counts up to N+1
-- (additional step to pass the neuron values)
counter2: counter12b PORT MAP(clk, reset, 0, rco, 201, q2, rco2);
index_i <= q2 + 1;

-- Third counter running with the fast clock to generate the enable (control) signals;
-- it counts up to 2*(N+1)
counter3: counter12b PORT MAP(clk, reset, 0, en, 402, q3, rco3);
index_k <= q3 + 1;

-- decoder generating the enable (control) signals
enable_theta1 <= '1' WHEN index_k=2 ELSE '0';
enable_theta2 <= '1' WHEN index_k=4 ELSE '0';
enable_theta3 <= '1' WHEN index_k=6 ELSE '0';
...
enable_theta200 <= '1' WHEN index_k=400 ELSE '0';

enable_tau <= '1' WHEN index_k=402 ELSE '0';
```

```
-- output layer
-- multiplexer choosing the proper weight for each neuron output
-- weights calculated by software (given with 20-bit resolution)
-- wi representing the numerical quantity according to s5.14

WITH index_i SELECT  wi <=
  "11111101111101011111" WHEN 1, --(-0.259807)
  "11111101100000011110" WHEN 2, --(-0.310665)
  "11111111011011011111" WHEN 3, --(-0.071355)
  ...
  "00000000000000011000" WHEN 200, --(0.001464)
  x"00000" WHEN OTHERS;

-- product of each output neuron by the corresponding output weight
zi(35 DOWNTO 0) <= xi * wi; -- s6.29
zib(19 DOWNTO 0) <= zi(35 DOWNTO 16);  -- s6.13 (keep the 20 most signitificant bits)

-- addition with the accumulated products (of previous nodes)
yi <= zib + yi_ant; --(s6.13)

-- flip-flop storing the previous accumulated products
ff_out: ffD_20b_en PORT MAP(yi, clk, enable_tau, enable_thetai, yi_ant);
enable_thetai <= rco; --habilitation for the previous flip-flop for each node time

-- flip-flop storing the final accumulated product (last node)
-- (final value stored a clock cycle before enable_tau is activated)
ff_out2: ffD_20b_en PORT MAP(yi_ant, clk, reset, enable_tau2, y_j);
enable_tau2 <= '1' WHEN index_k=401 ELSE '0';

-- assignation of the neuron states and final readout as network outputs
out_x1 <= x1;
out_x2 <= x2;
...
out_x200 <= x200;

yj <= y_j;

END net;
```

Bibliography

[ACI+17a] M. L. Alomar, V. Canals, E. Isern, M. Roca, and J. L. Rossello. Efficient parallel implementation of reservoir computing systems. *Neural Computing and Applications*, 2017. Submitted on Jun. 2017 (under review).

[ACI+17b] M. L. Alomar, V. Canals, E. Isern, M. Roca, and J. L. Rossello. Fault-tolerant Echo State Networks for temporal signal classification. In *Design of Circuits and Integrated Circuits (DCIS), 2017 Conference on*, pages 1–5, Nov 2017. Submitted on May 2017 (under review).

[ACM+16] M. L. Alomar, V. Canals, A. Morro, A. Oliver, and J. L. Rossello. Stochastic hardware implementation of liquid state machines. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 1128–1133, Jul 2016.

[ACMMR14] M. L. Alomar, V. Canals, V. Martinez-Moll, and J. L. Rossello. Low-cost hardware implementation of reservoir computers. In *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2014 24th International Workshop on*, pages 1–5, Sep 2014.

[ACMMR15] M. L. Alomar, V. Canals, V. Martinez-Moll, and J. L. Rossello. *Advances in Computational Intelligence: 13th International Work-Conference on Artificial Neural Networks, IWANN 2015. Proceedings, Part II*, chapter Stochastic-Based Implementation of Reservoir Computers, pages 185–196. Springer International Publishing, Cham, Jun 2015.

[ACPM+16] M. L. Alomar, V. Canals, N. Perez-Mora, V. Martinez-Moll, and J. L. Rossello. FPGA-based stochastic echo state networks for time-series forecasting. *Computational Intelligence and Neuroscience*, 2016.

[ACRMM16] M. L. Alomar, V. Canals, J. L. Rossello, and V. Martinez-Moll. Estimation of global solar radiation from air temperature using artificial neural networks based on reservoir computing. In *11th ISES EuroSun Conference, International Conference on Solar Energy for Buildings and Industry*, pages 1–6, Oct 2016.

[AE94] A. Achyuthan and M. I. Elmasry. Mixed analog/digital hardware synthesis of artificial neural networks. *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, 13(9): 1073–1087, 1994.

[AGAS+15] G. A. Abandah, A. Graves, B. Al-Shagoor, A. Arabiyat, F. Jamour, and M. Al-Taee. Automatic diacritization of Arabic text using recurrent neural networks. *International Journal on Document Analysis and Recognition*, 18(2): 183–197, 2015.

[AH13a] A. Alaghi and J. P. Hayes. Exploiting correlation in stochastic circuit design. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 39–46, Oct 2013.

[AH13b] A. Alaghi and J.P. Hayes. Survey of stochastic computing. *Transactions on Embedded Computing Systems*, 12(2 SUPPL.), May 2013.

[AIM10] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15): 2787 – 2805, 2010.

[AKK+16] M. F. Amir, D. Kim, J. Kung, D. Lie, S. Yalamanchili, and S. Mukhopadhyay. NeuroSensor: A 3D image sensor with integrated neural accelerator. In *2016 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pages 1–2, Oct 2016.

[AKS05] A. Alkan, E. Koklukaya, and A. Subasi. Automatic seizure detection in EEG using logistic regression and artificial neural network. *Journal of Neuroscience Methods*, 148(2): 167–176, 2005.

[Ald09] Irene Aldridge. *High-frequency trading: a practical guide to algorithmic strategies and trading systems*, volume 459. John Wiley and Sons, 2009.

[AMN14] F. Aghaeipoor, M. Mohammadi, and V. S. Naeini. Target tracking in noisy wireless sensor network using artificial neural network. In *Telecommunications (IST), 2014 7th International Symposium on*, pages 720–724, Sep 2014.

[AP00] A. F. Atiya and A. G. Parlos. New results on recurrent network training: unifying the algorithms and accelerating convergence. *IEEE Transactions on Neural Networks*, 11(3): 697–709, May 2000.

[AS15] E. A. Antonelo and B. Schrauwen. On learning navigation behaviors for small mobile robots with reservoir computing architectures. *IEEE Transactions on Neural Networks and Learning Systems*, 26(4): 763–780, Apr 2015.

[ASB13] F. Alvaro, J. A. Sanchez, and J. M. Benedi. Classification of online mathematical symbols with hybrid features and recurrent neural networks. In *2013 12th International Conference on Document Analysis and Recognition*, pages 1012–1016, Aug 2013.

[ASD+07] E. A. Antonelo, B. Schrauwen, X. Dutoit, D. Stroobandt, and M. Nuttin. Event detection and localization in mobile robot navigation using reservoir computing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4669 LNCS(PART 2): 660–669, 2007.

[ASEM+15] M. L. Alomar, M. C. Soriano, M. Escalona-Moran, V. Canals, I. Fischer, C. R. Mirasso, and J. L. Rossello. Digital implementation of a single dynamical node reservoir computer. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(10): 977–981, Oct 2015.

[ASG91] C. Alippi and G. Storti-Gajani. Simple approximation of sigmoidal functions: realistic design of digital neural networks capable of learning. In *Circuits and Systems, 1991., IEEE International Symposium on*, volume 3, pages 1505–1508, Jun 1991.

[ASS07] E. Antonelo, B. Schrauwen, and D. Stroobandt. Experiments with reservoir computing on the road sign problem. In *Proceedings of the VIII Brazilian Congress on Neural Networks (CBRN), Florianopolis. 2007*, 2007.

[ASS08a] E. Antonelo, B. Schrauwen, and D. Stroobandt. Mobile robot control in the road sign problem using reservoir computing networks. In *2008 IEEE International Conference on Robotics and Automation*, pages 911–916, May 2008.

[ASS08b] E.A. Antonelo, B. Schrauwen, and D. Stroobandt. Event detection and localization for small mobile robots using reservoir computing. *Neural Networks*, 21(6): 862–871, 2008.

[ASSC02] I. F. Akyildiz, Weilian Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8): 102–114, Aug 2002.

[ASVC07] E. A. Antonelo, B. Schrauwen, and J. Van Campenhout. Generative modeling of autonomous robots and their environments using reservoir computing. *Neural Processing Letters*, 26(3): 233–249, 2007.

[ASVDS+11] L. Appeltant, M.C. Soriano, G. Van Der Sande, J. Danckaert, S. Massar, J. Dambre, B. Schrauwen, C.R. Mirasso, and I. Fischer. Information processing using a single dynamical node as complex system. *Nature Communications*, 2(1), 2011.

[AZ26] E.D. Adrian and Y. Zotterman. The impulses produced by sensory nerve-endings: Part II. the response of a single end-organ. *The Journal of Physiology*, 61(2): 151–171, 1926.

[AZIPY93] F. Ahmed-Zaid, P. A. Ioannou, M. M. Polycarpou, and M. M. Youssef. Identification and control of aircraft dynamics using radial basis function networks. In *Control Applications, 1993., Second IEEE Conference on*, pages 567–572 vol.2, Sep 1993.

[BB97] M. P. Baze and S. P. Buchner. Attenuation of single event induced pulses in CMOS combinational logic. *IEEE Transactions on Nuclear Science*, 44(6): 2217–2223, Dec 1997.

[BB99] Sergio Benedetto and Ezio Biglieri. *Principles of digital transmission: with wireless applications*. Springer Science & Business Media, 1999.

[BC01a] R. Bakker and B. Currie. *The McMaster IPIX radar sea clutter database*, 2001. `http://http://soma.ece.mcmaster.ca/ipix/`.

[BC01b] B. D. Brown and H. C. Card. Stochastic neural computation. I. computational elements. *IEEE Transactions on Computers*, 50(9): 891–905, Sep 2001.

[BC01c] B. D. Brown and H. C. Card. Stochastic neural computation. II. soft competitive learning. *IEEE Transactions on Computers*, 50(9): 906–920, Sep 2001.

[Beh00] Parhami Behrooz. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000.

[BH94] S. L. Bade and B. L. Hutchings. FPGA-based stochastic neural networks-implementation. In *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*, pages 189–198, Apr 1994.

[Bis95] Christopher M. Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[BL91] M. R. Buhl and R. D. Lorenz. Design and implementation of neural networks for digital current regulation of inverter drives. In *Conference Record of the 1991 IEEE Industry Applications Society Annual Meeting*, pages 415–421 vol.1, Sep 1991.

[BL03] N. Brunel and P. E. Latham. Firing rate of the noisy quadratic integrate-and-fire neuron. *Neural Computation*, 15(10): 2281–2306, Oct 2003.

[BLAZ11] L. Boccato, A. Lopes, R. Attux, and F. J. Von Zuben. An echo state network architecture based on Volterra filtering and PCA with application to the channel equalization problem. In *The 2011 International Joint Conference on Neural Networks*, pages 580–587, Jul 2011.

[BM95] D. V. Buonomano and M. M. Merzenich. Temporal information transformed into a spatial code by a neural network with realistic properties. *Science*, 267(5200): 1028–1030, 1995.

[BM15] L. Brocki and K. Marasek. Deep belief neural networks and bidirectional long-short term memory hybrid for speech recognition. *Archives of Acoustics*, 40(2): 191–195, 2015.

[BMR07] S. Bhunia, S. Mukhopadhyay, and K. Roy. Process variations and process-tolerant design. In *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*, pages 699–704, Jan 2007.

[Boo04] O. Booij. Temporal pattern classification using spiking neural networks. *Master's thesis, Universiteit van Amsterdam*, Aug 2004.

[BPK02] S. M. Bohte, H. La Poutre, and J. N. Kok. Unsupervised clustering with spiking neurons by sparse temporal coding and multilayer RBF networks. *IEEE Transactions on Neural Networks*, 13(2): 426–435, Mar 2002.

[BR13] S. Basterrech and G. Rubino. Echo state queueing network: A new reservoir computing learning tool. In *2013 IEEE 10th Consumer Communications and Networking Conference (CCNC)*, pages 118–123, Jan 2013.

[Bro09] S. H. Brown. Multiple linear regression analysis: a matrix approach with MATLAB. *Alabama Journal of Mathematics*, 34: 1–3, 2009.

[BS05] B. Boukhezzar and H. Siguerdidjane. Nonlinear control of variable speed wind turbines without wind speed measurement. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 3456–3461, Dec 2005.

[BSE99] S. Bhowmik, R. Spee, and J. H. R. Enslin. Performance optimization for doubly fed wind power generation systems. *IEEE Transactions on Industry Applications*, 35(4): 949–958, Jul 1999.

[BSL10] L. Büsing, B. Schrauwen, and R. Legenstein. Connectivity, dynamics, and memory in reservoir computing with binary and analog neurons. *Neural Comput.*, 22(5): 1272–1311, May 2010.

[BSMF13] D. Brunner, M.C. Soriano, C.R. Mirasso, and I. Fischer. Parallel photonic information processing at gigabyte per second data rates using transient states. *Nature Communications*, 4, 2013.

[BSMH15] M. Bauduin, A. Smerieri, S. Massar, and F. Horlin. Equalization of the non-linear satellite communication channel with an echo state network. In *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*, pages 1–5, May 2015.

[BTdC02] K. Basterretxea, J. M. Tarela, and I. del Campo. Digital design of sigmoid approximator for artificial neural networks. *Electronics Letters*, 38(1): 35–37, Jan 2002.

[Bur98] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2): 121–167, 1998.

[Bur05a] H. Burgsteiner. On learning with recurrent spiking neural networks and their applications to robot control with real-world devices. *Doctoral thesis, Graz University of Technology*, 2005.

[Bur05b] H. Burgsteiner. Training networks of biological realistic spiking neurons for real-time robot control. In *Proceedings of the 9th international conference on engineering applications of neural networks, Lille, France*, pages 129–136, 2005.

[BVN+13] P. Buteneers, D. Verstraeten, B.V. Nieuwenhuyse, D. Stroobandt, R. Raedt, K. Vonck, P. Boon, and B. Schrauwen. Real-time detection of epileptic seizures in animal models using reservoir computing. *Epilepsy Research*, 103(2-3): 124–134, 2013.

[BVvM+11] P. Buteneers, D. Verstraeten, P. van Mierlo, T. Wyckhuys, D. Stroobandt, R. Raedt, H. Hallez, and B. Schrauwen. Automatic detection of epileptic seizures on the intra-cranial electroencephalogram of rats using reservoir computing. *Artificial Intelligence in Medicine*, 53(3): 215–223, 2011.

[BW73] C. R. Baugh and B. A. Wooley. A two's complement parallel array multiplication algorithm. *Computers, IEEE Transactions on*, C-22(12): 1045–1047, Dec 1973.

[CAM+15] V. Canals, M. L. Alomar, A. Morro, A. Oliver, and J. L. Rossello. Noise-robust hardware implementation of neural networks. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–8, Jul 2015.

[CdPdK05] S. Cristea, C. de Prada, and R. de Keyser. Predictive control of a process with variable dead-time. *IFAC Proceedings Volumes*, 38(1): 309 – 314, 2005. 16th IFAC World Congress.

[CFA+17] V. Canals, C. F. Frasser, M. L. Alomar, A. Morro, A. Oliver, M. Roca, E. Isern, V. Martinez-Moll, E. Garcia-Moreno, and J. L. Rossello. Noise tolerant probabilistic logic for statistical pattern recognition applications. *Integrated Computer-Aided Engineering*, 2017. Submitted on Feb. 2016 (accepted with minor changes).

[CG12] V. J. Canals Guinand. Implementación en hardware de sistemas de alta fiabilidad basados en metodologías estocásticas. *Doctoral thesis, Universitat de les Illes Balears*, May 2012.

[CGC90] S. Chen, G. J. Gibson, and C. F. N. Cowan. Adaptive channel equalisation using a polynomial-perceptron structure. *IEE Proceedings I - Communications, Speech and Vision*, 137(5): 257–264, Oct 1990.

[CGRS09] J. P. Cunningham, V. Gilja, S. I. Ryu, and K. V. Shenoy. Methods for estimating neural firing rates, and their application to brain-machine interfaces. *Neural Networks*, 22(9): 1235–1246, 2009.

[Che93] P. L. Chen. Some aspects of the Morris-Lecar model and the myelinated axon models with Morris-Lecar dynamics. *Mathematical and Computer Modelling*, 17(8): 85–97, 1993.

[CHK+13] E. Cambria, G. B. Huang, L. L. C. Kasun, H. Zhou, C. M. Vong, J. Lin, J. Yin, Z. Cai, Q. Liu, K. Li, V. C. M. Leung, L. Feng, Y. S. Ong, M. H. Lim, A. Akusok, A. Lendasse, F. Corona, R. Nian, Y. Miche, P. Gastaldo, R. Zunino, S. Decherchi, X. Yang, K. Mao, B. S. Oh, J. Jeon, K. A. Toh, A. B. J. Teoh, J. Kim, H. Yu, Y. Chen, and J. Liu. Extreme learning machines [trends controversies]. *IEEE Intelligent Systems*, 28(6): 30–59, Nov 2013.

[CLO+13] S. Conforto, A. Laudani, F. Oliva, F. R. Fulginei, and M. Schmid. Classification of ECG patterns for diagnostic purposes by means of Neural Networks and Support Vector Machines. In *2013 36th International Conference on Telecommunications and Signal Processing (TSP)*, pages 591–595, Jul 2013.

[CM05] M. Cernansky and M. Makula. Feed-forward echo state networks. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks.*, volume 3, pages 1479–1482 vol. 3, Jul 2005.

[CM15] D. Ciresan and U. Meier. Multi-column deep neural networks for offline handwritten Chinese character classification. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6, Jul 2015.

[CMA+16] J. A. Clemente, W. Mansour, R. Ayoubi, F. Serrano, H. Mecha, H. Ziade, W. El Falou, and R. Velazco. Hardware implementation of a fault-tolerant Hopfield neural network on FPGAs. *Neurocomputing*, 171: 1606–1609, 2016.

[CMGS10] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation*, 22(12): 3207–3220, 2010.

[CMO+15] V. Canals, A. Morro, A. Oliver, M. L. Alomar, and J. L. Rossello. An unconventional computing technique for ultra-fast and ultra-low power data mining. In *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2015 25th International Workshop on*, pages 40–46, Sep 2015.

[CMO+16] V. Canals, A. Morro, A. Oliver, M. L. Alomar, and J. L. Rossello. A new stochastic computing methodology for efficient neural network implementation. *IEEE Transactions on Neural Networks and Learning Systems*, 27(3): 551–564, Mar 2016.

[CMS12] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, Jun 2012.

[COA+16a] V. Canals, A. Oliver, M. L. Alomar, N. Perez-Mora, A. Moia, V. Martinez-Moll, M. Roca, E. Isern, E. Garcia-Moreno, and J. L. Rossello. Mid-term photovoltaic plant power generation forecast model using a time delayed artificial neural networks: application for a grid-connected PV plant at Balearic Islands, Spain. In *11th ISES EuroSun Conference, International Conference on Solar Energy for Buildings and Industry*, 2016.

[COA+16b] V. Canals, A. Oliver, M. L. Alomar, M. Roca, E. Isern, E. Garcia-Moreno, A. Morro, F. Galan, J. Font-Rossello, and J. L. Rossello. Robust stochastic logic for pattern recognition. In *Design of Circuits and Integrated Circuits (DCIS), 2016 Conference on*, pages 1–6, Nov 2016.

[Coc09] Marco Cococcioni. *MathWorks file exchange, Mackey-Glass time series generator*, June 7, 2009. `https://es.mathworks.com/matlabcentral/fileexchange/24390-mackey-glass-time-series-generator`.

[CR95] M. Chiaberge and L. M. Reyneri. Cintia: a neuro-fuzzy real-time controller for low-power embedded systems. *IEEE Micro*, 15(3): 40–47, Jun 1995.

[CR07] A. Carnell and D. Richardson. Linear algebra for time series of spikes. pages 363–368, 2007.

[CRBV07] E. F. Camacho, F. R. Rubio, M. Berenguel, and L. Valenzuela. A survey on control schemes for distributed solar collector fields. part I: Modeling and basic control approaches. *Solar Energy*, 81(10): 1240–1251, 2007.

[CSC13] A. Chauhan, S. Semwal, and R. Chawhan. Artificial neural network-based forest fire detection system using wireless sensor network. In *2013 Annual IEEE India Conference (INDICON)*, pages 1–6, Dec 2013.

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4): 303–314, 1989.

[dACA16] R. de Azambuja, A. Cangelosi, and S. V. Adams. Diverse, noisy and parallel: a new spiking neural network approach for humanoid robot control. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 1134–1142, Jul 2016.

[dAKS+16] R. de Azambuja, F. B. Klein, M. F. Stoelen, S. V. Adams, and A. Cangelosi. Graceful degradation under noise on brain inspired robot controllers. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9947 LNCS: 195–204, 2016.

[DCFEB13] I. Del Campo, R. Finker, J. Echanobe, and K. Basterretxea. Controlled accuracy approximation of sigmoid function for efficient FPGA-based implementation of artificial neurons. *Electronics Letters*, 49(25): 1598–1600, Dec 2013.

[DGLZ12] S. Decherchi, P. Gastaldo, A. Leoncini, and R. Zunino. Efficient digital implementation of extreme learning machines for classification. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 59(8): 496–500, Aug 2012.

[DN16] Z. Darojah and E. S. Ningrum. The extended Kalman filter algorithm for improving neural network performance in voice recognition classification. In *2016 International Seminar on Intelligent Technology and Its Applications (ISITIA)*, pages 225–230, Jul 2016.

[Dom95] P.F. Dominey. Complex sensory-motor sequence learning based on recurrent state representation and reinforcement learning. *Biological Cybernetics*, 73(3): 265–274, 1995.

[Doy92] K. Doya. Bifurcations in the learning of recurrent neural networks. In *[Proceedings] 1992 IEEE International Symposium on Circuits and Systems*, volume 6, pages 2777–2780 vol.6, May 1992.

[Dre05] Gérard Dreyfus. *Neural networks: methodology and applications*. Springer Science & Business Media, 2005.

[DS95] P. E. Dodd and F. W. Sexton. Critical charge concepts for CMOS SRAMs. *IEEE Transactions on Nuclear Science*, 42(6): 1764–1771, Dec 1995.

[DSS+12] F. Duport, B. Schneider, A. Smerieri, M. Haelterman, and S. Massar. All-optical reservoir computing. *Optics Express*, 20(20): 22783–22795, 2012.

[DSVC+09] X. Dutoit, B. Schrauwen, J. Van Campenhout, D. Stroobandt, H. Van Brussel, and M. Nuttin. Pruning and regularization in reservoir computing. *Neurocomputing*, 72(7-9): 1534–1546, 2009.

[DVT+15] H. N. Do, M. T. Vo, V. S. Tran, P. V. Tan, and C. V. Trinh. An early flood detection system using mobile networks. In *2015 International*

Conference on Advanced Technologies for Communications (ATC), pages 599–603, Oct 2015.

[DW16] H. Duan and X. Wang. Echo state networks with orthogonal pigeon-inspired optimization for image restoration. *IEEE Transactions on Neural Networks and Learning Systems*, 27(11): 2413–2425, Nov 2016.

[DZ07] Z. Deng and Y. Zhang. Collective behavior of a small-world recurrent neural system with scale-free distribution. *IEEE Transactions on Neural Networks*, 18(5): 1364–1375, Sept 2007.

[EBD+14] M. Eldib, N. B. Bo, F. Deboeverie, J. Nino, J. Guan, S. Van de Velde, H. Steendam, H. Aghajan, and W. Philips. A low resolution multi-camera system for person tracking. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 378–382, Oct 2014.

[Eck06] D. Eck. Generating music sequences with an echo state network. In *NIPS 2006 workshop on Echo State Networks and liquid state machines*, 2006.

[Edw15] C. Edwards. Growing pains for deep learning. *Communications of the ACM*, 58(7): 14–16, 2015.

[EMSFM15] M. A. Escalona-Moran, M. C. Soriano, I. Fischer, and C. R. Mirasso. Electrocardiogram classification using reservoir computing with logistic regression. *IEEE Journal of Biomedical and Health Informatics*, 19(3): 892–898, May 2015.

[ERG15] J. Del Espiritu, G. Rolluqui, and R. C. Gustilo. Neural network based partial fingerprint recognition as support for forensics. In *2015 International Conference on Humanoid, Nanotechnology, Information Technology,Communication and Control, Environment and Management (HNICEM)*, pages 1–5, Dec 2015.

[ESCB13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. *IEEE Micro*, 33(3): 16–27, 2013.

[FE05] G. Fette and J. Eggert. Short term memory and pattern matching with simple echo state networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3696 LNCS: 13–18, 2005.

[FFD12] S. Fischer, P. Frey, and H. Drueck. A comparison between state-of-the-art and neural network modelling of solar collectors. *Solar Energy*, 86(11): 3268–3277, 2012.

[Fit61] R. FitzHugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical Journal*, 1(6): 445–466, 1961.

[FLP+13] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown. Overview of the spinnaker system architecture. *IEEE Transactions on Computers*, 62(12): 2454–2467, Dec 2013.

[FN93] K. i. Funahashi and Y. Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks*, 6(6): 801–806, 1993.

[FS91] James A. Freeman and David M. Skapura. *Neural Networks: Algorithms, Applications, and Programming Techniques*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1991.

[FS03] C. Fernando and S. Sojakka. Pattern recognition in a bucket. volume 2801, pages 588–597, 2003.

[FVVW+14] M. A. A. Fiers, T. Van Vaerenbergh, F. Wyffels, D. Verstraeten, B. Schrauwen, J. Dambre, and P. Bienstman. Nanophotonic reservoir computing with photonic crystal cavities to generate periodic patterns. *IEEE Transactions on Neural Networks and Learning Systems*, 25(2): 344–355, 2014.

[Gai67] B. R. Gaines. Stochastic computing. In *Proceedings of the AFIPS Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 149–156, New York, NY, USA, Apr 1967. ACM.

[Gal07] Alexander I. Galushkin. *Neural networks theory*. Springer Science & Business Media, 2007.

[GB14] A. Gruning and S.M. Bohte. Spiking neural networks: Principles and challenges. In *22nd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2014 - Proceedings*, pages 1–10, Apr 2014.

[GHLO15] L. Grigoryeva, J. Henriques, L. Larger, and J.-P. Ortega. Optimal nonlinear information processing capacity in delay-based reservoir computers. *Scientific Reports*, 5, 2015.

[GHLO16] L. Grigoryeva, J. Henriques, L. Larger, and J.-P. Ortega. Nonlinear memory capacity of parallel time-delay reservoir computers in the processing of multidimensional signals. *Neural Computation*, 28(7): 1411–1451, 2016.

[GK02] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models*. Cambridge University Press, 2002.

[GL81] A. Gersho and T. L. Lim. Adaptive cancellation of intersymbol interference for data transmission. *The Bell System Technical Journal*, 60(9): 1997–2021, Nov 1981.

[GLF+09] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber. A novel connectionist system for unconstrained

handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5): 855–868, May 2009.

[GM10] L. Glass and M. Mackey. Mackey-Glass equation. *Scholarpedia*, 5(3): 6908, 2010.

[GR95] A. Gutierrez and W. E. Ryan. Performance of adaptive Volterra equalizers on nonlinear satellite channels. In *Communications, 1995. ICC '95 Seattle, 'Gateway to Globalization', 1995 IEEE International Conference on*, volume 1, pages 488–492, Jun 1995.

[GSMOP12] J. M. Gutierrez, D. San-Martin, S. Ortin, and L. Pesquera. Simple reservoirs with chain topology based on a single time-delay nonlinear node. In *ESANN 2012 proceedings, 20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, pages 13–18, 2012.

[GSS15] A. Goudarzi, A. Shabani, and D. Stefanovic. Product reservoir computing: Time-series computation with multiplicative neurons. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Jul 2015.

[GTK+14] A. Gupta, B. Thomas, P. Kumar, S. Kumar, and Y. Kumar. Neural Network based indicative ECG classification. In *2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)*, pages 277–279, Sep 2014.

[HAM07] S. Himavathi, D. Anitha, and A. Muthuramalingam. Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3): 880–888, May 2007.

[Hay01] Simon S. Haykin. *Kalman filtering and neural networks*. Wiley Online Library, 2001.

[Hay15] J. P. Hayes. Introduction to stochastic computing and its challenges. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–3, Jun 2015.

[HCL+09] Y. Hu, Liangliang Cao, Fengjun Lv, Shuicheng Yan, Yihong Gong, and T. S. Huang. Action detection in complex scenes with spatial and temporal ambiguities. In *2009 IEEE 12th International Conference on Computer Vision*, pages 128–135, Sep 2009.

[Heb49] Donald O. Hebb. *The organization of behavior. A neuropsychological theory*. John Wiley, New York, NY, USA, 1949.

[Hen76] M. Henon. A two-dimensional mapping with a strange attractor. *Communications in Mathematical Physics*, 50(1): 69–77, 1976.

[HFN14] Helmut Hauser, Rudolf Marcel Fuechslin, and Kohei Nakajima. *Morphological Computation: The Body as a Computational Resource*, pages 226–244. Self-published, 2014.

[HH52] 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*, 117(4): 500–544, 1952.

[HH04] M. Hartmann and P. C. Haddow. Evolution of fault-tolerant and noise-robust digital designs. *IEE Proceedings - Computers and Digital Techniques*, 151(4): 287–294, Jul 2004.

[HH10] G. Holzmann and H. Hauser. Echo state networks with filter neurons and a delay-and-sum readout. *Neural Networks*, 23(2): 244 – 256, 2010.

[HIF+11] H. Hauser, A. J. Ijspeert, R. M. Füchslin, R. Pfeifer, and W. Maass. Towards a theoretical foundation for morphological computation with compliant bodies. *Biological Cybernetics*, 105(5-6): 355–370, 2011.

[HIF+12] H. Hauser, A. J. Ijspeert, R. M. Füchslin, R. Pfeifer, and W. Maass. The role of feedback in morphological computation with compliant bodies. *Biological Cybernetics*, 106(10): 595–613, 2012.

[Hir93] Y. Hirai. Hardware implementation of neural networks in Japan. *Neurocomputing*, 5(1): 3–16, 1993.

[HL09] Z. Huang and H. Liang. A novel radiation hardened by design latch. *Journal of Semiconductors*, 30(3), 2009.

[HLM14] G.M. Hoerzer, R. Legenstein, and W. Maass. Emergence of complex computational structures from chaotic neural networks through reward-modulated hebbian learning. *Cerebral Cortex*, 24(3): 677–690, 2014.

[HM07] S. Haeusler and W. Maass. A statistical analysis of information-processing properties of lamina-specific cortical microcircuit models. *Cerebral Cortex*, 17(1): 149–162, 2007.

[HMM03] S. Häusler, H. Markram, and W. Maass. Perspectives of the high-dimensional dynamics of neural microcircuits from the point of view of low-dimensional readouts. *Complexity*, 8(4 SPEC. ISS.): 39–50, 2003.

[HNBO03] T. Hammadou, M. Nilson, A. Bermak, and P. Ogunbona. A 96 x 64 intelligent digital pixel array with extended binary stochastic arithmetic. In *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, volume 4, pages IV–772–IV–775, May 2003.

[Hop88] J. J. Hopfield. Artificial neural networks. *IEEE Circuits and Devices Magazine*, 4(5): 3–10, Sep 1988.

[HR82] J. L. Hindmarsh and R. M. Rose. A model of the nerve impulse using two first-order differential equations. *Nature*, 296(5853): 162–164, 1982.

[HS97] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8): 1735–1780, 1997.

[HS00] P. Hazucha and C. Svensson. Impact of CMOS technology scaling on the atmospheric neutron soft error rate. *IEEE Transactions on Nuclear Science*, 47(6): 2586–2594, Dec 2000.

[HSBD14] M. Hermans, B. Schrauwen, P. Bienstman, and J. Dambre. Automated design of complex dynamic systems. *PLoS ONE*, 9(1), 2014.

[HSD+15] M. Hermans, M.C. Soriano, J. Dambre, P. Bienstman, and I. Fischer. Photonic delay systems as machine learning implementations. *Journal of Machine Learning Research*, 16: 2081–2097, 2015.

[HSR+15] N. D. Haynes, M. C. Soriano, D. P. Rosin, I. Fischer, and D. J. Gauthier. Reservoir computing with a single time-delay autonomous Boolean node. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 91(2), 2015.

[HWL11] G. B. Huang, D. H. Wang, and Y. Lan. Extreme learning machines: A survey. *International Journal of Machine Learning and Cybernetics*, 2(2): 107–122, 2011.

[IBA+10] M. S. Islam, M. S. Bhuyan, S. H. M. Ali, M. Othman, and B. Y. Majlis. VHDL implementation of fuzzy based handwriting recognition system. In *Semiconductor Electronics (ICSE), 2010 IEEE International Conference on*, pages 161–164, Jun 2010.

[Ien95] P. Ienne. Digital hardware architectures for neural networks. *Speedup Journal*, 1(9), 1995.

[IJK+07] I. Ilies, H. Jaeger, O. Kosuchinas, M. Rincon, V. Sakenas, and N. Vaskevicius. Stepping forward through echoes of the past: forecasting with echo state networks. *Short report on the winning entry to the NN3 financial forecasting competition*, 2007, http://www.neural-forecasting-competition. com/downloads/NN3/methods/27-NN3 Herbert Jaeger report.pdf.

[Ike79] K. Ikeda. Multiple-valued stationary state and its instability of the transmitted light by a ring cavity system. *Optics Communications*, 30(2): 257–261, 1979.

[ILBH+11] G. Indiveri, B. Linares-Barranco, T. J. Hamilton, A. van Schaik, R. Etienne-Cummings, T. Delbruck, S. C. Liu, P. Dudek, P. Haefliger, S. Renaud, J. Schemmel, G. Cauwenberghs, J. Arthur,

K. Hynna, F. Folowosele, S. Saighi, T. Serrano-Gotarredona, J. Wijekoon, Y. Wang, and K. Boahen. Neuromorphic silicon neuron circuits. *Frontiers in Neuroscience*, (MAY), 2011.

[Izh03] E. M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6): 1569–1572, Nov 2003.

[Izh04] E. M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5): 1063–1070, Sep 2004.

[Jae01] H. Jaeger. The 'echo state' approach to analysing and training recurrent neural networks. *Technical report GMD Report 148, German National Research Center for Information Technology*, 2001.

[Jae02] H. Jaeger. Tutorial on training recurrent neural networks, covering BPTT, RTRL, EKF and the 'echo state network' approach. *Technical report GMD Report 159, German National Research Center for Information Technology*, 2002.

[Jae03] H. Jaeger. Adaptive nonlinear system identification with echo state networks. 2003.

[Jae07a] H. Jaeger. Discovering multiscale dynamical features with hierarchical echo state networks. *Technical report No. 9, Jacobs University Bremen*, 2007.

[Jae07b] H. Jaeger. Echo state network. *Scholarpedia*, 2(9): 2330, 2007.

[JC10] Y. Jewajinda and P. Chongstitvatana. FPGA-based online-learning using parallel genetic algorithm and neural network for ECG signal classification. In *ECTI-CON2010: The 2010 ECTI International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, pages 1050–1054, May 2010.

[JC13] Y. Jewajinda and P. Chongstitvatana. A parallel genetic algorithm for adaptive hardware and its application to ECG signal classification. *Neural Computing and Applications*, 22(7-8): 1609–1626, 2013.

[JDP15] S. H. Jambukia, V. K. Dabhi, and H. B. Prajapati. Classification of ECG signals using machine learning techniques: A survey. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 714–721, Mar 2015.

[JH04] H. Jaeger and H. Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667): 78–80, 2004.

[JIM16] C. Jayne, L. Iliadis, and V. Mladenov. Special issue on the engineering applications of neural networks. *Neural Computing and Applications*, 27(5): 1075–1076, 2016.

[JLPS07] H. Jaeger, M. Lukoševičius, D. Popovici, and U. Siewert. Optimization and applications of echo state networks with leaky- integrator neurons. *Neural Networks*, 20(3): 335–352, 2007.

[JM99] L. C. Jain and L. R. Medsker. *Recurrent Neural Networks: Design and Applications*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1999.

[JM05] P. Joshi and W. Maass. Movement generation with circuits of spiking neurons. *Neural Computation*, 17(8): 1715–1738, 2005.

[JM15] L. L. Jiang and D. L. Maskell. Automatic fault detection and diagnosis for photovoltaic systems using combined artificial neural network and analytical based methods. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Jul 2015.

[JMP07] H. Jaeger, W. Maass, and J. Principe. Special issue on echo state networks and liquid state machines. *Neural Networks*, 20(3): 287–289, 2007.

[Jou16] N. Jouppi. *Google Cloud Platform Blog. Google supercharges machine learning tasks with TPU custom chip*, May 18, 2016). `https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html`.

[JTDM15] A. Jalalvand, F. Triefenbach, K. Demuynck, and J. P. Martens. Robust continuous digit recognition using Reservoir Computing. *Computer Speech and Language*, 30(1): 135 – 158, 2015.

[JTVM11] A. Jalalvand, F. Triefenbach, D. Verstraeten, and J. P. Martens. Connected digit recognition by means of reservoir computing. pages 1725–1728, 2011.

[JWW15] A. Jalalvand, G. V. Wallendael, and R. V. D. Walle. Real-time reservoir computing network-based systems for detection tasks on visual contents. In *Computational Intelligence, Communication Systems and Networks (CICSyN), 2015 7th International Conference on*, pages 146–151, Jun 2015.

[Kec01] Vojislav Kecman. *Learning and soft computing: support vector machines, neural networks, and fuzzy logic models*. MIT press, 2001.

[Key01] R. W. Keyes. Fundamental limits of silicon technology. *Proceedings of the IEEE*, 89(3): 227–239, Mar 2001.

[KFV11] R. V. Kulkarni, A. Forster, and G. K. Venayagamoorthy. Computational intelligence in wireless sensor networks: A survey. *IEEE Communications Surveys Tutorials*, 13(1): 68–96, Feb 2011.

[KHB+98] F. Kamran, R. G. Harley, B. Burton, T. G. Habetler, and M. A. Brooke. A fast on-line neural-network training algorithm for a rectifier regulator. *IEEE Transactions on Power Electronics*, 13(2): 366–371, Mar 1998.

[KKJ+15] D. Y. Kim, J. M. Kim, H. Jang, J. Jeong, and J. W. Lee. A neural network accelerator for mobile application processors. *IEEE Transactions on Consumer Electronics*, 61(4): 555–563, Nov 2015.

[Kni02] J. C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 547–550, May 2002.

[Kor66] Granino Arthur Korn. *Random-process simulation and measurements*. McGraw-Hill, New York, NY, 1966.

[KS89] G. Karam and H. Sari. Analysis of predistortion, equalization, and isi cancellation techniques in digital radio systems with nonlinear transmit amplifiers. *IEEE Transactions on Communications*, 37(12): 1245–1253, Dec 1989.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. volume 2, pages 1097–1105, 2012.

[Kun93] S. Y. Kung. *Digital Neural Networks*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[KW07] B. Keshari and S. Watt. Hybrid mathematical symbol recognition using support vector machines. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, volume 2, pages 859–863, Sept 2007.

[Kwa92] H. K. Kwan. Simple sigmoid-like activation function suitable for digital hardware implementation. *Electronics Letters*, 28(15): 1379–1380, Jul 1992.

[LB12] U. Lotric and P. Bulic. Applicability of approximate multipliers in hardware neural networks. *Neurocomputing*, 96: 57–65, 2012.

[LBBH98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324, Nov 1998.

[LBH02] J. Liu, M. A. Brooke, and K. Hirotsu. A CMOS feedforward neural-network chip with on-chip parallel learning for oscillation cancellation. *IEEE Transactions on Neural Networks*, 13(5): 1178–1186, Sep 2002.

[LBH15] Y. Lecun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553): 436–444, 2015.

[LBV06] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 8 pp.–, Apr 2006.

[LDL+16] S. Li, Y. Dou, Q. Lv, Q. Wang, X. Niu, and K. Yang. Optimized GPU acceleration algorithm of convolutional neural networks for target detection. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 224–230, Dec 2016.

[LGMARB+05] J. C. Lopez-Garcia, M. A. Moreno-Armendariz, J. Riera-Babures, M. Balsi, and X. Vilasis-Cardona. Real time vision by FPGA implemented CNNs. In *Proceedings of the 2005 European Conference on Circuit Theory and Design*, volume 1, pages I/281–I/284 vol. 1, Aug 2005.

[LH94] J. H. Lala and R. E. Harper. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82(1): 25–40, Jan 1994.

[LHP+16] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung. Fpga-based low-power speech recognition with recurrent neural networks. In *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 230–235, Oct 2016.

[Lic13] M. Lichman. *UCI Machine Learning Repository*, 2013. `http://archive.ics.uci.edu/ml`.

[LJ09] M. Lukoševičius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3): 127–149, 2009.

[LJS12] M. Lukoševičius, H. Jaeger, and B. Schrauwen. Reservoir computing trends. *KI - Künstliche Intelligenz*, 26(4): 365–371, 2012.

[LL88] Y. C. Lim and B. Liu. Design of cascade form FIR filters with discrete valued coefficients. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(11): 1735–1739, Nov 1988.

[LL11] P. Li and D. J. Lilja. Using stochastic computing to implement digital image processing algorithms. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 154–161, Oct 2011.

[LM13] M. Lukoševičius and V. Marozas. Noninvasive fetal QRS detection using Echo State Network. In *Computing in Cardiology 2013*, pages 205–208, Sep 2013.

[LR03] T. J. Lewis and J. Rinzel. Dynamics of spiking neurons connected by both inhibitory and electrical coupling. *Journal of Computational Neuroscience*, 14(3): 283–309, 2003.

[LRS+12] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. PocketWeb: Instant web browsing for mobile devices. pages 1–12, 2012.

[LSB+12] L. Larger, M.C. Soriano, D. Brunner, L. Appeltant, J.M. Gutierrez, L. Pesquera, C.R. Mirasso, and I. Fischer. Photonic information processing beyond Turing: An optoelectronic implementation of reservoir computing. *Optics Express*, 20(3): 3241–3249, 2012.

[LSL96] Hongjun Lu, R. Setiono, and Huan Liu. Effective data mining using neural networks. *IEEE Transactions on Knowledge and Data Engineering*, 8(6): 957–961, Dec 1996.

[LSM05] Hui Li, K. L. Shi, and P. G. McLaren. Neural-network-based sensorless maximum wind energy capture with compensated power coefficient. *IEEE Transactions on Industry Applications*, 41(6): 1548–1556, Nov 2005.

[Luc65] R. W. Lucky. Automatic equalization for digital communication. *The Bell System Technical Journal*, 44(4): 547–588, Apr 1965.

[LW15] X. Li and X. Wu. Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4520–4524, Apr 2015.

[LZF06] H. Li, D. Zhang, and S. Y. Foo. A stochastic digital implementation of a neural network controller for small wind turbine systems. *IEEE Transactions on Power Electronics*, 21(5): 1502–1507, Sep 2006.

[LZPH14] Y. Li, J. Zhang, D. Pan, and D. Hu. A study of speech recognition based on RNN-RBM language model. *Jisuanji Yanjiu yu Fazhan/-Computer Research and Development*, 51(9): 1936–1944, 2014.

[MAAI+14] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197): 668–673, 2014.

[Mac87] Ronald J. MacGregor. *Neural and brain modeling.* Academic press, Inc., 1987.

[Mac02] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

[MAL88] N. McCulloch, W. A. Ainsworth, and R. Linggard. Multi-layer perceptrons applied to speech technology. *British Telecom technology journal*, 6(2): 131–139, 1988.

[Mat91] V. J. Mathews. Adaptive polynomial filters. *IEEE Signal Processing Magazine*, 8(3): 10–26, Jul 1991.

[MB99] Wolfgang Maass and Christopher M. Bishop. *Pulsed Neural Networks*. MIT Press, Cambridge (USA), 1999.

[MCO+15] A. Morro, V. Canals, A. Oliver, M. L. Alomar, and J. L. Rossello. Ultra-fast data-mining hardware architecture based on stochastic computing. *PLoS ONE*, 10(5), May 2015.

[MCO+17] A. Morro, V. Canals, A. Oliver, M. L. Alomar, F. Galan-Prado, P. J. Ballester, and J. L. Rossello. A stochastic spiking neural network for virtual screening. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–5, 2017.

[Mea89] Carver Mead. *Analog VLSI and Neural Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[MF04] A. V. Mezhiba and E. G. Friedman. Scaling trends of on-chip power distribution noise. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(4): 386–394, Apr 2004.

[MFS96] P. D. Moerland, E. Fiesler, and I. Saxena. Incorporation of liquid-crystal light valve nonlinearities in optical multilayer neural networks. *Applied Optics*, 35(26): 5301–5307, 1996.

[MG77] M. C. Mackey and L. Glass. Oscillation and chaos in physiological control systems. *Science*, 197(4300): 287–289, 1977.

[MH16] Zhihong Man and Guang-Bin Huang. Guest editorial: Special issue on extreme learning machine and applications (I). *Neural Computing and Applications*, 27(1): 1–2, 2016.

[MHO04] K. Martinez, J. K. Hart, and R. Ong. Environmental sensor networks. *Computer*, 37(8): 50–56, Aug 2004.

[MHS08] A. Muthuramalingam, S. Himavathi, and E. Srinivasan. Neural network implementation using FPGA: issues and application. *International journal of information technology*, 4(2): 86–92, 2008.

[MIC+11] E. Monmasson, L. Idkhajine, M. N. Cirstea, I. Bahri, A. Tisan, and M. W. Naouar. FPGAs in industrial control applications. *IEEE Transactions on Industrial Informatics*, 7(2): 224–243, May 2011.

[Mit97] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1997.

[Mit16] S. Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys*, 48(4), 2016.

[MJM12] M. Mohanraj, S. Jayaraj, and C. Muraleedharan. Applications of artificial neural networks for refrigeration, air-conditioning and heat pump systems - a review. *Renewable and Sustainable Energy Reviews*, 16(2): 1340–1358, 2012.

[MJPV15] S. Mahdiani, V. Jeyhani, M. Peltokangas, and A. Vehkaoja. Is 50 Hz high enough ECG sampling frequency for accurate HRV analysis? In *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 5948–5951, Aug 2015.

[MJS05] W. Maass, P. Joshi, and E. D. Sontag. Principles of real-time computing with feedback applied to cortical microcircuit models. *Advances in Neural Information Processing Systems*, pages 835–842, 2005.

[MKM15] D. Manatunga, H. Kim, and S. Mukhopadhyay. SP-CNN: A scalable and programmable CNN-based accelerator. *IEEE Micro*, 35(5): 42–50, Sep 2015.

[MLM02] W. Maass, R. Legenstein, and H. Markram. A new approach towards vision suggested by biologically realistic neural microcircuit models. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2525: 282–293, 2002.

[MM15] A. Maknickas and N. Maknickiene. Investment support system using the EVOLINO recurrent neural network ensemble. In *2015 7th International Joint Conference on Computational Intelligence (IJCCI)*, volume 3, pages 138–145, Nov 2015.

[MMR97] Kishan Mehrotra, Chilukuri K. Mohan, and Sanjay Ranka. *Elements of artificial neural networks*. MIT press, 1997.

[MNM02] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11): 2531–2560, 2002.

[MNM04] W. Maass, T. Natschläger, and H. Markram. Computational Models for Generic Cortical Microcircuits. In *Computational Neuroscience: A Comprehensive Approach*, pages 575–605. Chapman & Hall/CRC, 2004.

[MOM12] Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. *Neural Networks: Tricks of the Trade*. Springer Science & Business Media, 2nd edition, 2012.

[MOPU93] M. Marchesi, G. Orlandi, F. Piazza, and A. Uncini. Fast neural networks without multipliers. *IEEE Transactions on Neural Networks*, 4(1): 53–62, Jan 1993.

[MP43] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4): 115–133, 1943.

[MP69] Marvin Minsky and Seymour Papert. *Perceptrons*. MIT press, Cambridge, MA, USA, 1969.

[MP93] M. Meyer and G. Pfeiffer. Multilayer perceptron based decision feedback equalisers for channels with intersymbol interference. *IEE Proceedings I - Communications, Speech and Vision*, 140(6): 420–424, Dec 1993.

[MP08] E. Mathews and A. Poigne. An Echo State Network based pedestrian counting system using wireless sensor networks. In *2008 International Workshop on Intelligent Solutions in Embedded Systems*, pages 1–14, Jul 2008.

[MPR09] M. Moradi, M. A. Poormina, and F. Razzazi. FPGA implementation of feature extraction and MLP neural network classifier for Farsi handwritten digit recognition. In *Computer Modeling and Simulation, 2009. EMS '09. Third UKSim European Symposium on*, pages 231–234, Nov 2009.

[MR99] S. McInerney and R. B. Reilly. Hybrid multiplier/CORDIC unit for online handwriting recognition. In *Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on*, volume 4, pages 1909–1912 vol.4, Mar 1999.

[MRC+16] P. Mitra, R. Ray, R. Chatterjee, R. Basu, P. Saha, S. Raha, R. Barman, S. Patra, S. S. Biswas, and S. Saha. Flood forecasting using Internet of things and artificial neural networks. In *2016 IEEE 7th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 1–5, Oct 2016.

[MS10] J. Misra and I. Saha. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, 74(1-3): 239–255, 2010.

[MSZ+05] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *Computer*, 38(2): 43–52, Feb 2005.

[MW79] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1): 2–9, Jan 1979.

[MW11] J. Malone and M. A. Wickert. Practical Volterra equalizers for wideband satellite communications with TWTA nonlinearities. In *2011 Digital Signal Processing and Signal Processing Education Meeting (DSP/SPE)*, pages 48–53, Jan 2011.

[MYWW15] X. Ma, H. Yu, Y. Wang, and Y. Wang. Large-scale transportation network congestion evolution prediction using deep learning theory. *PLoS ONE*, 10(3), 2015.

[MZD12] C. Ma, S. Zhong, and H. Dang. High fault tolerant image processing system based on stochastic computing. In *Computer Science Service System (CSSS), 2012 International Conference on*, pages 1587–1590, Aug 2012.

[NDMM07] N. Nedjah and L. De Macedo Mourelle. Reconfigurable hardware for neural networks: Binary versus stochastic. *Neural Computing and Applications*, 16(3): 249–255, 2007.

[NHK+13] K. Nakajima, H. Hauser, R. Kang, E. Guglielmino, D. G. Caldwell, and R. Pfeifer. Computing with a muscular-hydrostat system. In *2013 IEEE International Conference on Robotics and Automation*, pages 1504–1511, May 2013.

[NHLP15] K. Nakajima, H. Hauser, T. Li, and R. Pfeifer. Information processing via physical soft body. *Scientific Reports*, 5, 2015.

[Nic05] M. Nicolaidis. Design for soft error mitigation. *IEEE Transactions on Device and Materials Reliability*, 5(3): 405–418, Sep 2005.

[NMM03] Thomas Natschläger, Henry Markram, and Wolfgang Maass. *Computer Models and Analysis Tools for Neural Microcircuits*, pages 123–138. Springer US, Boston, MA, 2003.

[NMSG11] A. Naderi, S. Mannor, M. Sawan, and W. J. Gross. Delayed stochastic decoding of LDPC codes. *IEEE Transactions on Signal Processing*, 59(11): 5617–5626, Nov 2011.

[Nor96] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6): 2742–2750, Dec 1996.

[NPT95] A. Namajunas, K. Pyragas, and A. Tamaševičius. An electronic analog of the Mackey-Glass system. *Physics Letters A*, 201(1): 42–46, 1995.

[NR15] E. Najibi and H. Rostami. SCESN, SPESN, SWESN: Three recurrent neural echo state networks with clustered reservoirs for prediction of nonlinear and chaotic time series. *Applied Intelligence*, 43(2): 460–472, 2015.

[NS16] M. H. Najafi and M. E. Salehi. A fast fault-tolerant architecture for Sauvola local image thresholding algorithm using stochastic computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2): 808–812, Feb 2016.

[NUSM09] D. Nikolic, S.H. Usler, W. Singer, and W. Maass. Distributed fading memory for stimulus properties in the primary visual cortex. *PLoS Biology*, 7(12), 2009.

[NVDVDS14] R. M. Nguimdo, G. Verschaffelt, J. Danckaert, and G. Van Der Sande. Fast photonic information processing using semiconductor lasers with delayed optical feedback: Role of phase dynamics. *Optics Express*, 22(7): 8672–8686, 2014.

[OKM+15] N. Onizawa, D. Katagiri, K. Matsumiya, W. J. Gross, and T. Hanyu. Gabor filter based on stochastic computation. *IEEE Signal Processing Letters*, 22(9): 1224–1228, Sept 2015.

[OR06] Amos R. Omondi and Jagath C. Rajapakse. *FPGA Implementations of Neural Networks*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[OSP+15] S. Ortin, M. C. Soriano, L. Pesquera, D. Brunner, D. San-Martin, I. Fischer, C. R. Mirasso, and J.M. Gutierrez. A unified framework for reservoir computing and extreme learning machines based on a single time-delayed neuron. *Scientific Reports*, 5, 2015.

[OXP07] M.C. Oztuik, D. Xu, and J. C. Principe. Analysis and design of echo state networks. *Neural Computation*, 19(1): 111–138, 2007.

[OYSK04] K. Osada, K. Yamaguchi, Y. Saitoh, and T. Kawahara. SRAM immunity to cosmic-ray-induced multierrors based on analysis of an induced parasitic bipolar effect. *IEEE Journal of Solid-State Circuits*, 39(5): 827–833, May 2004.

[OZJM+16] F. Ortega-Zamorano, J. M. Jerez, D. Urda Munoz, R. M. Luque-Baena, and L. Franco. Efficient implementation of the backpropagation algorithm in FPGAs and microcontrollers. *IEEE Transactions on Neural Networks and Learning Systems*, 27(9): 1840–1850, Sep 2016.

[OZW+16] W. Ouyang, X. Zeng, X. Wang, S. Qiu, P. Luo, Y. Tian, H. Li, S. Yang, Z. Wang, H. Li, C. C. Loy, K. Wang, J. Yan, and X. Tang. DeepID-Net: Deformable deep convolutional neural networks for object detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PP(99): 1–1, 2016.

[PAE67] W. J. Poppelbaum, C. Afuso, and J. W. Esch. Stochastic computing elements and systems. In *Proceedings of the AFIPS Fall Joint Computer Conference*, AFIPS '67 (Fall), pages 635–644, New York, NY, USA, Nov 1967. ACM.

[PCFB09] K. Patel, C. P. Chua, S. Fau, and C. J. Bleakley. Low power real-time seizure detection for ambulatory EEG. In *2009 3rd International Conference on Pervasive Computing Technologies for Healthcare*, pages 1–7, Apr 2009.

[PCK+14] J. Park, D. Cho, S. Kim, Y. B. Kim, P. Y. Kim, and H. J. Kim. Utilizing online learning based on echo-state networks for the control of a hydraulic excavator. *Mechatronics*, 24(8): 986–1000, 2014.

[PDS09] V. Petridis, B. Deb, and V. Syrris. Detection and identification of human actions using predictive modular neural networks. In *2009 17th Mediterranean Conference on Control and Automation*, pages 406–411, Jun 2009.

[PDS+12] Y. Paquot, F. Duport, A. Smerieri, J. Dambre, B. Schrauwen, M. Haelterman, and S. Massar. Optoelectronic reservoir computing. *Scientific Reports*, 2, 2012.

[PDW14] H. Palangi, L. Deng, and R. K. Ward. Recurrent deep-stacking networks for sequence classification. In *2014 IEEE China Summit International Conference on Signal and Information Processing (ChinaSIP)*, pages 510–514, Jul 2014.

[Ped04] Volnei A. Pedroni. *Circuit design with VHDL*. MIT press, 2004.

[PK11] F. Ponulak and A. Kasinski. Introduction to spiking neural networks: Information processing, learning and applications. *Acta Neurobiologiae Experimentalis*, 71(4): 409–433, 2011.

[PLK+16] J. Park, B. Lee, S. Kang, P. Y. Kim, and H. J. Kim. Online learning control of hydraulic excavators based on echo-state networks. *IEEE Transactions on Automation Science and Engineering*, PP(99): 1–11, 2016.

[PMC09] J. C. Patra, P. K. Meher, and G. Chakraborty. Nonlinear channel equalization for wireless communication systems using Legendre neural networks. *Signal Processing*, 89(11): 2251–2262, 2009.

[PML11] A. Petrenas, V. Marozas, and A. Lukoševičius. Ventricular activity cancellation in ECG using an adaptive echo state network. In *Proceedings of the 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems*, volume 1, pages 378–382, Sep 2011.

[PMSL12a] A. Petrenas, V. Marozas, L. Sornmo, and A. Lukoševičius. An echo state neural network for QRST cancellation during atrial fibrillation. *IEEE Transactions on Biomedical Engineering*, 59(10): 2950–2957, Oct 2012.

[PMSL12b] A. Petrenas, V. Marozas, L. Sornmo, and A. Lukoševičius. Reservoir computing for extraction of low amplitude atrial activity in atrial

fibrillation. In *2012 Computing in Cardiology*, pages 13–16, Sep 2012.

[PPBP99] J. C. Patra, R. N. Pal, R. Baliarsingh, and G. Panda. Nonlinear channel equalization for QAM signal constellation using artificial neural networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 29(2): 262–271, Apr 1999.

[PS00] R. Plamondon and S. N. Srihari. Online and off-line handwriting recognition: a comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1): 63–84, Jan 2000.

[PVCL06] C. Paul, F. J. Valero-Cuevas, and H. Lipson. Design and control of tensegrity robots for locomotion. *IEEE Transactions on Robotics*, 22(5): 944–957, Oct 2006.

[PZD+03] E. M. Petriu, Lichen Zhao, S. R. Das, V. Z. Groza, and A. Cornell. Instrumentation applications of multibit random-data representation. *IEEE Transactions on Instrumentation and Measurement*, 52(1): 175–181, Feb 2003.

[Qia09] W. Qiao. Echo-state-network-based real-time wind speed estimation for wind power generation. In *2009 International Joint Conference on Neural Networks*, pages 2572–2579, Jun 2009.

[QLR+11] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja. An architecture for fault-tolerant computation with stochastic logic. *IEEE Transactions on Computers*, 60(1): 93–105, Jan 2011.

[QZAH08] W. Qiao, W. Zhou, J. M. Aller, and R. G. Harley. Wind speed estimation based sensorless output maximization control for a wind turbine driving a DFIG. *IEEE Transactions on Power Electronics*, 23(3): 1156–1169, May 2008.

[RAM+16] J. L. Rossello, M. L. Alomar, A. Morro, A. Oliver, and V. Canals. High-density liquid-state machine circuitry for time-series forecasting. *International Journal of Neural Systems*, 26(5), 2016.

[RAVPM15] J. J. Rodriguez-Andina, M. D. Valdes-Pena, and M. J. Moure. Advanced features and industrial applications of FPGAs; a review. *IEEE Transactions on Industrial Informatics*, 11(4): 853–864, Aug 2015.

[RCMO12] J. L. Rossello, V. Canals, A. Morro, and A. Oliver. Hardware implementation of stochastic spiking neural networks. *International Journal of Neural Systems*, 22(4), 2012.

[RCO+14] J. L. Rossello, V. Canals, A. Oliver, M. Alomar, and A. Morro. Spiking neural networks signal processing. In *Design of Circuits and Integrated Circuits (DCIS), 2014 Conference on*, pages 1–6, Nov 2014.

[RCOM14] J. L. Rossello, V. Canals, A. Oliver, and A. Morro. Studying the role of synchronized and chaotic spiking neural ensembles in neural information processing. *International Journal of Neural Systems*, 24(5), 2014.

[RGW+09] S. Raghunathan, S.K. Gupta, M.P. Ward, R.M. Worth, K. Roy, and P.P. Irazoqui. The design and hardware implementation of a low-power real-time seizure detection algorithm. *Journal of neural engineering*, 6(5): 056005, 2009.

[RHL08] M. Rabinovich, R. Huerta, and G. Laurent. Neuroscience: Transient dynamics for neural processing. *Science*, 321(5885): 48–50, 2008.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088): 533–536, 1986.

[Rib67] S. T. Ribeiro. Random-pulse machines. *IEEE Transactions on Electronic Computers*, EC-16(3): 261–276, Jun 1967.

[RM86] David E. Rumelhart and James L. McClelland. *Parallel distributed processing*, volume 1 and 2. MIT press, Cambridge, MA, USA, 1986.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6): 386–408, 1958.

[Ros62] F. Rosenblatt. *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*. Washington DC: Spartan, 1962.

[RT11] A. Rodan and P. Tino. Minimum complexity echo state network. *IEEE Transactions on Neural Networks*, 22(1): 131–144, Jan 2011.

[RT12] A. Rodan and P. Tino. Simple deterministically constructed cycle reservoirs with regular jumps. *Neural Computation*, 24(7): 1822–1852, Jul 2012.

[RWRI09] S. Raghunathan, M. P. Ward, K. Roy, and P. P. Irazoqui. A low-power implantable event-based seizure detection algorithm. In *2009 4th International IEEE/EMBS Conference on Neural Engineering*, pages 151–154, Apr 2009.

[SAC+10] M. U. Saleheen, H. Alemzadeh, A. M. Cheriyan, Z. Kalbarczyk, and R. K. Iyer. An efficient embedded hardware for high accuracy detection of epileptic seizures. In *2010 3rd International Conference on Biomedical Engineering and Informatics*, volume 5, pages 1889–1896, Oct 2010.

[San00] N. Sano. Increasing importance of electronic thermal noise in Sub-0.1 micrometer Si-MOSFETs. *IEICE Transactions on Electronics*, E83-C(8): 1203–1211, 2000.

[SBEM+15] M. C. Soriano, D. Brunner, M. Escalona-Moran, C. R. Mirasso, and I. Fischer. Minimal approach to neuro-inspired information processing. *Frontiers in Computational Neuroscience*, 9(JUNE), 2015.

[SBG+10] J. Schemmel, D. Briiderle, A. Griibl, M. Hock, K. Meier, and S. Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 1947–1950, May 2010.

[Sch15] J. Schmidhuber. Deep Learning in neural networks: An overview. *Neural Networks*, 61: 85–117, 2015.

[SCL+12a] X. Sun, H. Cui, R. Liu, J. Chen, and Y. Liu. Multistep ahead prediction for real-time vbr video traffic using deterministic echo state network. In *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, volume 02, pages 928–931, Oct 2012.

[SCL+12b] X. C. Sun, H. Y. Cui, R. P. Liu, J. Y. Chen, and Y. J. Liu. Modeling deterministic echo state network with loop reservoir. *Journal of Zhejiang University: Science C*, 13(9): 689–701, 2012.

[SDVC07] B. Schrauwen, M. D'Haene, D. Verstraeten, and J. Van Campenhout. Compact hardware for real-time speech recognition using a liquid state machine. In *2007 International Joint Conference on Neural Networks*, pages 1097–1102, Aug 2007.

[SDVC08] B. Schrauwen, M. D'Haene, D. Verstraeten, and J. V. Campenhout. Compact hardware liquid state machines on FPGA for real-time speech recognition. *Neural Networks*, 21(2-3): 511–523, 2008.

[SE05] A. Subasi and E. Ercelebi. Classification of EEG signals using neural network and logistic regression. *Computer Methods and Programs in Biomedicine*, 78(2): 87–99, 2005.

[SGMA15] K. Sanni, G. Garreau, J. L. Molin, and A. G. Andreou. FPGA implementation of a deep belief network architecture for character recognition using stochastic computation. In *Information Sciences and Systems (CISS), 2015 49th Annual Conference on*, pages 1–5, Mar 2015.

[SH07] M. D. Skowronski and J. G. Harris. Automatic speech recognition using a predictive echo state network classifier. *Neural Networks*, 20(3): 414–423, 2007.

[Shi82] E. Shichor. Fast recursive estimation using the lattice structure. *The Bell System Technical Journal*, 61(1): 97–115, Jan 1982.

[SHP11] H. Sumioka, H. Hauser, and R. Pfeifer. Computation with mechanically coupled springs for compliant robots. In *2011 IEEE/RSJ*

International Conference on Intelligent Robots and Systems, pages 4168–4173, Sep 2011.

[SJM04] Dongsheng Shen, Lianwen Jin, and Xiaobin Ma. *Advances in Neural Networks – ISNN 2004: International Symposium on Neural Networks, Dalian, China, August 2004, Proceedings, Part I*, chapter FPGA Implementation of Feature Extraction and Neural Network Classifier for Handwritten Digit Recognition, pages 988–995. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[SK08] D. Shutin and G. Kubin. Echo State wireless sensor networks. In *2008 IEEE Workshop on Machine Learning for Signal Processing*, pages 151–156, Oct 2008.

[SK10] S. Soltic and N. Kasabov. Knowledge extraction from evolving spiking neural networks with rank order population coding. *International Journal of Neural Systems*, 20(6): 437–445, 2010.

[SKK+02] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389–398, 2002.

[SMA07] A. W. Savich, M. Moussa, and S. Areibi. The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study. *IEEE Transactions on Neural Networks*, 18(1): 240–252, Jan 2007.

[SMK+00] P.N. Steinmetz, A. Manwani, C. Koch, M. London, and I. Segev. Subthreshold voltage noise due to channel fluctuations in active neuronal membranes. *Journal of Computational Neuroscience*, 9(2): 133–148, 2000.

[SNJ14] Z. Sadrnezhad, A. Nekouie, and M. V. Jahan. Online handwriting Farsi character and number recognition based on hand movement direction using hidden Markov models. In *Technology, Communication and Knowledge (ICTCK), 2014 International Congress on*, pages 1–6, Nov 2014.

[SNS11] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[SOK+14] M.C. Soriano, S. Ortin, L. Keuninckx, L. Appeltant, J. Danckaert, L. Pesquera, and G. Van Sande. Brief papers delay-based reservoir computing: Noise effects in a combined analog and digital implementation. *IEEE Transactions on Neural Networks and Learning Systems*, 2014.

[SOP07] S. Seth, M. C. Ozturk, and J. C. Principe. Signal processing with echo state networks in the complex domain. In *2007 IEEE Work-*

shop on Machine Learning for Signal Processing, pages 408–412, Aug 2007.

[SP05] M. Salmen and P. G. Ploger. Echo state networks used for motor control. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 1953–1958, Apr 2005.

[SS98] N. Sundararajan and P. Saratchandran. *Parallel Architectures for Artificial Neural Networks: Paradigms and Implementations*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1998.

[SSC16] N. Shaetti, M. Salomon, and R. Couturier. Echo state networks-based reservoir computing for MNIST handwritten digits recognition. In *19th IEEE International Conference on Computational Science and Engineering*, pages 1–8, Aug 2016.

[SSR+15] H. Sak, A. Senior, K. Rao, F. Beaufays, and J. Schalkwyk. *Google Research Blog. Google voice search: faster and more accurate*, September 24, 2015). `https://research.googleblog.com/2015/09/google-voice-search-faster-and-more.html`.

[Ste67] R. B. Stein. Some models of neuronal variability. *Biophysical Journal*, 7(1): 37–68, 1967.

[STP13] J. Schumacher, H. Toutounji, and G. Pipa. An analytical approach to single node delay-coupled reservoir computing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8131 LNCS: 26–33, 2013.

[SvHS+13] H. Safaai, M. von Heimendahl, J.M. Sorando, M.E. Diamond, and M. Maravall. Coordinated population activity underlying texture discrimination in rat barrel cortex. *Journal of Neuroscience*, 33(13): 5843–5855, 2013.

[SVVC07] B. Schrauwen, D. Verstraeten, and J. Van Campenhout. An overview of reservoir computing: Theory, applications and implementations. pages 471–482, 2007.

[SZ07] A. M. Schaefer and H. G. Zimmermann. Recurrent neural networks are universal approximators. *International Journal of Neural Systems*, 17(4): 253–263, 2007.

[TBCC07] M. H. Tong, A. D. Bickett, E. M. Christiansen, and G. W. Cottrell. Learning grammatical structure with echo state networks. *Neural Networks*, 20(3): 424–432, 2007.

[TDM14] F. Triefenbach, K. Demuynck, and J. P. Martens. Large vocabulary continuous speech recognition with reservoir-based acoustic models. *IEEE Signal Processing Letters*, 21(3): 311–315, Mar 2014.

[TDVR01] S. Thorpe, A. Delorme, and R. Van Rullen. Spike-based strategies for rapid processing. *Neural Networks*, 14(6-7): 715–725, 2001.

[TI04] K. Tan and S. Islam. Optimum control strategies in energy conversion of PMSG wind turbine system without mechanical sensors. *IEEE Transactions on Energy Conversion*, 19(2): 392–399, Jun 2004.

[TJSM10] F. Triefenbach, A. Jalalvand, B. Schrauwen, and J. P. Martens. Phoneme recognition with large hierarchical reservoirs. 2010.

[Tom03] M. T. Tommiska. Efficient digital implementation of the sigmoid function for reprogrammable logic. *Computers and Digital Techniques, IEE Proceedings* -, 150(6): 403–411, Nov 2003.

[UHS16] T. Ujiie, M. Hiromoto, and T. Sato. Approximated prediction strategy for reducing power consumption of convolutional neural network processor. In *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 870–876, Jun 2016.

[VDS+08] K. Vandoorne, W. Dierckx, B. Schrauwen, D. Verstraeten, R. Baets, P. Bienstman, and J. Van Campenhout. Toward optical signal processing using photonic reservoir computing. *Optics Express*, 16(15): 11182–11192, 2008.

[Ver04] David Verstraeten. Een studie van de liquid state machine : een woordherkenner. *Master's thesis, Ghent University, ELIS department*, 2004.

[Ver09] D. Verstraeten. Reservoir computing: computation with dynamical systems. *Doctoral thesis, Ghent University*, Aug 2009.

[Vid07] A. K. Vidybida. Input-output relations in binding neuron. *BioSystems*, 89(1-3): 160–165, 2007.

[VMB15] S. I. Venieris, G. Mingas, and C. S. Bouganis. Towards heterogeneous solvers for large-scale linear systems. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sep 2015.

[VMVV+14] K. Vandoorne, P. Mechet, T. Van Vaerenbergh, M. Fiers, G. Morthier, D. Verstraeten, B. Schrauwen, J. Dambre, and P. Bienstman. Experimental demonstration of reservoir computing on a silicon photonics chip. *Nature Communications*, 5, 2014.

[Vre02] J. Vreeken. Spiking neural networks, an introduction. *Technical report UU-CS-2003-008, Institute for Information and Computing Sciences, Utrecht University*, 2002.

[Vre04] Jilles Vreeken. On real-world temporal pattern recognition using liquid state machines. *Master's thesis, Utrecht University, Institute for Information and Computing Sciences*, 2004.

[VSDS07] D. Verstraeten, B. Schrauwen, M. D'Haene, and D. Stroobandt. An experimental unification of reservoir computing methods. *Neural Networks*, 20(3): 391–403, 2007.

[VSS05] D. Verstraeten, B. Schrauwen, and D. Stroobandt. Reservoir computing with stochastic bitstream neurons. In *Proceedings of the 16th Annual ProRISC Workshop*, pages 454–459, Nov 2005.

[VSS06] D. Verstraeten, B. Schrauwen, and D. Stroobandt. Reservoir-based techniques for speech recognition. In *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, pages 1050–1053, Jul 2006.

[VSS07] D. Verstraeten, B. Schrauwen, and D. Stroobandt. Adapting reservoirs to get Gaussian distributions. pages 495–500, 2007.

[VSSVC05] D. Verstraeten, B. Schrauwen, D. Stroobandt, and J. Van Campenhout. Isolated word recognition with the liquid state machine: A case study. *Information Processing Letters*, 95(6 SPEC. ISS.): 521–528, 2005.

[WDH15] A. J. Wootton, C. R. Day, and P. W. Haycock. An Echo State Network approach to structural health monitoring. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, Jul 2015.

[web09] *Reservoir Computing, ORGANIC project (Self-Organized Recurrent Neural Learning for Language Processing)*, (March 2009). `http://reservoir-computing.org/organic`.

[web16a] *Astra*, (accessed December 20, 2016). `http://www.onastra.com`.

[web16b] *Eutelsat*, (accessed December 20, 2016). `http://www.eutelsat.com`.

[web16c] *Biomedical Advances: Neuron Damage*, (accessed June 7, 2016). `http://biomedicalengineering.yolasite.com`.

[web16d] *Altera FPGAs*, (accessed May 1, 2016). `https://www.altera.com/products/fpga/cycloneseries.html`.

[web17a] *MathWorks product documentation, multiple linear regression*, accessed February 8, 2017. `https://es.mathworks.com/help/stats/regress.html`.

[web17b] *Sharp Microelectronics. Sharp Distance measuring sensor*, (accessed January 16, 2017). `http://www.sharpsma.com`.

[web17c] *Altera download center, Quartus software*, accessed March 17, 2017. `http://dl.altera.com/`.

[Wer90] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10): 1550–1560, Oct 1990.

[WG15] A. S. Weigend and N. A. Gershenfeld. *The Santa Fe Time Series Competition Data*, (accessed September 1, 2015). `http://www-psych.stanford.edu/~andreas/Time-Series/SantaFe.html`.

[WGM+03] S. Wing, R. A. Greenwald, C. I. Meng, V. G. Sigillito, and L. V. Hutton. Neural networks for automated classification of ionospheric irregularities in HF radar backscattered signals. *Radio Science*, 38(4): 2–1–2–8, Aug 2003.

[WGS05] D. Wierstra, F. J. Gomez, and J. Schmidhuber. Modeling systems with internal state using evolino. pages 1795–1802, 2005.

[WHCE15] R. Wang, J. Han, B. Cockburn, and D. Elliott. Design and evaluation of stochastic FIR filters. In *Communications, Computers and Signal Processing (PACRIM), 2015 IEEE Pacific Rim Conference on*, pages 407–412, Aug 2015.

[WHH+89] 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*, 37(3): 328–339, Mar 1989.

[WHS13] T. Waegeman, M. Hermans, and B. Schrauwen. MACOP modular architecture with control primitives. *Frontiers in Computational Neuroscience*, (JUL), 2013.

[Wid60] B. Widrow. Adaptive "adaline" neuron using chemical "memistors". *Technical report 1553-2, Stanford Electronics Laboratory, Stanford, CA*, Oct 1960.

[WLL16] Q. Wang, Y. Li, and P. Li. Liquid state machine based pattern recognition on FPGA with firing-activity dependent power gating and approximate computing. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 361–364, May 2016.

[WRK15] F. Walter, F. Rohrbein, and A. Knoll. Computation by time. *Neural Processing Letters*, pages 1–22, 2015.

[WSLW95] Jhing-Fa Wang, An-Nan Suen, Jia-Ru Lee, and Chung-Hsien Wu. A bayesian neural network chip design for speech recognition system. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 2027–2030 vol.4, Nov 1995.

[WSS08a] F. Wyffels, B. Schrauwen, and D. Stroobandt. Stable output feedback in reservoir computing using ridge regression. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5163 LNCS(PART 1): 808–817, 2008.

[WSS08b] F. Wyffels, B. Schrauwen, and D. Stroobandt. System modeling with reservoir computing. In *Proceedings of the Benelearn*, volume 2008, pages 103–104, 2008.

[WTS08] Ben Williams, Marc Toussaint, and Amos J Storkey. Modelling motion primitives and their timing in biologically executed movements. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1609–1616. Curran Associates, Inc., 2008.

[XYH07] Y. Xue, L. Yang, and S. Haykin. Decoupled echo state networks with lateral inhibition. *Neural Networks*, 20(3): 365–376, 2007.

[YAT16] T. Yeoh, H. E. Aguirre, and K. Tanaka. Clothing-invariant gait recognition using convolutional neural network. In *2016 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, pages 1–5, Oct 2016.

[YD15] Dong Yu and Li Deng. *Automatic Speech Recognition: A Deep Learning Approach*. Springer Science & Business Media, 2015.

[YLN12] T. Yamanishi, J. Q. Liu, and H. Nishimura. Modeling fluctuations in default-mode brain network using a spiking neural network. *International Journal of Neural Systems*, 22(4), 2012.

[YM12] J. Yin and Y. Meng. Reservoir computing ensembles for multi-object behavior recognition. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Jun 2012.

[YT07] T. Yamazaki and S. Tanaka. The cerebellum as a liquid state machine. *Neural Networks*, 20(3): 290–297, 2007.

[ZB01] G. P. Zhang and V. L. Berardi. Time series forecasting with neural network ensembles: An application for exchange rate prediction. *Journal of the Operational Research Society*, 52(6): 652–664, 2001.

[ZL08] D. Zhang and H. Li. A stochastic-based fpga controller for an induction motor drive with integrated neural network algorithms. *IEEE Transactions on Industrial Electronics*, 55(2): 551–561, Feb 2008.

[ZM06] Quming Zhou and K. Mohanram. Gate sizing to radiation harden combinational logic. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(1): 155–166, Jan 2006.

[ZNS+13] Q. Zhao, K. Nakajima, H. Sumioka, H. Hauser, and R. Pfeifer. Spine dynamics as a computational resource in spine-driven quadruped locomotion. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1445–1451, Nov 2013.

[ZS06] M. Zhang and N. R. Shanbhag. Dual-sampling skewed CMOS design for soft-error tolerance. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 53(12): 1461–1465, Dec 2006.

[Zur92] Jacek M. Zurada. *Introduction to Artificial Neural Systems*. West Publishing Co., St. Paul, MN, USA, 1992.

[ZVDF96] Ming Zhang, S. Vassiliadis, and J. G. Delgado-Frias. Sigmoid generators for neural computing using piecewise approximations. *IEEE Transactions on Computers*, 45(9): 1045–1049, Sep 1996.

[ZYCS15] Qingchun Zhao, Hongxi Yin, Xiaolei Chen, and Wenbo Shi. Performance optimization of the echo state network for time series prediction and spoken digit recognition. In *2015 11th International Conference on Natural Computation (ICNC)*, pages 502–506, Aug 2015.

Nomenclature

AI	Artificial intelligence
ALR	Adjacent-feedback loop reservoir
ANN	Artificial neural network
ANS	Artificial neural system
ASIC	Application-specific integrated circuit
B2P	Binary to pulsed
BN	Binding neuron
BTT	Backpropagation through time
CI	Computational intelligence
CMOS	Complementary metal-oxide-semiconductor
CNN	Convolutional neural network
CPU	Central processing unit
CRJ	Cycle reservoir with jumps
DCR	Delay-coupled reservoir
DDE	Delay differential equation
DER	Digit error rate
DLR	Delay line reservoir
DLRB	DLR with feedback connections
DMR	Dual modular redundancy
DNN	Deep neural network
DSP	Digital signal processor

ECG	Electrocardiogram
EEG	Electroencephalography
ELM	Extreme learning machine
EPSP	Excitatory postsynaptic potential
ESL	Extended stochastic logic
ESN	Echo state network
ESP	Echo state property
FFNN	Feed-forward neural network
FIR	Finite impulse response
FPGA	Field programmable gate array
FSA	Finite-state automata
FSM	Finite-state machine
GPU	Graphical processing unit
HDL	Hardware description language
HMM	Hidden Markov model
HNN	Hardware neural network
IF	Integrate-and-fire
IoT	Internet of things
IPSP	Inhibitory postsynaptic potential
LDPC	Low density parity check
LE	Logic element
LED	Light-emitting diode
LFSR	Linear feedback shift register
LIF	Leaky-integrate-and-fire
LMS	Least mean squares

LSM	Liquid state machine
LSTM	Long short-term memory
MAC	Multiply-accumulate
ML	Machine learning
MLP	Multi-layer perceptron
MPC	Model predictive control
MSE	Mean square error
NARMA	Nonlinear autoregressive moving average
NMSE	Normalized mean square error
NRMSE	Normalized root-mean-square error
P2B	Pulsed to binary
PC	Personal computer
PDA	Personal digital assistant
PV	Photovoltaic
RAM	Random access memory
RBF	Radial basis function
RC	Reservoir computing
RLS	Recursive least squares
RNG	Random number generator
RNN	Recurrent neural network
SC	Stochastic computing
SCR	Simple cycle reservoir
SDRAM	Synchronous dynamic RAM
SER	Symbol error rate
SEU	Single event upset

SNG	Stochastic number generator
SNR	Signal-to-noise ratio
SRAM	Static random access memory
SSN	Stochastic spiking neuron
SSNN	Stochastic spiking neural network
SVM	Support vector machine
TDR	Time-delay reservoir
TMR	Triple modular redundancy
VHDL	VHSIC hardware description language
VHSIC	Very high speed integrated circuit
VLSI	Very large scale integration
WSN	Wireless sensor network
WTG	Wind turbine generator